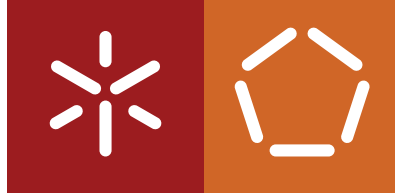


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Márcio da Silva Rocha

Distributed Game

May 2022



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Márcio da Silva Rocha

Distributed Game

Master dissertation
Master's in Informatics Engineering

Dissertation supervised by
José Orlando Roque Nascimento Pereira

May 2022

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

LICENSE GRANTED TO USERS OF THIS WORK:



CC BY

<https://creativecommons.org/licenses/by/4.0/>

ACKNOWLEDGEMENTS

I would like to thank Professor Doctor José Orlando Roque Nascimento Pereira for all the guidance throughout the work and availability demonstrated. The clarification of doubts and shared knowledge was fundamental to the success of the result achieved. Lastly, I would like to thank my parents, my brother and my friends for all the support, affection and advice during my academic path. This dissertation would not be concretizable without the help of all of them.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

The demand for online games has risen over the years, expanding multiplayer support for new and different game genres. Among them are Massively Multiplayer Online games, one of the most popular and successful game types in the industry. Nowadays, this industry is thriving, evolving alongside technological advancements and producing billions in revenue, making it an economic importance. However, as the complexity of these games grows, so do the challenges they face when constructing them.

This dissertation aims to implement a distributed game, through a proof of concept or an existing game, using a distributed architecture to acquire knowledge in the construction of such complex systems and the effort involved in dealing with consistency, maintaining communication infrastructure, and managing data in a distributed way. It is also intended that this project implements multiple mechanisms capable of autonomously helping manage and maintain the correct state of the system.

To evaluate the proposed solution, a detailed analysis is carried out with performance benchmark analysis, stress testing, followed by an examination of its security, scalability, and distribution's resilience.

Overall, the present research work allowed for a greater understanding of the technologies and approaches used in constructing a gaming system, establishing a new set of development opportunities to be further investigated upon the constructed solution.

Keywords: Massively Multiplayer Online Games, Distributed System, Gaming Architecture, Online Game, Game Server, Interest Management, Synchronization, Scalability, Networking, Replication, Consistency Control

RESUMO

A procura por jogos online aumentou ao longo dos anos, expandindo o suporte multiplayer para novos e diferentes géneros. Entre estes estão os jogos *Massively Multiplayer Online*, um dos tipos de jogos mais populares e bem-sucedidos na indústria. Atualmente, esta indústria está a prosperar, evoluindo com os avanços tecnológicos e gerando milhares de milhões em receita, tornando-se uma importância económica. Porém, à medida que a complexidade destes jogos aumenta, também aumenta os problemas encontrados durante a sua construção.

Esta dissertação tem como objetivo implementar um jogo distribuído, através de uma prova de conceito ou um jogo existente, usando uma arquitetura distribuída a fim de adquirir conhecimento na construção destes sistemas complexos e o esforço envolvido em lidar com consistência, manter a infraestrutura de comunicação e gerir dados de maneira distribuída. Para isto, é pretendido que este projeto também implemente vários mecanismos capazes de, de forma autónoma, ajudar a gerir e manter o correto estado do sistema.

Para avaliar o solução proposta, uma análise detalhada é realizada sobre o desempenho, segurança, escalabilidade e resiliência da distribuição do sistema.

De forma geral, o presente trabalho de pesquisa permitiu uma maior compreensão das tecnologias e abordagens utilizadas na construção de um sistema de jogos, estabelecendo um novo conjunto de oportunidades de desenvolvimento a serem investigadas sobre a solução construída.

Palavras-Chave: Jogo MMO, Sistema Distribuído, Arquitetura de Jogo, Jogo Online, Servidor de Jogo, Gestão de Interesse, Sincronização, Escalabilidade, Rede, Replicação, Controlo de Consistência

CONTENTS

Contents [iii](#)

1	INTRODUCTION	1
1.1	Objectives	1
1.2	Methodology	2
1.3	Structure	3
2	STATE OF THE ART	4
2.1	MMOGs	4
2.2	Design principles of an MMOG	5
2.2.1	Game States	5
2.2.2	Game Replication	6
2.2.3	Game Latency Tolerance	7
2.3	Foundations and Key Issues in a Gaming System	8
2.3.1	Communication	8
2.3.2	Bandwidth Requirements	9
2.3.3	Frame Rate	10
2.3.4	Interest Management	10
2.3.5	Consistency Control	12
2.3.6	Dead Reckoning	13
2.3.7	Lag Compensation	15
2.3.8	Synchronization	16
2.3.9	Security and Cheating	19
2.4	Architectures	20
2.4.1	Client-Server Architecture	21
2.4.2	Multi-Server Architecture	22
2.4.3	Peer-To-Peer Architecture	24
2.4.4	Comparison	25
2.4.5	Hybrid Architecture	26
2.4.6	Decentralized Virtual Environments	28
3	ARCHITECTURE AND IMPLEMENTATION	31
3.1	Game Developed	31

3.2	Approach	33
3.3	Implementation	35
3.3.1	GameClient - Client Application	36
3.3.2	GameServer - Server Application	37
3.4	Communication	39
3.4.1	Transport Layer Protocol	39
3.4.2	Message Queue	40
3.4.3	Data Encapsulation	41
3.5	Architecture	44
3.5.1	Server	47
3.5.2	Room	48
3.5.3	Client	49
3.5.4	Game	50
3.6	Application Protocols	54
3.6.1	Messages	54
3.6.2	New Connections	55
3.6.3	Matchmaking	56
3.6.4	Join Room	57
3.6.5	Room Start and End	57
3.7	Game Protocols	59
3.7.1	Server-side protocol	59
3.7.2	Client-side protocol	60
3.7.3	UDP Pong - Protocol Design	61
3.7.4	Tick based system	63
3.7.5	Entity Interpolation	66
3.7.6	Player synchronization	68
3.7.7	Lag Compensation	73
4	ANALYSIS AND DISCUSSION	75
4.1	Analysis	75
4.1.1	Experiments And Experimental platform	75
4.1.2	Performance	76
4.1.3	Scalability	82
4.1.4	Security	85
4.1.5	Resilience	86
4.2	Discussion	88
4.2.1	Performance	88
4.2.2	Scalability	88

4.2.3	Security	89
4.2.4	Resilience	90

5 CONCLUSION 91

5.1	Conclusions	91
5.2	Limitations	92
5.3	Future Work	93

LIST OF FIGURES

Figure 1	Research Onion Diagram	2
Figure 2	Object types and interactions	6
Figure 3	Multiplayer game components	8
Figure 4	Layers of a node	9
Figure 5	Bandwidth requirements of games	10
Figure 6	Aura-Nimbus model	11
Figure 7	Different game zoning mechanisms	11
Figure 8	Illustration of Dead reckoning	13
Figure 9	Illustration an example of dead reckoning	14
Figure 10	Illustration and example of lag compensation	16
Figure 11	Conservative Algorithms	17
Figure 12	Optimistic Algorithms	18
Figure 13	Topology for Multiplayer Online Game systems	20
Figure 14	Different maintained game worlds	21
Figure 15	Example of hybrid approaches	27
Figure 16	Illustration of Pastry message routing	29
Figure 17	Architecture Overview	33
Figure 18	Client application main menu (mockup)	36
Figure 19	Server console application	37
Figure 20	Server properties file	38
Figure 21	MessageQueuer update method	41
Figure 22	Serialization example	42
Figure 23	Deserialization example	42
Figure 24	System's packet types	43
Figure 25	GameServer's Class Diagram Overview	45
Figure 26	GameClient's Class Diagram Overview	46
Figure 27	Server Class Diagram Overview	47
Figure 28	Room Class Diagram Overview	48
Figure 29	Client Class Diagram Overview	49
Figure 30	Game Class Diagram Overview	50
Figure 31	Player Class Diagram Overview	52
Figure 32	RemotePlayer Class Diagram Overview.	53
Figure 33	New Connection Process	55

Figure 34	Server-Client message system overview	59
Figure 35	Client-Client message system overview	60
Figure 36	Ping message	62
Figure 37	Ping calculation	62
Figure 38	Heartbeat timer	63
Figure 39	Game Update	63
Figure 40	Logic Timer	65
Figure 41	PlayerState Class	66
Figure 42	Interpolator's update loop	67
Figure 43	Player update on the client	69
Figure 44	Player update on the server	70
Figure 45	Reconciliation	71
Figure 46	Server incoming network throughput per client	77
Figure 47	Server outgoing network throughput per client	77
Figure 48	Server packets sent per second	78
Figure 49	Server incoming request rate per client	79
Figure 50	Server request process rate per client	79
Figure 51	Server response rate at the client application	80
Figure 52	Server average processor usage per room	80
Figure 53	Server average processor usage per client	81
Figure 54	Server average processor usage by tick rate	81
Figure 55	Server average number of packets received by tick rate	82
Figure 56	Server incoming network throughput per client	83
Figure 57	Server outgoing network throughput per client	83
Figure 58	Server execution time breakdown	84
Figure 59	Server disconnection rate	85

LIST OF TABLES

Table 1	Command classifications for Quake	12
Table 2	Characteristics of the three architectures' systems	25

ACRONYMS

- AFK** away from keyboard. 61, 85
- AI** Artificial Intelligence. 4
- CS** Client-Server. 20–23, 25–27, 34, 61, 91
- DHT** Distributed Hash Table. 28, 29
- DNS** denial-of-service. 20
- FPS** First-Person Shooter. 6, 7, 10, 11, 17, 31, 32, 65
- GTV** Global Virtual Time. 17
- Hz** Hertz. 10
- IM** Interest Management. 10, 11, 88
- MMO** Massive Multiplayer Online. 1, 3, 4, 6, 8–10, 12, 20, 24, 31, 44, 91, 92
- MMOG** Massive Multiplayer Online Game. 1, 4, 5, 12, 21, 22, 24, 26, 28, 91
- MS** Multi-Server. 20
- MUD** multi-user dungeons. 4
- NPC** Non-Player Character. 5, 6
- OLTP** online transaction processing. 75
- P2P** Peer-to-Peer. 5, 19, 20, 24–30, 33, 34, 85, 89, 91
- PCM** Predictive Contract Mechanism. 13
- RTP** Real-time Transport Protocol. 28
- RTS** Real-Time Strategy. 7, 17
- RTT** round trip time. 62
- RUDP** Reliable User Datagram Protocol. 88, 93
- TCP** Transmission Control Protocol. 8, 13, 35, 39, 40, 48, 54–57, 77, 86, 88, 91, 93
- TSS** Trailing state synchronization. 18
- TWS** TimeWarp synchronization. 17, 18
- UDP** User Datagram Protocol. 8, 13, 28, 35, 39, 40, 48, 54, 55, 57, 61, 62, 77, 86, 88, 91, 93
- UI** User Interface. 36, 49, 51

UTC Coordinated Universal Time. 61

VM Virtual Machines. 92

WoW World of Warcraft. 1, 28

INTRODUCTION

A [Massive Multiplayer Online Game \(MMOG\)](#), also commonly denominated as [Massive Multiplayer Online \(MMO\)](#), is an online game with often hundreds or thousands of players playing simultaneously on the same server. Such games are a thriving business, being one of the most famous [World of Warcraft \(WoW\)](#). Due to their complexity and magnitude, they can create tremendous processing loads and network traffic ([Hu et al. \(2006\)](#); [Chen et al. \(2005\)](#); [Suznjevic and Matijasevic \(2012\)](#)), making scalability an enormous challenge when creating this type of game.

Traditionally, multiplayer games have used a client-server communication architecture. Although having a lot of advantages and simple to secure, it is also a source of many drawbacks, which will later be discussed into further detail, being one of the most critical the fact that it isn't scalable. Many researchers have been studying alternative architectures, such as Peer-To-Peer, which solves the problem of scalability but brings others, such as the lack of centralized power to maintain consistency. Even though there are various architectural options for the construction of a distributed game, the game itself has to be considered when selecting them due to its characteristics and requirements.

In addition, to successfully create a distributed game, the architecture alone is not sufficient to maintain the correct distribution and management of the information across all participants. For this, there are several strategies, algorithms, and mechanisms that a game system can implement to maintain the correct game state while distributed.

1.1 OBJECTIVES

Given the extensive research regarding the development of gaming architectures and their mechanisms, the formal objectives of this work are:

- Acquisition of skills in the analysis and design of multiplayer game systems;
- Identify common development strategies and approaches utilized in multiplayer game development;
- Select appropriate technology to develop concerning the game at hand, and necessary mechanisms to ascertain consistency, synchronization, and playability;
- Develop a multiplayer game system based on the selected technology as a result of compliance with the third objective, as a prototype of the distributed game system;

- Understand the effort faced in the development of scalable and network resistance applications for multi-player game;
- Evaluate the constructed system's aspects of performance, scalability, security and distribution.

1.2 METHODOLOGY

After reflection and research, it was pertinent to conduct the work following the research methodology *Research Onion*. The *Research Onion* diagram represented in Figure 1, and developed by Saunders (Mark N.K. Saunders (2019)), is used recurrently to describe the issues underlying the choice of data collection techniques and analysis procedures that occur during an investigation. It describes the issues that an investigator must respond to in such a way as to define a coherent and complete methodology for its investigation, beginning from the outermost layers to the core. According to the diagram, six different divisions are identified, these being: Philosophies, Approaches, Chosen methods, Strategies, Temporal horizon, and Techniques and procedures.

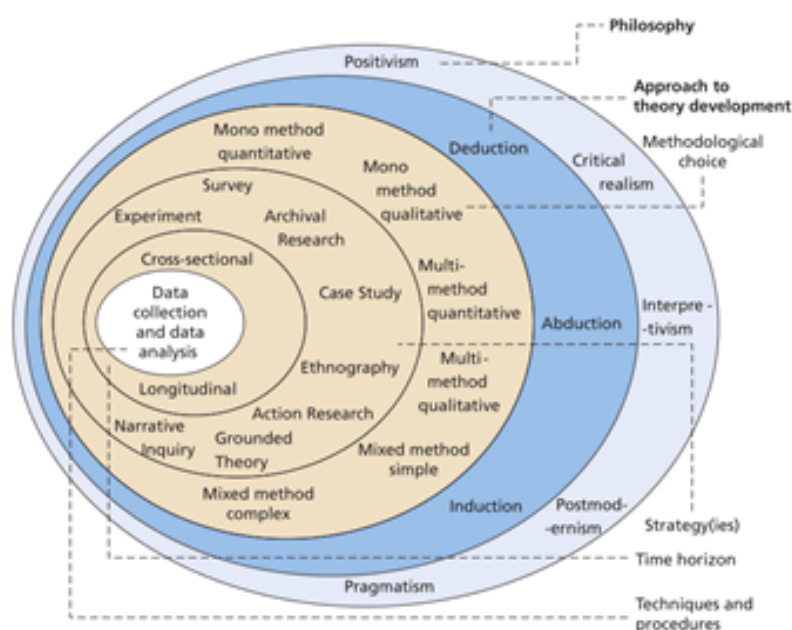


Figure 1: Research Onion Diagram. source: Mark N.K. Saunders (2019)

Following this diagram, the first question is related to the philosophy's identification. This work followed the *Realism* philosophy, since objects exist independently of our knowledge about their existence, and that the senses show us that it is reality, it is the truth. The work also faces epistemology, which is based on observable phenomena and which are likely to be studied through data and facts (Mark N.K. Saunders (2019)).

The research's topic addressed consists of developing and implementing a multiplayer game system, which aimed to distribute a game and evaluate its capabilities. Chosen the topic, a bibliographic review was held with the research topic following a *Deductive* approach, which involves the development of a theory and strategy

research that subjects the theory referred to a rigorous test through a series of propositions (Mark N.K. Saunders (2019)). In this approach, when confirming that premises are true, it is confirmed that the conclusion is correct.

Referring to the methodological choice, it was used the *concurrent mixed methods* study research design, which involves the separate use of quantitative and qualitative methods within a single phase of data collection and analysis (single-phase research design).

The research strategy followed the form of *Case Study* due to the development of detailed and intensive knowledge of one case, which the same is obtained through the use of appropriate techniques. Therefore, an investigation occurs at the same time as the production process, where it is considered as a tool for the resolution and simplification of problems (Mark N.K. Saunders (2019)).

Finally, the temporal horizon of this study was *transversal*, which represents a study of a particular phenomenon at a specific time, set by the University of Minho.

1.3 STRUCTURE

The remainder of this work is structured as follows. [Chapter 2](#) examines common game design principles used in most multiplayer games, along with several key concepts of mechanisms used in their design and construction. Then, it finishes with the discussion of several architectures used to construct an [MMO](#) and a comparison of their characteristics.

[Chapter 3](#) starts by addressing the game developed, used as the object of distribution in the system, and also provides some key insights to consider in its distribution. Furthermore, it presents the proposed architecture selected to develop, explaining how it works, the different components, roles, and capabilities. Lastly, it describes the implementation of the system developed, giving detail about its workflow, functionalities, structure, and application and game protocols.

[Chapter 4](#) discusses the results acquired from the system's analysis through multiple experiments conducted. It also presents the dissertation's conclusions through a retrospective assessment of the realized work and raises some issues for future work.

Finally, [Chapter 5](#) concludes the document, giving an overview of the central findings of the research and its limitations, presenting as well future work as a complement to the work studied in this document.

STATE OF THE ART

This chapter provides an understanding of basic concepts in a multiplayer game that is independent of the architecture chosen to do it on. Furthermore, explains some mechanisms used in distributed gaming architectures to achieve a functional multiplayer game. Moreover, it analyzes key issues to the understanding and construction of distributed gaming systems. Afterward, discusses the various architectures used to develop MMO games and some other approaches.

2.1 MMOGS

A video game is an electronic game involving interaction from the user via an interface or an input device, giving them visual feedback as their interactions are processed by the game. Games can also be multiplayer, meaning more than one person can play the game simultaneously, this is possible by either playing locally (on the same machine) or online (over the internet). Players can compete against other players, in teams, or even against the game, meaning Artificial Intelligence (AI). These games can be non-networked, such as in shared screen games, or networked, allowing players to play together over a greater distance, entering the MMOGs.

Massively Multiplayer Online Games or MMOGs are, as the name suggests, games that are played by a massive number of players simultaneously online. Usually, these games take place in a single shared virtual gaming environment that, after downloading the game software, players can enter. The game world is often persistent, meaning immutable and always present, running independently from the existence of active players. However, it can also be instantiated, being created for temporary necessity and disposed of afterward.

The explosive growth of MMOGs has prompted many game designers to build online multiplayer modes into many traditionally singleplayer games, and today represent a billion dollars business ¹, being also used in simulation and for military purposes. The first MMOGs were the evolution of multi-user dungeons (MUD), a very limited text-based game ². This was a very simple text-oriented game because the computational power and bandwidth were very limited at the time (late 1970s). Nowadays MMOGs offer complex 3D graphics and a massive virtual world to explore.

Early MUDs were very modest applications based on a Client-Server architecture. However, with the increasing complexity of games and players' interactions, these systems could not keep up. Even though computers were in a high development to become more powerful and faster, it still was a far too great load of messages

¹ <https://www.statista.com/statistics/346515/leading-f2p-mmo-games/>

² https://pt.wikipedia.org/wiki/Multi-user_dungeon

to be treated by a single machine. Nowadays we continue to use this same client-server architecture, but with clusters of servers that can handle that same load of messages to process.

As MMOGs are resource-intensive and complex, resulting in a lot of problems and complications along the way of production and still afterward, make them a very popular research topic in distributed architecture studies. Developing an MMOG takes a lot of time and resources, because of this a lot of work is put into them in order to achieve an optimized result. For this, there is intensive research put into developing better and more efficient ways to distribute the load among the network, while guaranteeing some level of availability, security, and performance.

As result, the majority of research found has been focused on improving the existing Client-Server architecture that is predominant in much of the MMOG still today. This research has also been focused on developing a more efficient way to distribute the server architecture and the communication between various parts that form the system, to increase the amount of load that it can handle. Another major part of the research done for MMOG is, so far, mostly for academic studies³, focused on replacing outdated systems for a distributed architecture, such as Peer-to-Peer (P2P), in new and innovative ways. These architectures try also to eliminate the need for such a big and expensive cluster of servers, which are not affordable for most start-up companies developing products that depend on having them. There is also interest in developing relatively low complexity architectures that may catch the interest of developers that prefer Client-Server architecture due to its overall simplicity, and incentivize them to use these alternative systems.

2.2 DESIGN PRINCIPLES OF AN MMOG

The core premise in most multiplayer games is that the player plays the role of a character in a fictional world, taking on missions or quests, completing objectives, or slaying enemies, alone or as part of a group, requiring him to traverse different parts of the world, engaging with various players or Non-Player Character (NPC)s, while being rewarded with money or experience points, which allows the character to evolve, get new abilities and so on (Knutsson et al. (2004)). Despite the variation of the game premise, there are some basic concepts that every multiplayer online game follows, independently from the game or architecture chosen to develop it on. Subtracting this superficialities, this section goes into further detail on the concepts involved and execution patterns in the design of a multiplayer game.

2.2.1 Game States

In most modern multiplayer games, the game world is usually made of four object types (Knutsson et al. (2004)): (1) immutable objects, such as the terrain or landscape objects, usually designed by a game artist, not changing during any state of the game; (2) characters or avatars (Suznjevic et al. (2009)), controlled by the player; (3) mutable objects, such as weapons, tools, food, or even landscape information, for example, players can interact with food on the ground and eat it, changing the food's position and deleting its object after the player is done

³ <http://vast.sourceforge.net/relatedwork.php>

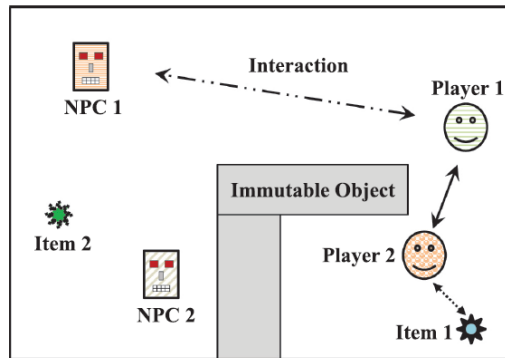


Figure 2: Object types and interactions. source: [Yahyavi and Kemme \(2013\)](#)

eating it; (4) **NPC**, such as villagers or characters, are part of the simulation taking a role in the game world and are not controlled by players but by automated algorithms. In [Figure 2](#) it is shown an example of these different object types.

These interactions controlled by the player, also called player interactions ([Knutsson et al. \(2004\)](#)), are usually confined to three types: (1) player update, consisting of interactions with the game world that only affect himself. Depending on the game, player updates may contain his position in the game world, the state of his character, such as health, money, abilities, items, etc. A character can be permanent until the removal of the player from the game or instantiated for a specific purpose and then deleted; (2) player-object interaction, is an interaction between players and mutable objects, including **NPCs**, in the game world; (3) player-player interaction, is an interaction between players and other players in the game world, for example, trading an item with someone would change both players' inventory and fighting another player would change both of their player objects' health.

These types of interactions are important when dealing with concurrent and conflicting updates due to their importance over others. Cheat prevention techniques are only applied to certain types of interactions ([Knutsson et al. \(2004\)](#)) that might benefit a player over others, for example, a player giving himself great amounts of in-game currency out of thin air, or instantaneously kill all other players in the game world. In **First-Person Shooter (FPS)** games, proof-cheating is crucial when dealing with player interactions, much advantage can be taken from these, such as a player allowing himself to see other players through walls. Combined with architecture features, **MMO** games are coordinated through either a centralized or a distributed structure. These differ in the location wherein the game state is maintained, and the mechanism used to achieve synchronization. As such, these mechanisms may vary depending on the architecture ([Hsu and Kuo \(2003\)](#)) and the results they are trying to achieve, as discussed ahead.

2.2.2 Game Replication

In a multiplayer game, when a player enters the game, they receive the information needed to create an instance of the game environment based on the server's simulation and its current game states. It is a combination of

local resources oriented through remote information. Both entity states are managed under the Client-Server architecture by the server, which determines the game state based on client inputs and tells them of the new calculated game state. Thus, an instance of their character is created on their local game environment, which the player can control through input messages. This is because, for each mutable object and character presented on the local game world, there is also an authoritative copy called the primary or master copy, which remains in the simulation of the server.

For most game engines first, there is an alteration on the primary copy (server's object) and then on the secondary copy (player's local object) (Yahyavi and Kemme (2013)). For a player to make an update on their copy they first have to send an update message to change the authoritative copy, and then the holder decides whether or not to accept the update received and send the updated object (or just components of what is needed to change the object) to all of those who have the secondary copy. However, under a distributed system each player retains its own entity status, tells other players of its decisions, and addresses any issues of consistency without the use of any centralized authority. Another difference between them is that in a centralized system each client sees only a partial vision of the game world (only the server sees it entirely), while in a distributed system each player manages his own world.

Apart from persistent player states and immutable objects, most game states are rebuilt at some point later (Knutsson et al. (2004)), an example of this is the eventual depletion of food that a player can grab from the ground or the finite number of enemies a player can kill in the game world without replenishment through respawn. For this to happen, these examples also require a follow up message to be sent, by the holder, to every player in the game world, in order to see these changes happen locally on the clients.

2.2.3 Game Latency Tolerance

Latency is referred to as the interval between the execution of an update to the primary copy of the object and the replica receiving an update to the object. This latency depends, as it will be addressed, on the architecture and the delays in networking.

As there are immense types of multiplayer games, the player tolerance for network latency varies from these types. What for some games can be considered crucial, to others may not be as important for an optimal gameplay. Higher latencies than the game acceptable limits can harm game playability and customer retention. The gameplay and architecture have a huge impact on the latency and its tolerance. Thus, an architecture is only viable if it fulfills the criteria of game latency. Most games can usually accommodate latencies of between 100 and 300 milliseconds, although there are exceptions that have higher latency tolerance.

Some examples of latency requirements' variation are FPS and Real-Time Strategy (RTS) games (Yahyavi and Kemme (2013); Armitage (2003)). In an FPS game, the gameplay is centered on simple and short-lived actions, focused on the direct control of the player's avatar, and as result, this type of games can only tolerate low latencies. An example of this is *Quake 3* with a latency requirement of fewer than 180 milliseconds (Armitage (2003)). Yet, games such as RTSs can tolerate lower responsiveness because their core gameplay is focused on strategy rather than direct interaction. Usually in this type of games the player commands the character to make

long and lasting actions, and in most cases, the movement type and/or actions are limited options or pre-planned, making it a predictable interaction, where their ordered execution is more important.

2.3 FOUNDATIONS AND KEY ISSUES IN A GAMING SYSTEM

This section introduces general concepts, execution patterns, and mechanisms used to achieve a functional game when implementing a distributed game architecture. In Figure 3 it is shown an overview of the various components of the a multiplayer game system sample which some are discussed in this chapter.

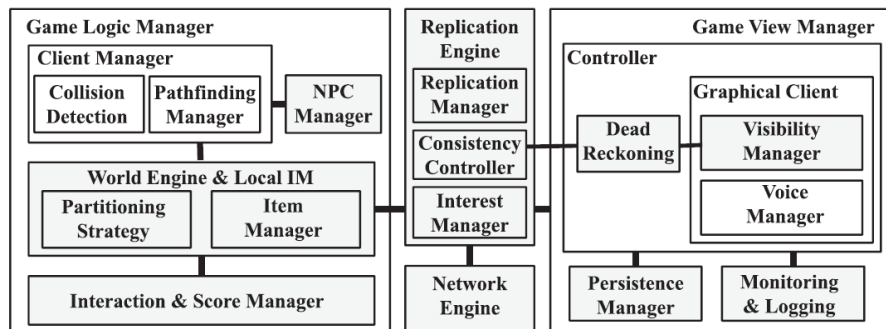


Figure 3: Multiplayer game components. source: [Yahyavi and Kemme \(2013\)](#)

2.3.1 Communication

As previously mentioned, a multiplayer game needs to share information across all participants in order to mirror the game in every instance being run. Figure 4 shows an example of the process that multiplayer games perform to receive and treat game information. This process is explained following an example of player replication from Node A to Node B, through player state messages.

In order to share information between nodes, a multiplayer game has to employ *transport layer protocols*, such as [Transmission Control Protocol \(TCP\)](#) and [User Datagram Protocol \(UDP\)](#). Different protocols implement different rules in how information should be transmitted across the network. For example, some ensure the reliability of message delivery while others focus on fast delivery. Therefore, these must be used depending on the information sent since the choice between them, in turn, impacts other aspects of game design and implementation. For instance, using a transport layer that does not ensure reliability will make the information susceptible to being lost along the way. This missing information can affect the mirroring of the game, potentially causing some problems, which can be only be solved by implementing mechanisms capable of handling such events, as will be discussed in this chapter. Nevertheless, considering that information between Node A and Node B is ensured, the transport layer protocol sees that the player state from Node A arrives at Node B.

Upon receiving a player state, Node B adds it to a *message queue*, which is a sophisticated protocol that guarantees a message is received and correctly processed. [MMO](#) games tend to implement a priority system in

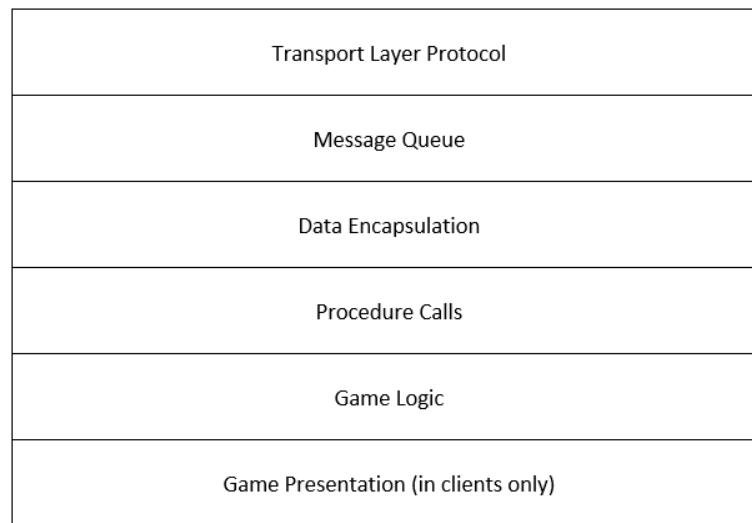


Figure 4: Layers of a node.

the message queue, processing specific messages that are more crucial to the game's well-functioning first, and then the rest.

To share information across the network, the game must format data in a way that can be used by a transport layer protocol. The process is called serialization, which converts an object into a stream of bytes with the main purpose of saving its state. In addition, the process can be reversed, called deserialization, which restores the object to its original format. This process is executed by a *data encapsulation* library that covers high-level (serialization) and low-level (deserialization) conversions. So, when Node B receives a player state message, and the message queue sees fit to process it, the data encapsulation library uses deserialization methods to transcribe it into usable game information. Finally, using a procedure call, the information is passed onto the game logic that handles and applies it to the game. Thus achieving a replication of the player between the two nodes.

2.3.2 Bandwidth Requirements

Bandwidth requirements for MMO games can be determined based on the average message size, number of players, and update rate (Suznjevic and Matijasevic (2012); Chen et al. (2006)), considering also its architecture. The more players active, the more bandwidth the overall system will require. Therefore, games with thousands of players, will require high bandwidth to support high amounts of players taking place in one location. Moreover, events in the game world like raids, battles against other players, which require/attract many players or increase the world activity in general, can cause spikes in the requirement of a higher bandwidth size. Game servers also deal with these spikes by over-provisioning (Yahyavi and Kemme (2013)).

Fast-paced games like an FPS, such as *Halo 3*, while not holding as much players in one place, will require a higher update rate, as shown in Figure 5. This, consequently, increases the bandwidth requirements, since more messages are transmitted across the nodes. In general, due to the limited bandwidth resources from clients,

application/platform	avg.	payload size (bytes)		avg. bandwidth requirement	
		min	max	pps	bps
Anarchy Online(PC)‡	98	8	1333	1.582	2168
World of Warcraft (PC)	26	6	1228	3.185	2046
Counter Strike (PC)	36	25	1342	8.064	19604
Halo 3	247	32	1264	27.778	60223
Gears of War (XBOX 360)	66	32	705	2.188	10264
Tony Hawk's Project 8 (XBOX 360)	90	32	576	3.247	5812
Test Unlimited (XBOX 360)	80	34	104	25	22912

Figure 5: Bandwidth requirements of games. source: [Harcsik et al. \(2007\)](#)

games tend to compress data packet in order to cover more people. As data packets tend to be very small and have high heterogeneity, classical compression shows little efficiency. A much more effective way to compress the data packets is a clever content consideration. This can be translated into only sending relevant changes since the last update (delta synchronization) or applying a round down of the values, since most values that need to be exact on the server state, don't need the same accuracy on the client side.

2.3.3 Frame Rate

In relation to the synchronization, games implement a *discrete event loop* ([Yahyavi and Kemme \(2013\)](#)), also referred to as a frame, which represents all events executed between the last event loop execution and now. The frequency of updates is also referred to as frame rate in [Bharambe et al. \(2008\)](#), and [Hsu and Kuo \(2003\)](#) states that a game's loop should be executed 30 times per second in order to maintain real-time property. However this is not always needed, allowing for the possibility of even lower frame rate. Note that a game's frame rate is separate from the screen it is being displayed on (graphical frame rate) as a display has its own frequency (or "refresh rate"), measured in [Hertz \(Hz\)](#). A decrease in the frame rate can be explained by an increase in the necessary computation needed to be calculated and executed in a determined loop. When games demonstrate low frame rates the gameplay experience can degrade or even become unplayable. For fast-paced competitive games, such as [FPSs](#), it is required from the server to run at high frame rates for fast and deterministic responses.

2.3.4 Interest Management

In most [MMO](#) games, players do not need to know the activities of all other users in the game, who may be far away from what the player can see. The communication of an [MMO](#) game network is expected to contain only the critical information to save as much bandwidth as possible for scalability reasons. [Interest Management \(IM\)](#) ([Smed et al. \(2002b\)](#); [Yahyavi and Kemme \(2013\)](#)) is an important mechanism used to reach this goal, serving as a filter to remove irrelevant messages. It can minimize network traffic and reduce the burden on the client and on the server. The general idea is that in a multiplayer game, players have restricted vision and movement

capabilities, which does not require them to know everything happening inside the game world. As result many games show a fraction of the game world to the player, limiting the data that they can access to only receive the game states relevant to them. Thus, **IM** is important for scalability, because it minimizes the number of messages that the protocol should send out to make the player's game world consistent, keeping the network overhead low.

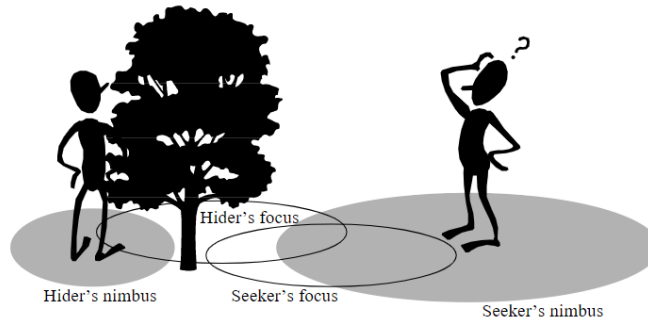


Figure 6: In hide-and-seek, the seeker is not aware of the hider because the hiding person's nimbus is smaller than the seeker's. However, because of the seeker's nimbus being larger and overlapping the hider's focus, the hider can see the seeker. source: [Smed et al. \(2002b\)](#)

The calculation of **IM** is centered on the space-based aura-nimbus model ([Benford and Fahlén \(1993\)](#)) that in turn corresponds to the system's sensing capabilities that is being modeled. Aura also called area of interest, is the area around the player's character, in **IM** the areas are always symmetric ([Smed et al. \(2002a\)](#)), meaning if they overlap, then both entities receive messages from each other. Furthermore, the aura can be split into a nimbus and a focus, which represent, respectively, the observed object's perceptivity and the observer's perception, meaning in order for a player to gain awareness of the other, their focus must overlap with them. This filter is very critical to games like **FPSs** because it would not be fair if two players could not see each other if looking directly at each other. An example of **IM** is shown in [Figure 6](#).

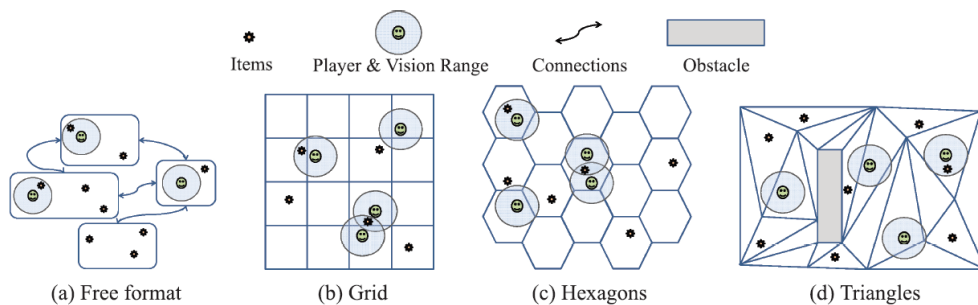


Figure 7: Different game zoning mechanisms. source: ([Yahyavi and Kemme \(2013\)](#))

The most commonly used mechanism for **IM** is Zoning ([Yahyavi and Kemme \(2013\)](#)). It works by partitioning the game world into smaller sections, denominated regions, or zones. This partitioning has different approaches, shown in the [Figure 7](#) and also discussed in [Chen et al. \(2005\)](#), which can result in the different shapes and how

they are mapped. Zones can also be totally different instances, and in that case, players must migrate to a new zone, often in a different server. This migration can also be transparent thus not affecting the player's experience.

2.3.5 Consistency Control

Event ordering, consistency, and synchronization form the backbone of any distributed system (GauthierDickey (2004)). In a distributed architecture such as a multiplayer game, concurrent and possibly conflicting updates can be made on different machines in different orders depending on when it is received, resulting in inconsistent states. These inconsistencies exist due to the execution of simultaneous and overlapping updates, for which mechanisms are put in place to avoid or correct these events. For example, if two players try to finish off an enemy at the same time, a consensus must be reached and then both players should see the same order of events happen in their local simulation of the game. This consensus is reached depending on the architecture and the algorithm implemented. In a client-server system the server will receive both events, determine in which order to execute them and share the results with all replicas in the network. Peer-to-Peer usually reaches the consensus by assuming one of the two is correct propagating their outcome or assigning a leader to take charge of that task.

Consistency requirements may vary widely depending on events type of events in the game and their variation in complexity (Bharambe et al. (2008)). These may represent different object and interaction types (Zhang and Kemme (2011)), implementing different levels of consistency, by various mechanisms, ranging from no consistency to exact consistency, impacting the synchronization as well. For example, frequently executed actions (at real-time) might not require a high level of consistency if they can be easily repeated. Some further examples of the mentioned various can be seen in Table 1, enumerating different Quake commands. In this case, consistent real-time events are the most challenging events as they have both strict timeliness and consistency requirements. Furthermore, in many MMOs there are objects considered to be valuable that can be sold or traded for real money (Yahyavi and Kemme (2013)). This makes consistency control a critical requirement for this type of games.

Event Type	Event Description	Event Property
Move event	An avatar moves to a new position	Real-time
Fire event	An avatar fires a rocket	Consistent real-time
Impact event	The rocket impacts and detonates	Consistent real-time
Damage/Die event	An avatar takes damage and possibly die	Consistent
Spawn event	An avatar is reborn in a random part of the map	Consistent

Table 1: Command classifications for Quake. source: (Hsu and Kuo (2003))

Stale views are also a problem of consistency state in MMOGs. As mentioned before, architectures that implement the client-server solution have to replicate the game from the primary copy on the server. This solution also requires that all updates must be first executed on the primary copy and then distributed among all clients. Although this simplifies consistency management, replicas will only receive the new update sometime after it

has occurred in the server. During this waiting period replicas are stale, and players might initiate invalid update requests to the primary copy, based on these stale values. For example, as explained in [Varvello et al. \(2011\)](#), in the game *Second Life*, on a crowded region, players have an inconsistent view of their neighbor's avatars, meaning they see them at a wrong location. Ideally, the primary copy would directly send updates to all replicas in order to minimize staleness. However, this is expensive and other methods may be employed that may increase the latency and staleness experienced by the replicas.

However, inconsistent states can also be the result of packet loss, due to most games using unreliable **UDP** messaging protocol where message loss can occur. The solution is to use a reliable **TCP** messaging protocol. Still, they contribute to a delay in communication, as shown in [Kim et al. \(2005\)](#), thus it should only be used for important messages, that everyone must receive and execute, like a message to spawn a player in their local simulation of the game world. **UDP** messages should only be used when it is not important if some packets are lost, such as movement.

Several consistency techniques are introduced to the game to hide inconsistencies and staleness. These typically fall under two categories: **Predictive Contract Mechanism (PCM)** and multi-resolution simulation ([Yahyavi and Kemme \(2013\)](#)). Dead reckoning is the most commonly used **PCM**, which will be explained further over. Moreover, these techniques can also hide the packet loss, for example, if dead-reckoning is introduced correctly into the game implementation, this loss of information may become invisible to the player ([Mauve et al. \(2004\)](#); [Yahyavi and Kemme \(2013\)](#); [Smed et al. \(2002b\)](#)).

2.3.6 Dead Reckoning

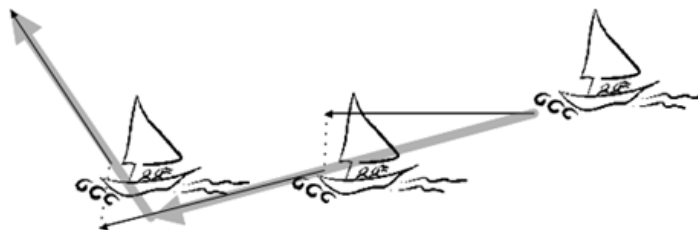


Figure 8: Illustration of Dead reckoning. The real movement is represented by the gray arrows, the predicted movement is represented by the black arrows, and the dotted lines represent a shift of position (or "warp") caused by an update message. source: [Smed et al. \(2002a\)](#)

As stated before, reducing the packet update rate when possible, can lower the bandwidth requirements and improve the scalability of the platform. However, in doing so the player's game state is left susceptible to differ from the correct state. To keep consistency or an approximation of consistency, the game somehow must counteract the absence of data between packet updates. Dead reckoning ([Smed et al. \(2002b\)](#)) comes to play when the deviation threshold is exceeded. Its methods utilize the object's parameters such as position and velocity to predict one's events based on some rules. The same concept can be used to predict the data coming from other clients, allowing to extend the period between transmission of messages and at the same

time, eliminate the network latency at the cost of data consistency. A use case of this is a character moving at a fixed velocity, until its velocity vector changes it will not send further updates.

Dead reckoning involves two segments: the prediction technique and the convergence technique ((Smed et al. (2002b)). To calculate the predicted state, dead reckoning uses the prediction technique which uses the object's last position, until new data is received. Depending on it, the prediction may be far off from what it should be, affecting all the next predictions. The most used technique is derivative polynomials, which vary in order. A low order derivative polynomial, such as zero-order, only transmits the position, which is not helpful for objects with velocity. On the other hand, a high-order derivative polynomial, such as second-order, considers the position, velocity, and acceleration of the object. This is the most commonly used technique. However, higher-order derivatives trade low computational burden, due to additional terms, for higher accuracy which ends up consuming limited bandwidth resources. In order to balance both, hybrid systems can dynamically change from second-order to first-order when it sees fit. An example would be, the object's acceleration is constantly changing, making it very improbable to apply the correct value to the prediction technique. Furthermore, the source node can transmit only absolute positions instead of transmitting higher polynomial terms.

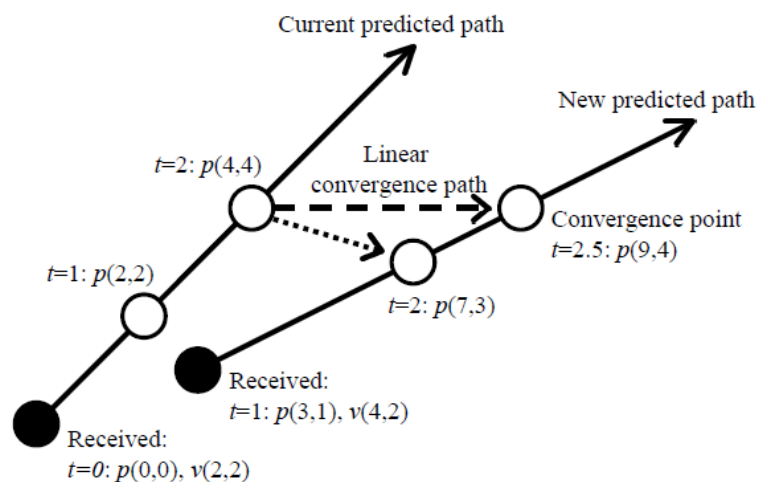


Figure 9: Illustration an example of dead reckoning. The black circles correspond to the data received regarding the object position p and velocity v at a certain time t , while the white circles represent the corresponding position of the object at a certain time. At the instant $t = 2$ when the direction of the object is expected to be $(4, 4)$, new data is obtained that the real position of the object was $(3, 1)$ and the velocity was $(4, 2)$ at time $t = 1$ due to latency. Rather than to automatically "warp" the entity to its newly predicted position $(7, 3)$, a convergence point at 0.5 seconds later is determined from the new predicted path. In doing so, the entity is moved smoothly, during the convergence period, along the linear converge path to the newly predicted path, continuing it later on and consequently avoiding intensive jitter. source: Smed et al. (2002b)

The convergence technique is used when new information about the object's position is received and must be applied, so in order to help smooth the transition and avoid jitter (when moving), this algorithm is applied. The predicted position of the object may differ from the based position sent by the source code whenever a node receives an update message. Depending on the distance between the current prediction position and the

position received, the convergence algorithm can opt for interpolation or direct change to the new position. The easiest approach for convergence is zero-order, in which the object is simply moved to the new predicted position, however, this may be visible to the player being seen as jitter. Linear convergence would be a better method (Hsu and Kuo (2003)). This method determines a future convergence point through a path from where the object is and where it must be. The object is then moved along the direct path. Although, being a more transparent method, it still creates some problems, such as unnatural curves. Curve-fitting techniques can be applied to resolve them, smoothing out these movements. However, the higher-order it is the more computational power it will require from the application.

2.3.7 Lag Compensation

Contrary to a singleplayer game, multiplayer games have to constantly deal with several problems caused by network-based communications. For instance, packets sent over the network take a certain amount of time to travel between the server and the client. Therefore, every client will ultimately have a different network latency on the server, which may vary due to other traffic that the client machine may sustain over time. This delay can cause logical problems in the game, worsening as the latency increases. For example, Player A sees Player B and shoots them at a given time, sending the shot information to the server. However, while the packet is on its way, the server continues to simulate the world, allowing Player B to move to a new location. When the shot message is received on the server, a few milliseconds later, the server uses the information to verify if Player A has indeed hit Player B. However, since the server simulation has moved on from when Player A shot Player B, the server won't detect the hit, even though Player A aimed precisely at Player B. Consequently, this results in a massive problem affecting the game's fairness and player experience. Due to this, having low latency is a huge advantage, especially in fast-paced games where several messages are being created and sent. Yet, achieving it at all times may be difficult for clients with limited resources, making the game's actions unfair.

To cope with these problems, architectures that contain a central arbiter can use techniques such as Lag Compensation, which are invisible to the client, and provide fairness for players with slower connections. Lag compensation works by storing all recent player positions for a certain amount of time and creating a history of what occurred until now. If a user command, sensitive to the time in which it occurred, is executed, then Lag compensation will estimate at what given time the action happened through the following equation:

$$\text{CommandExecutionTime} = \text{CurrentServerTime} - \text{PacketLatency} - \text{ClientViewInterpolation}$$

This equation accounts for the interpolation delay applied to movement messages on the client's application and the packet travel time, removing them from the server's current time to obtain the time that Player A shot Player B on the server. Concluding the command execution time, the central arbiter moves all other players back to where they were when the command was executed, running the command and correctly detecting if Player A hit Player B. This way, no matter the time the command arrives at the central arbiter, the hit is always accurately

calculated. After the command has been processed, the players are reverted to their original positions, and the game can continue normally.

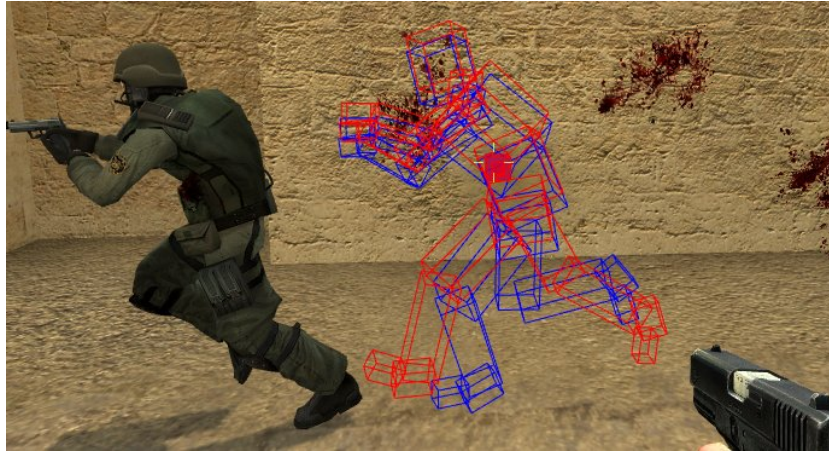


Figure 10: Illustration and example of lag compensation of a user with 200 ms delay when they hit a player. The red wireframe shows the remote player position when the user shot them. The player on the left is the player object that continued to move while the shot command traveled to the server, which is why it is ahead. The blue wireframe represents the player on the server when the user has executed the command. source: Valve® (2021)

Ultimately, Lag Compensation makes the game fairer for Player A but can also cause other problems. For example, after Player B has moved behind a wall, they could still get shot a fraction of a second later by Player A, even though they thought they were safe. This is the resultant trade-off of lag compensation, which may seem partially unfair to Player B. However, it would be considerably worse if Player A were to miss a shot deemed unmissable.

2.3.8 Synchronization

Synchronization is the foundation of a multiplayer online game (Hsu and Kuo (2003)). It is responsible to maintain a consistent game base, allowing the players to play the game without occurring any strange behavior. Moreover, the implement synchronization algorithm is an important factor on other key issues such as scalability, cheat-proofing, etc.

Conservative Algorithms

Conservative algorithms solve the synchronization issue through directly preventing out-of-order events (Cronin et al. (2003b)). A common conservative algorithm used is lockstep synchronization (Steinman (1995)). This is the simplest technique available, used in military simulations (Mauve et al. (2004)). Lockstep synchronization consists of no member (or in the case of an authoritarian server, server/host) will advance its simulation until all players' commands are received (Cronin et al. (2003a)), shown in Figure 11(a). This prevents out-of-order

events from even being generated, making it impossible for inconsistencies to ever occur since no member of the network has performed calculations to advance its simulation.

Another conservative algorithm is bucket synchronization (Steinman (1995)). The principle of bucket synchronization, also designated as local-lag in (Diot and Gautier (1999)) which consists of artificially delaying the updating of the game world until a short fixed-length of time has passed, shown in Figure 11(b). The value of this latency must be chosen to allow enough time for updates to be received in the different recipients of the network.

Still today many FPS games use this algorithm. A use case of this algorithm is in Cronin et al. (2003b), where the authors apply this principle to a fully replicated game. In the described game, it is given enough time for the state update to reach remote peers through delayed the update of the state by a fixed period of 100 milliseconds. While these algorithms perform poorly in fast-paced games such as FPSs, these may still be adequate to more slow-paced games such as RTSs, due to being design as a turn-based game. This makes it so that each player has to wait for every other player to complete their turn, allowing them then to advance in the simulation.

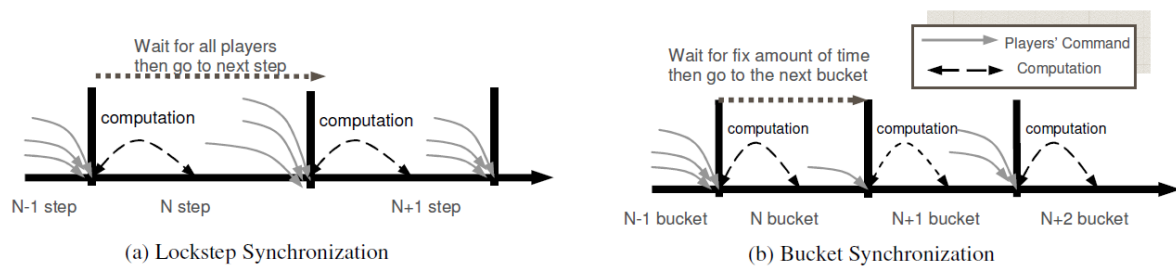


Figure 11: Conservative Algorithms. source: Hsu and Kuo (2003)

While these algorithms allow fairness despite latency variations and give more control over update dissemination costs, it comes at an unacceptable trade-off in multiplayer games because it is impossible to maintain a correlation between the wall-clock time and the simulation time (also referred to as Global Virtual Time (GTV) or game time (Hsu and Kuo (2003))).

Optimistic Algorithms

Optimistic algorithms resolve the synchronization issue by using a mechanism that detects and corrects inconsistencies in the state, and keeps a consistent rate of simulation. These algorithms execute events optimistically, updating the game simulation, before knowing if all earlier events have arrived, and then correct inconsistencies when their state is wrong. These algorithms are far better suited for more complicated interactive situations that multiplayer games can introduce. The strategy is to use dead reckoning to correct it in case of a wrong prediction, rolling back the wrong predictions and updating the simulation with the correct states.

To evaluate optimistic algorithms, there are three factors to take into account (Cronin et al. (2003b)): (1) Overhead: penalty paid for rollback. (2) Memory Usage: memory required to store the backup game states. (3) Complexity: complexity to fix inconsistencies detected on the game state.

TimeWarp synchronization (TWS) (Steinman et al. (1998)) consists of doing a snapshot of the game state at each execution and if an event prior to the last executed event is received then it is issued a rollback to a previous

game state, as shown in Figure 12(a). When in rollback, first the game state is restored to the previous snapshot, then all events that occurred between the current execution time and the snapshot time are re-executed, including the last command. This rollback also dispatches anti-messages to cancel previously produced events that now have become invalid. When a client receives them, it triggers their own rollback, which in turn triggers even more anti-messages, and so on. Clearly, this considerable size of messages may congest the network creating problems, as stated before. Servers may tie up processing these anti-messages instead of processing the execution of the game state. However, there are breathing algorithms (Steinman (1993)) developed to attempt to solve this problem alongside the excessive rollbacks by restricting the number of commands that can be executed optimistically. Instead of a fully optimistic execution, breathing algorithms limit their optimism to events only within the event horizon.

Trailing state synchronization (TSS) (Cronin et al. (2001)), shown in Figure 12(b), developed for Quake and introduced in (Smed et al. (2002a)) implements rollback intelligently to avoid high memory and processor overheads visible on TWS. Instead of keeping snapshots at every command, TSS keeps a fixed number of copies of the game world. When rollbacks are required, instead of copying the game state from a snapshot taken just prior to the conflicting command, this algorithm copies the game state from a secondary copy (named Trailing). The reason for this is because the secondary copy is trailing the primary copy (named Leading), meaning it is behind. This makes it so the secondary copy has more time to reorder commands and that it does not have the inconsistency that must be repaired. In other words, the leading state, which has the shortest synchronization delay, is used to render the current events to the clients, while the trailing states are used to detect and correct inconsistencies in the game state, copying the states from the trailing to the leading state and then performing all commands between the inconsistency point and the present point.

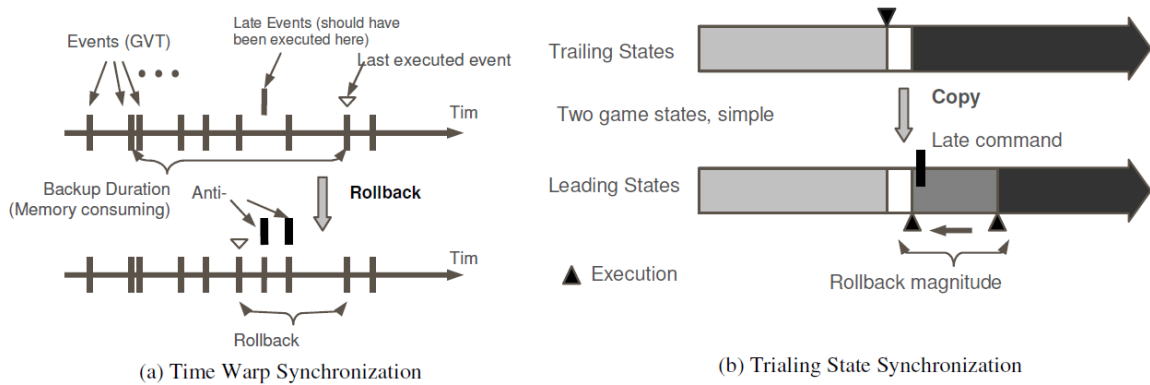


Figure 12: Optimistic Algorithms. source: Hsu and Kuo (2003)

This actually means that TSS does not resolve the problem of rollback originated from TWS. This means that it will only have better performance than TWS when the following two situations are evident: (1) The game state is very expensive to hold the snapshot. (2) The gap between the delay of the state is small.

2.3.9 Security and Cheating

Online security is not only something most of us use on a daily basis but also a major topic nowadays. This can also be said about the gaming industry. Game websites often report cyber attacks and warn about misbehavior and cheating in online gaming. Kirmse and Kirmse in [Kirmse and Kirmse \(1997\)](#) identify two security goals for online games: (1) Protect sensitive information, such as credit card numbers. (2) Provide a fair playing field, through anti-cheating measures. Safety and security also are important issues inside the game world. However, anti-cheating measures do not fall under the scope of this document, for that see [Sanderson \(1999\)](#).

Online cheaters are usually motivated by vandalism or dominance ([Moraal \(2007\)](#)). Gauthier in [GauthierDickey et al. \(2004\)](#), distinguishes cheats into three different categories: protocol-level cheats, game-level cheats, and application-level cheats, affecting confidentiality, integrity, and availability. Pritchard ([Pritchard \(2000\)](#)) and Kirmse ([Kirmse \(2000\)](#)), also go over some of the common methods used in online cheating. Protocol-level cheats occur by modifying the protocol. This can be achieved by the cheater through focusing on the network and eavesdrop, injecting, delaying, or even dropping messages. They can for example, “listen” passively to the private communications that they are only supposed to forward. There are five common protocol-level cheats ([Neumann et al. \(2007\)](#)): (1) Fixed-Delay cheat adds a fixed-time delay to all outgoing packets. This makes it so they can react faster to actions, in the game world, than every other player. A protocol that eliminates the possibility for this type of cheats can be seen in ([GauthierDickey et al. \(2004\)](#)). However, these can generate delays in bandwidth use creating another set of problems to be dealt with. (2) Timestamp cheat sends wrong timestamps on actions so those same actions can favor them in relation to other players. (3) Suppressed Update cheat, suppresses updates of any intended actions, while still receiving updates from others. (4) Inconsistency cheat sends different updates of their actions to different players. This makes the rest of the players have a game world out of sync, with possibly strange behaviors. (5) Collision cheat retrieves unauthorized information from shared updates sent by other players. On the other hand, game-level cheats occur by violating the rules of the game. For example, moving long distances in an instant, that normally would take some time to accomplish. This type of cheats affects mostly P2P solutions, while in client-server solutions players can benefit from a trusted server to manage the game states making it very difficult for cheaters to alter them. Furthermore, application-level cheats materialize by modifying the code of the game. The cheater can also reverse-engineer the game application to obtain more information on how it works. A common use of this cheat is modifying the graphics engine so that walls become invisible, or even highlight enemy players through walls and other visual barriers, making it easy for cheaters to know where players are. For this type of cheats, both client-server and peer-to-peer have the same level of vulnerable status.

To prevent cheating, there are two approaches to be considered ([Yahyavi and Kemme \(2013\)](#)): (1) Cheating-resistant systems. (2) Cheating-evident systems. Cheating-resistant systems basis on preventing cheating from happening, through various means and services. Some of these services are player authentication, accountability of their actions, information and communication confidentiality, application integrity, and tamper-proof devices. Cheating-evident systems basis on detecting cheaters and punish them accordingly.

Furthermore, software and network traffic are the only vulnerable places in a multiplayer online game. Design defects can also create loopholes that cheaters can use to exploit the game. For example, in a system where each client trusts every other client on the network, receiving updates and implementing them blindly. The game is clearly unshielded and susceptible to client authority abuse. In this system, a compromised client can, for example, alter the game state as they please and the rest of the clients will accept this information as correct. In addition, heterogeneity of network environments in distribution can be the source of unexpected behavior, such as “features” that only become visible when the latency is extremely high or when the server is under a [denial-of-service \(DNS\) attack](#).

2.4 ARCHITECTURES

This section gives an introduction of architectures in [MMO](#) games. The three most common architectures today are [Client-Server \(CS\)](#), [Multi-Server \(MS\)](#), and [Peer-To-Peer architectures \(P2P\)](#), shown in [Figure 13](#). All these different architectures try to attain scalability by different means ([Pellegrino and Dovrolis \(2003\)](#)). The server-based architectures achieve it by increasing the number of resources through clustering servers to distribute the load. In [P2P](#) architectures each peer that joins the network helps to distribute the load.

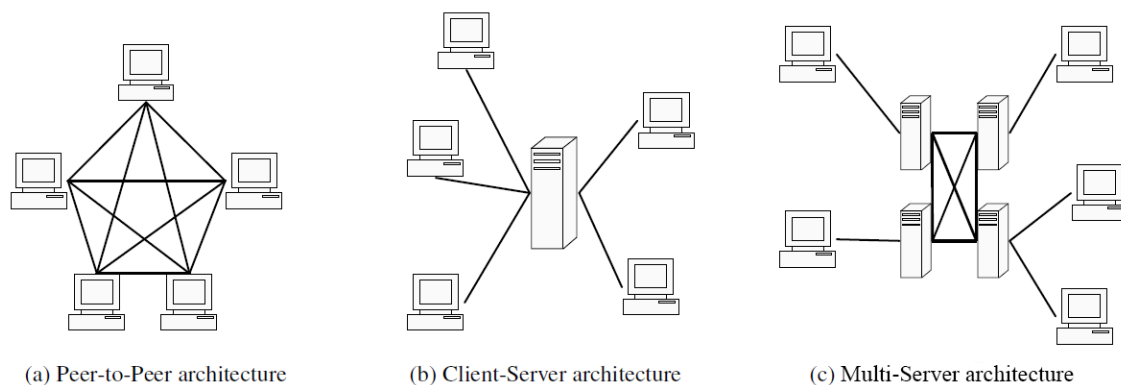


Figure 13: Topology for Multiplayer Online Game systems. source: [Hsu and Kuo \(2003\)](#)

In a more game-related overview, as shown in the examples below in the [Figure 14](#), these different gaming architectures maintain the game world in different ways. In a [CS](#) architecture, the server is responsible for maintaining the entire game world and its correct execution. In an [MS](#) architecture, the game world is maintained as separate instances of the game world and/or partitioned into multiple different, or parallel, worlds spreading the users over them. In a [P2P](#) architecture, each peer holds a part of the world acting as region servers for other players ([Hsu and Kuo \(2003\)](#)).

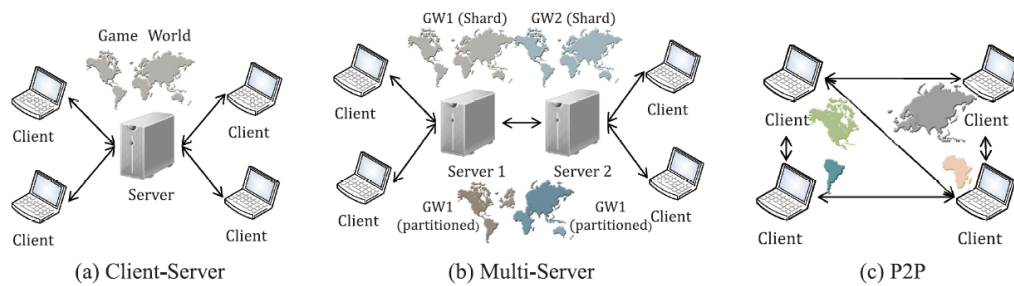


Figure 14: Different maintained game worlds. source: [Yahyavi and Kemme \(2013\)](#)

2.4.1 Client-Server Architecture

Client-Server architecture is the most common architecture for networked multiplayer games ([Hsu and Kuo \(2003\)](#)), being the most popular architecture for MMOGs since the first developed. In this architecture, the server is designated as the holder of the master copies of the game world, which other players connect to, receiving the necessary information about the game world to replicate the simulation locally. In some systems, the player is also the host providing the same central server authority upon other recipients in the network ([Moraal \(2007\)](#)).

The basis of this architecture is the following, each player sends updates (actions) to the server, which are used to update the simulation. This makes conflicts easy to resolve due to being a single authoritarian execution. After the simulation's update, the server sends back the updated game state to the players who then apply it to their local game world.

A CS architecture, having a centralized authoritarian server provides a high level of control over the game world ([Yahyavi and Kemme \(2013\)](#); [Hsu and Kuo \(2003\)](#)) because it completely controls the game state. This architecture has the following key characteristics:

Single game state on the server. Since all participants receive information directly from the server, global consistency is guaranteed since it is based on a single authoritarian server. Moreover, the server introduces a natural synchronization among all recipients, displaying the same game state at the same time. Participants can go temporarily out of sync, which affects the consistency of the local game state, but will never affect the rest of the network since after reconnecting the local game state is corrected by incoming updates from the server. This architecture is usually simple to implement, but is also less efficient and not scalable, as even well-equipped servers can only support a limited number of users until their limited bandwidth runs out ([Hsu and Kuo \(2003\)](#)). The capacity that a server has is defined by a fixed number of clients that a single machine can handle, related to the game in hand. To increase this capacity, more servers need to be added to the network so the workload can be balanced between them. As this enters on multi-servers, this topic will be continued in [Section 2.4.2](#), which discusses the multi-servers' characteristics.

Easy to update. Used mostly in games with a static or persistent game world, this architecture allows to easily change, update, and have control over any certain update because the game rules and logic are handled solely by the server. That is, when a new update is to be implemented, only the server has to be updated. However, the

client software in MMOGs is not often as naive as in more traditional client-server applications. These may have some data on the player-side to help lower requests and interactions from the client with the server, creating the need for this client-side data to also be updated when game rules are modified. Still, some small updates may be only needed on the server and not on the client. For bigger sized changes, both server and client software have to be updated.

One governing authority. As every command must go through the server and then be sent back to everyone in the network, this makes it a natural governing authority to prevent cheating, especially because a reasonable part of the game is controlled by the server and thus trustworthy. This also allows the server to authenticate the requests made by clients, certifying that all requests executed by the server are legal and allowed in relation to who requested them.

Limited scalability and load flexibility. However, since the server must collect all the messages sent from the clients, execute all the updates and resolve all the conflicts for the game to be updated, this causes the load to be solo handled by the server. Clearly, this forms a natural bottleneck. The overall amount of data expected to be transmitted and evaluated takes additional time as the number of participants increases. Once the servers reach their maximum capacity on the CPU, storage, and bandwidth, the game state computations and/or its distribution will be affected, causing it to possibly slow down or to even completely stop. Consequently, this reduces playability for players. As solution clients can implement prediction techniques to mask this inconsistency of constant updates from the server.

Single point of failure. Due to being a centralized system, based on a single authoritarian server, in case of failure all participants, being dependent on it, will be disconnected from the collapsing network, interrupting the game and losing non-persistent game states (Moraal (2007)). Consequently, a CS architecture is less robust than a distributed system because of its single-point-failure (failure of the server). While this issue can be addressed by adding backup servers, for the eventuality of the main server going offline, this can however lead to more complexities and costs, further reducing its scalability and viability. This can make the network very fragile, even more in MMOGs that require high availability. Frequent complaints from MMOGs' players is the unavailability, often because servers are unstable or even down (Yahyavi and Kemme (2013)).

2.4.2 Multi-Server Architecture

A Multi-Server architecture or server-cluster (Chen et al. (2005)) consists of a group of connected servers. Usually, large game companies maintain such server farms to provide service for many clients (Knutsson et al. (2004)). Depending on the Multi-Server architecture the application may work in different ways, these variations can be categorized into two categories. The first category consists of a basic Client-Server architecture but with an increased number of resources, meaning servers. Here the entire game world is mirrored across several servers. Each server follows the traditional Client-Server architecture and is responsible for its own complete copy of the game world and a different set of clients, usually defined by where they are located in the real world and/or where they have a better connection. In most games there is usually no need for communication between

these servers, making it also impossible for players to communicate or even interact with players from other servers.

In the second category, the game world is partitioned into several regions and given to servers to maintain. However, in contrast with the first, this category servers may communicate between themselves. As referred before, regions are kept separated in different servers, but all players continue to be in the same game world and may interact with each other if they are in the same region. Most architectures only give one region to a server to handle, however, some also allow one server to handle more if its current region load is low. This creates a problem if one of the regions suddenly becomes overloaded. For this, schemes to dynamically load-balance regions have been presented (Pellegrino and Dovrolis (2003)), allowing to allocate overloaded regions to new servers.

To allow this free movement between regions (servers), the system requires the support of a hand-off mechanism that allows this movement (Yahyavi and Kemme (2013)). This mechanism can be transparent through the use of elements of the game to mask the transition. It is used as a player approaches the edge of the region and tries to cross it. In doing so, the mechanism will send the necessary information to the correspondent server and when the transition is complete the player may walk freely and interact with the players in the newly entered region. Although it allows free movement this system also brings new challenges to be tackled, such as the flocking of players in a single region.

A Multi-Server architecture inherits the benefits of the CS architecture while improving on its problems (Hsu and Kuo (2003)). This architecture has the following key characteristics:

Scalable cluster of servers. Contrary to the CS architecture, in this architecture, there is a cluster of servers that is scalable. The bottleneck from the large computational power required, used for an increase of users and objects in the game world referred to previously in CS, is eliminated by distributing the computational load over the multiple servers in the cluster. To increase the capacity of the system machines can be added to a cluster of servers. Companies try to predict accurately the capacity that will be needed from the cluster because adding new machines will require some time to reconfigure the server cluster, which makes it not very flexible. If they predict too much load, they will end up with more capacity than needed resulting in wasted money. If they predict too little, their servers will be unable to handle the load, making the game unplayable, and likely fail on the market.

Efficient synchronization. This architecture allows the game world to be divided into different regions or zones, and each instance is maintained by a different server, consequently supporting a higher number of players simultaneously. However, this brings new challenges to the architecture. The main drawback is that in most games, people from different servers cannot play together, being isolated from each other. This is because, by separating in instances, it forces players to only interact with others that share the instance of the game world. Allowing players to migrate to other zones can be a complex system to implement and requires attention to maintain game state consistency. In the case of the players flocking in a region, the corresponding server will only be able to handle a limited number, creating a problem for the remaining players. A solution to this would be to have more servers handling the same region.

Large costs. A major drawback of this architecture is setting up and maintaining server farms, making it only a viable option for companies with funds, preventing start-up or small game companies from entering the MMO game market (Hsu and Kuo (2003)).

2.4.3 Peer-To-Peer Architecture

In a Peer-To-Peer gaming architecture, each node acts both as a server and a client, meaning each peer is responsible for executing its own game simulation, updating, and disseminating game updates to other peers (Neumann et al. (2007)). Due to the lack of central authority upon the clients, these networks are not very popular. Furthermore, it is hard to maintain state consistency among players, prevent cheating and regulate access.

In a P2P architecture, due to the elimination of the central authoritative server and the distribution of the load over all users in the network, new characteristics appear, being the following:

Scalability and flexibility. As there is no need to send updates to a central server and from the server to the peers, updates are directly sent between peers, achieving low latency. Moreover, each peer in the network helps to distribute the computational power by the players, through contributing in CPU, storage, and bandwidth resources, making game updates between themselves. The more players in the game who are involved, the greater the network load will be. This removes the need for large server-side bandwidth and reduces the need for powerful central server clusters, thus lowering the cost to maintain them. However, it may lead peers to surpass their bandwidth capacities in cases where they need to send updates to far too many peers. For example, if a game world is too overcrowded.

Low cost for running the game. As mentioned before, P2P architectures use the participants' resources that are active on the network. In doing so, the need for major infrastructure investments to keep servers running is not required. In the case of MMOGs, it is challenging to make them solely from a P2P system. There would have to be some sort of central server in effect to maintain order and consistency, but because of the low complexity and low workload, it will most certainly not be a big expense.

Robustness. Due to all participants being equivalent, independent and having the necessary information to compute the state of game at any time, makes the P2P architecture very robust (Neumann et al. (2007)). If a peer node fails, the rest of the network will continue to operate normally. However, this is not the case in all P2P architectures, nevertheless it is a common advantage in a P2P approach. In MMOGs this will also mean that the game will always be available.

No easy identifiable global game state. As a result of a distributed game state, in P2P systems it is very hard to ensure a consistent game state between peers, as there are multiple peers handling the same objects simultaneously and there is no particular entity that holds the entire game state. For these systems, it is required some form of synchronization between clients to ensure that each copy of the state is the same. Without synchronization clients' game, states would diverge over time due to network delays and their characteristics. Some studies

achieve global consistency (Hampel et al. (2006)), however, it only works properly in systems where games are at a low scale.

No governing authority to counter cheating. Cheating in P2P architectures is a sophisticated problem (Moraal (2007)) because part of the virtual world and game logic is run on the player's peer node. This allows players to make their own decisions without there being some sort of authority to check its legally, making it easy to cheat. This means that peers cannot be regarded as trustworthy and that special mechanisms must be developed to ensure that compromised peers cannot endanger the consistency or stability of objects and network traffic. In P2P other mechanisms have to be implemented in order to defend against cheating, such as those mentioned in Section 2.3.9. However, it is unclear, whether cheating without any kind of a global monitor can be completely removed (Neumann et al. (2007)).

2.4.4 Comparison

Here it is discussed the different characteristics of each of the previously described architectures. A summary of comparison is provided in Table 2.

	Peer-to-Peer	Multi-Server	Client-Server
Robustness	Good	Medium	Poor
Scalability	Good	Medium	Poor
Delay	Good	Medium	Poor
Consistency	Poor	Medium	Good
Cheat-proof	Poor	Medium	Good
Commerciality	Poor	Medium	Good

Table 2: Characteristics of the three architectures' systems.

Robustness. In a CS architecture, due to being a centralized system, all participants are dependent on it. Consequently, it is the less robust of all of the three architectures. Contrary to CS, in a P2P architecture, all participants are independent, equivalent, and have the necessary information to compute the state of the game at any time, resulting in a very robust system.

Scalability. CS has poor scalability due to just being able to hold a fixed number of players. A solution for this is to add new machines to the network of servers to increase the number, forming a Multi-Server. However, maintaining these farms of servers is costly. P2P architecture has the highest potential for scalability as each peer connected to the network contributes with CPU, storage, and bandwidth resources, distributing the computational power by the clients.

Delay. In a CS architecture, the server must collect all the messages sent from the clients and execute all the updates, resolving conflicts and updating the simulation, only then sends updates to the clients. As the number of players goes up there is more information to be received, handled, and sent to the clients. This can cause delays in the response time since the bandwidth's limit is reached. In a Multi-Server architecture, the load can be balanced between all servers. In P2P architectures, the delay of messages is very small as peers connect

directly to each other. This means that in contrast to CS, the messages do not have to be relayed from a central server.

Consistency. A CS architecture, having a centralized authoritarian server provides the highest level of control over the game world and naturally guarantees global consistency. This architecture is usually simple to implement in comparison with P2P architectures, but it is also less efficient in the resources used. In a P2P architecture is very complicated to maintain consistency as everyone can change the primary copy, leading to inconsistencies.

Cheat-proof. In a CS architecture, a server manages all updates, thus verifying if a player is following the game rules and correct them if not. As the server is owned or controlled by the developers, players cannot tamper with it, being regarded as trustworthy. In contrast, P2P systems are very hard to prevent from cheating because all peers have the game logic locally, being able to change it as they please and propagate it through the network. Due to this, P2P networks are more susceptible to protocol-level cheats, since peers are involved in message routing, which the same cannot be said about server-based architectures where each player communicates only with the server. The same can be said about game-level cheats because there is no governing authority that handles all actions and can reject them if they should not be possible. In P2P solutions, game-level cheats can be partly avoided by randomly assign items to peers, as done in Moraal (2007) where peers have no say about what aspects of the game state they are responsible for. Some other approach is to replicate over multiple peers the state of all objects and make them check every update its legality.

Commerciality. Server-based architectures (Client-Server and Multi-Server) are very easy to commercialize and exploit (Yahyavi and Kemme (2013)). A lot of MMOGs nowadays present a monthly subscription fee in order to have access to the game. Due to being centralized, it is very easy to implement this business model, as the game company owns or controls the servers. In order to play, users will have to request the servers to join, easily controlling who has access to them. In P2P architectures this is not as feasible due to being very hard to enforce. Users can simply form their own network. However, there are solutions (Moraal (2007)), such as having a central server that manages users' accounts and guarantees that the network is accessed only by users who have paid. Still, since the client software contains all of the game logic and the server will be technically simple, the system would be extremely susceptible to piracy.

2.4.5 Hybrid Architecture

When examining the upsides and downsides of both CS and P2P architectures, one can doubt whether both architectures can be merged in a way that provides all the advantages that both offer with none of the downsides. Studies have been made, proposing various combinations of these two architectures, as shown in Figure 15.

The approaches can be divided into several categories according to what is being handled by the P2P system. Yahyavi in Yahyavi and Kemme (2013) classifies them as: (1) Cooperative message dissemination: the game state is maintained by one or multiple servers, however, the update dissemination uses a P2P approach. In this system, the players normally send their actions' requests directly to the server, which executes them. Then, the peer uses a P2P multicast mechanism to update every peer on the network. (2) State distribution: the game state is distributed among the peers. Each player can hold primary copies of the game objects, making them also

Architecture	Type	Network Overlay	Interest Management	Replication & Consistency Control
Dist. Avatar Mgmt. [Varvello et al. 2009b]	Hybrid: Server + Unstructured	Direct(Neighborhood)	Dynamic: Region + Delaunay Triangulation	N/A
Hydra [Chan et al. 2007]	Hybrid: Server + Unstructured	Region Server + Proxies	Static Region	Primary Copy: Region(slice) Controller
Distributed Event Delivery System [Yamamoto et al. 2005]	Hybrid: Server + Structured	Server + Load balancing tree	Static Regions: Rectangular Cells	Primary Copy: Region Controller
IRS [Goodman and Verbrugge 2008]	Hybrid: Server + Structured	Server Message Relaying + Proxy Execution	Server	Primary Copy: Proxy peer
MM-VISA [Ahmed et al. 2009]	Hybrid: Server + Structured	AL Multicast	Hexagonal Regions + Player Clusters	Primary Copy: Region Controller
MOPAR [Yu and Vuong 2005]	Hybrid: Structured + Unstructured	DHT: Pastry + AL Multicast: Scribe	Hierarchical AOI + Static Regions: Hexagonal cells	Primary Copy: Region Controller
VoroGame [Buyukkaya et al. 2009]	Hybrid: Unstructured + Structured	Direct(Neighborhood) + DHT	Dynamic: Voronoi (Convex Polygon)	Primary Copy: Each Peer (Random DHT)
Mammoth [Kienzle et al. 2009]	Hybrid: Server + Unstructured	Region Controller(All-to-All)	Dynamic Regions: Rectangular tiles	Primary Copy: Region Controller
Zoned Federation [Iimura et al. 2004]	Hybrid: Server + Structured	DHT	Static Regions: Rectangular Cells	Primary Copy: Region Controller

Figure 15: Some examples of hybrid approaches. source: [Yahyavi and Kemme \(2013\)](#)

responsible for executing the player's actions. However, part or all of the communication between peers can be managed by the servers. Furthermore, the servers are also responsible for authentication and admission control. Clearly, distributing the cost of state execution among clients, makes the approach highly scalable. (3) Basic server control: both message dissemination and state distribution are done only through the P2P overlay. In this approach, the servers' main role is to keep highly sensitive data, for example, the players' state and progress, user logins, and payment information. Still, in some games, servers may also have additional roles to fulfill, such as coordinate interactions between peers and performing admission control to enter or leave the game session.

However, creating an ideal combination with all the benefits and no downsides is difficult because these disadvantages are almost mutually exclusive. In a CS architecture, the main disadvantages come from being based around a central server. The cost of maintaining, the limited scalability and flexibility of the network, and a single point of failure are truly the most problematic factors on this architecture. While in P2P architectures much of its downsides occur by not having a central server. With this comes no referee to counter cheating and it becomes very difficult to maintain a consistent game state throughout the entire network.

This does not necessarily imply that merging P2P aspects with client-server's is never advantageous, much of the disadvantages from p2p architectures do not always apply to all games. Some games may offer some simplicity or specific elements that make them not susceptible to these downsides. For example, games that do not need to pass over the game state from one instance to another or games with a small number of players. It might not be viable to completely remove the central server in MMOGs, and it is likely better to keep the complete game state in the server (Moraal (2007)). What can be used in a P2P manner depends on the game's specifics. In some games' systems, P2P can be even used in other ways, for example, in WoW, P2P systems are used to distribute software updates to the clients.

2.4.6 Decentralized Virtual Environments

Decentralized virtual environments have been explored in the past (Barrus et al. (1996); Frécon and Stenius (1998); Hu et al. (2006)), to provide scalability and bandwidth savings for online games. MiMaze (Gautier and Diot (1998)) was one of the first attempts to design a fully distributed (i.e., serverless) online game. It uses an unreliable communication system that is based on Real-time Transport Protocol (RTP) (Casner et al. (2003)) over UDP/IP multicast (SE (1998)). Each player has an entire replication of the virtual world on their machine (as opposed to having just a partial view), and multicasts their actions to all other players. This technique can be considered as P2P since each player communicates their actions directly to the other players. Based on how nodes connect to each other, fully-distributed peer-to-peer gaming solutions overlays are generally divided into two types: structured overlays or unstructured overlays. Some of these types of architectures are shown in Figure 15.

Structured P2P Game Architecture

Structured P2P systems use a deterministic protocol to form a specific graph structure, which consists of the determination of the connections between peers. The core of several of these architectures uses a Distributed Hash Table (DHT) as the underlying mechanism for game state distribution and update dissemination. Players subscribe to receive updates from different game regions, which are mapped locations in the DHT, and the protocol guarantees that a node is able to route messages to every other node and/or find an object in the network overlay. This is done by exchanging $O(\log(N))$ messages, where N is the number of nodes. To help the lookup, key-based routing mechanisms are provided, these services are comparable to a hash table, however differing since the (key, value) pairs are scattered around the nodes in the network. These protocols can also, with low overhead, introduce or exclude nodes from the overlay. Some known examples of these P2P substrates are, Pastry (Rowstron (2001)) and Chord (Stoica et al. (2002)). DHT uses these substrates for the fundamental mechanism. Knutsson, in Knutsson et al. (2004) presents a peer-to-peer solution for MMOGs based on Pastry.

In some MMOGs, a common approach to creating these hierarchical overlays is to divide the game into regions and assign a super-peer that acts as a regional server. Players can then exchange updates with the node responsible for that region. However, structured P2P games can be problematic. One key issue is distributing the load when the number of players becomes high in a certain region. There are approaches that can be taken into

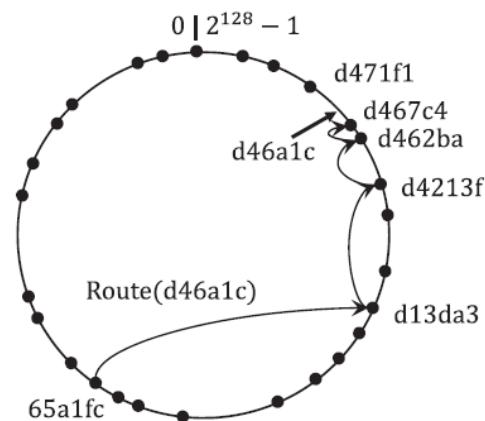


Figure 16: Illustration of Pastry message routing. source: (Yahyavi and Kemme (2013))

consideration to resolve this, such as load-balanced trees (Yamamoto et al. (2005)) and the sub-server approach (Lee and Sun (2006)). With load-balanced trees, the responsible node builds a delivery tree among the players in the region and uses the tree to deliver events for the region. With the sub-server approach, the responsible node subdivides its region into smaller areas and assigns an area server for each of these pieces, resulting in a three-level hierarchy.

An additional alternative to this somewhat top-down hierarchy is to instead create a bottom-up hierarchy based on event scoping (GauthierDickey et al. (2005)). The players join at the leaves of the tree, which is the smallest scope, and players within the small scope exchange game update directly. Furthermore, this approach uses a cryptographic protocol that orders local events and prevents cheating (GauthierDickey (2004)). When events occur at larger scopes, they are delivered to all affected nodes using the scoped tree.

Unstructured P2P Game Architecture

Unstructured P2P systems do not use deterministic algorithm used to arrange and optimize connections between peers in the network. In other words, it lacks a global mechanism, such as DHT, that would manage and maintain the overlay. The network connections are generally established by probabilistic mechanisms or at random, with the goal of achieving a suitable overlay. For example, a mechanism that has a greater probability of connecting peers semantically close. Usually, the lookup on this network is performed in a probabilistic way.

Also, redundancy measures have been introduced to help speed up the such as content replication or flooding, described in Leontiadis et al. (2006); Sozio et al. (2008); Costa et al. (2003). An example of this is, if a peer wants to find a piece of data in the network, a query has to be created and flooded through the network in order to reach as many peers as possible that share the data. However, in these networks, the queries may not always be resolved. Popular content is likely to be easily available to several peers, but if the content to be looked for is a rare or not popular data shared by no more than a few other peers, then it is very likely that the search will be unsuccessful. Flooding also causes a high amount of signaling traffic in the network, thus such networks usually

have very poor search efficiency. Some popular unstructured P2P networks are Napster (David (2016)), Gnutella (Taylor and Harrison (2009)) and KaZaA (Liang et al. (2004)).

ARCHITECTURE AND IMPLEMENTATION

This chapter describes the approach used to develop the distributed game and its implementation. It begins by describing the game developed and providing various insights into its requirements in order to work correctly in a distributed manner. Going further, this chapter introduces the selected network architectural approach and the capabilities it provides. Additionally, the system's implementation is explained, describing the functionalities, the communication workflow, the various structural components, and lastly, the system and game protocols.

3.1 GAME DEVELOPED

Since well-documented, open-source, and relatively simple MMO games are hard to stumble upon, it was chosen to develop a game from scratch using free game-making tools available. Constructing the game allowed to keep things simple and build it as seen most desirable in order to allow for an easy adaption and integration of network modules or for further expansion of the game's functionalities, to create new exciting problems to be explored. Moreover, it helped tremendously during the distribution process to already know how the game and its components worked due to building them.

The constructed game is called *DummyGuys* and was made in *Unity*¹, using mainly C# language. This game is heavily inspired on *Fall Guys*² due to its popularity at the time of researching a game to distribute. Its popularity also sparked others to take interest in developing games around the same concept and art style as *Fall Guys*, for example, *Dani* (2020) and *Sykoo* (2020).

DummyGuys is an MMO of the Battle Royale genre. This type of games traditionally involves dozens or hundreds of players, enclosed in a big map area, fighting for survival, alone or in teams, against other players, with the winner being the last team or player alive (*last-standing-man*³), with some famous examples being *Apex Legends*⁴, *PUBG*⁵, and *Fortnite*⁶. *DummyGuys* is a slight variation of this. It introduces up to 60 players to a small map with a series of obstacle courses that players must go through in order to reach the finish line and win. The players' actions are bound to running, jumping, and diving movements, whereas player-on-player

1 A cross-platform game engine. - <https://unity.com/>

2 <https://www.fallguys.com/en-US>

3 Multiplayer deathmatch gameplay mode is featured in some FPS games and is also the essence of battle royale games. games.

4 <https://www.ea.com/games/apex-legends>

5 <https://na.battlegrounds.pubg.com/>

6 <https://www.epicgames.com/fortnite/en-US/home>

interactions consists of pushing and grabbing each other, and, occasionally, bumping or blocking other players with their bodies. Collision with players or other objects can cause the player's character to be knocked out, transition into a ragdoll state, where they can't move until the character gets back up again.

The several obstacles in the game describe a stable and predictable movement, represented in the world as boulders, moving walls, blades, etc., each with a different movement type. When players collide with them, they may be pushed off map, which results in being respawned at the latest checkpoint they have passed. So, players must avoid these obstacles in order to win, as being struck by one them may considerably waste their time when competing to reach the end of the course as soon as possible.

Some of the game aspects to take into consideration when selecting an appropriate approach for its distribution, are the following:

- The game's synchronization consists of ensuring that the game state (race started, slots available, etc.), the players state (ragdolled, position, rotation, etc.), and the obstacles state (position and rotation) are the same on all simulations of the game;
- Systems like the finish line and checkpoint system are dependent on the player position. So, it is one of the most crucial attributes of the player's character;
- The character's state is vulnerable to cheating since it is controlled by a user;
- Events such as, players finishing the race or pushing and grabbing each other can happen simultaneously, requiring an agreement protocol to guarantee the same results on all instances of the game in order to achieve consistency;
- The game represents fast-paced and short-lived interactions, such as an FPS game, resulting in a low tolerance to the latency.

3.2 APPROACH

For the distribution of the game developed, this document proposes a hybrid architecture approach (based on the hybrid architectures mentioned in [Section 2.4.5](#)) with the introduction of some adaptations to fit the game's requirements already discussed.

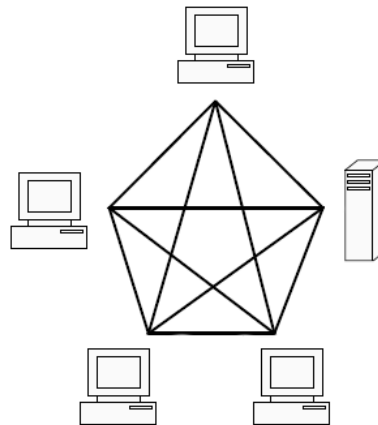


Figure 17: Architecture overview.

The hybrid architecture is comprised of two components: (1) **P2P** network paired with a (2) central server, represented in [Figure 17](#). Their roles in the system are designed as follows:

P2P Network. The **P2P** network represents the clients in the game session who communicate directly with each other, disseminating their player state to the rest of the peers in the network. Thus keeping it updated on all simulations as new actions are created and shared. The clients themselves are not responsible for the distribution of the game state, nor events. Their responsibility is only to themselves, meaning they can only control and be responsible for their player state, having no authority towards others. Since they do not participate in disseminating player states from other clients, nor game events, clients cannot harm others by sharing false information. However, adopting this approach causes the system to lose any cost savings from using the clients' bandwidth and computational resources to share or calculate these events.

Central Server. To ensure security in the network, it is employed a central server that also provides consistency and authority over all clients. The server has a great deal of power in the system because the ultimate goal of most games is to acquire or control the game state, whether that is a treasure, power, or the lives of other players. Anytime the players' actions interact with the game state or other players, the server acts as the arbiter, ordering state-changing events and issuing game state updates, assuring that every client's simulation follows the same outcome. In addition, player states are monitored by the server so it can keep up and, possibly, detect inconsistencies. Apart from this, the central server is also responsible for registering and managing clients, and when necessary, evicts them from the network if it detects they misbehaved.

In conclusion, the proposed architecture provides a distribution model that makes use of client resources to take the load off the central server while also providing inconsistency detection, network management, and

cheat-resistance, to which P2P is vulnerable. Lastly, revising the previously mentioned *DummyGuys* aspects, the current architecture presents the following capabilities:

- The architecture maintains control over important popular states, such as game state and obstacles state, and shares them with the clients, ensuring synchronization and consistency.
- Although player states are disseminated by clients, these are monitored by the server, which ensures their validity. Also, this allows for the implementation of proof-cheating mechanisms.
- The central server can act as an arbiter in events, such as players finishing the race at the same time or pushing and grabbing each other simultaneously, requiring an uncomplicated agreement protocol and ascertaining consistency on the outcome.
- The direct dissemination of player states results in low latency travel, especially compared to regular a CS architecture, which helps fulfill the small tolerance requirement from the game.

3.3 IMPLEMENTATION

The constructed system is comprised of two applications, *GameClient* and *GameServer*, corresponding, respectively, to the client and the server referred in the previous [Section 3.2](#). Seeing that the game was developed in Unity, both applications are also constructed in *Unity*.

Considering the game's specific communication requirements, the system supports three transport protocols: [TCP](#), [UDP](#) and IP Multicast using [UDP](#), which their use in the system is explained in [Section 2.3.1](#). Both applications follow an asynchronous communication with multiple blocking sockets for receive, that using a multi-threaded solution, achieve a responsive effect on both server and client. The messages treatment is handled by a multi-receiver, first-in-first-out message queue. When messages are received, they are handled on a separate thread, which adds them to the message queue to be processed sequentially.

An application-specific protocol is a particular set of permitted message sequences specific to the game, which facilitates its operation ([Anthony \(2016\)](#)). The specific game application protocol to this system requires the following activity sequence:

- The game server is started and runs continuously as a service;
- A user starts an instance of the game client when they wish to join in;
- The user chooses an alias name, commonly known as username, and other information;
- The client connects to the server and shares their information;
- The server directs the client to an available lobby, which their information is propagated to a group of clients as part of a list of players;
- The server selects a random map to play and starts the race;
- The game is played. The server mediates events and keeps track of the game state to determine their outcome and if the game has ended, sharing game information with the clients. The client also propagates their gameplay moves (player states) to the other clients, so their simulation can be updated;
- The clients are notified of the game result by the server;
- The client's connection to the server is closed.

Each of these operations involves sending messages between the system's components to update the game state and synchronize the behavior. These messages are sent to each connected client and server in rapid succession. These contain a careful selection of content to send as little as possible, specific to a task. Therefore, the message processing is performed contextually based on the message type code. Depending on the transport layer protocol used to share them, additional mechanisms are implemented at the game level (depending on the application) to take care of latency, message sequence, or message loss. Moreover, a broader explanation of the system protocols and workflow of the multiplayer game is provided in [Section 3.6](#) and [Section 3.7](#), respectively.

3.3.1 GameClient - Client Application

GameClient application is the executable of the user's application, which they use to play the multiplayer game. This application interprets the user's interactions and expresses them in the game. It also exchanges messages with the server and other clients in the game session, applying them to the game to achieve a mirrored simulation. Furthermore, to help the application achieve from the problems described before, some mechanisms at the game level are implemented, for example, entity interpolation, player replication, game tick system, dead reckoning, etc.

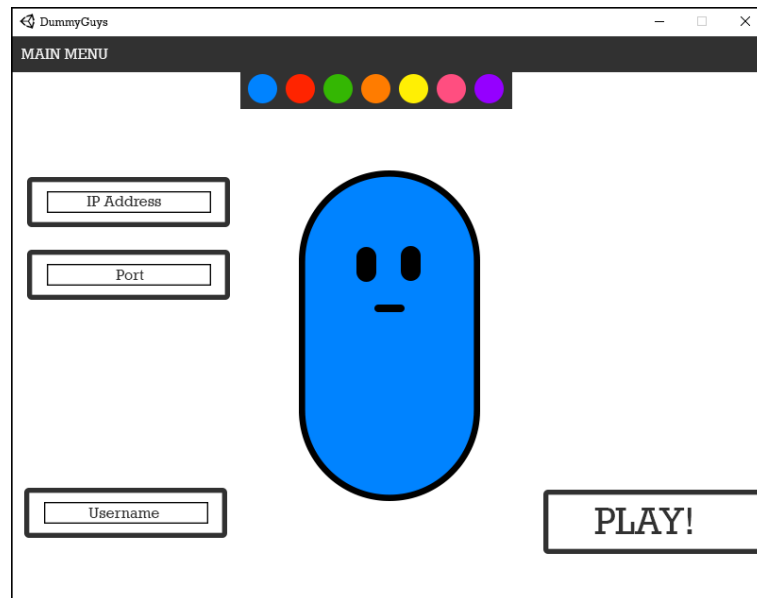


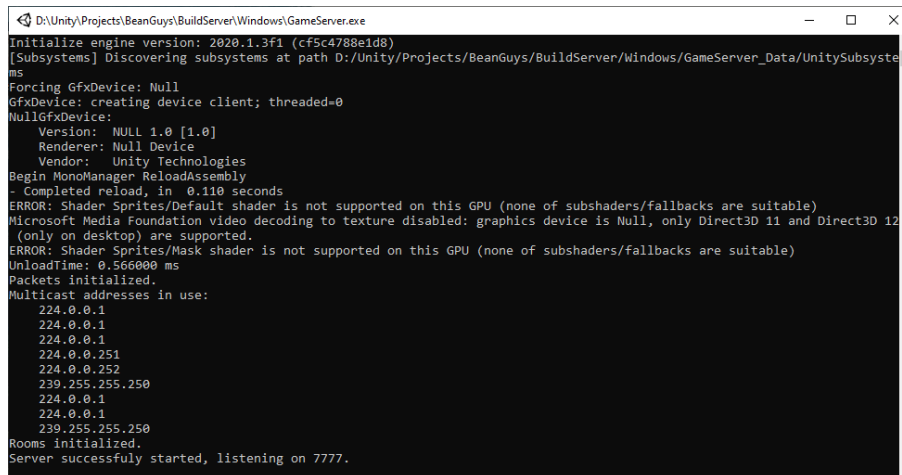
Figure 18: Client application main menu (mockup).

Figure 18 shows the mockup of *DummyGuys*' main menu. To participate in the game, the user presses the *Play!* button, initiating the connection process to the server. Nevertheless, the user must define the server port and IP address beforehand, using the [User Interface \(UI\)](#) provided, so the application knows who to contact and exchange messages. The user can also change their username and color applied to their character in the game world. While playing the game, the user can press the *escape* key to bring up the options menu. There, the user can exit the game or change settings such as the graphic's quality or audio volume.

If the client is disconnected from the server, another menu pops up, alerting them, which then leads them back to the main menu. Similarly, in case the client application crashes or closes while connected to the server, if not correctly disconnected, a heartbeat mechanism on the server notifies it. This prompts the server to remove the client from the connections and the game world, also informing the rest of the clients in the network of the client's disconnection.

3.3.2 GameServer - Server Application

GameServer application is the executable of the *DummyGuys*' server that starts its service of accepting connections, managing clients, and exchanging messages. The server is a console application, shown in Figure 19 because it provides a simple user interface and requires little to no user interaction. Additionally, a console application requires fewer implementation resources since it removes unnecessary functionalities, such as rendering objects, generating lighting, and effects or graphics.



```

D:\Unity\Projects\BeanGuys\BuildServer\Windows\GameServer.exe
Initialize engine version: 2020.1.3f1 (cf5c4788e1d8)
[Subsystems] Discovering subsystems at path D:/Unity/Projects/BeanGuys/BuildServer/Windows/GameServer_Data/UnitySubsystems
Forcing GfxDevice: Null
GfxDevice: creating device client; threaded=0
NullGfxDevice:
  Version: NULL 1.0 [1.0]
  Renderer: Null Device
  Vendor: Unity Technologies
Begin MonoManager ReloadAssembly
- Completed reload, in 0.110 seconds
ERROR: Shader Sprites/Default shader is not supported on this GPU (none of subshaders/fallbacks are suitable)
Microsoft Media Foundation video decoding to texture disabled: graphics device is Null, only Direct3D 11 and Direct3D 12
(only on desktop) are supported.
ERROR: Shader Sprites/Mask shader is not supported on this GPU (none of subshaders/fallbacks are suitable)
UnloadTime: 0.566000 ms
Packets initialized.
Multicast addresses in use:
  224.0.0.1
  224.0.0.1
  224.0.0.1
  224.0.0.251
  224.0.0.252
  239.255.255.250
  224.0.0.1
  224.0.0.1
  239.255.255.250
Rooms initialized.
Server successfully started, listening on 7777.

```

Figure 19: Server console application.

The server is event-driven, relying heavily on the client's messages. Each event is modeled as a separate activity, which is initiated when the relevant action is detected. Sockets are used to receive activities, invoking a handler to treat them. In addition to many user activity events, which start when a specific user event occurs, for example, when the player tries to push another player in the game world, there are also timer-based events, such as race start countdown.

The server allows for multiple game sessions to be created and run simultaneously, independently and isolated from one another, called rooms. Each of these rooms has different clients inside it, which are not allowed to communicate between them. This is because, while in the same machine, rooms have little to no communication with each other and need little to no access to information not explicitly owned to them. Therefore, they become clear candidates for units of parallelization. A good solution would also be to separate them into different machines. However, for a simpler study of testing the system's capabilities, workload caused by clients, and maintaining a well-functioning distributed game, it was decided to keep everything in the same machine. Additionally, it is inside them the game state and events are managed on the server application. Moreover, rooms are created at the start of the server or on-demand. Still, these are kept dormant until the server has notified them of their necessity.

Clients enter rooms, routed through a matchmaking mechanism after successfully joining the server. Each client on the system is associated with an id, taking different ids in the server and the room (explained further ahead). Seeing that clients do not have persistent data such as an account, and the server game worlds are

instantiated on command and discarded afterward. Therefore, the system does not integrate any persistence mechanisms in case of failure.

```
#DummyGuys server properties
28/12/2021 14:29:49
tickrate=30
vsync-count=0
max-player-per-room=60
max-rooms=5
player-interaction=true
server-ip=192.168.1.10
server-port=7777
room-min-port=7778
debug=true
local=true
count-processes=false
dorment-rooms=true
lag-compensation=true
```

Figure 20: Server properties file.

Lastly, the server application uses an external I/O file, called *server.properties* (shown in [Figure 20](#)), at its start-up. The data contained in it controls specific functionalities, which are used to configure the server when required. For example, when the `DEBUG` variable is true, the server ceases to execute additional verifications and other implemented processes, only needed when running on a normal execution. This file was indispensable when making tests during development and analysis since it avoided building different versions of the same system only to change some values or test new mechanisms.

3.4 COMMUNICATION

Following the very same communication workflow explained in [Section 2.3.1](#), this section presents the mechanisms constructed and used in the system for the Transport Layer Protocol, Message Queue, and Data Encapsulation communication layers.

3.4.1 *Transport Layer Protocol*

Given the need for networking reliability in most information sent and the need for fast delivery in others, it makes the decision of what transport layer protocol to use very important and dependent on the specific game information at hand. [UDP](#) is ideal for sending fast game updates, but messages are not guaranteed to arrive at the recipient. In contrast, [TCP](#) guarantees message delivery, but its speed is considerably slower and more expensive because of its use of ACK packets and their round-trip times. Furthermore, considering that the system needs to share the same information with all clients and the required bandwidth to do so when holding a large number of players in the room, Multicast (Network-Layer Multicast) shows itself as a viable option to implement. Therefore, the transport layer protocols implemented and their use in the system are as follows:

- *Client-Client* - Multicast with peers inside the room.
- *Client-Server* - [TCP](#) with the server and room, Multicast and [UDP](#) when in a room.

Client-Client communication is characterized by only using Multicast due to the client being the only one responsible for disseminating their player state with other clients. Since player state messages represent fast-paced, short-lived actions created several times per second, there is no problem losing some. So, [UDP](#) is more than sufficient for its dissemination. However, in conjunction with the fact that each client has to share their player state with every other client in the game makes Multicast more suitable, providing a more efficient way for clients to communicate directly with each other.

Client-Server communication employs all three communication channels due to the high variation in the relevance of messages sent between them and the number of recipients that need them. For instance, for the server to connect with a client and receive their information, it uses [TCP](#) due to the transmission of critical data. Without the arrival of this information at the server, the client would not successfully connect. On the other hand, [UDP](#) is used for less critical messages, and with a high creation rate, that does not affect the system if they are lost from time to time. Lastly, *Client-Server* communication also uses Multicast to take advantage of the fact that clients already use it to share player state messages. This way, the server can also receive and monitor these packets, allowing an efficient way of sending other necessary messages to all clients.

As a result of the implementation, the server maintains one [TCP](#) socket for listening to clients' connection requests, two others per connected client ([TCP](#) and [UDP](#)), and another per room to Multicast. On the client-side, clients maintain only three sockets, [TCP](#), [UDP](#), and Multicast, where the [TCP](#) and [UDP](#) are only used to communicate with the server (and room) while multicast is used for both clients and server.

Upon receiving data on these sockets, a check on the received message length is performed. On the **TCP** socket, receiving a zero-byte message (as opposed to no message) implies that in the context of the communication logic of this system, the connection has been lost. On the **UDP** and Multicast socket, receiving a four-byte message (as opposed to no message) is also taken to imply that the connection has been lost. If sockets happen to receive a message, its bytes are copied to an array, and in order to continue reading data from the stream, *stream.BeginRead*⁷ asynchronous read operation is called. The copied data is then given to the system's message queue for processing.

3.4.2 Message Queue

As mentioned, messages upon being received create a thread to process the data received without blocking the main thread of the receive loop. Due to this, messages are concurrent, needing to assess the game logic and write data in shared memory. Although writing in shared memory from different threads can work, it is also problematic to implement and very error-prone. In addition, *Unity* does not allow *Unity* components, such as *GameObjects*⁸, which makes up the player's character or everything else in the game world, to be accessed from outside the main thread. Therefore, it was decided not to pursue these more concurrent solutions.

To facilitate the messages' proper treatment, the system implements *MessageQueuer*, which allows a received message to schedule their treatment to be run on the same specific thread, avoiding unforeseen errors. This thread is *Unity's Update*⁹ loop, which permits *GameObjects* to be accessed.

MessageQueuer is used by every component in the system that handles the treatment of network data, namely server, room, and client. Although structured slightly differently on the server (and therefore in the room), it follows the same basis. On the client, *MessageQueuer* holds a list of *Actions*¹⁰ that each update is iterated and executed all code inside. New *Actions* are stored when information is received. To add them, the list of *Actions* inside *MessageQueuer* is locked and then adds them. When it is time to update, all code meant to run on the main thread (*Unity's Update* loop) is executed using *UpdateOnMain*, shown in [Figure 21](#). The process is accomplished by locking the list of *Actions* and copying its contents to a new list. The list of *Actions* is then cleared and unlocked to add more *Actions* over time, while the copied list is iterated, executed and then emptied on the next run.

7 <https://docs.microsoft.com/en-us/dotnet/api/system.io.stream.beginread?view=net-6.0>

8 <https://docs.unity3d.com/ScriptReference/GameObject.html>

9 <https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html>

10 An Action delegate is an encapsulated method that returns no value, or a void encapsulated method. - <https://docs.microsoft.com/en-us/dotnet/api/system.action-1?view=net-5.0>

```

private static void UpdateOnMain()
{
    queueCopy.Clear();
    lock (queue)
    {
        queueCopy.AddRange(queue);
        queue.Clear();
    }

    for (int i = 0; i < queueCopy.Count; i++)
        queueCopy[i]();
}

```

Figure 21: MessageQueuer update method.

On the server, *MessageQueuer* still has the responsibility to run all *Actions* on the main thread, but it does not store them. Since the server has multiple rooms, receiving and treating incoming messages, to not mix *Actions* from different sources, they are stored in the separate *MessageQueue* classes. Therefore, each room and server will create a *MessageQueue* and add it to a list of *MessageQueues* in *MessageQueuer*, which it goes through to execute all *Actions* inside. Moreover, since the system has multiple rooms going dormant after the game instance has finished, if the *Actions* of each room and the server are separate it creates a better approach to clearing them later. This is because a *MessageQueue* can be easily removed from *MessageQueuer*.

Furthermore, these *Actions* are simple processes, such as add, remove, modify and update, with as much little data to handle as possible. Intensive processes are avoided from being executed using a lock. When such operations are required, they are run asynchronously to avoid disrupting the system and, possibly, other active game instances. It is the case for loading a game scene on the server. This task is heavy since the application has to instantiate multiple game scene objects and scripts while other game instances are running. By processing it asynchronously, the scene's load process does not put at risk the correct behavior of the server.

3.4.3 Data Encapsulation

For the data encapsulation approach, the system uses *Packet*, which allows for low-level data to be converted into high-level data (bytes) and vice-versa. The following [Figure 22](#) and [Figure 23](#) show an example of the read and write process when dealing with a packet. Since *Packet* inherits from *IDisposable*¹¹, the system manually disposes of it when it has finished using the class for the message. For this, a packet instance is defined inside a using block, which disposes of it afterward.

[Figure 22](#) shows the *Add* method that converts a given attribute to a list of bytes and adds it to the packet buffer. After inserting the values, the buffer length is inserted at the beginning of the packet using *InsertLength*. This way, when the message is received, the recipient can handle the data correctly.

¹¹ <https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=net-6.0>


```

public static void PlayerMovement(PlayerState _state)
{
    using (Packet _packet = new Packet((int)ClientPackets.playerMovement))
    {
        _packet.Add(ClientInfo.instance.ClientRoomId); // int

        _packet.Add(_state.tick); // int
        _packet.Add(_state.position); // Vector3
        _packet.Add(_state.rotation); // Quaternion
        _packet.Add(_state.ragdoll); // Bool
        _packet.Add(_state.animation); // int

        _packet.InsertLength();

        MulticastUDPData(_packet);
    }
}

```

Figure 22: The following serialization example shows the creation of a player state message on the client. After being constructed, a client send data method is called (*MulticastUDPData*) to transmit the data through the appropriate socket.

```

public static void PlayerMovement(int _clientRoomId, Packet _packet)
{
    int _tick = _packet.GetInt(); // int
    Vector3 _position = _packet.GetVector3(); // Vector3
    Quaternion _rotation = _packet.GetQuaternion(); // Quaternion
    bool _ragdoll = _packet.GetBool(); // bool
    int _animation = _packet.GetInt(); // int

    GameManager.instance.PlayerMovement(_clientRoomId, _tick, _position,
        _rotation, _ragdoll, _animation);
}

```

Figure 23: The following deserialization example shows the treatment of player state messages on both client and server. After its contents are retrieved, they are given to a Procedure Call (*PlayerMovement*) that applies them to the corresponding player character inside the game world.

For the deserialization, [Figure 23](#) shows the various *Get* methods that convert the bytes received back to their original format. These methods must consider the object they are trying to retrieve from the packet. For example, *GetInt* tries to get an *integer* by reading 4 bytes inside the packet's array. Therefore, the complete process needs to regard the order in which objects were written in the packet, as seen comparing [Figure 23](#) with [Figure 22](#). Otherwise, it won't successfully retrieve them.

A system as complex as a multiplayer game requires a lot of information to be shared and synced. Therefore it requires a lot of different types of messages, holding just the necessary data to realize a specific task. Additionally, the system needs to know the packet content, its order, and its task. To accomplish this, all packets in the system are identified through an *integer* inserted at the beginning of its content. So, using this *integer* as an identifier,

the system knows the packet's contents and order. However, from a programmer's view, it is hard to identify a message through a simple *integer*, especially considering that there are various types of messages. Thus, it was used *enums*, shown in [Figure 24](#).

```

// Sent from server
18 references
public enum ServerPackets
{
    accept,
    refuse,
    joinedRoom,
    disconnected,
    playerJoined,
    playerLeft,
    playerRespawn,
    playerFinish,
    playerCorrection,
    playerGrab,
    playerLetGo,
    playerPush,
    map,
    startGame,
    endGame,
    serverTick,
    serverClock,
    pong
}

// Sent from client
10 references
public enum ClientPackets
{
    introduction = 18,
    playerMovement,
    playerRespawn,
    playerReady,
    playerGrab,
    playerLetGo,
    playerPush,
    ping
}

```

Figure 24: System's packet types.

Additionally, it is used a dictionary, denominated *packetHandler*, containing all packet types and the designated method for proper treatment of its contents. Therefore, upon receiving a message, the system reads the packet id and uses the *packetHandler* to connect it to the correct method and treat it correctly. If a message received has no id, it is deemed garbage and immediately discarded. Since the server only deals with incoming messages from the client, its *packetHandler* only needs to contain *enums* from the client's packets. However, since clients deal with incoming packets from both the server and other clients, its *packetHandler* needs to have values from both *enums*. Initially, this would have caused conflicts because *enums* start at zero, resulting in two different packets with the same id. Thus, client packets begin on the eighteenth, corresponding to the seventeen distinct server packets ids plus one, as shown in [Figure 24](#).

Finally, apart from the information needed for a specific task, packets also contain the room's id (if sent from the server) or the client's id (if sent from a client), as seen in both [Figure 22](#) and [Figure 23](#). Therefore, packets are auto-contained and possess all the necessary data to identify which information was sent or requested and who sent it.

3.5 ARCHITECTURE

This section explains the *GameServer*'s and *GameClient*'s various components, shown in [Figure 25](#) and [Figure 26](#), that make up the network and game logic. These two layers were kept separate, providing a clean development method by successfully abstracting critical communication problems, such as consistency and reliability issues from the game logic code. It also represents a good practice since most MMO game companies have big teams working simultaneously at different levels of the game. Thus, a game programmer should be able to work abstracted from the network domain, and a network programmer should be able to work unconcerned with the game domain. Moreover, each component of the applications represents clear competencies and interactions, implemented by a set of clearly defined protocols, making it possible to develop, update or adapt components independently. These components can be counted up to four critical models, being the following:

- *Server*, contained in *GameServer*, is responsible for handling new client connections and managing the server service;
- *Room*, contained in *GameServer*, is responsible for managing a room's state, events, and messages;
- *Client*, contained in *GameClient*, is responsible for managing a client's state, events, and messages;
- *Game*, contained in both *GameServer* and *GameClient*, is responsible for the game logic, state progress, and direct connection between the logical and physical (in the game world scene) game.

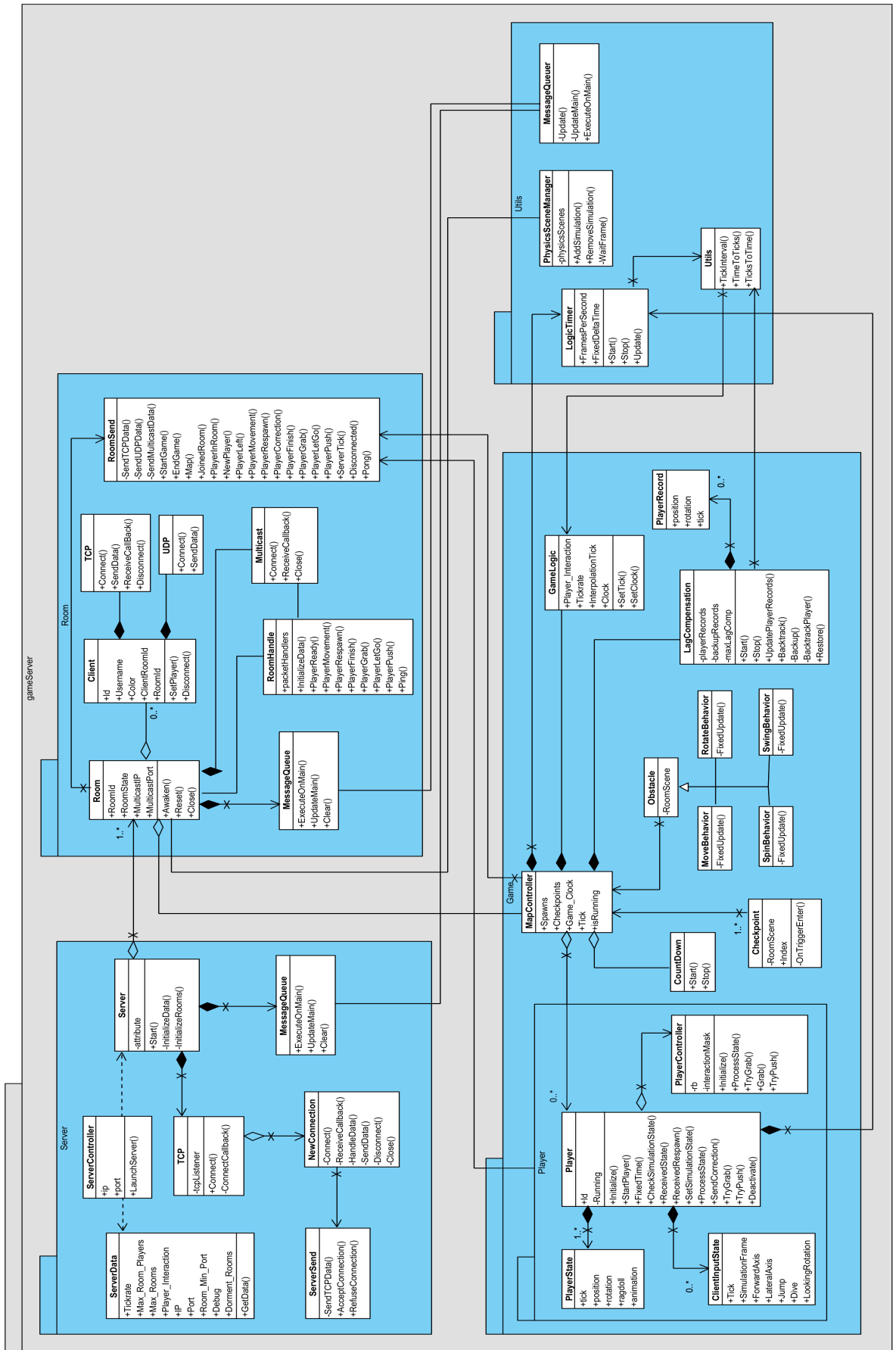


Figure 25: GameServer class diagram overview.

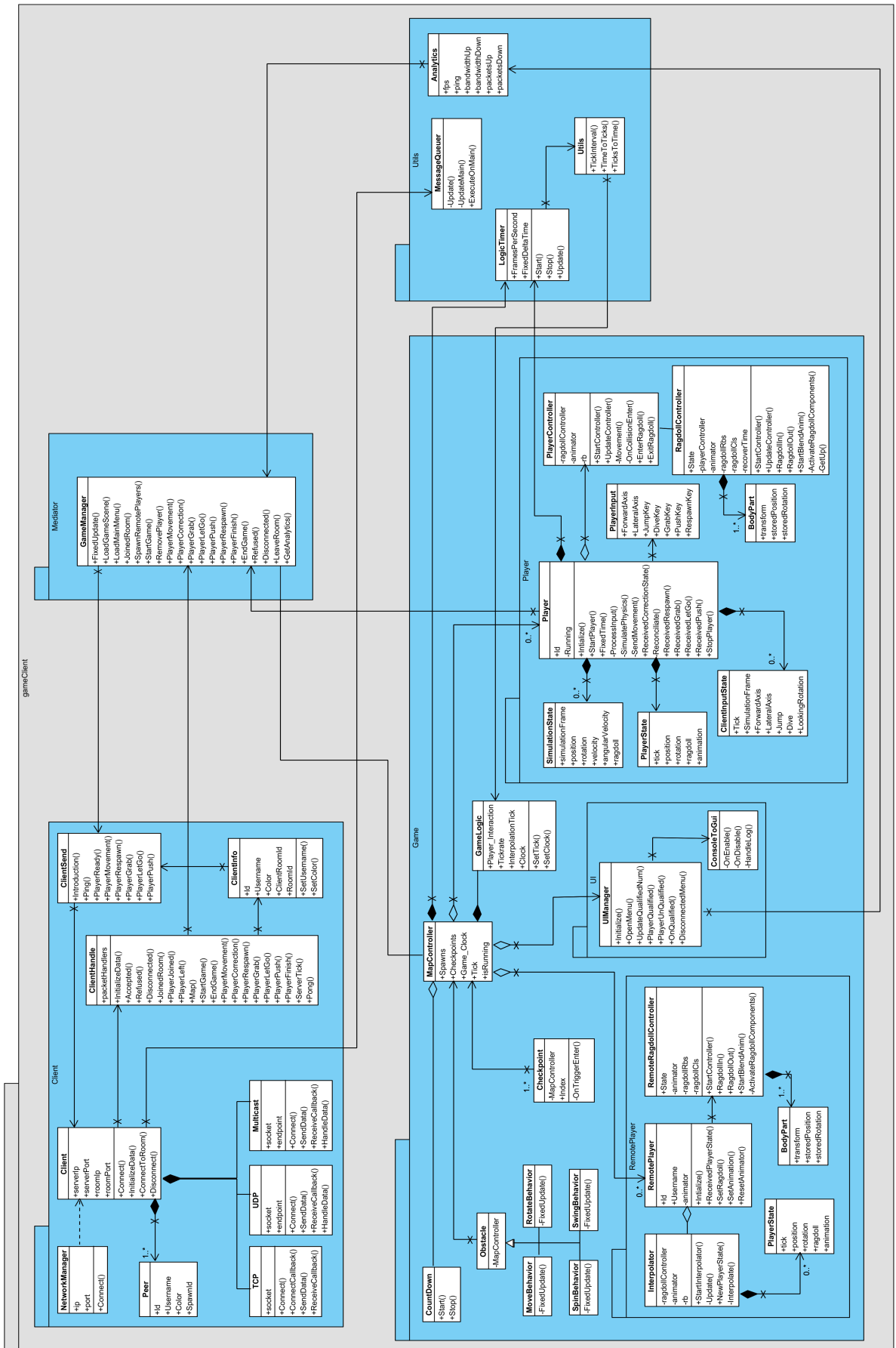


Figure 26: GameClient class diagram overview.

3.5.1 Server

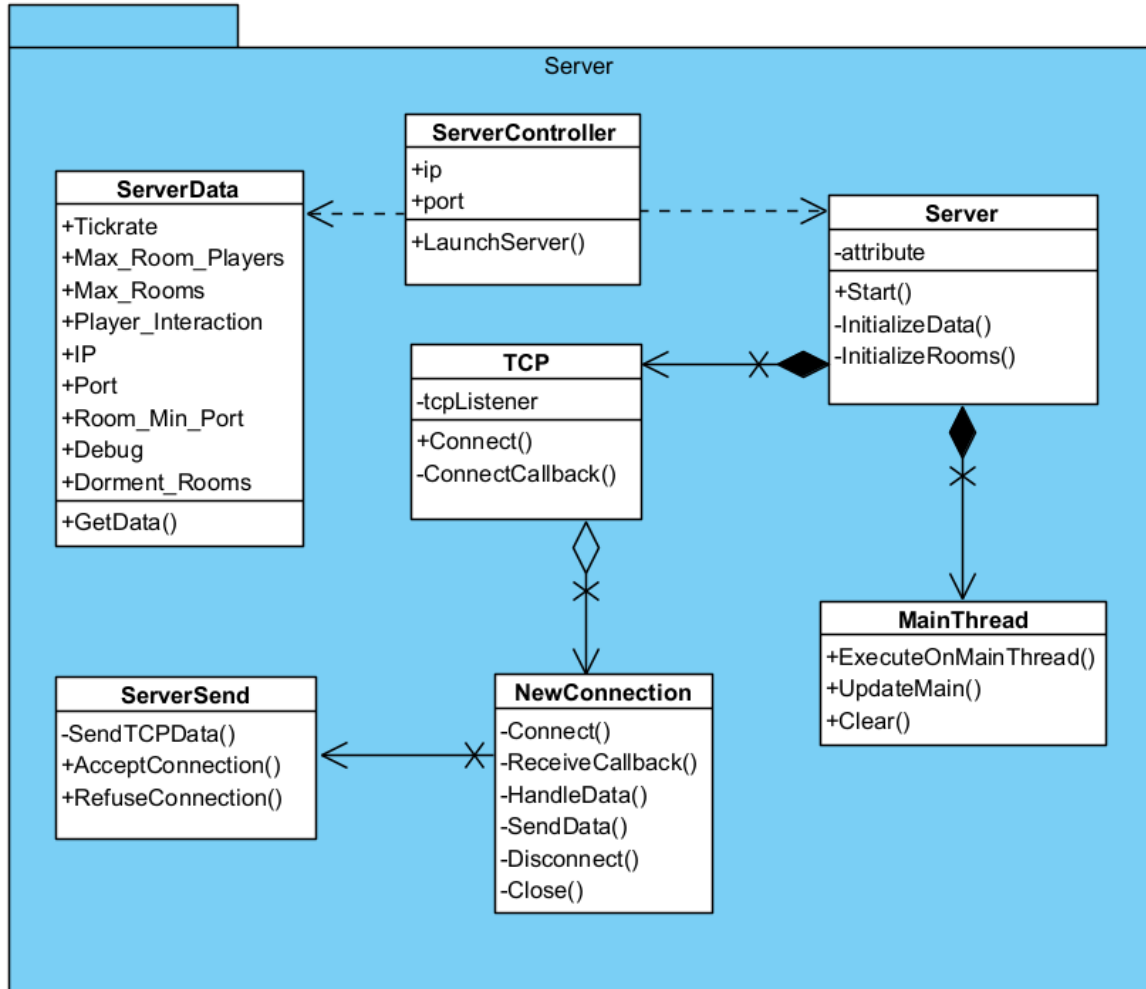


Figure 27: Server model’s class diagram overview.

The *Server* model contains the server’s main logic, which initiates all its processes and configurations needed for correct behavior. Without this model, all other models inside the *GameSever* application would not issue, and its service would not start.

ServerController is used to initiate the server, using its *LaunchServer* method when the *GameSever* application is launched. First, this method retrieves the data from the already mentioned *server.properties* file and saves it in *ServerData* to later configure the server. Then, it initiates the *Server*, which contains the server’s logic and information needed for its network components, such as rooms and ports being used. When initiated, it creates a fixed number of rooms and assigns them a multicast address and port available in the network.

TCP is used to handle incoming new connections from clients, which prompts the creation of a *NewConnection* that handles the connection process for each of them. This process involves an exchange of messages that the

server creates by using *ServerSend*'s various methods. Since all the game logic is implemented in the room, the server only handles new connections and directs them to the available rooms (matchmaking), keeping the tasks that has to handle and send at a minimum.

3.5.2 Room

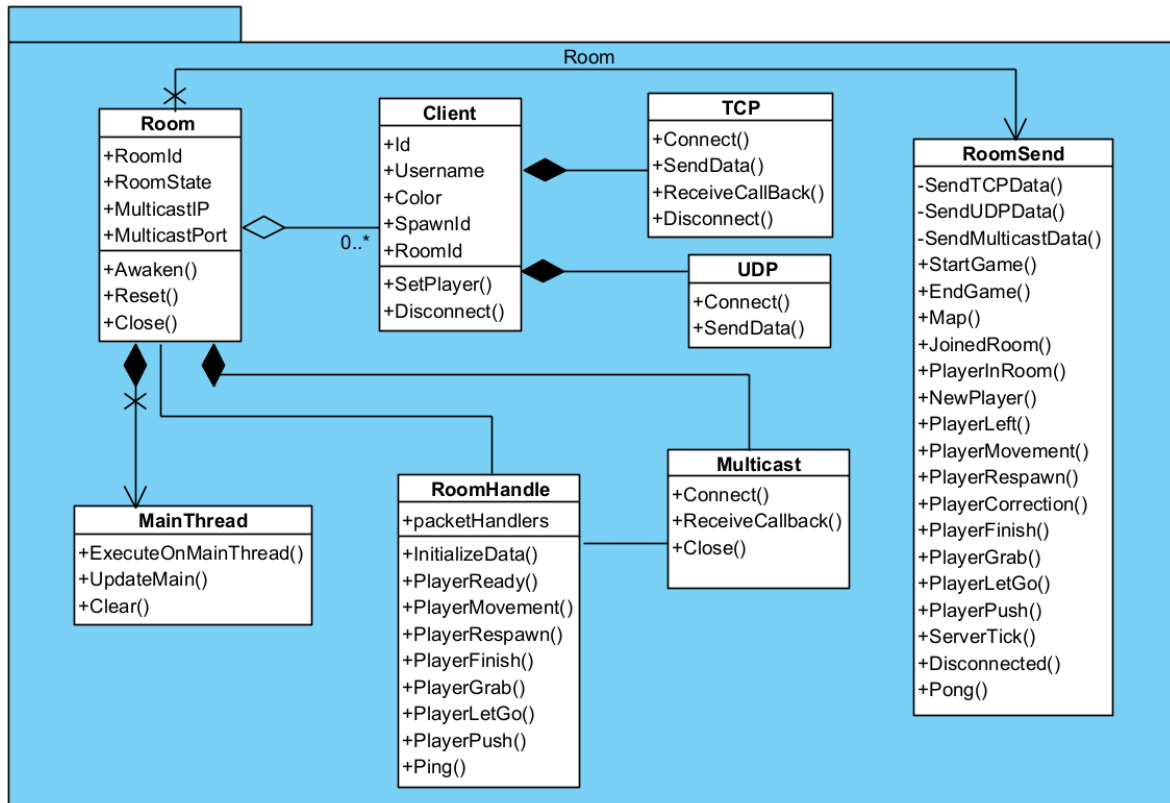


Figure 28: Room model’s class diagram overview.

Seeing that in *Room*, everything game-network-related takes place in the server, most of the messages and interactions involving clients are calculated here. *Room* holds the main logic of a server’s room, controlling the game session configurations, information, and room state (*RoomState*), which can be translated into the following states: *dormant*, *looking*, *textitfull*, *playing*, and *closing*. Moreover, *Room* has a dictionary of clients connected to it, each represented by a *Client*. This class holds all information in the server about one particular client and the character representing them in the game world. Therefore, this class is used for tasks, such as instantiating the client’s character and directing the player state messages received to the player object.

To transmit information, the room uses *Multicast* with the multicast address and port assigned by the server to open a socket to Multicast with all clients inside the room. To communicate with the clients, using the TCP and UDP sockets, it uses *TCP* and *UDP* that each *Client* contains. Ultimately, all messages received on the room end up at *MessageQueue* to be later be processed, invoking a *RoomHandle*'s method to unpack the message

content and then apply it to the game through procedure calls. In contrast, to create messages using game data, the room uses methods from *RoomSend* and afterward transmits them in the appropriate socket using send data methods, such as *SendTCPData*, *SendUDPData*, etc., shown in [Figure 28](#).

3.5.3 Client

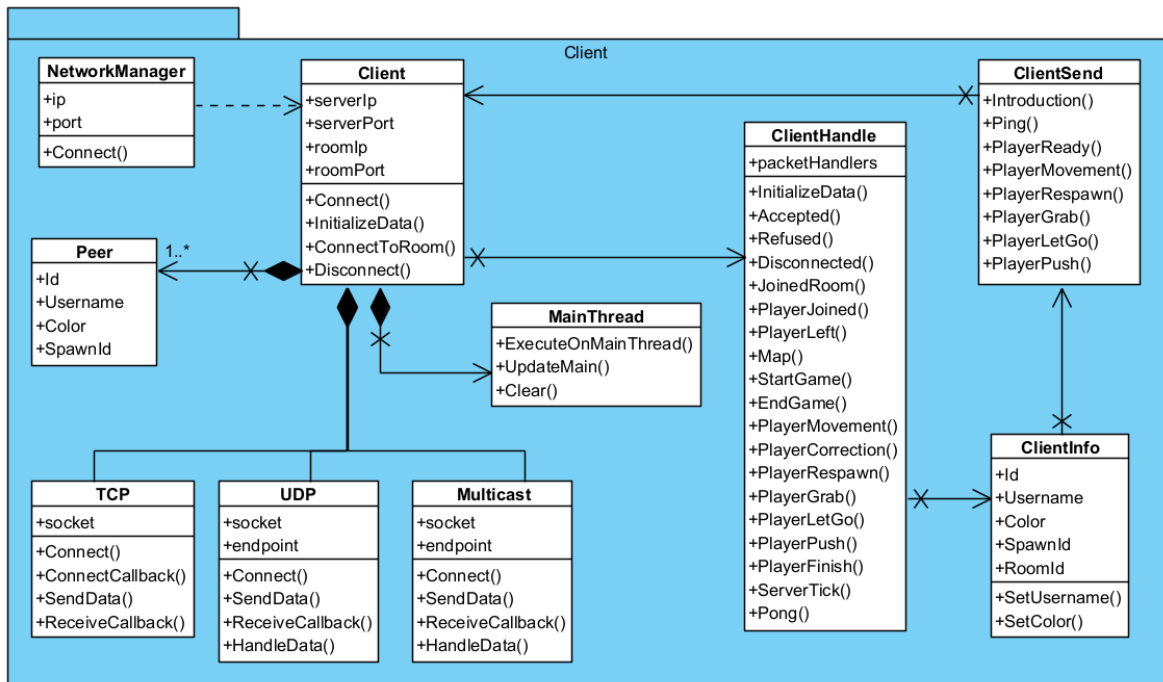


Figure 29: Client model’s class diagram overview.

The *Client* model contains all the network logic of the *GameClient* application. *NetworkManager* initiates the process of connecting the client to the server when the user presses the application’s play button. Given the IP address and port, the information from the *UI* is passed onto *Client*, using *Connect* method to let it know where the client has to connect.

Client contains the main network logic and stores information, such as the multicast address and port used for the room communication. If other clients join the room, when the client is already connected to it, they will store their information in a dictionary of *Peer*. *Peer* contains all necessary information about one particular peer in the room, which the client can use, for example, to instantiate their character in the game world, according to their information. Furthermore, as *Client* treats messages, it has a *packetHandler* dictionary and a *MessageQueuer* class to store and process them. To unpack incoming messages, the client uses *ClientHandle* to process the data accordingly and then passes it to procedure calls that apply it to the game. Moreover, for the client to share information with the server and peers, it uses *ClientSend*’s various methods to create messages. To transmit them, *TCP*, *UDP*, and *Multicast* classes contain the sockets, which the clients choose to use based on the aspects of the message, as already explained in [Section 3.4.1](#).

Before the client can connect to the server, it needs to generate a *Guid* to serve as their identifier inside it. The reason for the use of a *Guid* instead of a normal *integer* is because it produces a statistically unique sequence. Therefore, every client in the system is distinctively identified, avoiding situations where clients have the same id, which would cause several issues. This identifier is the *Id* value in *ClientInfo*. This class stores important information about the client's information in the server for multiple purposes. For instance, *Id* identifies the client in the server and in packets sent, while *Color* defines the color of the client's character in the game world.

3.5.4 Game

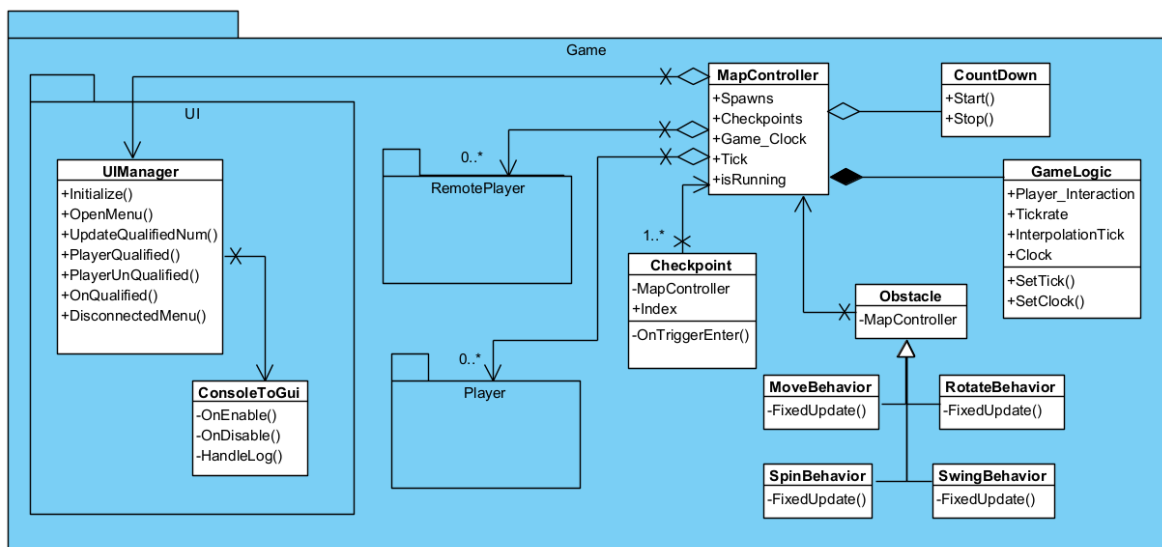


Figure 30: Game model's class diagram overview.

The *Game* model, isolated from the network logic, holds the game logic and components that make up the logical game entity in the applications. The model shown in [Figure 30](#) represents the *Game* model of the *GameClient* application since it has the full capabilities implemented on the *Game* model. On the *GameServer* application, *Game* model follows the same structure but without *RemotePlayer* and *UI* objects. The difference can be seen better in [Figure 25](#).

This model consists of two vital components: the characters and the game scene. The characters are divided into two different types, the local player (*Player*) and the remote player (*RemotePlayer*). Even though they seem similar, both describe a different logic behind their behavior, as discussed ahead.

MapController class is an internal logical entity representing the game and its state. In conjunction with the game scene, instantiated in *Unity*, *MapController* updates the game, advancing the simulation of each loop, triggering events and determining their outcome, changing the game state, and synchronizing the clients. This class stores the list of players and their characters. It also is directly connected to the session's *Room*, using its procedure calls (in *RoomSend*) to share events across the room network. Moreover, *MapController* stores other

various *GameObjects*, such as spawn points used for instantiating the players at the correct position at the start of the game session. When the game is about to start, *Countdown* starts a countdown, which reaching zero triggers the race's start event in *MapController*.

GameLogic stores the game session settings, such as tick rate and player interaction, defined by the room, and all game clients must follow. Most importantly, it stores the simulation's game tick and clock, although managed by *MapController*. The game tick creates the basis for game consistency and synchronization while the game clock controls all obstacles in the game world. Every obstacle object in the game world implements *Obstacle* and can adopt different behaviors, shown in [Figure 30](#), dependent on the game clock value to move. For instance, if its value is continuously increasing, the obstacles perform their movement behavior as implemented. Otherwise, if the game clock stops advancing, so do all the obstacle objects.

Moreover, *Checkpoint* is used for the checkpoint system in the game, where each checkpoint object, when touched by a client, triggers a checkpoint event in *MapController*. This way, the game keeps track of every client's last checkpoint and uses it when a player requests or needs to respawn. *FinishLine* is used on the finish line object when touched by a player, triggers an event in *MapController* to check if the client has qualified or not.

Lastly, the *Game* model employs *UI* elements (only on the client-side) to inform the user about the system and game. These are controlled by *UIManager*, which has various methods to control the information displayed on the screen, such as game information, menus, and player information. Also, it retrieves information from *Analytics* (in *Utils* model) to display relevant information when debugging, such as ping, bytes/sec, and packets uploaded or downloaded per second. Finally, *ConsoleToGUI* translates logs from the game engine, which are not visible on the game build, to a GUI window, allowing for proper debugging while developing the system.

Player

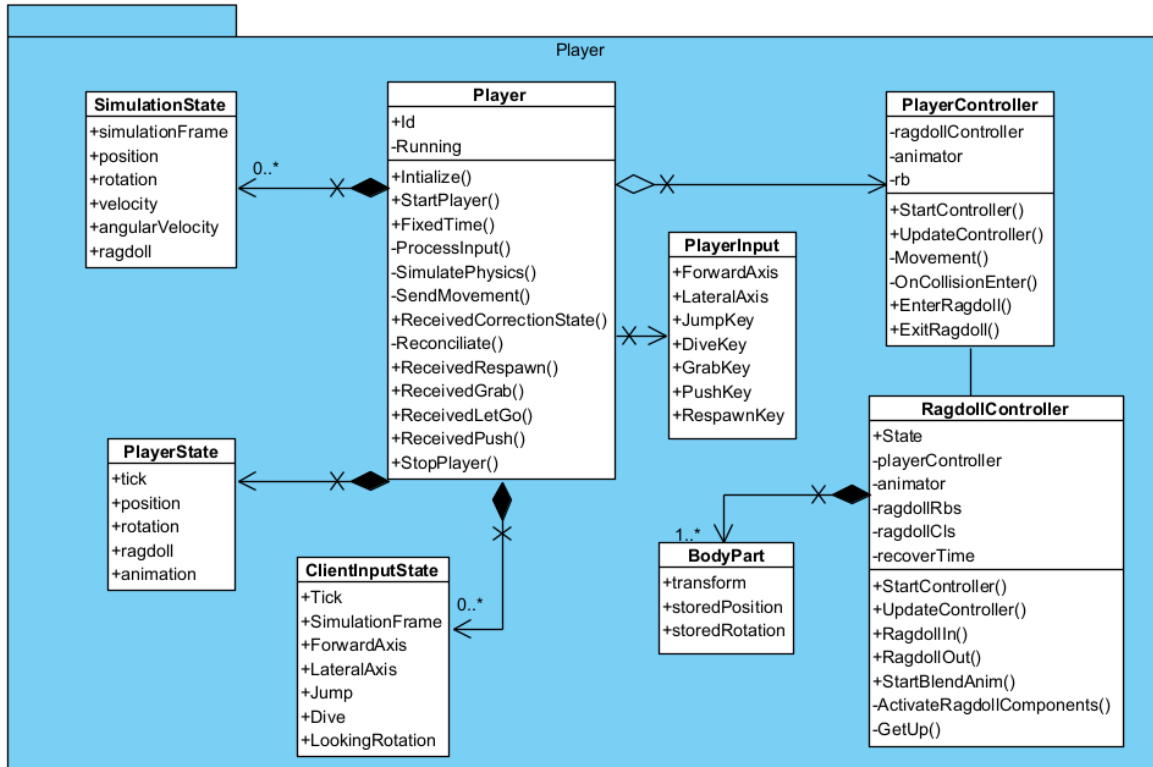


Figure 31: Player model’s class diagram overview.

The player’s character represents the user’s actions in the game world. This object poses one of the most substantial difficulties in distributing a multiplayer game, dependent on the architecture, transport protocol, and gameplay.

Player handles incoming game data from the network layer, intended for the local player. It controls the networked side of player behavior, the main component of the player replication mechanism. *PlayerState*, *SimulationState*, and *ClientInputState* are also used for the replication system, corresponding to data packs stored or sent over the network. *PlayerController* only handles the player’s behavior at the game level. These are the different types of movement behaviors and actions executed by the user, gathered from *PlayerInput* that reads the input data generated from the user and send it to *PlayerController*. These two classes act as an interface between the player object and the human player controlling it.

When a player collides with an object, it can lead them to transition into ragdoll state, meaning they can’t control the character until it gets up. The transition from animated to ragdoll state, and vice-versa, is managed by *RagdollController*. When initialized, the various ragdoll bodyparts are created and stored in an array of *BodyParts*. *RagdollController* also controls the force applied to the ragdoll when hit and the time it needs to get back up. Lastly, it also ensures that the necessary actions for the transition from ragdoll to animated state are met, such as arranging a proper location to position the player object and then get up.

Remote Player

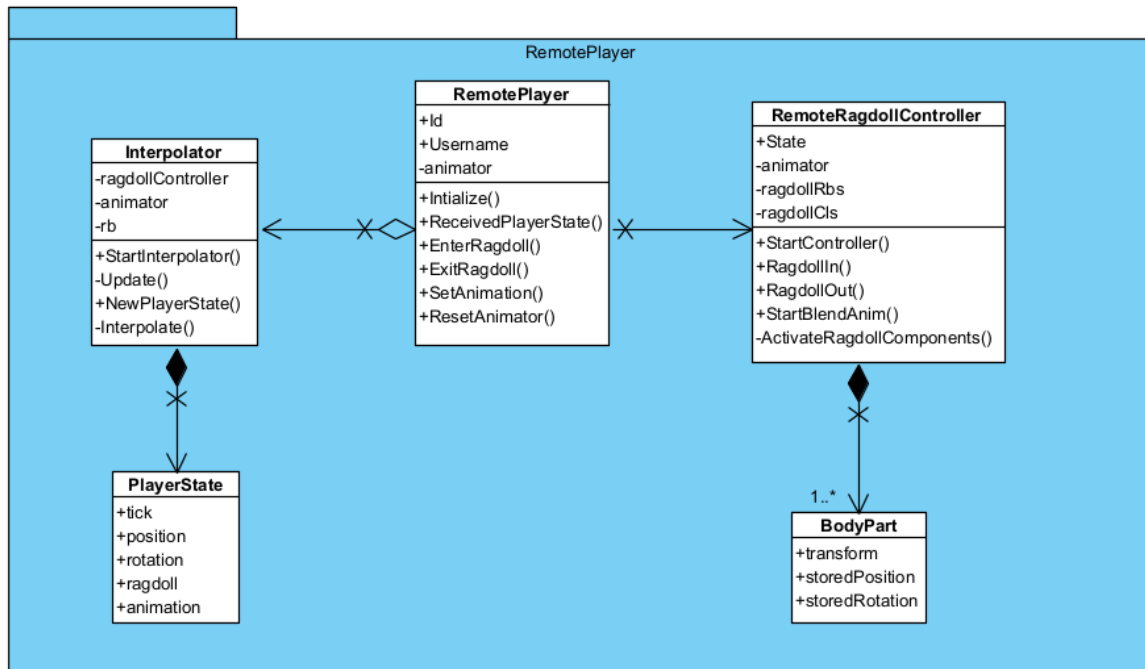


Figure 32: RemotePlayer model's class diagram overview.

Remote players are not controlled by the local user but instead by the user's peers, which typically are on a remote machine. These objects represent most of the players in the game simulation of a client application. Contrary to the local player's object, their movement is not dependent on physics. Instead, these are highly dependent on information sent over the network to function. This information is given to the *RemotePlayer*, which handles all the networked side of the object's behavior, just like in the local player. The received information composes a *PlayerState*, which holds all the necessary information about a player state. *Interpolator* uses these player states to replicate the original player movement.

Similar to the local player's *RagdollController*, *RemoteRagdollController* creates an array of *BodyPart* classes that compose a ragdoll and control its state. However, some of its functions were removed and solely follow the information received. For example, the get-up process and calculated position to stand up are controlled by foreign data and not by the physics of the local player's simulation.

3.6 APPLICATION PROTOCOLS

This section explains the application protocols that facilitate the game's operation, from a client being connected to the server, playing the game, and being disconnected from the server.

3.6.1 Messages

The protocols involve exchanging messages between the client and the server. Before getting into their proper explanation, this section goes through the multiple types of messages employed while also describing their use and the transport layer protocol employed:

Server messages:

- *TCP* **accept** - Sent to the client trying to connect to the server to notify them the server accepted their connection;
- *TCP* **refuse** - Sent to the client trying to connect to the server to notify them the server refused their connection.

Room messages:

- *TCP* **startGame** - Sent to the client to notify them the game has started;
- *TCP* **endGame** - Sent to the client to notify them the game has ended;
- *TCP* **map** - Sent to the client to notify them of the map to load;
- *TCP* **joinedRoom** - Sent to the client to notify them they have joined the room;
- *TCP* **playerJoined** - Sent to the client to notify them a client has joined the room;
- *TCP* **playerLeft** - Sent to the client to notify them a client has left the room;
- *UDP* **playerCorrection** - Sent to the client to notify them of a correction that must be applied;
- *TCP* **playerGrab** - Sent to the client to notify them they have grabbed a player or they have been grabbed;
- *TCP* **playerLetGo** - Sent to the client to notify them they have let go of a player or they have been let go;
- *TCP* **playerPush** - Sent to the client to notify them they have pushed a player or they have been pushed;
- *TCP* **playerRespawn** - Sent to the client to notify them their character was respawned;

- **TCP** **playerFinish** - Sent to the client to notify them a player has finished the race;
- **Multicast** **serverTick** - Sent to the clients to notify them of the game tick;
- **Multicast** **serverClock** - Sent to the clients to notify them of the game clock;
- **UDP** **pong** - Sent to the client to notify them of their ping.

Client messages:

- **TCP** **introduction** - Sent to the server requesting to connect;
- **Multicast** **playerMovement** - Sent to the server and the clients indicating the client's player state;
- **TCP** **playerRespawn** - Sent to the server requesting respawn of their character;
- **TCP** **playerReady** - Sent to the server indicating the client is ready to start the game session;
- **TCP** **playerGrab** - Sent to the server requesting to grab a player;
- **TCP** **playerLetGo** - Sent to the server requesting to let go a player;
- **TCP** **playerPush** - Sent to the server requesting to push a player;
- **UDP** **ping** - Sent to the server requesting a pong message.

3.6.2 New Connections

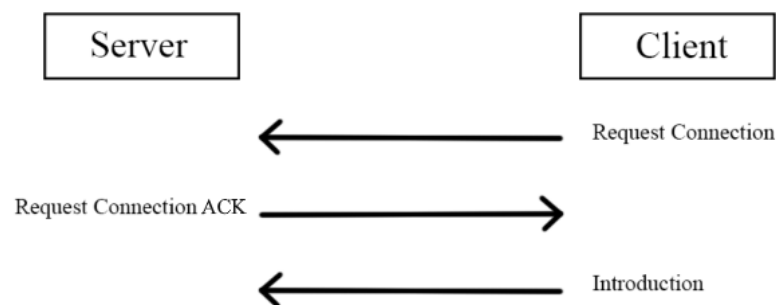


Figure 33: New Connection Process.

For a user to play the game, they has to join the server by introducing themselves in order to create their *Client* class on the server. For this, the system has implemented the following **TCP**-based process, represented in **Figure 33**, to successfully connect them to the server. This process consists of the following:

- The client sends a *requestJoin* packet, creating a *NewConnection* on the server meant to handle the client's messages as they try to connect, representing the client on the server until they have successfully joined a room.

- Once a *NewConnection* is created, it immediately tries to respond to the incoming TCP connection by sending an *accept* packet, informing the client that they have reached the server and that it is waiting for their detailed information to be sent over.
- Upon receiving this message, the client sends an *introduction* packet holding the information needed to represent the client on the server, such as *Id*, *Username*, and *Color*, completing the exchange of messages for the client's introduction.

However, the server still has to validate the information received. This consists of checking if the client *Id* is not already in the system and if the rest of the variables are valid. If the contents are acceptable, the client proceeds to the next phase, matchmaking. In case the information sent over is not in compliance with what was already mentioned, the server automatically assumes something is wrong with the client, and terminates their connection attempt by sending an additional *refused* packet, disconnecting them from the server.

3.6.3 Matchmaking

Even after connecting to the server, clients cannot play right away. They still have to join a room where the game session takes place. This happens through matchmaking, which consists of connecting players to an available online game session. There are many ways to go about implementing it, such as allowing players to connect and create a session at their will, or doing it for them. The constructed system follows the second option, where the matchmaking algorithm implemented instructs the server to find an available room and directs the clients to it.

This mechanism consists of checking the room state, previously mentioned as *RoomState*. If a room is in the *looking* state, it means it is not full and the game session has not yet started, allowing for the client to join in. Since the server has a limited number of rooms, no room will be found if all them are full, which results in the client being disconnected from the server. However, if it does find one, the server transfers the client connection (*NewConnection*) to the room. Afterward, the client is notified of the room's information by receiving a *joinedRoom* packet, which they use to connect to it.

Going on a more commercial route, a good way to implement matchmaking is by having a game instance per server and a discovery server that directs clients to an available game instance. This removes the load (CPU and bandwidth) of directing clients from the same machine that runs the game, which increases the server availability to receive and manage connections due to having a lower load. An example of this is Unity's hosting platform, *Multiplay*¹², which has multiple game servers that are kept dormant or running at low tick speeds until needed. After identifying a suitable group of clients, a matchmaker server searches for a server to host the game session. When one is acquired, the matchmaker sends the server's details to the clients, which they use to connect to it.

Lastly, the constructed matchmaking system does not allow for grouping, which could, optionally, be a good feature to implement. It would provide the possibility to introduce a ranking system centered on the player's skill,

¹² <https://docs.unity.com/multiplay/>

if the game allowed, such as *CS:GO*¹³ and *Rocket League*¹⁴ have. Another alternative could be to group clients by similar ping to assure better connection and fairness in latency.

3.6.4 Join Room

The first interaction involving the room and the client happens when the server notifies the room that a client is going to be directed to it, awaking the room if dormant and setting its state to *looking*. Following this, the room adds a *Client* to its dictionary of clients, storing their information gathered from the server during the introduction process. Apart from the information sent by the client, the room assigns them an id (*integer* between 1 and 60), corresponding to their id inside the room (*clientRoomId*). All game packets sent by the clients inside the room use this id, instead of the *Guid* id previously mentioned. Considering the high amount of players that a room can hold and that each player sends messages like player states every second, sending their *Guid* id, a heavy object, would create a substantially higher bandwidth requirement. Moreover, the *integer* id is only used in events inside the room, where every client must take a unique slot between 1-60. So, two players with an equal id are never simultaneously in the same room.

After the client's information is stored on the room, they receive a *joinedRoom* packet containing the information about the room they have been directed to, such as the room's multicast address and port. At this point, only the client's **TCP** channel is connected to the room (transferred from the introduction). However, upon receiving a *joinedRoom* packet, the client proceeds to also connect their Multicast and **UDP** sockets to the room's IP address and port. Thus having all transport layer protocols initiated and ready to send/receive data, completing the connection setup between the room and client.

Using **TCP**, the room sends multiple *playerJoined* packets containing the existing members inside the room to the client. This way the client is aware of all the clients already inside the room and their information. Oppositely, the room also sends the new client's information to every client inside it, by sending a *playerJoined* packet. In conclusion, all clients in the room end up being aware of every other client.

Lastly, upon adding a new client to the network, the room checks if the number of clients inside it has finally reached the room's maximum capacity, which by default is 60 players, although changeable in the *server.properties* file. If the room happens to reach the maximum allowed capacity, it changes the state to *full*, so no more clients join in, and initiates the process starting the game session.

3.6.5 Room Start and End

The game session initiation process starts by choosing a random map where the game session takes and then instantiates it. The instantiating process is managed by *PhysicsSceneManager*, in the *Utils* model, which is responsible for creating and destroying scenes in the game server application. The game's worlds, also known as scenes, are loaded in *Additive* mode, which *Unity* uses to allow multiple parallel scenes to run simultaneously

¹³ https://pt.wikipedia.org/wiki/Counter-Strike:_Global_Offensive

¹⁴ <https://www.rocketleague.com/>

while staying independent and isolated from each other. This way, the system can instantiate as many game world instances as it needs in order to have one per active room. After the scene has been created, *PhysicsSceneManager* assigns it to the room that requested it. To retrieve the scene generated, *PhysicsSceneManager* issues a *Coroutine*¹⁵ that waits for one frame to pass so that *Unity* can load the game scene and assign it to a room.

After receiving the scene template, the room instantiates the client's characters in their corresponding spawn id, and sets up the necessary game components. After the scene has been setup and is ready to start the game, the room sends a *map* packet to the clients, containing the map index, which in turn they will use to instantiate the game world in their application. When every client has finished setting up their game world instance, they send a *playerReady* packet to the server, informing the room that they are ready for the game to start. After all players have sent this packet, the room begins the game session by sending a *startGame* packet to every client, and setting its state to *playing*.

Lastly, when the race has finished, the room sets its state to *closing* and sends a *endGame* packet to let the clients know that the game has ended. On the client application, this stops the game and shows the race result. Additionally, it starts a timer that will disconnect the client from the room and lead them to the main menu. Meanwhile, the room waits for every client to disconnect and then resets itself, issuing the deletion of the game scene and the associated values no longer needed. Furthermore, it closes the Multicast socket and returns its state to *dormant* until asked to house a new group of clients, starting the process all over again.

15 A coroutine allows tasks to be stretched across many frames. - <https://docs.unity3d.com/Manual/Coroutines.html>

3.7 GAME PROTOCOLS

One of the most complex tasks that the system faces is the game's distribution. To better understand it, the multiple gaming mechanisms implemented in both server-side and client-side of the system are briefly introduced, which together achieve and keep the game operational, assuring consistency, synchronization, and playability across all game simulations. Additionally, since most implemented processes represent complex solutions to complex problems, they each have a separate session detailing their use and construction.

3.7.1 Server-side protocol

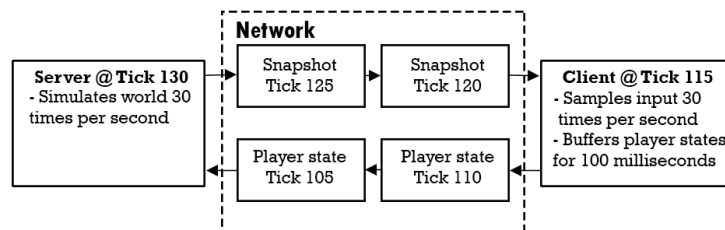


Figure 34: Server-Client message system overview, adapted from Valve® (2021).

During the race, both the clients and the server communicate with each other by sending small data packets at a high frequency, around 30 packets per second, depending on the tick rate. Using a custom logic timer developed, both reliably simulate the game in discrete time steps, called ticks. For each tick, there is an update of the simulation. By default, the time step is 33 ms, which correlates to 30.30 ticks simulated per second. However, this can be overwritten through the *TICKRATE* property in the *server.properties* file. The tick, or game tick, remains at zero until the race has begun. Considering packets take a certain amount of time to travel between the client and the server, it results in having the tick on the server a little bit ahead of the clients' tick since it is the server that starts the race by sending a *startGame* packet. Therefore, the server always deals with messages behind its time, as shown in Figure 34.

Although running at the same tick rate, simulations may fall out of sync. So, every time the tick is updated, the server sends a *serverTick* packet containing the current, correct tick to be applied on the client simulation, ensuring synchronization across all members. Moreover, using the game clock system mentioned in Section 3.5.4, the server sends a *serverClock* packet each second containing the game clock. So, in case that the client's game clock is wrong, it will accelerate or slow down to reach the correct value that was read from the server's packet. This strategy, although minor, provides a centralized, efficient and easy way to control all obstacles through the game clock. The need for the server to individually send an obstacle state to every obstacle object is also removed, hence lowering the average network load.

In each tick, the server: processes incoming packets, runs a physical simulation step, checks the game rules, and updates the game objects. The synchronization algorithm adopted by the server follows an optimistic

algorithm in its execution, which means the server will continuously update the game simulation, even before knowing if messages from all clients have arrived for the current game tick. This is because messages from clients can arrive late or drop on certain occasions, so it would heavily impact the system if the server had to wait for them, becoming even worse if one of the clients were to have high latency. This way, the server only updates the clients' state when a message has arrived, detecting inconsistencies and sending a *playerCorrection* packet when necessary. Apart from the game state updates and requests, the server remains silent in the absence of inconsistencies.

Moreover, the optimistic algorithm can cause a few problems, where the player state received in the server is not processed at the same game tick it was made. Therefore, the server's game simulation could be different, which is not fair for the client since it can lead to a different result from what it should be. Additionally, since *DummyGuys* basis on fast-paced player actions, even a delay of a few milliseconds can cause the message to arrive later than other clients, despite occurring at the same time. Therefore, low latency is a significant advantage but not always achievable. Both issues are an enormous problem for actions sensitive to the players' state, such as grabbing players, which directly affects the game's playability and fairness between players. Therefore, the server has also implemented lag compensation, which solves both problems.

3.7.2 Client-side protocol

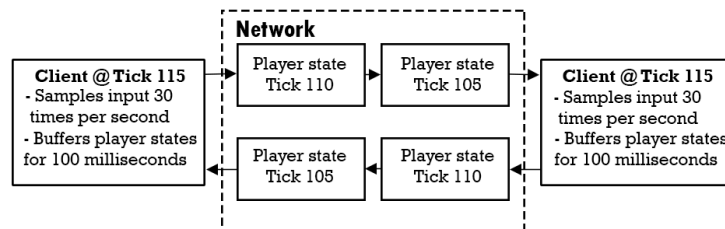


Figure 35: Client-Client message system overview, adapted from Valve® (2021).

Sharing the same logic, clients follow the game rules provided by the server, such as tick rate, updating the game simulation with a time step of 33ms. To inform all the client's peers and the server of their state, each client sends a *playerMovement* packet containing their player state every tick. Seeing that the game simulation advances 30 times per second, every client will have 30 new states. So, whenever something happens on the server, there will be a corresponding player state to each tick. Furthermore, these messages reach them after a certain amount of time, and since both clients and the server won't wait for all clients to send an update message, the game simulation may be ahead when it arrives. So, the client's peers will always deal with messages behind their time, as shown in Figure 35. To solve this, the client employs entity interpolation on the remote players to counteract the difference and ascertain good game playability.

In case a player state messages received from the client on the server represents misbehavior, they will receive a *playerCorrection* packet to execute the correction of the player state in their game instance. Since

these corrections may be in the past due to the travel time, the player may need to roll back their state to correct themselves. Moreover, since clients can crash at any given moment to let the server know of such events, each client on the server has a heartbeat mechanism, which also helps the server from allocating resources where they aren't needed. Similarly, each client also has an [away from keyboard \(AFK\)](#) timer to check if player messages are kept from being disseminated, threatening the system's synchronization.

Clients only control the state of their player object. Everything else, such as player respawn, is handled by the server. A client can only initiate the action by sending a request message to the server. In this case, a *playerRespawn* packet requesting their character's spawn back inside the map, which, in turn, the server validates and informs them of the outcome. Similarly, the server also controls actions such as grabbing and pushing players. This is because giving clients the authority to be the judge, jury, and executioner of events involving other clients would make them susceptible to cheating exploitation. Therefore, clients only make requests to the server by sending *playerGrab*, *playerPush*, *playerLetGo* packets and waiting for the outcome of the requested actions.

The absence of control and dependence can be felt by players when responses take a long time to arrive, creating a feeling of an unresponsive world towards the players. As such, the client implements a less dumb-client design, compared to a CS architecture design, employing client-side protocols to help make it resilient to the network-based communication problems, such as latency and dropping of messages. Additionally, to fake the client's involvement in server events, the game client fakes them. For instance, when a player passes a checkpoint, the server registers it as their new checkpoint, and in case of requesting respawn, they will spawn there. In contrast, the client does not take part in the process. It only provides player states that permit the server to deduct if the client has passed through the checkpoint. Therefore, the user is clueless if they went through a checkpoint or not. To solve this, the game client provides visual aid, such as particle effects, to seem like something happened on the client-side of things.

3.7.3 UDP Pong - Protocol Design

Since UDP is connectionless, to maintain a connection between the client and the server, the system has set up a ping mechanism, which begins when a client joins a room. This mechanism consists of a continuous exchange of messages between them, and it is implemented on the client's application *Client* and server's application *Client*, respectively. This system works by having the client send a *ping* packet to the server each second, storing the current time (*pingTimeSent* shown in), and waiting for the packet to be echoed back. Seeing that machines can be in different time zones, the current time is captured in the system using *UtcNow*¹⁶, which expresses the [Coordinated Universal Time \(UTC\)](#).

¹⁶ <https://docs.microsoft.com/en-us/dotnet/api/system.datetime.utcnow?view=net-6.0>

```

public static void Ping()
{
    using (Packet _packet = new Packet((int)ClientPackets.ping))
    {
        _packet.Add(ClientInfo.instance.ClientRoomId);
        SendUDPData(_packet);
    }
    Client.instance.pingSent = DateTime.UtcNow;
}

```

Figure 36: Ping message.

The ping message is received on the server through the UDP socket corresponding to the client, and a handler on *RoomHandle* is invoked, which in this case is *Ping* (shown in Figure 36). This method prompts the room to send a *pong* packet to the client using the UDP socket. When the client receives the server's ping response (*pong* packet), the *Pong* handler on *ClientHandle* is issued to treat the message (shown in Figure 37). This method calculates the client's ping from the difference between the current time and the time in which the client sent a *ping* packet to the server (*pingTimeSent*), resulting in the time, in milliseconds, that it takes for a message to reach the server and come back to the client, called ping or *round trip time (RTT)*.

```

public static void Pong(int not_needed, Packet _packet)
{
    Client.instance.ping = Math.Round((DateTime.UtcNow
        - Client.instance.pingSent).TotalMilliseconds, 0);
    GetAnalytics(_packet);
}

```

Figure 37: Ping calculation.

In addition, this ping/pong system allowed for a heartbeat mechanism in every client inside the room, shown in Figure 38. This system consists of a periodic signal used to monitor the connection between a manager (server) and an agent (client), helping the server automate the cleanup process when a connection between them is lost. Since the ping/pong system already does that, the heartbeat mechanism was built on top of it. It works by having a timer set to 0 every time a message arrives. If the client stops sending packets, this timer will eventually reach the timeout value, set to 10 seconds by default, which prompts the server to assume the client has crashed or lost connection. Therefore, the server disconnects the client and informs the rest of the clients by sending a *playerLeft* packet. Additionally, if a client disconnects during a game session, the server removes their character from the game world.

```

_heartbeat += Time.deltaTime;
if (_heartbeat > HEARTBEAT_TIMEOUT)
{
    // Disconnect player
    Server.Rooms[roomId].RemovePlayer(Id);
    Deactivate();
}

```

Figure 38: Heartbeat timer.

3.7.4 Tick based system

As aforementioned, for the game to progress, it needs to implement equations of motion using a numerical integrator, where everything is tied to the tick. Having a numerical identifier for each update allows for the synchronization of multiple game instances. This is because the system has an attainable value related to the game state it is on, which helps enforce the same state across all game instances in the network during the entirety of the game's duration. Despite being important in syncing the state, the game tick is also crucial in identifying when a determined action occurred and allows for the integration of supplementary mechanisms to help the game keep consistency. These being: client reconciliation, entity interpolation, lag compensation, and client monitoring.

The logic behind the tick system, shown in [Figure 39](#), is implemented on the server's *MapController*. The client application follows the same basis with a few logical modifications due to their different roles in the system. For example, the client does not use *SendServerTick* method, which sends a *serverTick* packet to every client in the room. Only the server has this role in the system.

```

private void FixedTime()
{
    if (!isRunning)
    {
        gameLogic.SetTick(0);
        return;
    }

    physicsScene.Simulate(logicTimer.FixedDeltaTime);

    SendServerTick();
    lagCompensation.UpdatePlayerRecords();
    gameLogic.SetTick(gameLogic.Tick + 1);
}

```

Figure 39: Game Update.

Game Tick

The tick is contained in *GameLogic* class and is updated every loop using *SetTick(tick + 1)*, shown in Figure 39. The game tick is updated at the end of *Update* after the code needed for the current game update has been executed. Then, the new tick is used to identify the next game simulation. Furthermore, the tick is only updated if the game race has started (*IsRunning=true*) since it is not meant to progress without the game. Additionally, the update loop manually updates the scene physics simulation. This is accomplished through *physicsScene.Simulate(...)*, which only targets the room's scene, which is defined as *physicsScene*, leaving the rest of the scenes, not owned to it, untouched. Typically, the physics simulation is automatic. However, to ensure that the physics do not perform without the tick, it was practical to put it on a manual step, which permits a more controlled update of the physics simulation towards the game.

Logic Timer

Implementing a game tick can be a trivial process, but there are many ways to update it. The simplest way is updating the tick with delta time, like 1/60th of a second. However, the update rate would only match clients with 60 Hz monitors. A better option is to use fixed delta time, which using the render framerate, makes the simulation behave the same way from one execution to the next, without any potential for different behaviors. Nevertheless, having physics tied to the render framerate is also a massive limitation and not always practical. For instance, if VSync were off, the rate would fluctuate, and since the game requires messages for each frame, it would, naturally, happen to create a lot of them if it reached a high frame rate.

The correct solution is having the best of both. This means the update method would have the ability to render at different frame rates and a fixed delta time value for the physics simulation. *Unity* offers this solution, having the *Update* loop, which uses the frame rate, and another specifically for the physics update called *FixedUpdate*¹⁷, which uses a fixed delta time. However, *FixedUpdate* is unstable due to depending on the *Update* loop, which is susceptible to frame drops.

The solution found to be appropriate was to implement a custom loop that allows for a more predictable behavior of the physics simulation. To function correctly, every object dependent on the physics behavior, such as players, must implement it. Otherwise, they would follow a different update rate that even if events were to occur simultaneously, they would have different ticks, resulting in inconsistency problems.

¹⁷ <https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html>

```

public void Update()
{
    FixedDeltaTime = Utils.TickInterval();
    long elapsedTicks = _stopwatch.ElapsedTicks;
    _accumulator += (double)(elapsedTicks - _lastTime) / Stopwatch.Frequency;
    _lastTime = elapsedTicks;

    while (_accumulator >= FixedDeltaTime)
    {
        _action();
        _accumulator -= FixedDeltaTime;
    }
}

```

Figure 40: Logic Timer.

Figure 40 shows the main logic of the custom loop implemented, called *LogicTimer*. This loop takes a given action, which in this case is the update function of an object, and after a given time, as passed, it executes its contents on *Unity*'s main thread. To know if an update is due, the custom loop continuously adds elapsed ticks to the *_accumulator* since the last loop. Additionally, it checks if the fixed delta time has passed, which corresponds to the ticks that must execute in one second (tick rate). When this is true, it means an update of the physics simulation is due, prompting the execution of the *_action* and rewinding the *_accumulator* a step back for the next update. The *_accumulator* is not set to zero because it can have some unsimulated time left over when the update was executed. This way, the time left over in the *_accumulator* is passed to the next loop, which considers as time already passed.

Tick rate

The tick rate is essentially the server's equivalent of a client's frame rate, absent from the rendering system, used by *LogicTimer* to know at what rate to update the game simulation. By default, the system's tick rate is set to 30 (ticks per second). This value was selected considering the number of messages each tick and the precision needed for the game simulation.

Although a higher tick rate increases the simulation precision, it eventually becomes futile to have such accuracy for this game. In relation with an FPS game, such as *CS:GO*, that uses a tick rate of 66, it requires a certain precision to determine if a shot has hit a player in the far distance. However, it is not the case in this particular game. Players are big capsules and can only grab or push each other when in direct contact, which gives a large margin of error. Additionally, considering the number of messages created per tick and the resources needed to send it and process it, it was chosen to set the tick rate to 30. A considerably smaller value than *CS:GO*'s tick rate, but sufficient during development and analysis.

3.7.5 Entity Interpolation

By default, each client receives around 30 player state messages per second from each of their peers. If the clients were to update and render the characters of the remote players upon receiving a new player state, the movement of these objects would look jittery. This is because, even though clients are receiving the data as fast as possible, the network makes no guarantees that the packets sent arrive nicely spaced apart from each other, much less in the same order they were sent in. Thus, the system has implemented entity interpolation, which essentially interpolates the player's object at a given delay. By going back in rendering time, these objects take longer to reach the intended state, allowing for new messages to arrive and for the object's state to be continuously interpolated between the two latest received states. For example, with 30 player state messages sent every second, a new player state will arrive around every 33 milliseconds. So by shifting back the render time by 33 milliseconds, the entities on the game client will always be able to be interpolated between the last received state and the state before that.

```
public class PlayerState
{
    public int tick;
    public Vector3 position;
    public Quaternion rotation;
    public bool ragdoll;
    public int animation; // Animations: 0 - idle, 1 - running, 2 - jumping, 3 - diving
}
```

Figure 41: PlayerState Class.

The entity interpolation is implemented on the *Interpolator* class and used by the *RemotePlayer*, previously shown in Figure 32. Additionally, only the client application uses this mechanism. To achieve the intended capabilities, the *Interpolator* buffers every player state received into a list. These player states contain the values shown in Figure 41, which upon arrival are translated into a *PlayerState*. Since packets can be jittered due to the network, clients may receive two packets on some frames and none in others, which ultimately causes the received packets to be out-of-order. As result, when buffering a new player state, the list of received player states is iterated through, using the new player state tick as the index until its ordered place in the list is found.

Figure 42 shows the update loop of the entity interpolation, that since it does not rely on physics like the local player, uses *Unity's Update* loop. In each update loop, the *Interpolator* iterates every player state in the list, checking if the game tick is equal or in front of it. If this is true, it signifies it is time for the player state to be applied to the remote player's character. To interpolate the player state, the *Interpolator* uses *from*, which is the state the player is currently on, and *to*, which is the state where the player needs to transition, shown in Figure 42. For this to work, the mechanism needs to correctly calculate the time interval for the transition between these states.

As aforementioned, the communication between two nodes in the network ends in dealing with packets containing information from the past. Considering the packet travel time and potential loss, the mechanism's default interpolation period *lerp* is 100 milliseconds (*lerpPeriod=0.1f*). This way, even if a packet is lost or delayed, there

```

for (int i = 0; i < futureTransformUpdates.Count; i++)
{
    if (MapController.instance.gameLogic.Tick >= futureTransformUpdates[i].tick)
    {
        previous = to;
        to = futureTransformUpdates[i];
        from = new PlayerState(MapController.instance.gameLogic.InterpolationTick,
            transform.position, transform.rotation, remotePlayer.Ragdolled, remotePlayer.currentAnimation);
        futureTransformUpdates.RemoveAt(i);
        timeElapsed = 0;
        timeToReachTarget = ((to.tick - from.tick) * MapController.instance.gameLogic.SecPerTick) + lerpPeriod;

        // Ragdoll state, if not in the correct ragdoll state then transition to it
        if (from.ragdoll != to.ragdoll)
            remotePlayer.SetRagdoll(to.ragdoll);

        // Animations
        remotePlayer.SetAnimation(to.animation);
    }
}

timeElapsed += Time.deltaTime;
Interpolate(timeElapsed / timeToReachTarget);

```

Figure 42: *Interpolator's* update loop.

are always two other valid player states to interpolate between, keeping a continuous and stable interpolation to the naked eye. Nevertheless, if more than two states in a row are lost, resulting in at least 99 milliseconds of silence, the interpolation won't, possibly, work. This is because it runs out of buffered states until the next one arrives at the 129 milliseconds mark. Therefore, if the player state does not arrive, it will result in a jump of a player state, which in some cases can create some strange behaviors in the movement of the player. This is because, the *Interpolator* will use a simple, and linear extrapolation instead of interpolation. Interpolation is done by adding points between data to smooth it, while extrapolation is done by assuming the values based on data.

When interpolating between states, the *Interpolator* calculates the lerp amount based on the time period needed to reach the desired target state, as shown in the following equation.

$$timeToReachTarget = ((to.tick - from.tick) * secPerTick) + lerpPeriod$$

This translates that to calculate the lerp amount, the system needs to get the difference between the ticks where the entity needs to be (*to* state) and where the entity is (*from* state), converting them into the seconds using $1/tickrate$. After having the time interval to transition between states, it is added the *lerpPeriod* to account for the problems mentioned before.

Apart from the calculation of position and rotation, the character's animated state and ragdoll state may also need updating. This is accomplished by comparing the current animation and ragdoll state against the ones on the new player state. If an update is due, the *SetAnimation* and *SetRagdoll* methods, in the *RemotePlayer*, are called.

Moreover, this mechanism is highly sensitive to the game tick, as without regulation from the server, the client's tick would completely fall out of sync and break the mechanism. For instance, if Client 1 is behind the server's

simulation and Client 2 is ahead, if Client 1 sends a player state, then Client 2, being ahead of the server's game tick, would not consider any of these old states from the Client 1.

3.7.6 Player synchronization

The player synchronization tends to be one of the most complex tasks in most multiplayer games since it is responsible for the correct representation of the player in the game world of all networked instances, controlling its information, movement behavior, actions, and correction. Before going into its explanation, there is to note that this mechanism is implemented in the player's object, more specifically in *Player*, already discussed in [Section 3.5.4](#).

Replication

The player replication model implemented is based on a decentralized design with server monitoring. This algorithm sends each tick a player state message to every node in the network, through Multicast, including the server. This message is the same to both the server and the clients because the algorithm basis on the client application having complete control over their player state inside the game world. However, this poses a threat to the system's security since users can cheat through their player state. Therefore, the server monitors these player states.

The replication process starts on the client's application *Player*, shown in [Figure 43](#), where *PlayerInputs* captures the user's inputs for every tick (*currentInputState*), and *ProcessInput* shares them with *PlayerController* to be processed, updating the current player state. Concluding the calculation of the new state, the client sends a *playerMovement* packet to everyone, containing the tick and player state. Additionally, the inputs and player state are also buffered in simple circular arrays (*simStateCache* and *inputStateCache*) using the tick as the index, calculated through $(\text{tick} \% \text{CACHE_SIZE})$. Since the physics tick rate is 30, the buffers will have 1024 elements, giving the system around 34 seconds of space, which is considerably more than it needs. Moreover, these are stored because they will be needed for reconciliation when a the server detects a misbehavior, as explained ahead.

The client's peers use this message to update the remote player object while the server uses it to monitor the client. The server asserts authority over the client's player state by checking its contents' validity and sending a *playerCorrection* packet if detected problems. This validation is focused on a cheater axis and consists of checking the tick and the player position. First, the server checks if the game tick on the message does not surpass the server's game tick, which would indicate that the client is tampering with their game's tick rate to update their simulation faster than other players for its character to react faster. Secondly, the server checks if the player position in the game world does not clip through any static object, such as walls or floor. This is done using *CapsuleCast*¹⁸, which casts a capsule (the actual physical object of the player) against the scene objects on the received position and rotation. If the player is found to be inside the scene objects, it indicates the client may be tampering with the game's collision objects or player state positions. These cheating processes represent

¹⁸ <https://docs.unity3d.com/ScriptReference/Physics.CapsuleCast.html>

```

private void FixedTime()
{
    if (Running)
    {
        // Update player's movement
        ProcessInput(currentInputState);

        if (Online)
            SendMovement();

        // Reconciliate
        if (serverSimulationState != null) Reconcile();

        // Determine current simulationState
        SimulationState simulationState = new SimulationState(currentInputState.SimulationFrame,
            transform.position, transform.rotation, velocity, angularVelocity, Ragdolled);

        int cacheSlot = simulationTick % CACHE_SIZE;

        // Store the SimulationState into the simulationStateCache
        simStateCache[cacheSlot] = simulationState;

        // Store the ClientInputState into the inputStateCache
        inputStateCache[cacheSlot] = currentInputState;

        // Move next frame
        ++simulationTick;
    }
}

```

Figure 43: Player update on the server.

two common ways that players tend to alter the game's simulation. Although it only runs two verifications, this system allows for further expansion to encompass more problems, which the player state may be susceptible in the system.

If the player state is within parameters, it is applied to the character. Furthermore, player state messages can arrive shuffled at the server. For instance, if a player state with tick 17 arrived at the server and is declared correct, later all other messages containing a player state behind tick 17 are also considered correct. Therefore, the server does not consider checking any other player state messages which occurred before tick 17.

Reconciliation

The reconciliation process begins when the client receives a *playerCorrection* packet due to the server detecting misbehavior. This message is stored in *Player* as *serverSimulationState*, which only considers the latest correction issued by the server. Therefore, if *serverSimulationState* has a stored correction with tick 17 and a new correction state is received with a lower tick, it is discarded. This is because there is no point in making corrections behind other corrections.

In each loop of the Player's update method, the *serverSimulationState* is checked to see if a player correction has arrived, as shown in Figure 44. If one is found, the *Reconcile* method is called, initiating the reconciliation

```

public void FixedTime()
{
    if (Running)
    {
        currentPlayerState = null;

        // Obtain CharacterInputState's from the queue.
        while (playerStates.Count > 0 && (currentPlayerState = playerStates.Dequeue()) != null)
        {
            _inputThisFrame = true;

            // Player is sending simulation frames that are in the past, dont process them
            if (currentPlayerState.tick <= lastFrame)
                continue;

            lastFrame = currentPlayerState.tick;

            // Check if the state received is inside certain params
            CheckSimulationState();
        }
    }
}

```

Figure 44: Player update on the server.

process. Once started, *Reconcile* checks if the correction state is in a tick ahead of the last corrected state, or else, it would be futile to correct it. Afterward, it searches for the buffered inputs and player states, saved during replication, with the correction state tick as the index, calculated through $(\text{tick} \% \text{CACHE_SIZE})$. In case that cache data is missing, for either input or player state, the current player state is immediately snapped to the correction state, but maintains the current tick. Additionally, the last corrected state is also updated, ending the reconciliation process and continuing the game simulation normally.

If the inputs and simulation state are in cache, the player object is still snapped to the correction state as before, but its tick is also set to the correction tick. However, since this correction is in the past, for instance, on tick 20, and the client's simulation is on tick 25, the *Reconcile* has to simulate all player states until it reaches tick 25 in order for the player object to match the current simulation. Having all the necessary stored inputs and states, this process is easily executed, shown in [Section 3.7.6](#).

```
// Loop through and apply cached inputs until it has reached the current simulation tick
while (rewindTick < simulationTick)
{
    // Determine the cache index
    int rewindCacheIndex = rewindTick % CACHE_SIZE;

    // Obtain the cached input and simulation states.
    ClientInputState rewindCachedInputState = inputStateCache[rewindCacheIndex];
    SimulationState rewindCachedSimulationState = simStateCache[rewindCacheIndex];

    // If there's no state to simulate, for whatever reason,
    // increment the rewindFrame and continue.
    if (rewindCachedInputState == null || rewindCachedSimulationState == null)
    {
        ++rewindTick;
        continue;
    }

    // Process the cached inputs. Simulate movement (and related), and physics.
    ProcessInput(rewindCachedInputState);

    // Replace the simulationStateCache index with the new value.
    SimulationState rewoundSimulationState = new SimulationState(rewindTick,
        transform.position, transform.rotation, velocity, angularVelocity, Ragdolled);
    simStateCache[rewindCacheIndex] = rewoundSimulationState;

    // Increase the amount of ticks rewinded.
    ++rewindTick;
}
```

Figure 45: Player Reconciliation.

The *rewindTick* corresponds to the player tick while the *simulationTick* corresponds to the simulation tick and where the client needs to go, which the method tries to reach by continuously updating the player. Inside the *while*, the inputs and state corresponding to the player's current tick are searched inside the arrays. Using the inputs, the client calculates the player state for that given tick and substitutes the player state buffered in the array. This is because, now that the player object is set to the correct and different player state, all the next player states will, potentially, be different. Therefore, the stored states (in the arrays) are substituted by the new ones. After this, the *rewindTick* is updated, signifying that the player has moved a simulation frame, and the process is repeated until the player finally reaches where the simulation currently is (*simulationTick*).

This correction of different positions and rotations on the player object might be noticeable to the user, appearing glitchy and jittery in the game world. Because of this, the player object was constructed in a way to avoid the user from viewing it on certain occasions. The player object is comprised of two main objects: the actual physics object of the player, which is invisible, and the visual object with the model of the player. When a correction is due, the physics object is immediately corrected to the correct state, but since the model object is only used for visual purposes, it is smoothed into the correct state, allowing for the best of both. However, there are exceptions, where if the correct position or rotation value is too distant (two meters in this game's case) from what the player

is currently on, the player model is also snapped to it. Otherwise, it would also look strange to see the player model slide into a new and distant position/rotation without the client's inputs to justify it.

Alternative

Another approach for the replication system would be following a more centralized and server authoritative design. This consists of the server mirroring the entire player state of the client in its game simulation to assess if the state on the client is correct. This requires the client to send two different messages: one for the server, which contains the inputs, the resulting player state (from the inputs), and the tick, and another for the clients, which has only the player state and tick. Further than this, the algorithms would be the same, storing inputs and player states for reconciliation.

Both algorithms advance their simulation independently from the server, meaning the actions of the local player take place immediately in the simulation, removing the problem of the game feeling unresponsiveness or laggy. Additionally, the server can still achieve consistency across the network by checking the client's states as they arrive. However, the alternative approach needs the client's player state to be identical on the server, or else, the client receives a correction. The implemented system follows a looser approach, verifying only two aspects focused on cheating. These two algorithms show a trade-off of more grip over the player state to a more open and less resource-intensive approach.

It was chosen not to implement this alternative approach because the number of resources needed to function is higher. This is due to each received player state requiring the server to process its inputs, simulate physics, and compare the resulting player state with the client. Additionally, the player movement and interactions are incredibly physics-based, which is never good when distributing games over the network. Collisions between players are dependent on information from both clients' simulations, which sends player states. This causes them to collide with each other at different times due to packet travel time and entity interpolation applied on the remote player. Since the alternative approach relies on the server's physics simulation to be identical to the client's simulation, it creates a problem where the collision on the client is irrefutably different from the one on the server's, especially considering that the server does not use an entity interpolation on any player object. All of this makes the player replication extremely hard to maintain (from the server's perception) in the alternative approach because it will lead to a lot of correction messages issued.

Due to this, the current implementation of player replication was chosen to stay, relying mainly on the client's resources for the player's calculations. Ultimately, this is considered an enormous amount of load (CPU and bandwidth) taken of the server, especially considering the number of players connected per room. Moreover, this load removal is seen in the difference of complexity between the *Player* classes on the client and the server, shown in [Figure 31](#). Nevertheless, the server still checks for potential cheating from the clients and allows room for improvement. Lastly, this system continues to allow the server to be a mediator in the system for events involving player-on-player interactions, controlling the outcome.

3.7.7 Lag Compensation

The following mechanism is implemented in *LagCompensation* and is only present on the server. This mechanism allows to backtrack the player's characters in order to provide fairness in actions sensitive to position and time, such as grabbing or pushing players. However, actions are only allowed to be backtracked if they occurred in the near past.

This mechanism works by storing a list of records corresponding to the player's former and current states, each represented as a *PlayerRecord*, and in a dictionary. This dictionary is initiated after every player has been instantiated in the game world, to hold all current players. To keep the records updated, the mechanism is called on *MapController*'s update, previously shown in [Figure 39](#), to update the current player states for each tick of the simulation tick.

Once called, *LagCompensation* goes through every player in their dictionary, checking if they are still in *MapController*'s player dictionary because clients could happen to be disconnected and removed from the game. If the player is in the game, their current state is added to their respective list of records. Additionally, *LagCompensation* goes through every record inside their list to confirm if all states contained in it occurred during the last second. This is because the lag compensation mechanism saves records only for a short period of time, else it would require a lot of memory. It only stores records for one second because it is the maximum time set that lag compensation is allowed to backtrack a player. *LagCompensation* knows to eliminate player records if the difference between their tick and the game is higher than one. There is to note that the maximum time allowed can be changed. However, a higher allowed time translates into more memory and CPU being used due to more actions needing to be backtracked, which results in more records being stored and checked.

The other part of the lag compensation is to use these records to backtrack the players when the room receives a player-on-player interaction, such as *playerPush* and *playerGrab* packets, which are passed down to *MapController* to be handled by *LagCompensation*. Actions like *playerLetGo* packets are not sensitive to position, so the server only checks if the player is grabbing someone and, if valid, accepts their request. The process of calculating and setting the players' positions is called backtracking and consists of four *LagCompensation* methods, being the following: *Backtrack*, *Backup*, *BacktrackPlayer*, and *Restore*.

- *Backtrack* initiates the process of backtracking players, given the player id and tick to where it needs to transition. It checks if the player exists in the game world and if the action happened a second ago. If this is valid, it calls for *Backup* and *BacktrackPlayer* methods for every player in the simulation.
- *Backup* adds a player's current state to the backup records' dictionary, which is used to store the players' states before being backtracked.
- *BacktrackPlayer* backtracks a player to the intended state, relative to the tick that the action took place. To achieve this, it gets the player state from their list of records. If it does not find it, it tries to get the state closest to the intended tick by looping through their records again and finding the record with the smaller tick difference. If a record still hasn't been found, the mechanism gives up on backtracking that player and moves on to the next one to do the same process until all players have been checked. If

the mechanism does manage to find a suitable record, the player character is interpolated to the state it was at that given tick. This calculation is similar to the one on the entity interpolation system and takes into account the interpolation delay added that a local player adds to the remote player on their client simulation by applying 0.1 seconds to their opponents' state (due to entity interpolation). Otherwise, the player on the server would not be interpolated to the same place as it was in the client's simulation. This method completes the backtracking of every player involved, allowing the *MapController* to execute the verifications needed to validate grab or push actions.

- *Restore* sets the player's character back to their former state (before being backtracked) through the backup records' dictionary. This finishes the lag compensation process and allows for the correct continuation of the game simulation on the server.

ANALYSIS AND DISCUSSION

4.1 ANALYSIS

This section describes the experiments executed with the proposed solution in order to evaluate the game's server behavior and non functional aspects, namely the performance, scalability, and security. Moreover, the study extends to the distributed game's perceived resistance to the network communication problems.

Realistic setups for these system types have characteristics similar to [online transaction processing \(OLTP\)](#), where the input is external to the application triggered by the client's system, so it doesn't match most traditional benchmarking methodologies used. Most information and events are generated by human users, making the system's state and behavior highly dependent on them. Therefore, a large-scale research on these applications involves hundreds or even thousands of clients, exceeding most university-level laboratory setups. Due to these factors, it becomes difficult to automate the benchmarking process and compare results across runs. To deal with these issues, the experimental setup and experiments run are explained in the next section.

4.1.1 *Experiments And Experimental platform*

The assessment of performance and scalability involves multiple clients to examine and understand the system's behavior under load. The scalability assessment presents an even large number of clients to determine the system's limits. The work is conducted on a cluster of 131 servers, Intel(R) Core(TM) i3-4170 3.70GHz, with the Ubuntu 20.04 LTS (YAML) operating system and connected to a private 1000Mb/s Ethernet network. The experiment, uses only four nodes, where one is dedicated to the DummyGuys' server (*GameServer*), and the rest is used as a client cluster (multiple *GameClient* in each machine). Measurements are performed by instrumenting the code and alterations of the *server.properties* file, configuring the server as needed for the experiments. For processes created in the server and their usage, the *top* tool is used, which provides a dynamic real-time view of the measurements. For the measurements of network traffic, bandwidth usage, etc., the *iftop* and *vnStat* monitoring tools are used.

It is found that in all experiment cases (related to performance and scalability), a few seconds of execution time is enough to capture the behavior of the game server. So, each experiment had a duration of one minute and is executed multiple times to verify their consistency. Also, the initialization time for the runs is excluded. The

results portrayed in the following graphics solely represent the values captured during in-game. Nevertheless, information about other states is discussed when relevant. Lastly, these experiments use bots to populate the server and estimate a load proportional to actual players playing the game.

For the assessment of security, the system's architecture and implementation are explained, revealing the natural level of security it provides. Furthermore, it is laid out where the system provides security to the clients and game state authority, while also mentioning potential risks.

For the assessment of the distributed game resilience, the system is submitted to networking constraints, resembling a real gaming scenario and some extreme case scenarios. These aspects are latency, packet loss, out-of-order packets, and packet jitter. The experiments are conducted on a personal computer, Intel(R) Core(TM) i7-6700HQ 2.60GHz, connected to a private 100Mb/s Ethernet network. Both server and clients are executed on the same machine. This experiment only needs two players to test all game protocols. Measurements are obtained with Clumsy ¹, a third-party tool, which works by selecting inbound or outbound packets to capture. This selection is given by a filter, predefined by the program or inserted by the user, composed of the port, IP address, or a combination of the two, and available in TCP or UDP sockets. Using this filter, the program only affects the information contained in it, leaving the rest untouched.

Each run of the experiments lasts a different time period, since it is necessary to play the game and assess certain events under the conditions to obtain the results. These experimental runs are also executed multiple times to verify the consistency of the results.

4.1.2 Performance

Performance metrics: Although the interest is mainly on the game server, the game client is occasionally monitored to help assess the server measurements. On the server it is studied, the number of incoming and outgoing messages to determine bandwidth requirements of the system, the request processing time to show how computationally intensive the game is, the impact of tick rate, the number of client requests received and processed in each server time-slot, etc. On the client it is studied, the incoming and outgoing bandwidth requirements (for verifying the server's measurements), the request rate to the server, and the response time from the server.

¹ <https://jagt.github.io/clumsy/>

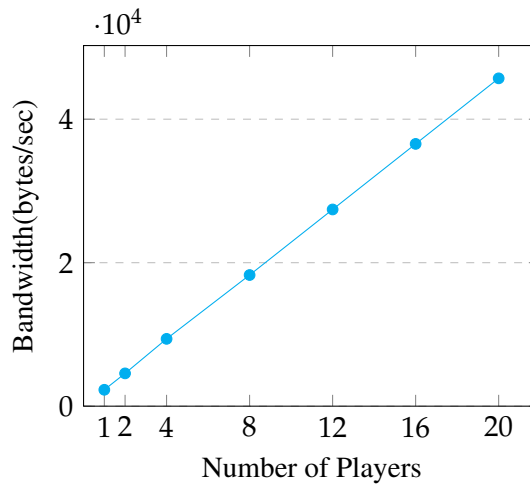


Figure 46: Server incoming network throughput per client. Dots represent the average for each run.

Bandwidth Requirements: Figure 46 shows the per player incoming network throughput in the server. It is seen that on average, each player sends about 2,285 Bytes/sec. Moreover, this throughput on the server is dependent of the number of players. This is due to the more players in the network, which results in more messages/requests being sent to the server. Additionally, the incoming packet size does not vary with the number of players, and therefore it is not plotted.

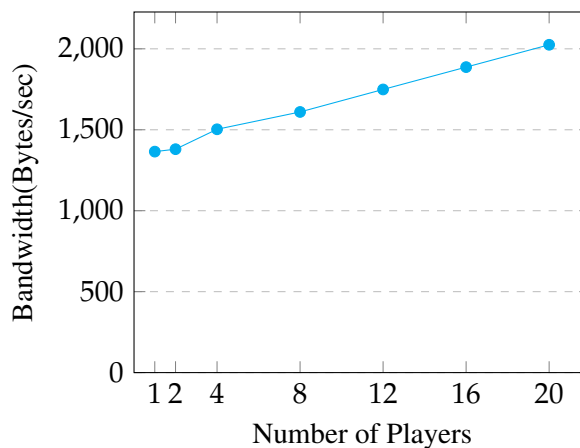


Figure 47: Server outgoing network throughput per client. Dots represent the average for each run.

Figure 47 shows the per client outgoing network throughput at the server for different numbers of total players. Although small, it is noticeable that the average throughput increases as more players are introduced to the system. Since the outgoing packet size does not vary with the number of players, the increase of outgoing throughput of the server can only be justified by more messages being sent each second. Figure 48 shows the average number of responses sent by the server. It shows a linear growth of messages, indicating the same conclusion. This increase is the result of more players interacting, meaning more corrections and responses to their requests. Furthermore, with every client comes more UDP (and TCP) connections constantly sending ping

packets, increasing the number of messages. Lastly, brief bursts of packets are inevitable while in a room that is waiting for clients to join (not shown). These events correlate to the server sending packets to the newly joined clients about all clients already in the room. As more join, the bigger the bursts are.

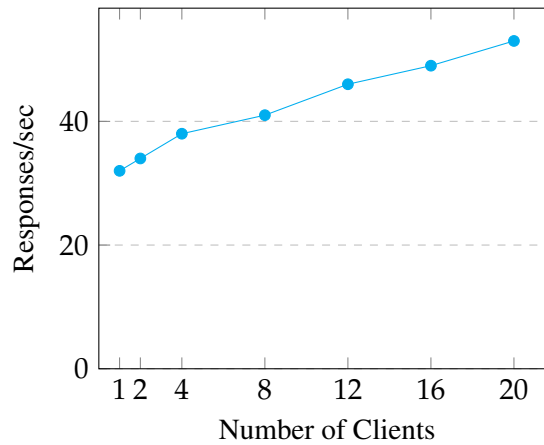


Figure 48: Server packets sent per second.

The minimum packet size sent is 8 bytes, corresponding to the *pong* packet from the server, used in the ping-pong mechanism. The maximum packet size contains 49 bytes, which corresponds to the *playerCorrection* packet sent by the server to correct a player when it detects a misbehavior in the player state. Moreover, while the server has to deal with much more information and share various events, the client has fewer messages that on average, are smaller in comparison with the server's. Nevertheless, the range between the minimum and maximum packet size of the client is similar to the server's. This is because of the *playerMovement* packet that, since the player replication is based on the client's application, forces them to send all information needed for the whole replication process, increasing the client's requirements.

In conclusion, the results show that the network throughput is not a problem, staying relatively small, and that a single 1 Mbps Ethernet connection could support hundreds of players. However, the same cannot be said about the incoming network throughput that with 20 players, reaches around 46 KB/sec, where 98% is coming from multicast. Furthermore, due to the use of multicast, a client receives roughly the same amount of data as the server, which can be especially impactful to the clients since these tend to have limited resources. Nevertheless, data suggests that a user with 1 Mbps Ethernet connection could still run the game with up to 30 players. However, when trying to scale beyond that, it may be required more resources.

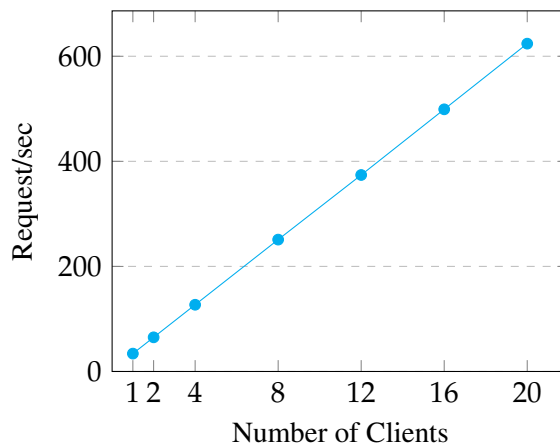


Figure 49: Server incoming request rate (request/s) per client.

Server Request and Response Rate: Figure 49 shows the average number of requests received by the server. It is noticeable that, as expected, the number of requests per second increases linearly with the number of players. Figure 50 shows the number of requests that are processed per second at the server. The average number of requests/sec processed increases linearly with the number of players indicating that the server is not saturated. Due to this, it can be concluded that the scalability of the server is limited, as it will eventually become saturated with messages.

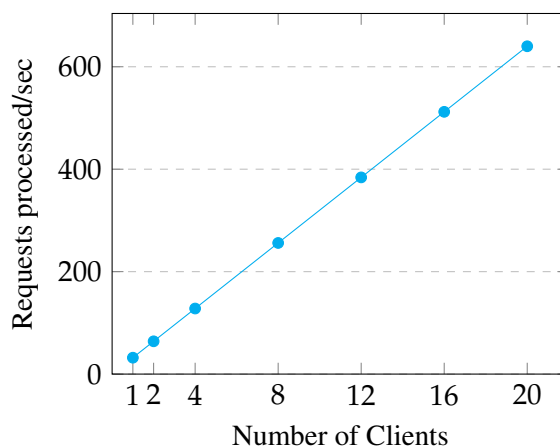


Figure 50: Server request process rate per client.

Client Received Response rate: Figure 51 shows the results from the server response rate measured at the clients using the ping mechanism on the client. Its value includes the server's processing time plus the transmission time. The average response time (from Figure 51) is 33-34 ms, indicating that clients receive server replies at a fast pace. Moreover, results show that response times may present a high variation with maximum values in the range of 67 ms and minimum values of 16 ms. Still, the minimum and maximum values remain constant throughout the player increase. These results are explained by the occasional loss of messages or multiple *pong* messages received at the same second. Additionally, the ping tends to spike momentarily at 200

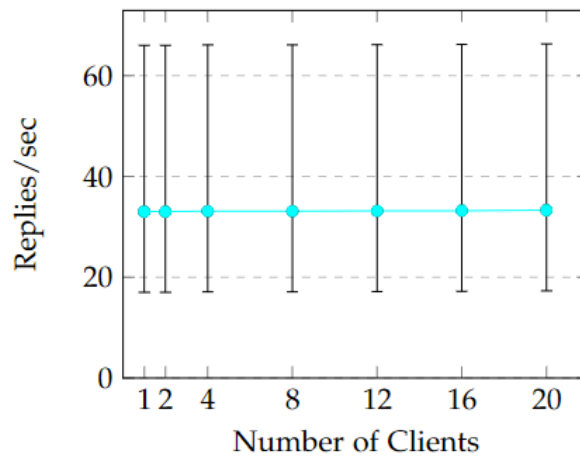


Figure 51: Server response rate (replies/s) at the client application.

ms when the server or client loads the game scene. However, this depends on the number of clients that the client cluster machine has, due to having to instantiate a scene per client, which impacts the machine.

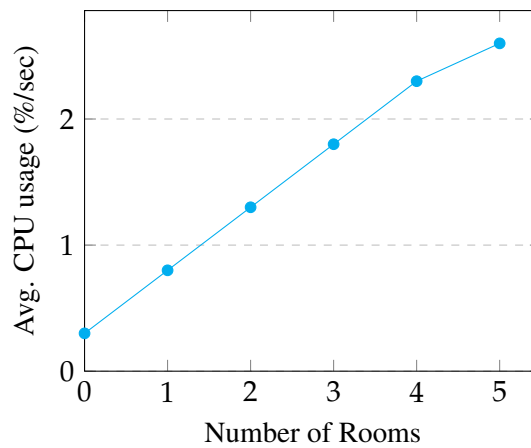


Figure 52: Server average processor usage per room.

Server Load: Figure 52 shows the measurements from the CPU usage of rooms in the server. These rooms remain empty and active with no clients inside it, showing the processor utilization needed just to update the game world. In order to obtain these results, the dormant optimization was deactivated, making the rooms generate and run the game world when initiated. The figure indicates a low increase of resource consumption. Furthermore, Figure 52 provides a frame of reference to the server’s load without an active room, showing that the processor’s usage offset that the application uses just by running is 0.3%.

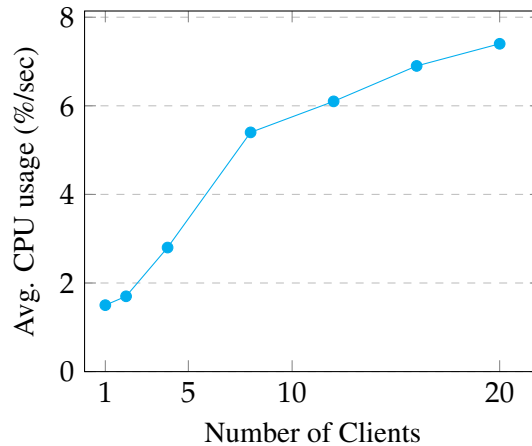


Figure 53: Server average processor usage per client.

Figure 53 shows the average CPU usage of players in the server. In conjunction with Figure 52, the overall results indicate that apart from the clients messages/requests, a room has a small number of tasks to perform each loop, having a low resource consumption. Moreover, since the server is a console application it does not have to render objects, generate lighting, effects or graphics, reducing even more the resources used. Since most of the room events are connected to the clients, it is expected that they bring a high load to the server's processor, specially because most of the information being sent in the system is player states and the server has to validate them. Lastly, comparing the resource usage from the player object in the client application and in the server (not shown) indicates that the chosen algorithm for player replication removed a big percentage of workload of the server, which was part of the reason for its implementation.

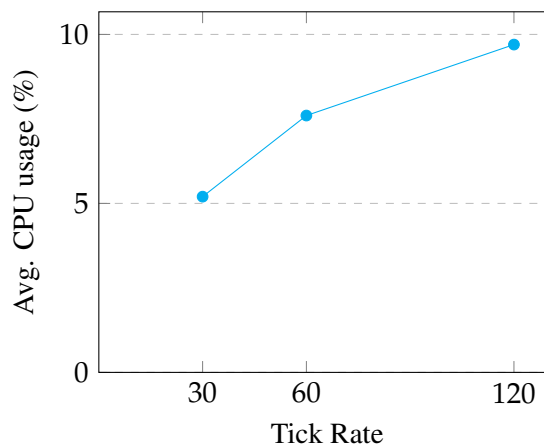


Figure 54: Server average processor usage by tick rate.

Server Tick Rate: Figure 54 shows the average processor utilization in the server depending on the tick rate and having a single room with 10 players. As expected, there is an increase of CPU usage with a higher tick rate, specially, with 120 ticks/sec that uses almost double the CPU power of 30 ticks/sec. Since the tick rate controls

the simulation update rate and the number of messages sent each second to match the simulation update, the higher the tick rate goes, the more updates need to be processed.

Figure 55 shows the average number of messages sent to the server depending on the tick rate. It indicates that part of the processor load comes from having to treat messages from clients, which increases heavily with the tick rate. It is seen that the amount of messages more than triples from 30 to 120 ticks/sec. Consequently, this increase of messages uses more bandwidth (not shown) from the system.

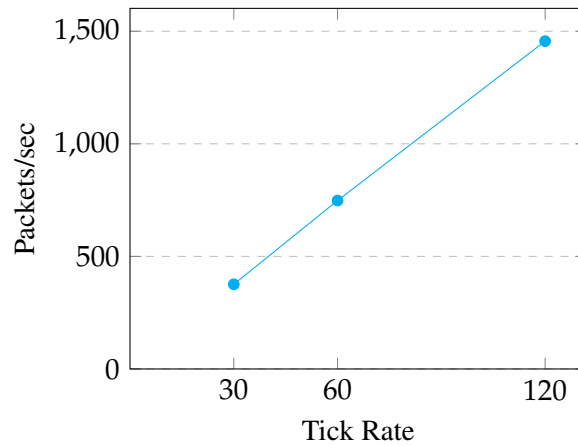


Figure 55: Server average number of packets received by tick rate.

4.1.3 Scalability

Up until now, the system was studied under a normal execution of up to 20 players. In this section, Figures 56-59 present the results of the system with a substantially higher number of players to evaluate scalability. Since each room was designed to sustain a maximum of 60 players, additional rooms were added to support more. Therefore, beyond 60 players, these are divided evenly across rooms. For instance, 90 players corresponds to 2 rooms with 45 players each. Furthermore, the study is bound by the number of players that each machine can instantiate, which during the experiments was 40 players, resulting in a total of 120 players.

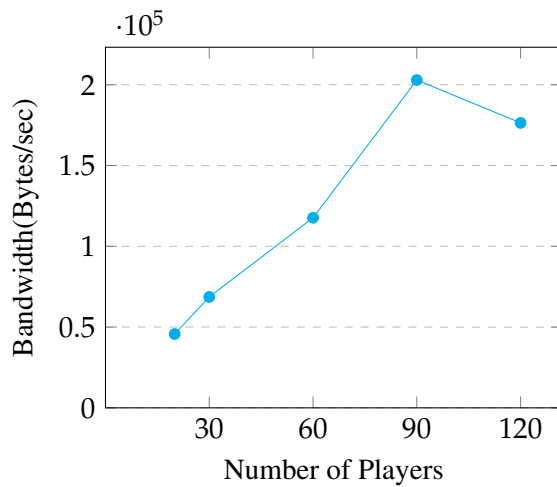


Figure 56: Server incoming network throughput per client.

Figure 56 describe the measurements of the incoming and outgoing network throughput of the server, respectively. It is found that these aspects continue to increase linearly with the number of players. However, when the server reaches 120 players, the bandwidth drops to around 127 KB/sec, when it should be around 217 KB/sec. This is due to client machines needing to instantiate 40 client applications each to reach the 120 players, which they cannot handle since it creates a lot of information to process. As result, their applications start to lag and send less messages, resulting in the decrease seen from 90 to 120 players.

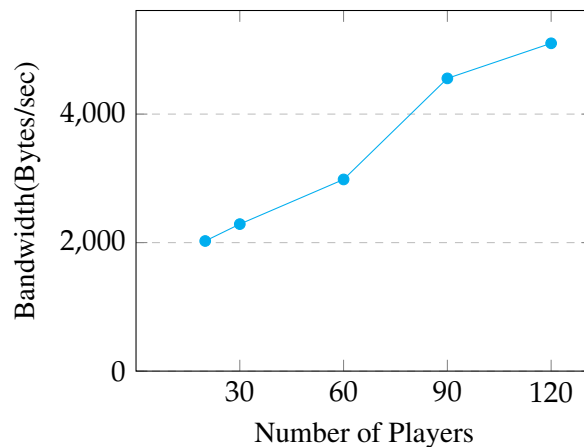


Figure 57: Server outgoing network throughput per client.

Figure 57 shows the measurements of the outgoing network throughput of the server. It follows a linear growth until 60 players. Furthermore, between 60 and 90 players there is a bigger increase of outgoing messages. This is because to accommodate 90 players, two rooms are needed. As result, this creates another multicast channel with more tick and clock messages to keep the second room synchronized. Therefore, resulting in a bigger jump of bandwidth used, even with players lagging on their machines. Additionally, the same would be seen going beyond 120 players.

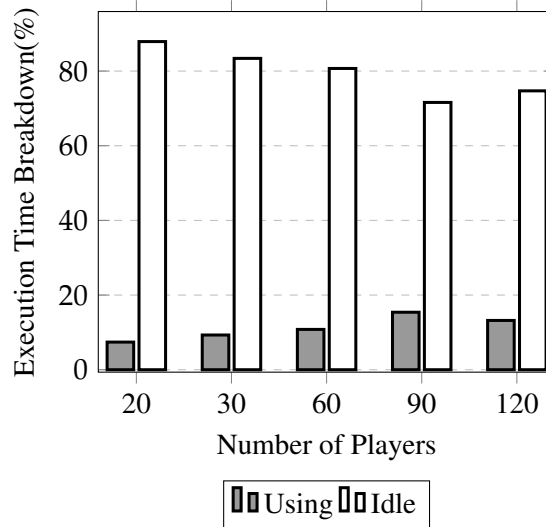


Figure 58: Server execution time breakdown.

Figure 58 shows the measurements captured from the server's CPU. Through the execution time breakdown, it is seen that the idle time is still high at 120 players, while processing time is low, allowing further expanding the amount of messages treated. Therefore, it can be concluded that the server can support hundreds of players and their messages. Although it was found that the server can handle these high amounts of messages, expanding as needed, the client machine used cannot (not shown). So, trying to reach a number of players any higher than 120 will result in multiple disconnections due to failing to send messages to the server in time.

Another factor influencing the system's scalability is tick rate. Figure 59 shows the disconnection rate of players depending on the tick rate. This experiment runs on a normal execution of the game with 120 players. As expected, a higher tick rate leads to a higher disconnection rate. Most players disconnect as soon as the race starts due to the client cluster machine instantiating multiple game scenes at once, affecting the machine's performance. Therefore, when 40 players try to do this all at once (in the same machine), they are unable to send messages. Even if these manage to not be disconnected at the beginning of the game, they may still be disconnected later because of having the client machine generate and process a lot of messages, which in time will become backed up, resulting in more disconnections until the client machine can tolerate the amount of load being created. Additionally, the opposite happens when lowered the tick rate. However, it is not recommended due to affecting the simulation's precision and the player's experience, and therefore it wasn't plotted in Figure 59.

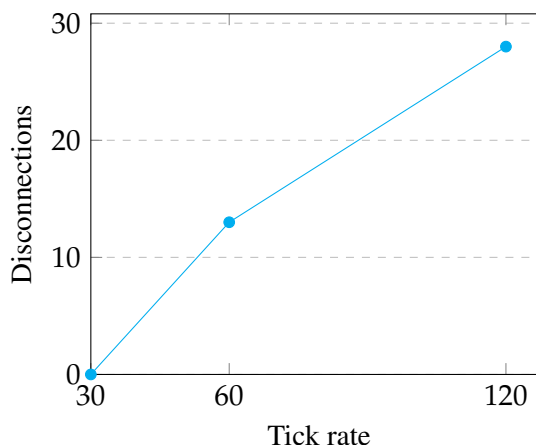


Figure 59: Server disconnection rate.

Lastly, the difference between the server's scalability comparing a small and big map was also tested. However, since the server's load comes almost entirely from clients and not the game world update, the data did not indicate a significant influence and therefore, it is not shown.

4.1.4 Security

Security metrics: Taking a pragmatic approach, the assessment of security follows the examination of the two security goals for online games described in [Kirmse and Kirmse \(1997\)](#). These are (1) client information and (2) anti-cheating measures. However, since the game-server doesn't have an account or verification system, it does not save sensitive data about clients. Therefore, client information is not considered for the system at hand, which consequently makes this system's security assessment focused solely on the anti-cheating measures that it naturally provides.

At the **protocol level**, the system's architecture doesn't allow clients to eavesdrop on messages not intended for them because, while **P2P**, clients on this system are not part of the dissemination process of messages that are not their own. Therefore, it impedes the possibility of clients harming others by sending messages about other objects or actions, which would result in the creation of inconsistencies across clients and overload the server with correction messages. Moreover, clients only know the player details about other clients, such as *clientRoomId*, *Username* and *Color*, and not their client information, such as address and port. Since clients are the only responsible entity in the architecture for sharing their player state, it creates the possibility for an exploit, such as Lag Switch², which unfortunately is irremediable. Nevertheless, to help combat it, the server implements an away from keyboard (**AFK**) timer, which in case a client stops sending player state messages for a determined time, they will be removed from the network. Therefore, the server forces them to send messages to it, which

² Lag Switch (or Artificial Lag) is when, in **P2P**, the stream of data between one or more players gets slowed or interrupted, causing movement to stutter and making opponents appear to behave erratically to other clients, while their client queues up the actions performed.

since the application uses multicast, they will also end up sending to the other clients listening to the multicast address of the room the client is in.

At the **game level**, the game-server has the authority over the player game state, such as if they have finished, passed a checkpoint, respawned, or grabbed someone. Therefore, it is concluded that the game server highly restrains the cheating problems at the game level. Moreover, since player states come from distrusted clients, the server ascertains security by monitoring, identifying, and correcting information coming from the clients that could be altered to their advantage. If they are caught misbehaving, the room issues a correction to be immediately implemented on the game client, asserting a correct state throughout the game session. However, the monitoring process is narrow, consisting only of checking the tick and if they are inside a wall.

4.1.5 Resilience

Resilience metrics: To study the influence of poor network conditions on the multiplayer game, the effects of latency, jitter, packet loss, and out-of-order packets are applied and measured. While the study doesn't provide a way to quantify the player's perceptions on the gameplay quality exhibited, comments and observed trends during and after the user studies are presented.

Latency: With the insertion of latency, the gameplay was somewhat sluggish when the delays induced on the client's connection were as low as 75 ms, being more impactful at latencies over 100 ms. However, the experiments showed that latency did not affect the local player, even at 200 ms. Since the player replication relies mainly on the client's application to function, being independent of the server's responses, it was found that it made them, for the most part, unaware of the delay in the game. Nevertheless, latency became clear when interacting with players or respawning (controlled by the server), which depending on the delay, took longer to happen. The most subjective impact was during the continuous variation of latencies, also known as packet jitter. It made other players start to teleport and, on some occasions, the local player would go into ragdoll state (due to colliding with someone) or be pushed for no apparent reason (at least from the client's perception).

Packet Loss: The examination of packet loss in the system showed that a player could occasionally notice a short strange behavior when induced to loss rates of at least 3%, becoming more noticeable at 10%. However, it was found that packet loss does not prove to impair the game's consistency since most messages are retransmitted due to **TCP**. For instance, when a player falls off the map, it takes a few milliseconds longer to be respawned. But since it uses **TCP**, it continues to occur, even if taking a bit longer, due to the need for retransmission. Even in messages that use **UDP**, which are not retransmitted, game protocols such as entity interpolation are implemented to take care of the problem of missing information by masking the loss.

Moreover, it was found that only when adding a loss rate of 60%, the entity interpolation of the remote players starts to break, aggravating as it goes higher. However, it is only spotted if remote players are not following a linear movement. These occurrences appear only in more complex behaviors, such as when a player drastically changes directions. So instead of following the original movement, the entity interpolation continues to move forward due to extrapolation since it is missing messages. Later, when a message is received, it quickly shifts to the correct position. Furthermore, as expected, players start to disconnect when induced to a loss rate of 100%.

This is due to the clients failing to send their ping messages to the server, which makes the server think they have crashed, removing them from the connections and game session.

Packet duplication and out of order: Lastly, the insertion of packet duplication and out-of-order packets in the system was also carried out. It is found that both do not affect the game content at all due to the game protocols implemented. For instance, even though entity interpolation is highly dependent on foreign information, where the ordered execution of states is fundamental, it contains countermeasures to fight these conditions, such as ordering the buffering player states received on a list. Similarly, pushing and grabbing still results in the same outcome since the server is the one who decides what happens.

In summary, players should avoid servers with latency over 180 ms and packet loss levels over 5%. Nevertheless, servers that do not meet these criteria may still not significantly affect the player's performance. However, it does make the gameplay experience less enjoyable, which at least partially defeats the purpose of playing games in the first place.

4.2 DISCUSSION

The first purpose of this study was to identify the performance, scalability, and security of the distributed architecture constructed. The second purpose of the study was the resilience that the system, under normal and poor circumstances of a network communications-based system, can maintain the distributed game at a certain quality that does not affect the playability. For this study, a discussion of significant findings discovered during and after the analysis is included here, also providing future research possibilities to pursue.

4.2.1 Performance

The analysis of the system allowed to study the server's behavior and performance under a reasonable player size of 20. It was concluded that most workload comes from the player's logic and that the player replication implemented led to a low processor utilization on the server because of only using the client's resources. Furthermore, the server's outgoing bandwidth depends on the amount of induced interactions from the clients but still does not exceed a few Bytes/s and kept constant with a fairly small increase due to sending ping messages to clients. The server's incoming bandwidth increased, also linearly, reaching a higher number of 46 KB/s, which is not an issue for modern systems, hence it can support hundreds of players.

Moreover, it was concluded that Multicast allowed a good server performance while increasing the number of players in the study. However, because clients use Multicast to share information about their player states, it doesn't allow them to choose the amount of information to send each peer. Therefore, they end up sending the same amount of data to their peers and server, even when it is not needed. As discussed in [Section 2.3.4](#), clients do not need to know every bit of information at all times, which could be the case in this game. It could be theorized that using interest management (IM), the bandwidth requirement would diminish considerably. Going forward, a solution to explore would be to remove Multicast and implement regular UDP between peers, reducing the bandwidth requirements of the clients and hopefully covering more users with lower available resources.

Additionally, the overall resources used by the system could be lower if a [Reliable User Datagram Protocol \(RUDP\)](#)³ protocol was used instead of TCP and UDP, as the assumptions made in [Huh \(2018\)](#). A RUDP based scheme, would be more efficient than using TCP, and would scale better because TCP will always handle out-of-order packets, meaning it won't deliver more messages until everything else has been received in an orderly fashion. This could be rather detrimental to the performance of a reasonably sized multiplayer game such as this, which in most cases it is not needed. Additionally, RUDP schemes handle out-of-order packets when strictly required, avoiding the TCP problem of holding messages until the one in front is confirmed to be received.

4.2.2 Scalability

To evaluate the *scalability*, the system has undergone experiments with a higher number of players, reaching 120 total. The server can hold hundreds of clients since its CPU usage is still minimal with 120 players. However,

³ https://en.wikipedia.org/wiki/Reliable_User_Datagram_Protocol

the bandwidth used continues to grow linearly with the number of players introduced. This analysis supports the hypothesis that a limiting factor in the server's scalability is the eventual extinguishment of both CPU and bandwidth resources. Regardless, due to the lack of resources, the experiments cannot confirm it.

Furthermore, reducing the tick rate can be a viable option to improve the system's scalability since it decreases the load on the server. However, it can be detrimental to the simulation precision and the player experience. Another solution to the system's scalability may be server clustering, which achieves it through adding more resources. This implementation could feature the limitation of executing a certain number of rooms or clients per machine. Moreover, it would be valuable to look at micro-service architectures that incorporate multiple small services, which are very scalable and highly modular, to help the system achieve better results.

Lastly, to all benchmarking efforts, the work done does not capture every aspect of the server behavior, given the bots used are not actual human players. While these proved to produce load on the server in order to study it, they did not allow, for instance, to observe the trade-off of lag compensation in the server because they do not possess AI for grabbing and pushing other bots. However, the data gathered is a decisive first step towards understanding and quantifying the problems faced in scaling a multiplayer game system.

4.2.3 Security

The proposed architecture provides a limiting window to the client's ability to disrupt the system due to their role in it. This is supported by the fact that the clients cannot harm others by disseminating false information. However, the use of Lag Switch cannot be avoided due to using a P2P system to disseminate player states. Although the implementation deals with this exploit, it is also limited.

Furthermore, the study was extended to the security at the game-level, where the server provides security and control over important game state events, such as respawn and player-on-player interaction, having also full authority over their dissemination to every single client. During development its importance was shown to be crucial in the game's well-functioning, suggesting the assumptions discussed in [Yahyavi and Kemme \(2013\)](#), wherein large games there is a need for a central governing authority.

Although the player replication implemented proved to be a good choice for scalability, having the client control their state, made the game's security susceptible to cheating since players can still cheat in their simulation and send wrong information to the server. Even though the game server focuses on detecting client inconsistencies, it still leaves room for improvement since the detection is limited to a few key points. A more effective way would be to prevent these misbehaves from being sent to the server in the first place. However, for a more comprehensive examination of this, it would be required anti-cheating software on the game client, which falls out of the scope. Therefore, it would be interesting for future research to understand what techniques and tools are used to help prevent cheating in multiplayer games.

4.2.4 Resilience

The data collected during the several user studies indicates that latency affects the system and agrees with assumptions made before about the game's low tolerance to latency. Added latencies between 50 and 100 ms showed to have nearly equal low influence on the game, yet noticeable, while in the range of 100 (above) and 180 ms latency had taken impact on the game's performance. However, the player performance remained the same due to the replication system relying solely on the client's application. Similarly, packet loss does not affect player performance and has an insignificant influence on the game since most mechanisms can mask this loss of information under normal conditions. Similarly, results from packet duplication and out-of-order suggest that it does not affect the system at all due to the mechanisms implemented.

The general study exhibited how few game components are affected by poor network conditions when mechanisms to lessen the high dependency on foreign information are implemented. Consequently, the client's application requires less resources from the server, doing more with what it has, and lowering the per-client requirements of the system.

Moreover, the experiments showed the importance of the order the messages arrive at the server for actions such as pushing players. During the analysis it was shown a reason for implementing lag compensation. This is because, before, if two players were to push each other at the same tick, the message arriving first was the one that decided who gets to do the action and who cannot (because they were pushed and transitioned into ragdoll state). This proved to be unfair, and therefore it was changed to consider both actions instead of only the first. Moreover, these slight changes in the fairness of game outcomes can highly affect the players' enjoyment. Although it hasn't been considered the player's perception in the scope of implementing these alterations in the system, it would be interesting for future work to do so as to know if their implementation is required.

CONCLUSION

5.1 CONCLUSIONS

This research aimed to investigate and construct a multiplayer game system, taking into account the technologies, strategies, and mechanisms used in the industry, and afterward evaluate its performance, security, scalability, and network resilience generated from the solution.

The initial study of relevant literature showed that Massive Multiplayer Online games (MMOG) are a thriving business industry with a range of problems and challenges. Most MMOGs exclusively use client-server (CS) architectures due to their simplicity. However, due to the lack of scalability, there has been increasing interest in designing peer-to-peer (P2P) (structured or unstructured) and Hybrid architectures for these types of games. So, these architectures are discussed and compared against each other concerning their robustness, scalability, delay, consistency, cheat-proof, and commerciality for a better understanding of their capabilities and the lack of. Furthermore, an overview of some of the core elements in multiplayer games is made to understand how depending on the game it can influence the appropriate architecture. Many techniques are also proposed to solve the problems of consistency, synchronization, and playability in a distributed game. However, while some of these can be applied to multiple architectures, many are dependent on the underlying architecture. The exploration and study of these architectures, protocols, and strategies used to create MMO games aimed to build a foundation on multiplayer games in the effort to better select a solution to distribute the game at hands.

The game developed, *DummyGuys*, has a Hybrid architecture model that attempts to combine features from CS and P2P architectures. It consists of two applications, *GameServer* and *GameClient*, and allows for multiple game sessions to be run at the same time while staying independent and isolated from each other. To communicate, it employs TCP, UDP and IP Multicast using UDP, which was found to be helpful in the dissemination of information to several clients. The applications' structure follows the division of network and game logic to provide a clean development method by successfully abstracting basic communication problems, such as consistency and reliability issues, from the game logic code. Lastly, to ensure that the distributed game performs correctly, the system also implemented numerous protocols to achieve a functional game with consistency, synchronization and good playability.

The system's capabilities are studied through a series of experiments with a set of appropriate methodological approaches. The performance is examined through experiments with up to 20 players (bots). The results indicate a continuous increase of incoming bandwidth with the introduction of clients. The incoming required bandwidth

per player is constant and in the order of a few Bytes/s, whereas the outgoing bandwidth per user depends on the total number of players and the induced interactions, but does not exceed a few Bytes/s also. Thus, current network connections on the game server can support hundreds of clients. Additionally, the processing load on the server proved to be mostly caused by the players, but continued to be small due to the player replication using mostly the client's resources. Moreover, the use of client-side metrics is proposed, such as the server response rate and the average response time, for evaluating the overall server performance. To better understand the implication of the performance results, future studies could address experiments with actual players.

Further experiments with a higher number of players are executed to examine the system's scalability, showing that the system can handle hundreds of players. However, resources on the server are finite, and their use continues to grow linearly with the number of players. So, their maximum capacity will be met at some point. However, this could not be tested due to the limited resources at hand. Therefore, further research is needed to determine under what conditions the system reaches the maximum load possible. Lastly, a few solutions to the scalability problem are suggested to be considered for future work.

For the security assessment, both the system's architecture and implementation are analyzed. The client's role in the system denies the possibility of harming other clients since they can only share information about themselves and are not part of the dissemination process to other clients. Also, the most vulnerable point of the system is the information sent from the client that cannot be trusted, hence it is examined. Additionally, being a central arbiter with authority over the clients, the server provides an easy and secure way to control and manage the game state and events.

Lastly, under poor network conditions, the system continues to function normally and provide a good player experience. Latency between 50 and 100 ms has limited influence on the game, while 180 ms shows to be more noticeable and impact player experience. However, the player behavior is not affected due to its implementation relying only on the client's application. Packet loss is found to have no actual effect on the player experience since most messages are re-transmitted, still it can be noticeable by players. Similarly, packet duplication and out-of-order seems to not affect the system due to the various game protocols implemented, which have counter-measures to such events.

5.2 LIMITATIONS

Due to using IP Multicast (Network-Layer Multicast), the current system does not consider the network layer. This is because IP Multicast requires all the routers in the path of the multicast packets to have multicast routing configured. However, since we are trying to use public internet, we don't own or control the routers. Therefore, use of IP Multicast was solely due to being an extremely efficient way to share information in one-to-many destinations.

Furthermore, to simplify the system and help in testing, the rooms were integrated into the server application, allowing one server application to have multiple rooms. However, this is not the most desirable outcome when developing an MMO server. It is preferred that rooms are kept separate, in different [Virtual Machines \(VM\)](#) or physical machines, and that a discovery server directs the clients to the ones available. Regardless, the rooms,

even if in the same application, were designed and constructed to be fully autonomous and self-contained, which helps for future development following this approach.

In all benchmarking experiments, the work conducted does not capture every aspect of the server behavior, given the restricted bots' behavior used. Therefore, the information gathered from these metrics does not state the entirety of the game server. However, they provided a decisive first step towards understanding and quantifying the system's behavior. Additionally, the number of clients used in the study was limited to what the three client machines used allowed, representing 120 players in total.

Lastly, while the system employs several mechanisms to ascertain consistency, synchronization, and playability, many challenges have not been addressed in this document, such as cheating, persistence, and availability, because it falls out of the scope of the document or were not seen to be necessary.

5.3 FUTURE WORK

The work carried out leaves some opportunities that would be beneficial to investigate or develop in the future. Thus, it is suggested the following work proposals:

- Investigate and implement an Reliable User Datagram Protocol (**RUDP**) following an **ACK** (acknowledgment) or **NACK** (negative acknowledgment) communication protocol to compare and assess reliability, resource usage, and speed with a **TCP** and **UDP** scheme;
- Investigate the micro-services-oriented architecture advantages and applicability to large-scale games, followed by an implementation;
- Investigate cheating in multiplayer games and measures to prevent it, such as encryption layer under packet communication sockets and anti-cheating software, followed by an implementation in the system constructed in this work;
- Analyze the constructed system in an actual context using actual human players or bots with an AI similar to human player behavior;
- Address the aspects of persistence and availability in multiplayer games on the solution constructed, which would require a further expansion of the system and game functionalities.

BIBLIOGRAPHY

- Richard John Anthony. Chapter 3 - the communication view. In Richard John Anthony, editor, *Systems Programming*, pages 107–201. Morgan Kaufmann, Boston, 2016. ISBN 978-0-12-800729-7. doi: <https://doi.org/10.1016/B978-0-12-800729-7.00003-0>.
- Grenville Armitage. An experimental estimation of latency sensitivity in multiplayer quake 3. pages 137 – 141, 01 2003. ISBN 0-7803-7788-5. doi: 10.1109/ICON.2003.1266180.
- J.W. Barrus, Richard Waters, and D.B. Anderson. Locales: Supporting large multiuser virtual environments. *Computer Graphics and Applications, IEEE*, 16:50 – 57, 12 1996. doi: 10.1109/38.544072.
- Steve Benford and Lennart Fahlén. A spatial model of interaction in large virtual environments. pages 107–, 01 1993. ISBN 978-94-010-4928-3. doi: 10.1007/978-94-011-2094-4_8.
- Ashwin Bharambe, John Douceur, Jacob Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. volume 38, pages 389–400, 01 2008. doi: 10.1145/1402958.1403002.
- Stephen Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. 07 2003. doi: 10.17487/RFC3550.
- J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. pages 289–300, 01 2005.
- Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. Game traffic analysis: an mmorpg perspective. *Computer Networks*, 50:3002–3023, 11 2006. doi: 10.1016/j.comnet.2005.11.005.
- Paolo Costa, Matteo Migliavacca, Gian Picco, and Gianpaolo Cugola. Introducing reliability in content-based publish-subscribe through epidemic algorithms. page 1, 01 2003. doi: 10.1145/966626.966629.
- Eric Cronin, Burton Filstrup, and Anthony Kurc. A distributed multiplayer game server system. 07 2001.
- Eric Cronin, Burton Filstrup, and Sugih Jamin. Cheat-proofing dead reckoned multiplayer games (extended abstract). 01 2003a.
- Eric Cronin, Anthony Kurc, Burton Filstrup, and Sugih Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools Apps*, 23, 03 2003b. doi: 10.1145/566500.566510.
- Dani. I made fall guys in 1 week... and tricked them to think it's real! <https://www.youtube.com/watch?v=vXQpgq1aVoU>, 2020. [Online; accessed 21-December-2021].

- Matthew David. *The Legacy of Napster*, pages 49–65. 01 2016. ISBN 978-1-137-58289-8. doi: 10.1057/978-1-137-58290-4_4.
- C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications. *Network, IEEE*, 13:6 – 15, 08 1999. doi: 10.1109/65.777437.
- Emmanuel Frécon and Mårten Stenius. Dive: A scalable network architecture for distributed virtual environments. *Distributed Systems Engineering*, 5:91–100, 09 1998. doi: 10.1088/0967-1846/5/3/002.
- Chris GauthierDickey. Distributed architectures for massively-multiplayer online games. 2004.
- Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr. Low latency and cheat-proof event ordering for peer-to-peer games. pages 134–139, 01 2004. doi: 10.1145/1005847.1005877.
- Chris GauthierDickey, Virginia Lo, and Daniel Zappala. Using n-trees for scalable event ordering in peer-to-peer games. pages 87–92, 01 2005. doi: 10.1145/1065983.1066005.
- L. Gautier and C. Diot. Design and evaluation of mimaze a multi-player game on the internet. pages 233 – 236, 04 1998. ISBN 0-8186-8557-3. doi: 10.1109/MMCS.1998.693647.
- Thorsten Hampel, Thomas Bopp, and Robert Hinn. A peer-to-peer architecture for massive multiplayer online games. page 48, 01 2006. doi: 10.1145/1230040.1230058.
- Szabolcs Harcsik, Andreas Petlund, Carsten Griwodz, and Pål Halvorsen. Latency evaluation of networking mechanisms for game traffic. pages 129–134, 01 2007. doi: 10.1145/1326257.1326280.
- Alex Hsu and C.-C. Jay Kuo. On the design of multiplayer online video game systems. *Proceedings of SPIE - The International Society for Optical Engineering*, 11 2003. doi: 10.1117/12.512201.
- Shun-Yun Hu, Jui-Fa Chen, and Tsu-Han Chen. Von: A scalable peer-to-peer network for virtual environments. *Network, IEEE*, 20:22 – 31, 08 2006. doi: 10.1109/MNET.2006.1668400.
- Jun-Ho Huh. Reliable user datagram protocol as a solution to latencies in network games. *Electronics*, 7:295, 11 2018. doi: 10.3390/electronics7110295.
- Jaecheol Kim, Jaeyoung Choi, Dukhyun Chang, Ted Kwon, Yanghee Choi, and Eungsu Yuk. Traffic characteristics of a massively multi-player online role playing game. pages 1–8, 01 2005. doi: 10.1145/1103599.1103619.
- Andrew Kirmse. A network protocol for online games. 01 2000.
- Andrew Kirmse and Chris Kirmse. Security in online games. *Game Developer*, 4, 01 1997.
- Bj Knutsson, Honghui Lu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. 01 2004.
- Hsiu-Hui Lee and Chin-Hua Sun. Load-balancing for peer-to-peer networked virtual environment. page 14, 01 2006. doi: 10.1145/1230040.1230079.

- Elias Leontiadis, Vassilios Dimakopoulos, and Evaggelia Pitoura. Creating and maintaining replicas in unstructured peer-to-peer systems. pages 1015–1025, 08 2006. ISBN 978-3-540-37783-2. doi: 10.1007/11823285_107.
- Jian Liang, Rakesh Kumar, and Keith Ross. Understanding kaza. 06 2004.
- Adrian Thornhill Mark N.K. Saunders, Philip Lewis. "Research Methods for Business Students" (8th ed.), page 872. 03 2019. ISBN 9781292208787.
- Martin Mauve, Jürgen Vogel, and Volker Hilt. Local-lag and timewarp: Providing consistency for replicated continuous applications. *Multimedia, IEEE Transactions on*, 6:47 – 57, 03 2004. doi: 10.1109/TMM.2003.819751.
- Martijn Moraal. Massive multiplayer online game architectures. 2007.
- Christoph Neumann, Nicolas Prigent, Matteo Varvello, and Kyoungwon Suh. Challenges in peer-to-peer gaming. *Computer Communication Review*, 37:79–82, 01 2007. doi: 10.1145/1198255.1198269.
- Joseph Pellegrino and Constantine Dovrolis. Bandwidth requirement and state consistency in three multiplayer game architectures. pages 52–59, 01 2003. doi: 10.1145/963900.963905.
- M. Pritchard. How to hurt the hackers: the scoop on internet cheating and how you can combat it. 01 2000.
- Antony Rowstron. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Middleware*, 07 2001.
- Derek Sanderson. Online justice systems. 04 1999.
- Deering SE. Host extensions for ip multicast. 01 1998.
- Jouni Smed, Timo Kaukoranta, and Harri Hakonen. Aspects of networking in multiplayer computer games. *The Electronic Library*, 20:87–97, 10 2002a. doi: 10.1108/02640470210424392.
- Jouni Smed, Timo Kaukoranta, and Harri Hakonen. A review on networking and multiplayer computer games. 05 2002b.
- Mauro Sozio, Thomas Neumann, and Gerhard Weikum. Near-optimal dynamic replication in unstructured peer-to-peer networks. *Lenzerini, Maurizio; Lembo, Domenico: PODS'08 : Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM, 281-290 (2008)*, 01 2008. doi: 10.1145/1376916.1376956.
- Jeffrey Steinman. Breathing time warp. *ACM SIGSIM Simulation Digest*, 23:109–118, 07 1993. doi: 10.1145/174134.158473.
- Jeffrey Steinman. Scalable parallel and distributed military simulations using the speedes framework. 01 1995.

- Jeffrey Steinman, J. Wallace, D. Davani, and D. Elizandro. Scalable distributed military simulations using the speedes object-oriented simulation framework. 01 1998.
- Ion Stoica, Robert Morris, David Liben-nowell, David Karger, M. Frans, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. 04 2002.
- Mirko Suznjevic and Maja Matijasevic. Player behavior and traffic characterization for mmorpgs: a survey. *Multimedia Systems*, 19, 06 2012. doi: 10.1007/s00530-012-0270-4.
- Mirko Suznjevic, Ognjen Dobrijevic, and Maja Matijasevic. Mmorpg player actions: Network performance, session patterns and latency requirements analysis. *Multimedia Tools Appl.*, 45:191–214, 10 2009. doi: 10.1007/s11042-009-0300-1.
- Sykoo. I made fall guys in 12 hours, but it's impossible to win. <https://www.youtube.com/watch?v=YRur1EJsgvU>, 2020. [Online; accessed 21-December-2021].
- Ian Taylor and Andrew Harrison. *Gnutella*, pages 181–196. 01 2009. doi: 10.1007/978-1-84800-123-7_10.
- Valve®. Source multiplayer networking. https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking, 2021. [Online; accessed 14-November-2021].
- Matteo Varvello, Stefano Ferrari, Ernst Biersack, and Christophe Diot. Exploring second life. *IEEE/ACM Trans. Netw.*, 19:80–91, 02 2011. doi: 10.1109/TNET.2010.2060351.
- Amir Yahyavi and Bettina Kemme. Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Computing Surveys (CSUR)*, 46, 10 2013. doi: 10.1145/2522968.2522977.
- Shinya Yamamoto, Yoshihiro Murata, Keiichi Yasumoto, and Minoru Ito. A distributed event delivery method with load balancing for mmorpg. pages 1–8, 01 2005. doi: 10.1145/1103599.1103610.
- Kaiwen Zhang and Bettina Kemme. Transaction models for massively multiplayer online games. pages 31–40, 10 2011. doi: 10.1109/SRDS.2011.13.

