UNIVERSIDADE D
COIMBRA

João Rafael Santos Calhau

MULTIPLAYER MODULE FOR SCIENCE4PANDEMICS' GAME

Dissertation in the context of the Master in Informatics Engineering, specialization in Intelligent Systems, advised by Professor Licínio Gomes Roque and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September of 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

João Rafael Santos Calhau

Multiplayer Module for Science4Pandemics' Game

Dissertation in the context of the Master in Informatics Engineering, specialization in Intelligent Systems, advised by Prof. Licínio Gomes Roque and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2023

Acknowledgements

I would like to express my gratitude to professor Licínio Roque for the opportunity to participate in the Science4Pandemics project through this internship and for his availability and guidance from the beginning of the internship.

I would also like to extend my thanks to my family and my girlfriend for their unwavering support. Their support was crucial to conclude this journey that began in 2018, with my admission to the Informatics Engineering course.

This dissertation was partially conducted within the scope of the Science4Pandemics Project, an EITHealth initiative, and funded by FCT - Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), as part of the research and development unit CISUC - UIDB /00326/2020 or project code UIDP/ 00326/2020.

Abstract

This thesis describes the implementation of a stateless backend server for Science4Pandemics' multiplayer game. It starts by investigating the architectures of Massive Multiplayer Online Games and expands on how they can be addressed. The primary purpose is to examine the two distinct architectures that have dominated the industry over the years, client-server and peer-to-peer and present a new alternative after determining how they relate to one another. The majority of Massive Multiplayer Online Games (MMOGs) developers have up to now implemented their games using client-server architectures. However, with the emergence of blockchain technologies, there has been an increase in peer-to-peer architectures. This thesis will examine the feasibility of using these peer-to-peer technologies to operate MMOGs. This is done by first looking at the existing client server architectures' characteristics and advantages and disadvantages. After that, peer-to-peer architectures will be reviewed, and their effects on MMOGs will be considered. Based on this, the two architectures will be compared, and a new solution will be explained. The solution encompasses using Erlang's powerful lightweight processing to distribute the load of the server's processing logic through different isolated processes. This combined with an HTTP server to distribute HTTP requests through these processes and a MQTT Broker to distribute messages allows the backend server to use a specified port to listen to requests and another to send the responses, and because MQTT uses a publish subscribe model, meaning it has one input channel and as many output channels as the number of clients subscribed to it, the bandwidth of the MQTT port is not overloaded. This allows the backend server to run with minimal maintenance and because of the messages being distributed, allows the game developers to keep their previously developed code intact, making it an economic and fast solution for transforming a single-player game to multi-player.

Abstract

Esta tese descreve a implementação de um servidor de backend sem estado para o jogo multijogador do Science4Pandemics. Começa por investigar as arquiteturas de Jogos Online Massivamente Multijogador (MMOGs) e expande sobre como podem ser abordadas. O objetivo principal é examinar as duas arquiteturas distintas que dominaram a indústria ao longo dos anos, cliente-servidor e peer-to-peer, e apresentar uma nova alternativa após determinar como se relacionam entre si. Até agora, a maioria dos desenvolvedores de MMOGs implementou os seus jogos usando arquiteturas cliente-servidor. No entanto, com o surgimento das tecnologias de blockchain, tem havido um aumento nas arquiteturas peer-to-peer. Esta tese irá examinar a viabilidade de usar essas tecnologias peer-to-peer para operar MMOGs. Isto é feito primeiro ao olhar para as características e vantagens e desvantagens das arquiteturas cliente-servidor existentes. Em seguida, serão revistas as arquiteturas peer-to-peer e serão considerados os seus efeitos nos MMOGs. Com base nisso, as duas arquiteturas serão comparadas, e uma nova solução será explicada. A solução envolve o uso do poderoso processamento leve do Erlang para distribuir a carga da lógica de processamento do servidor através de diferentes processos isolados. Isto, combinado com um servidor HTTP para distribuir pedidos HTTP através destes processos e um Broker MQTT para distribuir mensagens, permite que o servidor de backend utilize uma porta específica para ouvir pedidos e outra para enviar as respostas, e como o MQTT utiliza um modelo de publicação e subscrição, o que significa que tem um canal de entrada e tantos canais de saída quantos os clientes inscritos nele, a largura de banda da porta MQTT não é sobrecarregada. Isto permite que o servidor de backend funcione com uma manutenção mínima e, devido à distribuição de mensagens, permite que os desenvolvedores de jogos mantenham o seu código previamente desenvolvido intacto, tornando-o uma solução económica e rápida para transformar um jogo single-player num jogo multi-player.

Contents

Acronyms

CCU Concurrent Users.

DSS Decision Support Systems.

LDAP Lightweight Directory Access Protocol.

MMO Massive Multiplayer Online.

MMOGs Massive Multiplayer Online Games.

MQTT Message Queuing Telemetry Transport.

OLTP Online Transaction processing.

P2P Peer-to-Peer.

List of Figures

List of Tables

Chapter 1

Introduction

Massive Multiplayer Online Games, commonly referred to as MMOGs. They are complex video games, usually with 3D graphics and an extensive virtual world where players can engage in a variety of interactions.

Massive Multiplayer Online Games used to be a hot topic of study for scientists. Numerous articles have been written concerning problems with these games such as [Jon, 2010].

The social features of these games have received much attention from these researches [Jon, 2010]. Players frequently devoted a significant amount of time to them and to forming communities and relationships with other players. It has long been a popular topic in social studies to discuss why these players spend so much time playing these games, how these communities function, and the effects of all of this in the real world.

This document explores the different architectures used in today's Massive Multiplayer Online (MMO) and analyzes and compares these in order to determine which is most suited for them. Because the primary purpose of the thesis is the implementation of a backend server for Science4Pandemics and considering the game has been in development for a year and only has a single player version, the main focus of this thesis is:

> Design and development of an efficient and scalable backend server to convert the Science4Pandemics project from singleplayer to multiplayer.

1.1 Context

For this thesis, the author was asked to create a stateless backend server using Erlang, which is meant to host the multiplayer version of Science4Pandemics' game and to extract data from user actions for later analysis. The server must hold multiple connections without failing, meaning scalability is the priority. Quoting the game's design document the game concept would be an educational game

focused on sensemaking of everyday tasks, resource management and decision making during an epidemic or pandemic event.
A game for citizen science, through a digital experiential learning approach. It presents a progression from singleplayer to multiplayer simulation game directed at small teams (typically 3 or 4 people, to achieve social involvement while reducing constraints). Players will first gain understanding of basic concepts of infectious diseases (singleplayer learning stage) and then will face new challenges to consolidate knowledge, and understand the socio-technical and human interdependence factors in epidemics/pandemics (multiplayer challenge stage).

1.2 The Problem

The main problem this thesis aims to solve is implementing a scalable and sustainable backend server to convert Science4Pandemic's game from singleplayer to multiplayer. The solution aims to be efficient in terms of development costs and it should retain good performance in an economic infrastructure.

1.3 Goals

To achieve a scalable and sustainable backend server which can be used to convert Science4Pandemic's game from singleplayer to multiplayer with reduced development and maintenance costs, the work has been split into three different goals:

- Development of the backend server for Science4Pandemics using erlang and MQTT.
- Generalization of the solution(e.g. if another game similar to S4play is being build by the University, the backend should also be portable or reusable by the developers).
- Analysis of the scalability of the solution.

1.4 Document Structure

An introduction to massively multiplayer online games will be provided in Chapter 2, where The definition of MMO will be discussed, along with the state of research in this area. The two MMOG topologies that could be used and their characteristics will be introduced: client-server and peer-to-peer. Chapter 3 will cover the main objective of this thesis and Chapter 4 will describe the steps taken to reach that objective. In Chapter 5, the implementation will be discussed and on Chapter 6 an evaluation of this implementation is presented.

Chapter 2

State of the Art

2.1 Massive Multiplayer Games Background

Massive Multiplayer Online Games are, as the name suggests, games that are played by a massive amount of players simultaneously online. Two key features distinguish these games.

- A single gaming world's capacity to accommodate several players at once, typically ranges from 1,000 to 10,000.
- Have an ongoing gaming environment. This implies that the game world is continuously updated and evolved and cannot be reset, paused or halted. Even when some of the players are not present, it will still go on. Furthermore, every choice we make towards characters or objects has a definitive consequence and cannot be undone.

MMOGs originate from minimal text-based games called MUDs (Multi-User Dungeons). The first MUDs appeared in the late 1970s on university networks and later bulletin boards. These were fundamental text-based games because bandwidth and computer capacity were highly constrained. As computer technology advanced throughout the 1980s and 1990s, MMOGs transformed into graphical games with an expanding range of options. Modern MMOGs provide the user with intricate 3D graphics and an extensive virtual world. Although interaction with other players is a massive part of these games, they do not revolve around it solely; computer-controlled characters and other computer-controlled objects often play a significant role, which can be observed in games like World of Warcraft[Games, b] and newer titles like New World[Games, a] or Lost Ark[Smilegate].

The first MUDs were fundamental client-server applications on university mainframes and played using telnet or vt100. The load placed on the servers which run the games rise along with their complexity and potential. Even with the rapid development of faster and more powerful computers, the resources required to run these games are too much to be handled by a single machine. Like early MUDs, modern MMOGs still use client-server topologies, but their servers are large clusters of 10 to 100 state-of-the-art machines.

2.2 Literature Review

MMOGs are a popular study topic because of their complexity and resource requirements. It takes much work to optimise them and create better and more effective ways to split the load among several servers so they can maximise player engagement and enjoyment with the least amount of resources required. New and better operational solutions must be devised as such games continue to draw an increasing number of players, and the expectations for the quality, graphics, and level of interaction they offer continue to rise. Much research has therefore been focused on how to improve the architectures' effectiveness and the load they can handle.

The research in this field can be divided into two primary fields. The first is research being conducted to enhance the present client-server architecture currently employed in most games. This may take the shape of a distributed server architecture that is more effective inter-part communication. The second focuses on creating a peer-to-peer architecture for these games, eliminating the need for a heavy and complex server.

2.2.1 Client Server

Most of the MMOGs developed and brought to the market so far use the classic client-server architecture. There are specific issues with this architecture. In this architecture, the server is responsible for handling the entire load of managing the virtual environment. No single machine can handle this, so complex architectures must be developed to distribute the load over different servers. Therefore, the research in this area is primarily focused on how to distribute the various server functionality over various physical machines efficiently, for example: cloning the virtual world, dividing it into zones, and distributing the various server functionalities (database, login handling, etc.) on different machines. Some examples of previous research are architectures where the world is divided into zones that are dynamically assigned to different servers based on the load they experience [Van Den Bossche et al., 2006], using generic middleware for multiple MMOG projects [Hsiao and Yuan, 2005], [Caltagirone et al., 2002] presented an architecture for the inner workings of both server and client, [Assiotis and Tzanov, 2006] describes an algorithm for handling the interactions between different zones of the virtual world.

2.2.2 Peer to Peer

Developing a peer-to-peer architecture for an MMOG is a hot topic in the scientific world. Numerous articles have been written proposing architectures without a central server where clients communicate with each other directly and are each in charge of a small amount of the game state. The expense of maintaining a game is drastically decreased by removing the requirement for a central server. However, many issues arise when designing a peer-to-peer architecture for MMOGs.

To name a few issues, the distribution of updates and patches will be more difficult since there is not a centralized version of the game which forces every player to update their games making it also more challenging for businesses to use a peer-to-peer game for commercial purposes, it will be harder to stop cheating in the game since there is no governing authority, and the client bandwidth usage will be substantially higher than with the client-server design since each player needs to send and receive updates from every other player. Scientists believe they can counter all these arguments, and articles have been published presenting architectures or parts of architectures that can run peer-to-peer style MMOGs which led to the use of blockchain technologies to create a peer-to-peer game Nine Chronicles [Planetarium].

Research in this area is, for example, a distributed peer-to-peer architecture that guarantees a low latency for users and prevents cheating [GauthierDickey et al., 2004a], a system that can be used to turn classic client server-based games into peer-to-peer based ones [Kaneda et al., 2005] and a publisher/subscriber model is described for developing a distributed MMOG architecture [Fiedler et al., 2002].

2.2.3 Client Server Architecture

This chapter will present a model of the typical client-server architecture used by many of the MMOGs currently available on the market. There will also be a discussion of the advantages and disadvantages of different architectures.

Generic Client Server Architecture

Client-server architectures are not something that has been developed for Massive Multiplayer Games; they have been around for many years and are used in a wide variety of applications. They started to appear in the 1980s when people became aware of the shortcomings of the then-dominant mainframe/terminal designs. In the original two-tier client-server designs, the client computer hosts a program that communicates with the server(usually a database management system). The presentation logic (user interface), the business rules, and the database access were all contained in the client application, also referred to as a fat client. Even when the user interface was unaltered, the client application had to be updated, tested and redistributed whenever the business rules were adjusted.

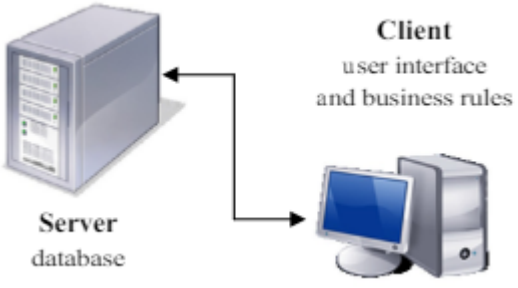

Figure 2.1: Two Tier Architecture

Because of the need to alter the client application every time the business rules are altered, the presentation logic and business rules started to separate, leading to the rise of the Three Tier architecture having that separation as its principle. Due

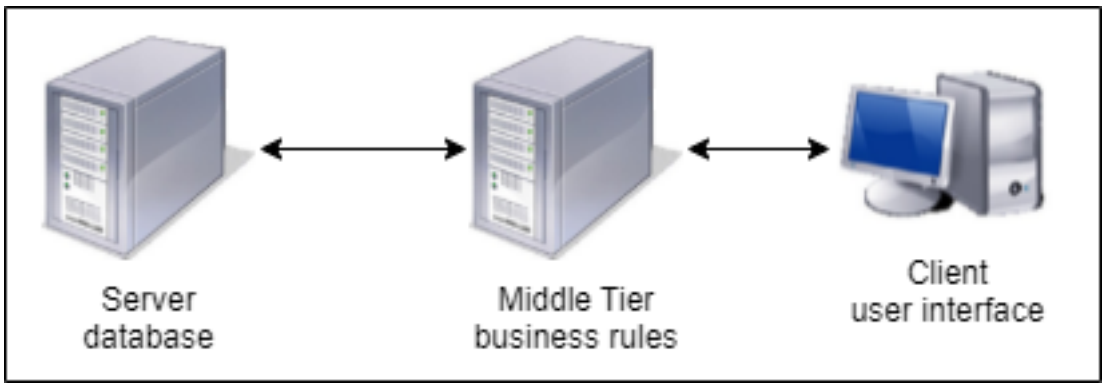

Figure 2.2: Three Tier Architecture

to their lack of business logic, the clients in these multi-tier systems are frequently referred to as thin. The client only communicates with the server to send and receive requests, and it graphically displays information to the user.

One example of this architecture would be the world wide web, in which web pages are stored on servers, and the client only requests and presents them graphically. This is currently the most used architecture throughout various systems, MMO included.

Client Server in MMOs

MMO developers have employed client-server architectures almost exclusively since the creation of the first MMO. The server manages all the rules and the status of the virtual world, and each player connects to it via the client application on their PC. One of the core characteristics of this architecture is that no single machine can handle the volume of clients that will connect to the server and the load they produce. Therefore, a distributed server design will be necessary to distribute the load among several machines.

Using client-server architectures in MMOGs and many other systems has led to much research in the field and the development of an abundance of information regarding how to use them most effectively. This has produced the standard multi-tier architecture in MMOGs nowadays, a four-tier server architecture consisting of a client layer, a proxy layer, an application/game layer and a database layer[Nawaz and Xu, 2014].

Client Layer

A single client represents each player in a MMO. Each participant establishes a connection to the server through a client application on their PC. The client displays the graphics and user interface to the user, notifies the server of all actions

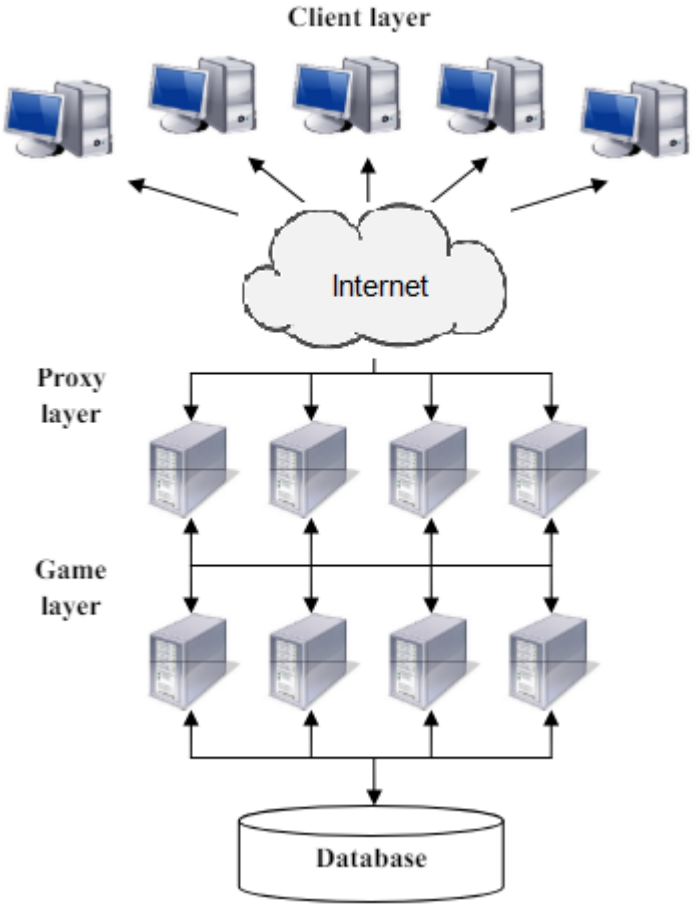

Figure 2.3: Four Tier Architecture

the user wishes to take, and displays any changes to the game environment that it gets from the server. The clients are viewed as thin since they have no control over the business logic of the game world. They only request and receive updates for the virtual world from the server and display them to the user in an appealing graphical manner. Clients are not, however, simple apps as a result. The most recent MMOGs frequently use sophisticated 3D graphic engines and sophisticated strategies to smooth out the game with little client-server interaction. Because both the server's and the client's bandwidth is constrained, interactions between the server and the client are kept to a minimum. The client will have to use complex techniques like motion prediction to present a smooth playing experience even though it receives the world updates only at intervals.

Proxy Layer

The proxy layer serves as a link between the client layer and the game layer. It manages all client connections and ensures all packets are delivered to the proper game layer servers. Additionally, a firewall can be used to address security concerns and ensure that only clients with access to the game can play the game (e.g. on a subscription model game like World of Warcraft, a player who does not renew his subscription loses access to his characters until the subscription gets renewed)

Game Layer

This layer is the game's central component because it contains, manages, and governs the virtual world. requests sent from clients are directed by the proxy layer to the appropriate game layer node. These commands are processed by the game servers, who check if they are valid, compute the changes to the virtual environment, update the game state, and send the updates to all the clients they will affect. The main task of this layer is to keep track of the states of all players and other world objects. All players' positions in the game's universe are managed, and all dynamic elements—including monsters, non-player characters, treasures, and the timer—are also under its control.

The complexity and challenge of MMOGs software architectures lies for a significant part in the efficient distribution of the game server functions over multiple machines, which will be discussed in the next section.

Database Layer

The game's database contains data elements for each action a player takes and every item, object, and PC and NPC agent/actor. However, even the most giant virtual worlds pale in comparison to the data warehouses used by thousands of companies worldwide. Additionally, the game databases' schemas and structures are simpler since most games specifically tailor their databases to have as few transactions as possible. As a result, they can accelerate transaction processing by using massive memory systems(in-memory databases), parallel databases, and other technologies.

It is worth noting that the transaction load that MMO's generate is very specific. The vast majority comprises colossal amounts(and more than 90%) of minimal reads and updates produced by players moving around the world and interacting with objects. They will not require extensive joins on star schemas. The Decision Support Systems (DSS) or Online Transaction processing (OLTP) systems that many organisations utilise are unlike games. The workload in the game is considerably more akin to Lightweight Directory Access Protocol (LDAP) or telephony directory lookup: several little operations across straightforward tables.

The database layer configuration for games is quite simple. This is because relational database technology has been developed and improved for many years for which game developers can create databases which are tailored to their game's needs. The capacity of contemporary relational databases is astounding, and they operate well on systems with multiple CPUs. The game database should ideally be kept entirely in memory. In this case, performance would be excellent, and transaction completion durations would be comparable to a function call. However, doing so would require a system with many terabytes of RAM. Although new database systems are being created, none of the MMO's now available employ them due to their high cost. These systems could theoretically allow for the running of the complete database in memory. However, with the increase in the size of these worlds, developers have not made these database systems their priority.

2.2.4 Techniques to Distribute Load

Although the various server layers can easily be spread across several servers, each layer will still need more resources than a single computer can provide. Since the proxy and database layers have been utilised for many years in many applications, there is abundant information about how to configure them effectively.

Sharding

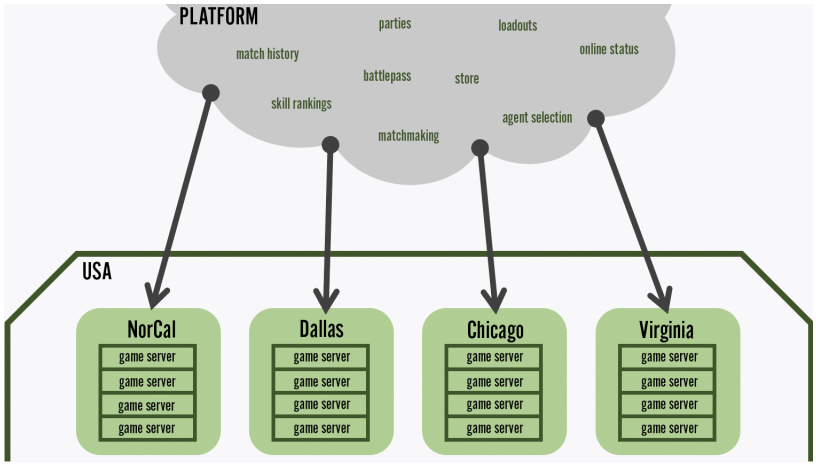

Figure 2.4: Sharding Example

The first technique for distributing server resources across several machines is sharding. Apart from a few exceptions, Massive Multiplayer Games use this in some form. Sharding is the process of running various game instances. There are several players in each copy who can communicate with one another but not with those on the other shards. These copies are referred to by various companies by different names, such as worlds, realms, or shards, but they all refer to the same thing. Each copy of the game may support a maximum number of players, and each copy runs on a distinct server (-cluster). Shards are added as the player count rises and removed as it falls. Reducing the number of players per shard helps reduce server strain. Therefore it makes the game very scalable if it has been designed to be playable using separate rounds, zones or sectors. Because MMOGs strive to be worldwide applications, they will have clients connect to their servers worldwide. This raises the issue of the speed of copper's electrons and the limiting speed of light. There will always be regions with substantial latency while connecting to a server running on a single location, which can seriously impair playability for those customers. In order to prevent this, it is necessary to run multiple instances of the game geographically distributed, ensuring that there is always a site to which anyone can connect with a adequate latency.

Sharding does, however, have the drawback of making the game feel smaller. The players on the various shards are unable to communicate with one another. Therefore, the aim of the massively multiplayer game is defeated if there are

too few participants per shard. Although sharding is commonly employed, each shard still has a prominent enough number of players that the load will be insufficient for a single computer, necessitating the adoption of additional strategies in addition to sharding to disperse the load throughout a cluster of servers.

Zoning

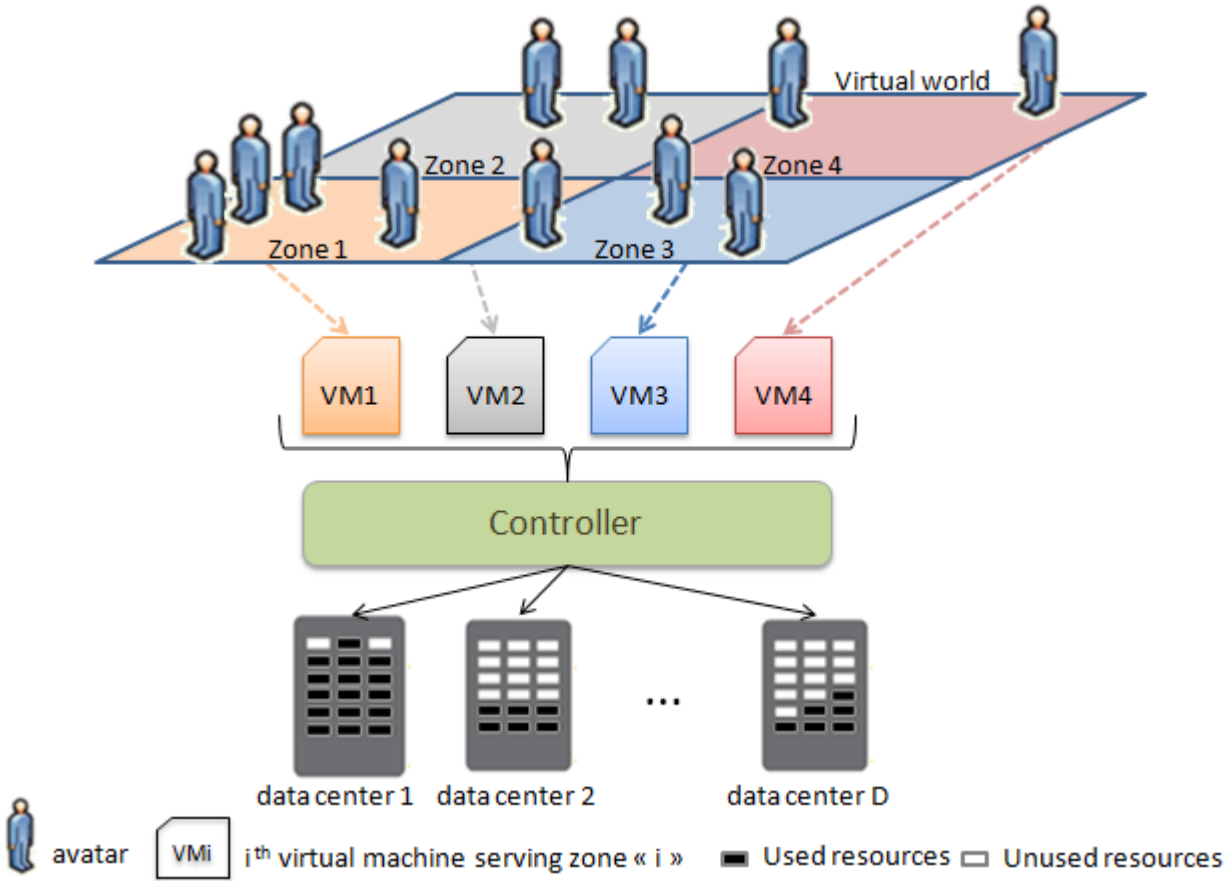

Figure 2.5: Zoning Example[Dhib et al., 2019]

Zoning is the process of creating distinct zones within the game world, each of which is managed by an individual server. In the game world, the zones are frequently divided by geographic boundaries and each zone server manages the activities and upholds the conditions unique to that zone meaning it is unaware of the of the state of the virtual world outside that zone.

The zones can be thought of as several areas of a world where the players are allowed to move around in. The borders in modern MMOGs are frequently invisible to the players. A loading screen appears when a player crosses the border between zones A and B, and they cannot see or interact with one another while doing so. However, it is technically feasible to make the players pratically unaware of the bounds. [Assiotis and Tzanov, 2006] outlines a method to make the player's use of zones fully transparent.As a result, it becomes very difficult to synchronise the various zones across the various servers (players must be able to view and communicate across border zones). Therefore, there is a significant

amount of overhead since all activities occurring in the border zone must be constantly updated on the various zone servers.

Almost all of the MMOGs available today use zones in some capacity. They often use a restricted number of zones with fixed, opaque borders along virtual world borders, such as mountains or water.

It is crucial to predict the load each zone will face while designing the game because zone borders are frequently static in the game world and cannot be changed. When the predictions don't match reality, genuine problems with certain zones being overwhelmed and others barely being used at all will happen. To address this, a system using micro-zones that are dynamically dispersed over a predetermined number of servers based on the demand they experience is suggested in [Van Den Bossche et al., 2006]. However, these zones cannot be too small. Zone transfers(player moves from one zone to another) increase as zones get smaller and these transfers put extra strain on the server since each zone handler must send the player's whole profile to the other meaning the overhead can quickly outweigh the initial benefit of using them as the zones become narrower and there are more transfers.

Cloning

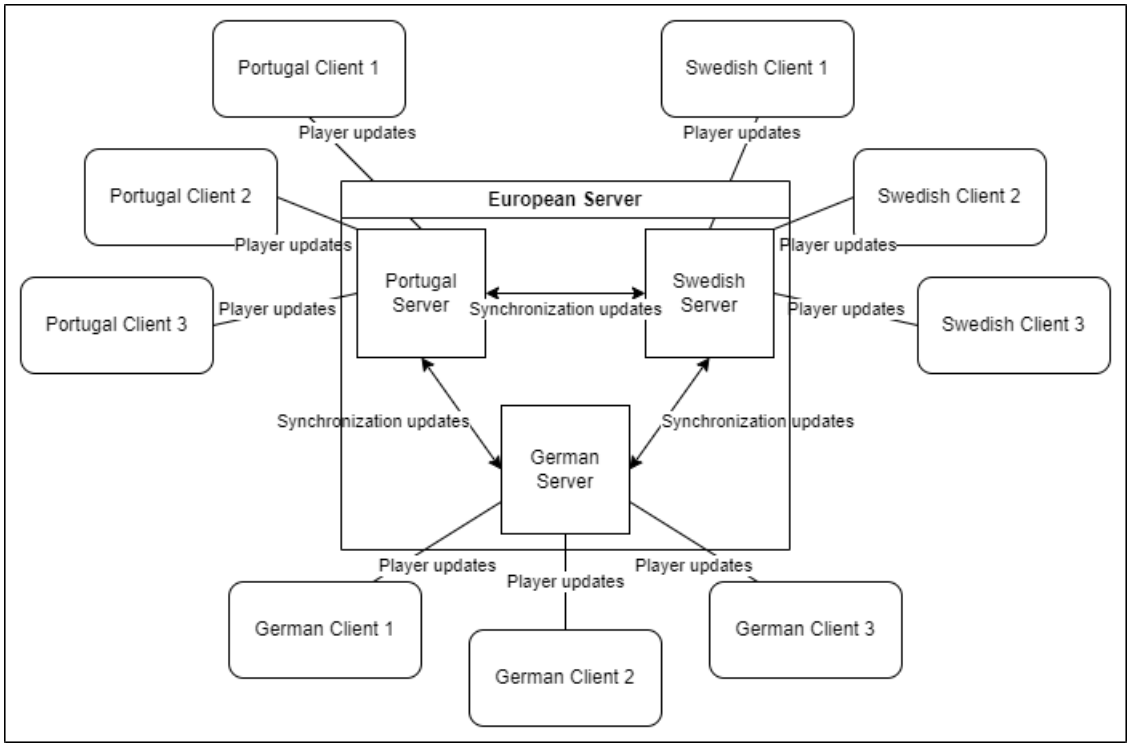

Figure 2.6: Cloning Example

Cloning is the process of simultaneously operating two identical copies of the game on different servers synchronized with one another, giving the user the impression that they are in a single virtual environment.

Cloning allows for a single virtual world to be maintained while reducing the

workload on each server. The drawback is that constant synchronization of the game state between the two servers will significantly increase traffic and CPU burden.

[Cronin et al., 2001] claims to have designed a more efficient system than any other using the cloning technique.

Instancing

Instancing is a combination of zoning and sharding. It implies that different instances of the same zone are possible. An instance is a replica of that particular zone in which only one player, or a small group of players, can communicate with the virtual world and other players within that zone. However, players cannot communicate with each other in the same zone but in a separate instance of that zone. Similar to how shards work, separate versions of the game are made that cannot communicate with one another. Players do not put much strain on the servers because there are only a few people per instance. Therefore multiple instances can be assigned to one zone. The system is very scalable because instances may be dynamically assigned to different servers based on load.

Because you can only interact with a small number of people inside an instance with this method, some of the immersion of being in a massive world of a MMOG is lost. However, it is possible to interact with everyone in the virtual world (out of the instanced zone) as soon as you exit the instanced zone, so an instance effectively operates as a limited or local gameplay context, enabling scalability through multiple parallel contexts. In most games, the virtual world is a single copy where everyone lives, with the exception of essential locations like dungeons. Instancing is also very attractive for playability reasons because a lot of different players can experience the same content at the same time.

2.2.5 Peer-to-Peer Architecture

This section will present a model of the typical peer-to-peer architecture used by many of the MMOGs currently available on the market. There will also be a discussion of the advantages and disadvantages of different architectures.

Generic Peer-to-Peer Architecture

Peer-to-peer network topologies are utilised in a wide range of applications and are growing in popularity. Only peer nodes with similar responsibilities and capabilities make up peer-to-peer architectures. Traditional client-server architectures only allow communication from client to server and vice versa since the server controls most resources, such as computing power and storage space. Clients never interact directly with one another or perform any business logic calculations. In P2P architectures with no centralized server, all resources are located in the peer nodes, and communications are direct between peers. Peer-to-peer ar-

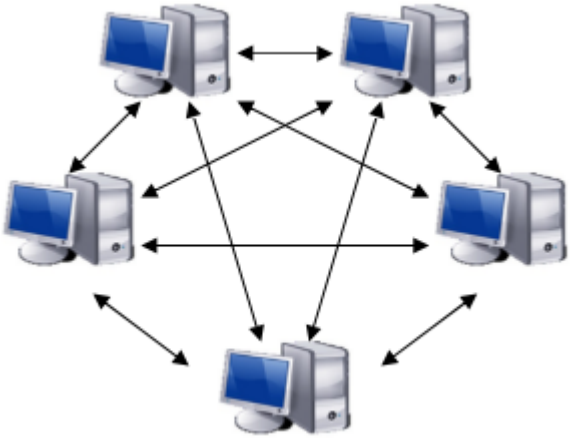

Figure 2.7: Peer-to-Peer Example

chitectures are utilised in a wide variety of contexts and formats. Peer-to-peer file-sharing technologies have grown in popularity since 2001 with the creation of torrent files. This technology allowed users to download files from other users directly. However, because torrent files frequently still need a central server like Piratebay or YTS to give an index for the available files, only the file sharing can be considered peer-to-peer. The system uses a more hybrid architecture.

Peer-to-peer architectures have several benefits. The entire burden on the network is distributed across the nodes due to sharing all resources, including network, bandwidth, storage, and processing power. The network's capacity will also expand due to the additional resources that new nodes provide, in addition to the load they add to the system. Peer-to-peer networks are very scalable as a result.

In client-server designs, since the server is the only component that can fail, the network as a whole also fails. Peer-to-peer networks can be configured so that even if one node fails, the rest keeps running normally, making them more robust than client-server.

Peer-to-Peer in MMOs

Much research has been done on using peer-to-peer style architectures and their implementation in MMOGs [Rutvji Joshi]. However, this has not yet been implemented in any significant MMOG with a few exceptions, such as Nine Chronicles[Planetarium]. Multiplayer games like MiMaze [Baughman and Levine, 2001] and Age of Empires are based on distributed peer-to-peer infrastructures. However, with only eight players able to play simultaneously, Age of Empires cannot be referred to as a Massive Multiplayer Game, and MiMaze is merely a research project.

Because the usage of peer-to-peer architectures in multiplayer games is an ongoing research phenomenon that has yet to reach the mainstream consumer market, there is no single dominant best practice design. Numerous potential architectural designs have been put forth by scholars in articles. However, they all have

different implementations. The majority of these designs continue to use central servers to conduct the necessary account and other administrative tasks and to index the peers currently playing the game at any given time. The peers, however, are responsible for doing all calculations related to the game's rules and preserving its game state.

The benefit of client-server designs is that a single authority controls all simulation aspects, including ordering events, resolving conflicts, and acting as the central store for data. Peer-to-peer designs lack this centralised authority, making presenting all users with a uniform virtual world far more challenging. Since the game state will be shared across all the peers, a successful approach must be devised for allocating various game world components to various peers. Additionally, all other peers on the network must be able to access and maintain consistency with these game world elements. This means that the game state must be the same across all the peers in a specific zone, while in hybrid architectures there's only one peer in charge of preserving the game state. Peers cannot be trusted since they might try to cheat to get an advantage in the game or because they could crash at any time, losing the state of the game for which they were responsible.

2.2.6 Issues in current Peer-to-Peer

Consistency and Synchronization

The game world must be uniform for all players to allow cooperative play and interaction. For instance, if player A opens a chest and removes its contents, player B, arriving at the location later, must find the chest empty. This particular game state is computed and kept by the server in client-server architectures. The benefit of having a single server to manage the game state is lost when adopting a peer-to-peer architecture; instead, each peer computes its local view of the game state using data from the other peers. Because network delays are different for all peers, some sort of synchronization protocol will have to be used to provide a game state as timely and consistent as possible for all peers.

In order to show the viability of their proposed protocol, Diot and Gautier created the game MiMaze [Laurent Gautier]. Their method, bucket synchronization, splits the game time into buckets so that all peers will process actions issued simultaneously. The synchronization mechanism ensures that any actions taken to assess the game's overall status at time t were issued near to t or within the same assessment period. For this reason, a bucket is connected to each evaluation period, and when it is time to compute this bucket, data is received the game will use that to compute the game state. In image 2.8, the action issued by the local peer at t0 would be processed significantly earlier than that issued by a remote peer at t1 (but received at t2), even though both actions were issued simultaneously. By placing actions in buckets to be handled simultaneously, bucket synchronisation delays the completion of the actions. The action issued at time t0 in the example above is combined with the action issued at time t1 in bucket d, and both will be processed simultaneously at time td. This synchronisation should

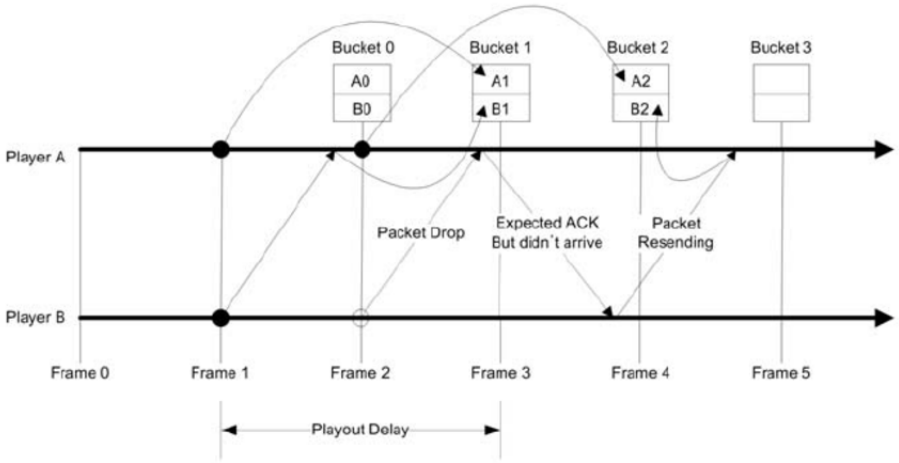

Figure 2.8: Bucket Synchronization [Moon et al., 2023]

cause all peers to show the same game state simultaneously. The receiver calculates the synchronisation delay to decide which bucket the incoming data should be stored in. The time it takes for participants to get information is measured using a global clock. However, the playability would be significantly hampered if the synchronisation delay grows too large, so a maximum delay limit must be established, along with a method to handle actions whose transmission delay exceeds the maximum delay.

Bucket synchronisation does not address the issue of peer cheating. To address this, [Baughman and Levine, 2001] created the lockstep protocol. Lockstep uses rounds for timing, which are divided into two steps:

- Dependably deliver a cryptographic hash of each peer's activity
- Have all peers send the plain-text version of that action.

This demands the peers to make their move while keeping it a secret, preventing anyone from knowing someone else's actions in advance. The effective latency is three times that of the slowest link due to the additional interactions occurring in lockstep. To improve this, the Sliding Pipeline protocol was developed by [Cronin et al., 2003], which adds an adaptive pipeline that allows the player to send out several actions in advance without waiting for acknowledgement packets from the other peers.

Game State synchronization

Four main features are commonly present in the game world of MMOGs: static terrain, uncontrollable objects, player-controlled characters, and non-playable characters (NPCs). The scenery comprises all the static objects and terrain that will not change during the game. It usually belongs to the client programme and does not contribute to the game state that needs to be shared and saved among peers because it will always be the same for everyone. The continuously changing dynamic game state comprises uncontrollable objects, player-controlled characters,

and computer-controlled characters. All players must experience the same game state. However, they are not required to constantly be aware of the status of every item in the game environment since it would consume far too much memory space, computing power and bandwidth to have every player constantly receiving updates on the game state and keeping track of it as a whole. A better method is to spread the state of all the game objects across the many peers so that each item has a peer in charge of maintaining its state. Afterwards, peers who want to engage with that object must ask the peer in charge of it for its current status and submit changes to it.

All the components of the game world will need to be efficiently distributed between the peers using some form of technique to make this happen. [Knutsson et al., 2004] described an architecture in which the game world is divided into smaller regions. Each peer and each area are then given an ID, and the peer with the closest ID match is designated as the region's coordinator. Each object in that zone is under the coordination of the that coordinating peer. One peer carrying the weight of all the objects in an area can be overburdened; to avoid this, objects can be assigned to several peers and given their IDs. The coordinator of a region is unlikely to be a member of the area due to the random mapping, although this has numerous advantages. First, by separating the objects from the players who access them, it lessens the potential for cheating. Second, only when peers join or leave the game does the coordinator of an area need to change instead of every time a peer departs the region. Random mapping also increases robustness by lessening the effect of localised events. For instance, losing the state of the region does not usually happen when repeated disconnections occur in the same area. One issue which [Knutsson et al., 2004] did not cover is what if there are not any or even a minuscule number of peers playing the game? It is conceivable that the resources needed to support this will be greater than what they have available when just a decent number of peers are active and they will have many objects assigned to them. Additionally, there is no saved game state, and all characters and objects in the game environment may lose their current states when no peers are present. This means this system is only feasible if a large amount of peers is guaranteed. These peers, however, can be provided by the developer itself.

Latency

Clients only connect to the server in client-server architectures to send and receive updates; the server manages the game's current state. This means that a player with high latency will transmit and receive updates slowly, which could make the game less playable. The other players' ability to send and receive updates quickly to and from the server does not, however, suffer from this one player's high latency. Since the server manages the player's state and can still change it, interactions with the player with the high latency will not cause significant issues. This creates more of a challenge in a peer-to-peer architecture. With synchronisation methods, as explained in the previous section, the slowest peer controls the game's overall pace. The algorithms can be changed to make the slow peers' consistency a trade-off for the other peers' ability to play. The issue remains, though, since interactions with these items and the character they represent will be de-

layed for all other peers because these slow peers may be in charge of their states in the game environment and in the character they represent.

Fault Tolerance

Unfortunately, peers do not always quit the game in a way that can be controlled. Without transferring their responsibilities and game state to the rest of the network, they can crash or disconnect, which may result in losing some game state components. When a coordinator of a region crashes, for example, the state of that entire region will be lost in the game state distribution system, as stated in the Game State section. To avoid this, a backup system must ensure that every object's state is constantly accessible to several peers. [Ahmed and Shirmohammadi, 2010] presents a model that forms several Peer-to-Peer (P2P) overlays, one for each cluster in each zone and restricts, for example, slowly moving players to be children of fast-moving payers. This means a leaving player only can break routing paths within its cluster, keeping other clusters untouched. In other words, the P2P routing problem faced due to a player disappearing or a change in characteristics is limited to a cluster. Clustering will therefore help stabilize the overlay networks used in zonal MMOGs. This approach categorizes players and creates clusters based on the assumption that the general features or attributes of the players are invariant, i.e., characteristics are invariable. However, a player's attribute might change while the game is running, either temporarily or permanently.

Cheating

Cheating is a fairly common problem that all multiplayer games must address. A single governing authority can stop players from getting an unfair advantage over other players in client-server architectures. Peer-to-peer architectures make this much more difficult.

Three types of cheats were described by [GauthierDickey et al., 2004b]: protocol level cheats, game level cheats, and application level cheats.

- By altering the protocol, such as by changing the contents of packets, protocol level cheats are possible. The most common protocol level cheats are:

  Fix-Delay Cheats: Your outgoing packets should have a defined amount of delay added so that you can respond to other people's activities more quickly than they can to yours.

  Timestamp Cheat: Send incorrect timestamps on your actions to make them seem as though they occurred earlier.

  Suppressed Update Cheat: To avoid being tracked by others, avoid sending updates about your own behaviour while still receiving updates about theirs.

  Inconsistency Cheat: Send various updates to various players in order to throw off the game's world's synchronization.

Collusion Cheat: Players sharing updates to get information from another that they shouldn't have gotten on their own.

These cheats can be prevented by using specific protocols, [GauthierDickey et al., 2004b] describes a possible protocol, however these protocols add delays for the users.

- Game level cheating occurs when the game's rules are broken. For instance, a player might go a long distance from point A to point B in a shorter time than he should've. Due to the lack of a central authority that controls all actions and has the authority to refuse them if necessary, peer-to-peer games are particularly susceptible to this. Additionally, since the peers are in charge of maintaining the objects' conditions, they are free to change them in order to gain an advantage. A good example of this would be opening and looting a container, to then proceed to refill it.

  A possible solution would be something similar to what [Knutsson et al., 2004] described, by assigning objects randomly to peers so that peers have no control over what part of the game state they are responsible for. Another possible solution would be to try and replicate the blockchain's consensus protocols, which implies that all the peers have a replicated state of the game and could then check if the update was legal or not.

- Application level cheating involves changing the game's code. For instance, changing the graphics engine to make walls invisible so that you can quickly find players and items is a frequent trick. However, when compared to other systems, peer-to-peer architectures are not more susceptible to this kind of cheating.

2.2.7 Benefits and Downsides of P2P

In this section the pros and cons regarding Peer-to-Peer architectures will be reviewed.

Benefits

- Flexibility and Scalability. Arguably their biggest advantage. The number of players actively playing the game directly affects the burden on the network and the game. The demand on the network will increase as more players are actively playing the game. In a peer-to-peer design, the network's overall capacity is equal to the sum of all users' capacities. The network should theoretically be able to support an endless number of users if the burden that each new user adds to the system does not exceed the capacity that he brings. Peers' bandwidth resources may be exceeded if they must send updates to an excessive number of other peers.

- Robustness. Peer-to-peer designs can provide a very robust network since the network can continue to function regularly even if one peer node fails. The ability to prevent node crashes from having a large influence on the

network as a whole is a typical benefit of peer to peer designs, however this isn't always true for all of the designs. This implies that the game will always be accessible for MMOGs.

- Low Costs. Peer to peer designs don't require significant infrastructure investments to function. There's no need for any pricey servers or system administrators to run the game. For the game developers, the creation and maintenance processes will essentially be the same as for any single-player game. Users can simply purchase the game software through customary routes of distribution, and updates can be made available. There is no requirement for extra network infrastructure.

Downsides

True Peer-to-peer systems have numerous drawbacks for game operation since there is no central server or authority.

- The lack of a central authority to ensure that the rules of the game are followed is one of the main downsides of peer-to-peer architectures. Each peer in a distributed architecture makes its own decisions, and there is no central authority to verify the acts' legality or find potential cheaters. For instance, players travelling faster than they ought to be able to or obtaining access to places, things, or abilities that they shouldn't be able to. Other measures must be built in peer-to-peer architectures to prevent cheating. Creating cheat-proof protocols and other systems to check or double check users' actions, like software agents, can partially help with this.

- Software complexity. A peer-to-peer based MMOG will have more complex algorithms, possibly harder to maintain or operate. The number of things that can go wrong when creating MMOGs, which are already extremely complicated systems that take years to develop, is substantially increased by using a peer-to-peer architecture. It's possible to claim that a large portion of peer-to-peer's increased complexity results from the fact that it is still relatively new, hence there aren't any best practises in place. While this is undoubtedly true, peer-to-peer designs are fundamentally more difficult than client-server ones because of all the extra problems they must address, including game state consistency and cheating.

- No easy identifiable global game state. In peer-to-peer architectures where the game state is divided among all of the network's peers, no single entity possesses the entire game state, making it significantly more difficult to guarantee that all users always have access to the game state. Corruptions and inconsistencies are can occur when peers are dynamically assigned an object's state. For instance, if an error occurs and two peers believe they are in control of the same object at once and change it in two different ways, the result is two distinct game states. It will be challenging to merge these again without some form of arbitrator to determine which object is the real one.

2.2.8 Client Server versus Peer-to-Peer

A relatively low amount of research is being done into the client-server architectures, which are used by the vast majority of current games, and a relatively high amount into peer-to-peer architectures, which are used by almost no games. The first question that arises is whether or not peer-to-peer architectures will become popular in games and whether or not they will surpass client-server architectures.

Client Server or Peer-to-Peer

Choosing an architecture will be driven not only by technological considerations but also by economic considerations. A single-player game takes nine to twelve months to develop and release compared to two to three years for an MMOG [Hsiao and Yuan, 2005]. Major PC game studios' MMOG projects can have a three-year timeline and hundreds of employees working on one game before it is released. For example, V Rising started development in 2019, was announced in 2021 and was released in early access in 2022. MMOGs are expected to be around for a very long time, during which time they will continue to be developed by fixing bugs and adding new content. Because creating an MMOG requires a significant investment from the development studio, it is logical that the studios would choose to use existing technologies rather than experiment with new ones.
Some of the concerns that studios face when considering Peer-to-Peer architectures for their games are:

- Software uncertainty/complexity. Since no major peer-to-peer MMOG has yet been created utilising this architecture, there are no industrially proven concepts or best practices. This argument only applies to the initial peer-to-peer MMOGs like [Planetarium], but even as best practises start to emerge, the inherent complexity of peer-to-peer architectures will remain higher, increasing the game's development time, cost, and/or bug count.
- Cheating. Cheating affects MMOGs far more than other games since they are continuous games with persistent states. If somebody cheats in a game of checkers he will just win that game, but the benefit of the cheat will not carry over to any of the next games he will play after. When a player in an MMOG acquires an item through cheating, he keeps it for the duration of the game. Inherently, cheating is simpler in games with peer-to-peer infrastructures. Developers will need to invest a lot of resources in trying to stop all the new cheats gamers create.

A balance will need to be found between the advantages of not having to set up and manage a server and its drawbacks (costs, limited scalability, single point of failure). For most development studios, the advantages of peer-to-peer architectures do not currently outweigh the disadvantages. Client-server architectures have some drawbacks, notably financial ones. If enough money is invested in it, it is possible to overcome restricted scalability by having much excess capacity, and it can lessen the effects of a single point of failure by implementing redundancy and ongoing server monitoring and maintenance. Even with the high costs

associated with creating and sustaining MMOGs, it is a very profitable business to capitalise on popular MMOGs. Due to this, there is little commercial motivation to move away from the proven client server concept to the risky peer-to-peer architectures.

It can be observed that the lower the number of players in a game, the more attractive it is to use peer-to-peer instead of client-server. For example, a peer-to-peer architecture might be more appealing for indie developers new to multiplayer games. For a non scalable infrastructure, when the number of consumers is large, the cost of maintaining the server's infrastructure does not amount to much, but as the number of players decreases, the cost remains the same, making it harder and harder to keep the infrastructure online. Additionally, since there are fewer players, interest management will not be necessary. Synchronisation will not be as complex because there are fewer players to synchronise with, and it is less costlier to maintain track of everyone else in the game to spot and stop cheating. Because of this, Peer-to-peer solutions are becoming more common for multiplayer games with 16 to 64 players. Nearly all of the advantages of peer-to-peer architectures (low or no expenses to run a server, high scalability, high robustness/availability) apply in these games that typically do not carry a game state over from one session to the next. Conversely, the disadvantages do not apply as much because these games are relatively simple. Cheating is less of a problem because it is not carried over from game to game. These games already use a one-time purchase business model. A few games that became popular over the last year which prove this are, for example, Stardew Valley, Terraria, and Core Keeper.

It remains to be seen if this progress will apply to massively multiplayer online games. There will be a massive increase in software complexity as the number of players rises. However, given the current popularity of client-server MMOGs, there is little incentive for the big development studios to switch since the reduction of infrastructure costs in the long term does not outweight the lack of evidence of peer-to-peer working. However, researchers and small development studios will likely keep working on peer-to-peer technology, producing little MMOGs with a small player base.

This open up the potential for possible hybrid solutions. With the use of something close to a fat client the client can be self-updating while maintaining the game logic but the updates will occur as responses from a server, meaning the server would act like a proxy distributing messages.

Chapter 3

Objectives and Methodology

3.1 Science4Pandemics Context

The Science4Pandemics' game has as its core objective to provide a learning opportunity about diverse pandemics, by letting the player manage manage actions during epidemic situations - Control and Mitigate the disease - without "crashing" society or the economy. Players will have to try and keep the population safe both from a health point of view while managing social, economic and political considerations.

The game is presented in single-player and Multiplayer modes in order.
The Single-player mode will introduce the player to the game, the principles he or she needs to learn to play, and basic concepts and actions for dealing with pandemics. This can be seen as a campaign or story mode where the player can explore and collect a variety of achievements or as a multiplayer training mode. The player will go through two stages of learning in the single-player mode: understanding the ideas and the interconnectedness between the regions.

To be able to start playing multiplayer games, players have to first play the single player mode. It is not necessary that they unlock every region, but it will be required that at least they have passed the phase of learning the basic mechanics of the game.

Multiplayer will work as a challenge mode. Players can form teams and play the multiplayer challenge where the team will be faced with a network with several regions opened, each a specific scenario, and will have to work together to reach the stipulated goal or solve the pandemic faster. These goals can vary from match to match.

Players will be able to share resources (buy/sell/donate) between them. There won't be a specific region assigned to each player. The players will start the challenge in a randomly assigned region that they will have to manage. A player will only be able to move/manage another region if that region doesn't have a player assigned.

Players can leave/quit the challenge at any time. If one or more players leave the

challenge before it is finished, the remaining players can keep playing to try to win. Although the task might prove even more difficult if not impossible.

If a player is AFK (away from keyboard, stops playing for some time), he/she will remain in the game until he/she comes back, quits, gets disconnected or the challenge ends. If all players leave the challenge it counts as a loss.

Players can engage in the multiplayer challenge without a team already formed. In that case, the system will look for other solo players and form a team.

Players can communicate through an in-game channel and the multiplier challenge is won and ended by achieving a stipulated goal.

The single-player game's mechanics will be included in the multiplayer, but there will also be an added coordination difficulty. The "small matches" it contains increase the variety of possibilities (geographical regions, networks, and diseases). This is the last stage of learning: realising how interdependent players are and learning how to work together to stop the epidemic.

Game Overview

Initially one region will be opened for the player as the game begins. When the player reaches the necessary experience level (has attained a specific achievement, number of achievements, or combination of achievements) or has played for a predetermined amount of time, further adjacent regions will be unlocked individually. Up to 10 regions can be managed and unlocked by the player.

The player's inputs consist of actions. These actions are different in each building and trigger results which influence the player's progression through the game, this progression can be the possibility of events being triggered or unlocking new action on other buildings.
Different diseases might be present in the player's network. However, for the first iteration, we are not considering the possibility of having two diseases present simultaneously in one area. This is primarily because the simulation is complex, and it is difficult to establish a scientific basis, but it could be something to investigate in the future.

Each region has a **population** with a set startup number; births are not considered throughout the simulation. It also has a set **budget** and a time since the first event that has been simulated. The player will be able to carry out many tasks inside each region, but they will cost money (**budget**) and require **time** (days). Each participant will be given a daily action/time budget. This will serve as a de-escalation measure and encourage rationalisation of the course of action, evaluation of alternatives, and coordination with others.

The **budget** periodically grows by a determined amount and by responding to infection points and donations the player's choices and the geo-socio-economic scenario's status determine how much is available. Additionally, there are some hypothetical situations and occurrences that could change the budget (for example, keeping the disease under control for a specific amount of time results in

a bonus; failing to stabilise the situation and running low on funds can cause an emergency fund to be activated; some characters may donate money; "third-party" interventions that need funding).

The **simulated time** will update in discrete units (days), but it will run continuously. The time shown is a simulation of the time since the initial case. Since actions take time to perform, they all have various completion times. By taking acts that improve the condition of the population as a whole, players can "gain time" for action and delay critical situations. The effects of those actions "buy" time for the player before the situation becomes unmanageable (usually by running out of funds).

The player's actions are related to information gathering, decisions and interventions, and resource management. When the player enters a region to start to play, s/he will go to the "Region View" screen. There, the player can start by gathering information on the current situation and disease affecting the region with the help of his/her team of characters/advisors. When the player begins to understand what is going on, s/he can choose to suggest or implement preventive, surveillance, or mitigation actions (for example, declare a lockdown or sponsor testing). These actions and suggestions will be followed to some extent (depending on fictitious public confidence in the accuracy of information and government action, as represented here by the player). The simulation's calculation will be affected by these activities or measures.

Equipment (hospital, research centre, testing centre, vaccine production) will be associated with various socio-economic regions. The basic equipment/buildings are as follows: research lab, hospital, government, media, and bank. A region will not have the resources needed to create every piece of equipment immediately. According to its geographical and economic criteria, only a handful will be possible. The player will have the option of creating, acquiring, or upgrading more equipment. Other equipment that can be unlocked or purchased includes testing facilities, sewage/water treatment centres, and vaccination centres.

When the player selects an equipment/building, s/he can perform activities such as buy/sell/share resources. The player can upgrade the facilities for the hospital, laboratory, and factory (hire specialists or boost capacity). Some indicators relate to the condition of the structure or equipment. The availability of people and supplies (such as doctors, hospital beds, lab technicians, scientists, lab materials, medical materials, tests, and vaccines) will impact the equipment's performance on relevant variables or indicators like recuperation rate, propagation rate, treatment rate, economic stress, disinformation, fear, and trust. The simulation processing will be affected appropriately by resource availability.

Action Analysis

The player will be exposed to several basic concepts related to infectious diseases and prevention measures throughout the game. Introducing these concepts and information, together with the game's action possibilities, will create learning opportunities for the player. Every action performed by the player(player actions

only, phenomenons like breakouts and such are not included) should be saved in some form of database. For Science4Pandemic's game an action table was devised with specified action codes for easier development. This table contains **AgentInteraction**, **#Execute**, **#Cancel**, **Action CODE**, **Action**, **Effect (why?)**, **PRE (requirements)**, **POST (consequences)**, **ecomodel (effects)**, **impact estim**, **sinais de actividade**, **mentorsuggestion**, **Description**, **informaCant**, **informa(V,X)**, **1st-feedback**, **2ndfeedback**, and **endfeedback**. The action table looks like the following table:

Agent	#Execute	#Cancel	Action
Research	0	1	ResearchDisease
Hospital	2	3	ResearchDiseaseClinicalCharacteristics
Research	4	5	ResearchDiseaseMeasures
Research	6	7	ResearchDiagnosticTool
Hospital	8	9	ResearchTreatments
Government	10	11	TestingContactTracing
Research	12	13	ResearchVaccine
Gov/MinSaude	14	15	CommunityConfinement
Gov/MinSaude	16	17	MandatoryQuarantine
Communication	18	19	SocialDistancing
Communication	20	21	PromoteMaskUse
Communication	22	23	VoluntaryIsolation
Gov/MinSaude	24	25	HandSanitizationAlcoholSanitizer
Gov/MinSaude	26	27	MandatoryMaskUse
Gov/MinSaude	28	29	UseFullPPE
Communication	30	31	SecureWaterSource
Gov/PublicWorks	32	33	WashProgram
Gov/MinSaude	34	35	FollowUpDiagnosedCases
Communication	36	37	UseCondom
Communication	38	39	UseImpregnatedNets
Communication	40	41	UseInsecticideSpray
Communication	42	43	Handwashing
Communication	44	45	VoluntaryCounselingTesting

Gov/Vaccination Center	46	47	—
Gov/Vaccination Center	48	49	—
Communication	50	51	VaccinePromotion
Gov/MinSaude	52	53	FreeMasksDistribution
Hospital	54	55	—
Hospital	56	57	—
Hospital	58	59	TemporaryHospitalExpansion
Laboratory	60	61	ProduceVaccine
Laboratory	62	63	UpgradeResearchSpeed
Laboratory	64	65	VaccineProductionUpgrade
Communication	66	67	RecommendBehaviours
Hospital	68	69	HospitalizationComparticipation
VaccinationCentre	70	71	Vaccination
Bank	72	73	RequestLoan
Public Construction Dept.	74	75	BuildEquipment - Factory
Public Construction Dept.	76	77	BuildEquipment - Water Treatment Station
Public Construction Dept.	78	79	BuildEquipment - Diagnostic Centre
Public Construction Dept.	80	81	BuildEquipment - Vaccination Centre
Public Construction Dept.	82	83	BuildEquipment - Hospital
Public Construction Dept.	84	85	BuildEquipment - Laboratory
Public Construction Dept.	86	87	BuildEquipment - Media Station
Public Construction Dept.	88	89	BuildEquipment - Bank
Public Construction Dept.	90	91	BuildEquipment - Government
Public Construction Dept.	92	93	BuildEquipment - Public Constr. Dept
Government	94	95	CreatePublicConstructionDept
Hospital	96	97	Minigame - Symptoms
DiagnosticCentre	98	99	Minigame - Test and Quarantine
Communication	100	101	Minigame - Fake News
VaccinationCentre	102	103	Minigame - Vaccination

Win/Lose condition

Since the game is based on a continuous simulated phenomenon, the player cannot really "win" it or put a stop to it; they can only try to manage or restore a balanced state. By progressing in the game and achieving a particular goal (such as a low infection rate), s/he can "win". Gaining levels can open up new playable zones or let the player acquire achievements. When a critical proportion of infected is reached, the player can "lose" the game because the game will then be

in a impossible to reverse with the available resources. A predetermined set of circumstances determines this state. A table was devised to make it easier for developer to add new achievement or milestones to the game, which looks like the following table:

3.2 Objectives

The main goal of this this dissertation project is to transform an almost complete game running in single player mode into a multiplayer game. The game needs to have action logs to keep track and perform analysis on the player's behaviour, and it also needs to be converted to multiplayer and deployed in about three months. With this in mind, the problem is in designing a multiplayer support to changing the current single-player logic to multiplayer logic in an economic friendly way. Considering that these developers had workload difficulties if they had to implement a backend server to run the game as a multiplayer game, the solution must be developer friendly, meaning that it should not force new concepts on the developers, increasing the time it would take to develop the rest of the game.

3.3 Methodology

Because the single player version of the game has been in development for over a year, there were multiple occasions where meetings with the lead developer for the game were needed. These meetings involved redesigning game logic, implementing the new logic as well as developing parts of the game which required the multiplayer module to be developed first. Hence, the best methodology to use when implementing a multiplayer module in an already existing single player solution is Agile, and in the case of Science4Pandemic's game, its Crystal variant(appropriate for 5 to 8-person teams).

Crystal is one of the most versatile frameworks, allowing the team to develop their processes freely. As a result, communication is a crucial factor. It places much more emphasis on people and how they connect than on the procedure or the tools. It is the best-suited methodology for this project since the development team comprises five elements (including the author). It emphasises concepts like People, Interactions, Community, Skills, Talent, and Communication to produce the best software development process possible. The interaction and synergy between the individuals assigned to the projects and processes—essential for the project's efficiency—are at the heart of this development process.

Achievement	Condition	Message	Reward	Budget Up
First on activity Research, Measure, Campaign Upgrade/expansion, Production, Travel	every first time	"Congratulations! You achieved X." Inform of XP	XP +10	—
Knowledge Leader	I_quadro, I_sintomas, I_measures,	"Congratulations! The new knowledge helps us identify new cases and take preventive measures." Inform of XP	XP +20	—
Lead Researcher	I_testkit, I_treatment,	"Congratulations! New tools for testing, isolation and treatment of cases will be an important way to contain the disease and reduce social and economic impact." Inform of XP	XP +50	Unlocks Hospital Expansion
The Doctor	1/1000 people cured (10M=>10.000)	"Congratulations! Our medical staff has now helped cure 10 000 people (under normal hospital capacity)."	XP +30	budget+500k
The Minister	3 combined measures; reduction of infections in the last 7 days	"Congratulations! You adopted combined measures that helped reduce propagation." Inform of XP	XP +30	—
Chief Scientist	I_vaccine R_Vac_factory	"Congratulations! You now have the vaccine and can begin production to satisfy our needs. Consider promoting vaccine adoption for maximum effect." Inform of XP	XP +50	budget+1M unlocks Vaccination Center
Vaccine Champion	V > 66%	"Congratulations! We now vaccinated more than 66% of the population. This will significantly reduce the effects of the disease."	XP +100	budget+1M
The Container	R0 below 1.0 for 50 days straight?	"Congratulations! Your combined measures kept R0 under 1.0 for 50 days, which reduced propagation." Inform of XP	XP +50	budget+1M
The Cleaner	cases below 100? during 100 days?	"Congratulations! Your combined measures kept new daily cases to X<100 over the last 100 days." Inform of XP	XP +100	—
The Philanthrope	spent 5M	"Congratulations! Your dedication to take action is admirable. You've spent half your initial budget!" Inform of XP	XP +100	budget open Bank loan (1M at 1% each 30 days)
The Global Citizen	acted on all regions	"You have now visited and helped on all regions. You are now perceived as a true global citizen, able to inspire global support."	XP +200	budget +5M

Table 3.1: Achievement List

Advantages

- Crystal requires frequent deliveries, in order to identify eventual problems at every stage;
- There is always space to improve characteristics, taking some time from software development and allowing for discussions about how to perfect processes;
- Facilitates closer communication within teams and promotes interaction and knowledge-sharing between team members;
- Requires a technical environment with automated tests, configuration management and frequent integration.

Disadvantages

- The fact that there are variants in the methodology family means that the principles might vary with the size of the team and the size of the project, resulting in projects that might not be so straightforward;
- It might not work best for geographically scattered teams, because of the constant need to communicate and reflect;
- Planning and development are not dependent on requirements;
- It is ideal for experienced, autonomous teams.

3.4 Requirements

Requirement 1: Web Compatibility

Because the single player version of the game has been developed under the context of being a web app, the backend should be compatible with web browsers

Requirement 2: User Friendly

Because the previous developers do not have experience with implementing backend connections on their code, the implementation should not force the developers to refactor their code, it should allow the same code to be used to treat player actions, when coupled with a server call and response.

Requirement 3: Backend Functionality and Endpoints

The player actions and gameplay mechanics should be processed through a set of RESTful endpoints.

Requirement 4: Multiplayer Functionality

The game should be able to support multiple connections at once, spread them amongst servers/lobbies and guarantee Real-Time updates

Requirement 5: Publish Subscribe Integration

A publish subscribe model should be used to fragment the whole server into lobby topics, allowing the backend server to create multiple processes, each of these being responsible for their own topic and players.

Requirement 6: Action Logging

Every action needs to be registered so that the progress, rythm and paths that players choose can be analysed.

Requirement 7: Documentation

Documentation must be written for the frontend developers, explaining how to make API calls, handle responses, and integrate backend functionality into the game. It should also provide guidelines on how to start the frontend-backend communication.

3.5 Non-Functional Requirements

Scenario 1 - Scalability

Stimulus: All the valid requests the server receives should be processed.

Source of the Stimulus: Increase in Concurrent Users (CCU)

Artifact: Processes handling the requests and propagating the messages for the respective individual game processes.

Environment: System runtime (normal operation)

Response:

- The number of processes handling Lobby Servers is increased to cope with the load.

Response Measure: Maximum number of CCU.

Scenario 2 - Reusability

Stimulus: The multiplayer integration must be easy to implement on the platforms of the already developed single player games.

Source of the Stimulus: Wanting to swap from a single player solution to a multiplayer solution.

Artifact: System

Environment: System runtime (normal operation)

Response:

- Regardless of the development time of the game, the change to multiplayer should be easy.

Response Measure: Number of supported devices.

3.6 Risk Management

Levels:

- Impact:
  - Medium (reach the proposed solution with extra work)
  - Critical(reach the proposed solution with extra hours and cut requirements)
  - Catastrophic (can't reach the proposed solution)
- Probability
  - Low (<30%)
  - Medium (<60%)
  - High (<80%)
  - Certain (>80%)
- Timeframe
  - Short (2 weeks)
  - Medium (1 month)
  - Long (2 months)

Four levels have been chosen so that an intermediate level can be achieved, there's no range in the 50%, so we must choose the intermediate levels.

Certain	Medium	High	Critical	Critical
High	Medium	High	Critical	Critical
Medium	Low	Medium	High	High
Low	Low	Low	Medium	High
Probability / Impact	**Little**	**Medium**	**Critical**	**Catastrophic**

Figure 3.1: Risk Exposure Matrix

1. The entity responsible for the game stops funding the game's development.
   (a) Impact: Critical
   (b) Probability: Low
   (c) Timeframe: Long
   (d) Mitigation Plan:
      i. Create a generic solution instead.
2. The developers responsible for the game so far leave the project.
   (a) Impact: Low
   (b) Probability: Low
   (c) Timeframe: Medium
   (d) Mitigation Plan:
      i. Make sure the latest version of the game can make use of the back-end server to swap to multiplayer.
3. The current implementation makes the integration of the multiplayer module impossible.
   (a) Impact: High
   (b) Probability: Low
   (c) Timeframe: Long
   (d) Mitigation Plan:
      i. Restructure the multiplayer module and adapt the single player code to allow for multiplayer integration.
4. Difficulties on familiarization with the multiplayer module which lead to development delays.
   (a) Impact: Medium
   (b) Probability: Low
   (c) Timeframe: Medium
   (d) Mitigation Plan:
      i. Accompany the developers and help them understand what should and should not be done when integrating the multiplayer module (pair-programming).

5. There's a fault on the concept which hinders performance.
    (a) Impact: High
    (b) Probability: Low
    (c) Timeframe: Medium
    (d) Mitigation Plan:
        i. Understand what part of the concept it hindering the performance and re-design the concept accordingly.

3.7 Work Plan

The work developed within the scope of this thesis encompasses one semester.

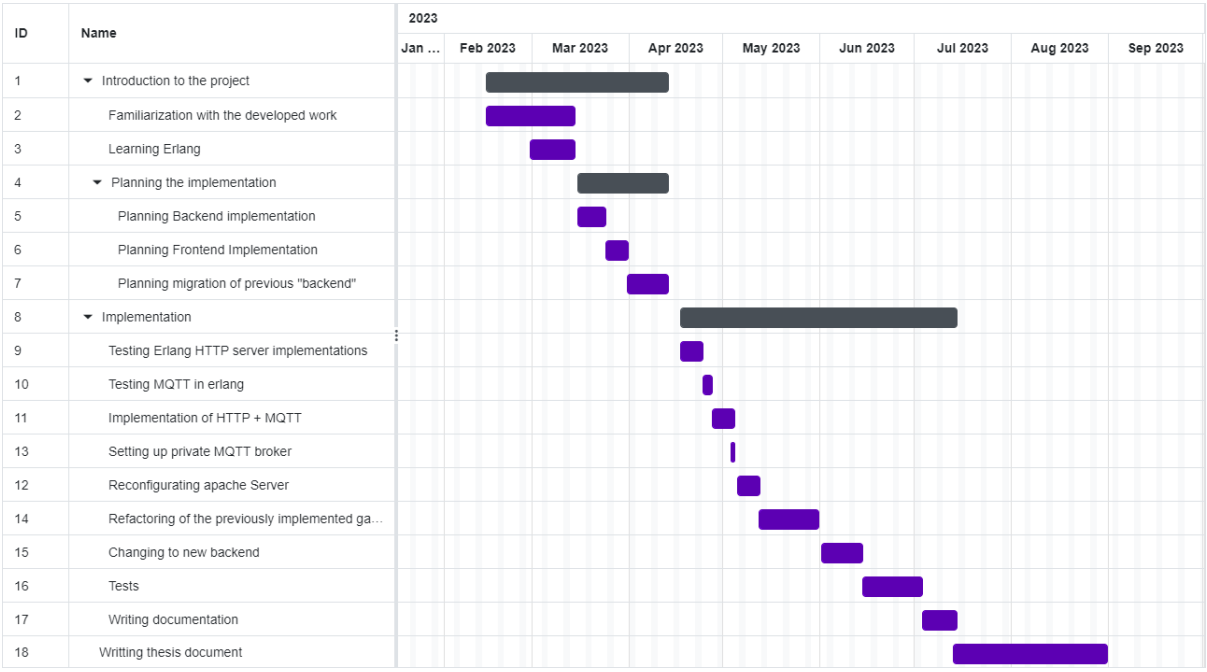

Figure 3.2: Work Plan

Chapter 4

Design Proposal

4.1 Quality Attributes

A centralized server is needed to create a functional multiplayer and have actions logged for further analysis. This, however, would force the developers to change their code to have a server running as their world which could potentially be detrimental if the machine hosting the server was not good enough. To avoid this, the client should be close to a fat client, processing all the game logic and self-updating.

Quality attributes, also known as non-functional requirements or system qualities, play a pivotal role in shaping the overall performance, usability, and effectiveness of software systems. In the backend servers which place strong emphasis on scalability and reusability, several critical quality attributes come into focus. These attributes contribute to the full evaluation of the system's capabilities beyond its functional aspects. In this section, the prominent quality attributes that underpin the design, implementation, and deployment of the backend server are looked at.

4.1.1 Scalability

Scalability is a key feature of the Science4Pandemic's backend server's architecture. It defines the system's ability to handle increasing loads while maintaining acceptable performance levels. Within our development framework, scalability is evaluated through:

- **Horizontal Scaling:** The backend server's capability to distribute load across multiple nodes or instances, thus accommodating growing user demands seamlessly.

- **Vertical Scaling:** The server's capacity to efficiently utilize available resources within a single node to cope with increased load.

4.1.2 Reusability

Reusability, a quality attribute often intertwined with modularity and adaptability, is paramount in our backend server's design philosophy. It contributes to the efficient transfer of knowledge and code across projects and contexts. Our reusability evaluation encompasses:

- **Modular Design:** The creation of well-defined, self-contained modules that can be easily extracted, reused, and integrated into different projects without extensive modification.
- **Adaptability:** Ensuring that reusable components can be adapted to fit varying scenarios without compromising their integrity or functionality.
- **Documentation and Guidelines:** Providing comprehensive documentation and guidelines that aid developers in understanding, integrating, and extending reusable components.

4.1.3 Reliability

Reliability addresses the system's ability to consistently deliver intended functionality, maintain data integrity, and uphold availability. For our backend server, reliability is evaluated through:

- **Fault Tolerance:** Designing the system to gracefully handle failures, crashes, and network disruptions without significant impact on user experience.
- **Error Handling:** Implementing robust error-handling mechanisms to minimize unexpected failures and ensure graceful degradation.
- **Availability Metrics:** Measuring metrics such as uptime and downtime to verify that the backend server meets its availability objectives.

4.2 Architecture

This section is meant to showcase the architecture of the backend server which will be developed. The architecture is detailed using the C4 Model, created by Simon Brown [Brown]. This model was chosen since it provides all the pertinent information required to clearly and unambiguously explain the system architecture.

4.2.1 Level 1 - Context Diagram

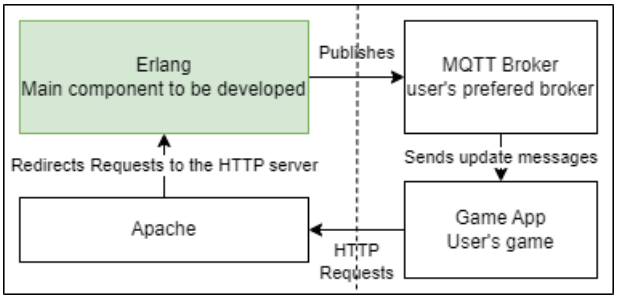

Figure 4.1: System Context Diagram

Starting by the first level (C1), we create the **context diagram** (Figure 3.7), which presents the component to be developed in the **Erlang** runtime environment and the components with whom it interacts, allowing us to view a higher level representation of the component's environment.
The Logic is inspired on peer-to-peer connections and fat clients, by using something similar to a fat client the logic can still be processed on the client side whilst the validations can be done on the server side. By maintaining the validations on the server side, there's a global authority which can prevent cheating. On the right are the **MQTT Broker** and the **Game Application**. These are the broker that the user decides to use and the game application(game being developed). In the case of this thesis, the broker is a private broker hosted by the project's team, and the game is being developed as a web app. The **Apache** receives requests from the Game App and redirects them to the **Erlang** component's endpoints. The **Erlang** component to be developed needs to process these requests and publishes whatever is needed on the **MQTT Broker**'s Topic. These requests are triggered by actions, which currently occur on the following logic:
There's other actions available to the player but this is the rudimentary logic

Figure 4.2: Single player Action Logic

for the actions which will need to be logged after. Logging the actions occurs whenever an action triggers a request which is sent to the endpoint responsible for saving these actions. Actions which do not need to be logged are usually triggered by the current Host client, and they server as updates for the other clients to maintain the same updated game state. For this to be validated, the Erlang component keeps a JSON formatted String which corresponds to the latest game state sent by the host.

The multiplayer logic is as following:

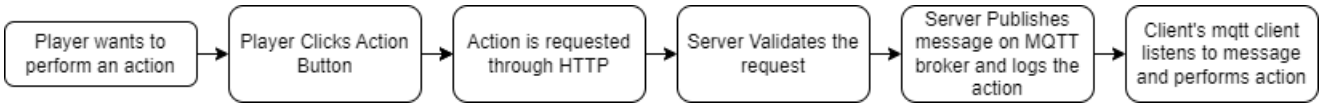

Figure 4.3: Multiplayer Action Logic

4.2.2 Level 2 - Container Diagram

Continuing to explore the architecture, the second level C2 can be reached, where the **container diagram** is presented (Figure 3.8), which enables a deeper comprehension of the Erlang server.

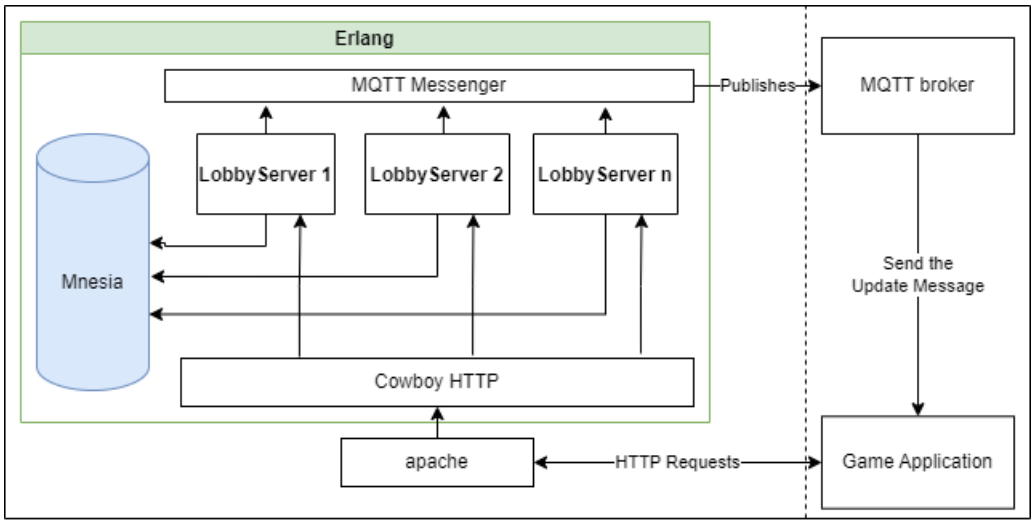

Figure 4.4: Container Diagram

C2 splits the component into multiple containers. The **Erlang** contains his **Cowbow HTTP** container, which refers to Erlang's HTTP server and all of the developed endpoints, which will be described later. It also contains multiple **Lobby Server**, the system's main component, allowing validations to occur and holding the player data. Each **Lobby Server** is responsible for a game instance. Besides this, there is also the **Mnesia** component. **Mnesia** is Erlang's local database which can be accessed from any Erlang process. For this system, Mnesia is used as an embedded database for data persistency. Finally, the component **MQTT Messenger** exists to receive messages from the lobby servers and publish them on the MQTT broker so that the **Game Application** can listen to these messages and change the game state accordingly.

4.2.3 Level 3 - Component Diagram

Reaching the third level (C3) of the architecture, the **component diagram** is presented for the first version of the Lobby Server component (Figure 3.9), where the inner components can be seen. It consists of three main components: **Game Session**, **Session Heartbeat Monitor** and **MQTT Messenger**.

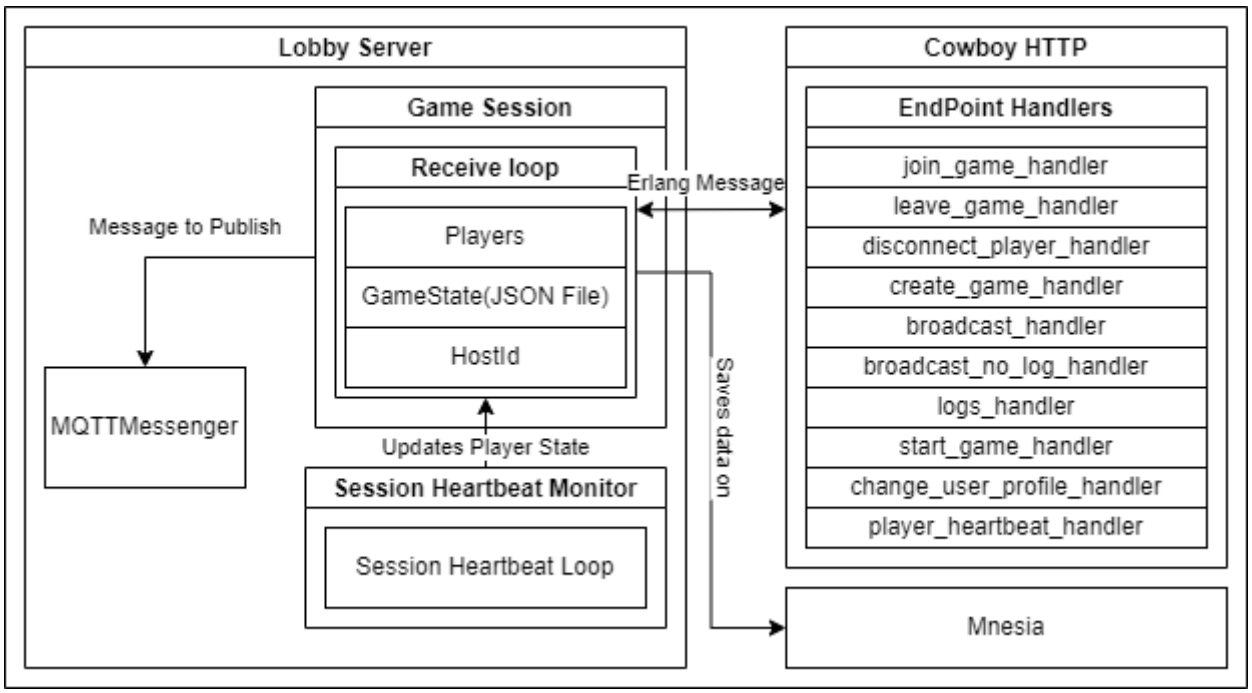

Figure 4.5: Lobby Server Component Diagram

We can go back to Erlang's behaviour to understand why only three components exist. Erlang uses lightweight execution threads referred to as processes. These operate simultaneously and are isolated from each other. However, they communicate with each other via messages. This means a process will die if not stuck on a persistent loop waiting for messages. Erlang's solution is a generic server model, meaning that a loop is running, which listens to messages from other processes and handles the messages accordingly. With this logic, it is possible to create a system that resembles the typical lobby system used in games (See figure 3.10). For Science4Pandemic's game this is ideal considering the maximum amount of players is 4 per game.

Game Session's main objective is to achieve a lobby-like system. It holds the player states, the last Game State as a JSON file (in case the player disconnects, this is a safe way not to lose all the progress) and the current HostId as variables which are changed according to the messages the process receives. This system, however, needs something to constantly manage the player state, and something to constantly check is connections were not severed, which is why **Session Heartbeat Monitor** is needed. Its primary purpose is to send messages to **Game Session** on an interval declaring that the players need to send an update. This guarantees that the player is still online. If the player misses two updates in a row, the player is considered disconnected. It's created simultaneously with a **Game Session**. **Session Heartbeat Monitor** is created when a **Game Session** is created, and it dies whenever the respective **Game Session** is considered finished.

Cowboy HTTP server has numerous processes to handle HTTP requests. Each one of these processes sends a specific message to **Game Session**, which handles them and returns a response to the respective handler with an error message or a success message containing the responses listed in the documentation. The

Figure 4.6: Age Of Empires Lobby [Studios]

desired solution uses HTTP to deliver private updates e.g. a response with the broker ID for the client to connect to, and MQTT for messages that need to be broadcasted, e.g. A player takes an action and every other player needs to be informed of this action to update their game state. The handler might or might not respond to the client depending on the client's desired endpoint. This is because the need for a response only exists on private updates. If a message needs to be sent to update every other client, the message is posted on the MQTT Messenger instead. If these are player actions in the game, they are also saved on Mnesia.

4.2.4 Publish Subscribe model

In this model, one device called the publisher delivers messages to any other device that is interested in receiving them, which might be a single sensor or another form of internet-connected machine like a server.
Subscribers are the devices that request data from the publisher, and they respond with an acknowledgement if they successfully received it.
There is no direct communication between the system's publishers and subscribers. An intermediary, known as a broker, manages the communication between the two parties by filtering all incoming messages and sending them to the appropriate subscribers.

4.2.5 MQTT

MQTT is an OASIS standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and

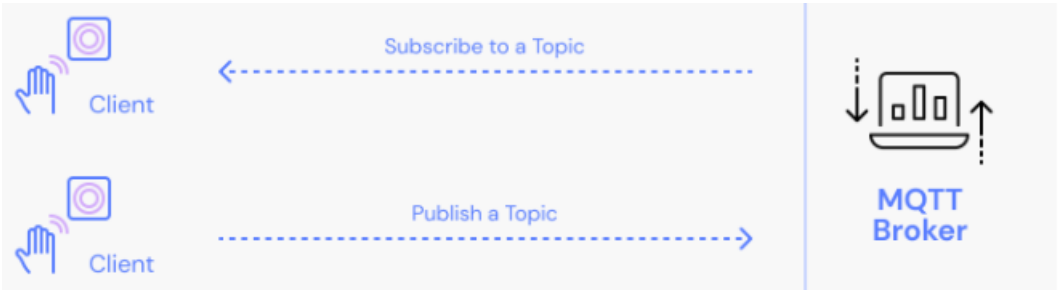

Figure 4.7: Publish Subscribe Example

minimal network bandwidth. MQTT today is used in a wide variety of industries, such as automotive, manufacturing, telecommunications, oil and gas, etc [Dr Andy Stanford].

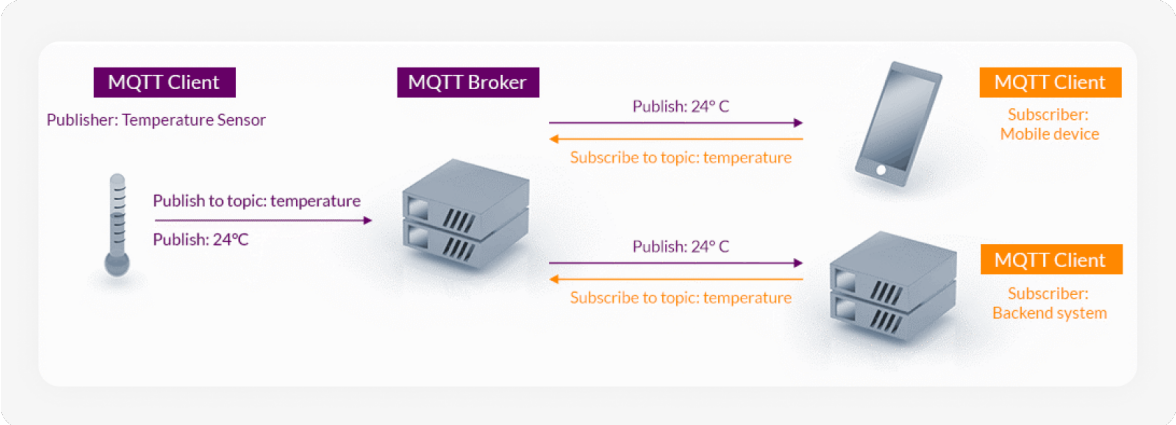

Figure 4.8: MQTT Publish/Subscribe Architecture [Dr Andy Stanford]

MQTT uses a publish-and-subscribe architecture. The publish and subscribe model is made to allow message transmission between client and server in both directions. This enables IoT devices to establish connectivity with one another regardless of where they are located. Even in the case of unstable or unresponsive networks, the MQTT protocol ensures message delivery. It makes use of an acknowledgment system to let both sides know whether or not data was correctly received.

MQTT Broker

The broker is at the centre of the system. All messages must be received, filtered, and sent to the subscribers, in this case the MQTT clients. Millions of connected MQTT customers can potentially be handled by a MQTT broker.

MQTT Client

Any device that can send and receive messages through the broker is considered a client. A client could be a small IoT sensor that sends data at regular intervals or a smart application that displays the IoT data graphically.

To receive communications related to a certain topic, a client may register to it in the broker. Similar to this, a client can publish messages under a specific topic for the broker to forward to the that topic's subscribers.

Access Control

Using Message Queuing Telemetry Transport (MQTT) allows the developers to be subscribed to a topic and self-update based on messages posted on that topic. Since Mosquitto's MQTT broker provides the configuration to alienize IPs, it is possible only to allow the server IP to publish, which provides us with a layer of security. This, however, does not provide a solution for logging the player actions for further analysis. To do this, the MQTT server must be running on another machine, which is also running a backend server, allowing players to send requests for updates to the backend server, and having the backend validate and publish the requested actions on the MQTT broker's topic. The client would then be self-updating while still having a governing authority, the server.

4.3 Proposed Solution

Because the server is supposed to hold numerous clients at once the proposed language to be used is Erlang.

4.3.1 Erlang

What is Erlang?

Erlang is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecommunications, banking, e-commerce, computer telephony and instant messaging [Laboratory].

Process Oriented

The key factor that distinguishes Erlang from other languages is its functional language, light-weight message processing model. It employs isolated, light-weight processes that send messages to one another.

These processes have the capacity to receive messages and, in response, modify their state, start up new processes, or communicate with other processes. Erlang, in other terms, adheres to the actor model [Storti].

The processes are isolated, quick to create, and consume little memory. By adding more of them, a system can easily be expanded. It is simple to expand both horizontally (by adding more machines) and vertically (by adding cores) because the

processes don't know if the other processes are on the same core or in a different location.

4.3.2 Advantages of Erlang

Erlang has three significant advantages over other programming languages, which mainly stem from the unique way the language is built.

- Concurrency. Erlang's virtual machine, BEAM, employs lightweight execution threads referred to as processes. These operate simultaneously on all CPUs, are isolated, and communicate via messages. Because of that and language's functional nature, it is easier to write concurrent programs in Erlang.

- Scalability. Erlang is ideal for today's multicore CPUs and distributed nature of modern computing. Erlang processes make it simple to grow systems, either by adding new machines or by giving current machines extra cores.

- Reliability. "Let it crash" is Erlang's motto. You can develop self-healing systems because of the supervisor system's swift restart of lightweight processes due to its unusual fault-tolerance strategy. Although it may not seem reliable, this fixes the majority of errors that are not the result of serious implementation errors.

Let It Crash and Fault tolerance

In truth, allowing it to crash has nothing to do with the user or the system going down. Erlang works quite hard to prevent that. Instead, it is about handling failure when it inevitably occurs since.

Put simply, an Erlang application is a tree of processes. At the bottom leaves of the tree, there's worker processes – the ones doing most of the work. Above them, there's supervisors, which launch the workers and check up on them.
Supervisors themselves can be supervised; A "Grand Supervisor" can easily be added on top of the tree.
A process sends a message to its supervisor in the event of a crash and depending on the chosen supervision technique, either the process or every process under its supervisor is restarted. The supervisor will terminate all of its children first, then itself if restarting the linked workers doesn't resolve the issue after a predetermined number of times in a time frame. At that point, the obligation to address the issue is passed up to the next layer of supervision.

This means that the program will not crash unless the last supervisor crashes, which guarantees a fairly amount of fault tolerance.

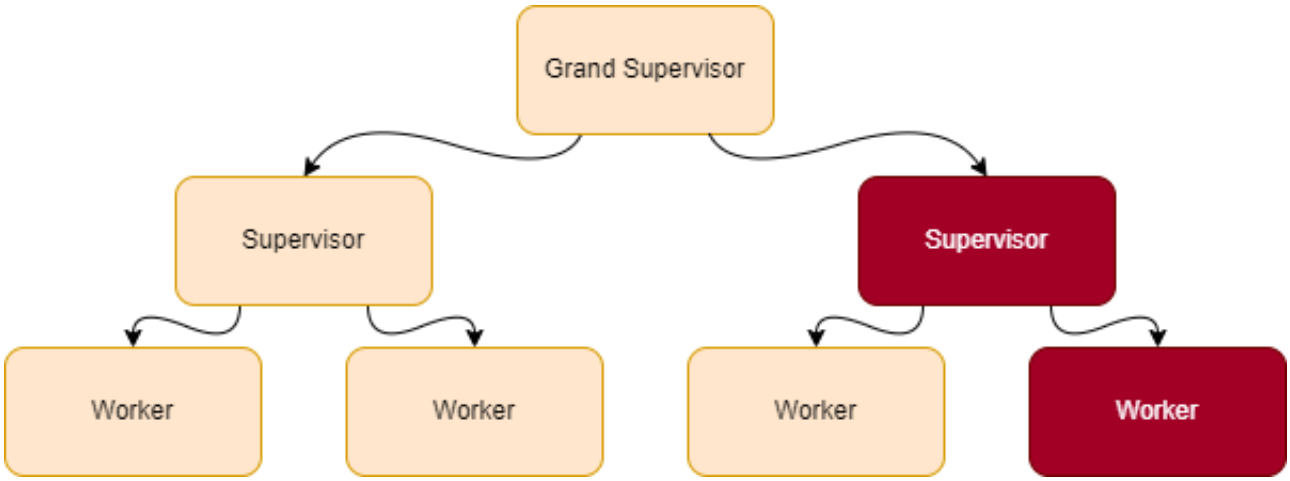

Figure 4.9: Supervisors in Erlang

4.4 Architecture Summary

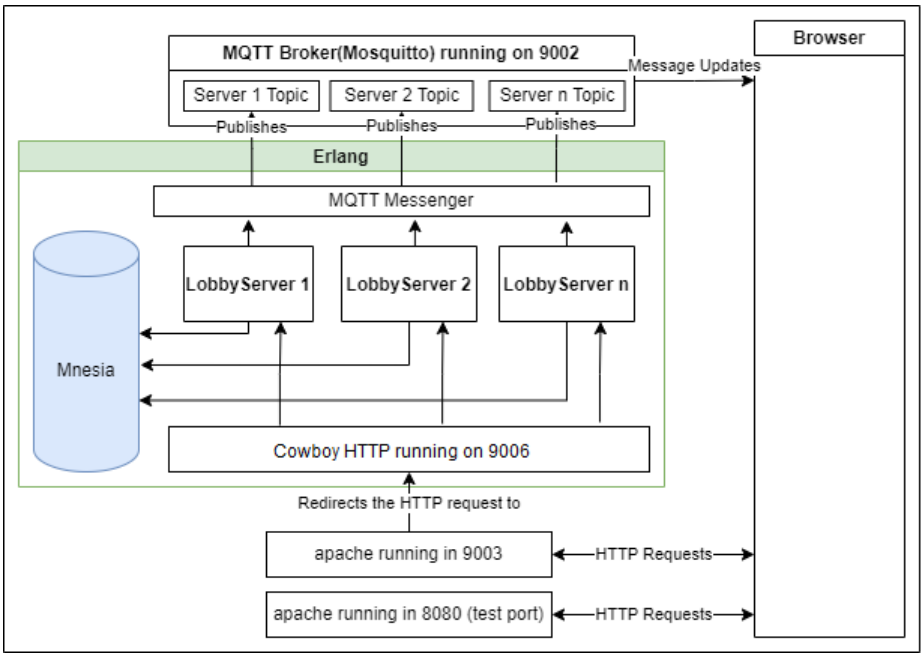

Figure 4.10: S4P architecture overview

The architecture of the developed backend server in Erlang is structured according to the C4 model, comprising three levels of diagrams that provide a comprehensive overview of the system's components, interactions, and inner workings.

Context Level (C1):
At the context level, the architecture is represented by a context diagram that outlines the primary component to be developed within the Erlang runtime environment. This component interacts with other components, forming an environment where its functionalities are situated. The Logic of the system draws inspiration from peer-to-peer connections and fat clients, employing a "fat client" model to process logic on the client side while conducting validations on the server side.

The architecture ensures global authority for validation, thus preventing cheating. The key components include the MQTT Broker, Game Application, and Apache. The Erlang component processes requests from the Game Application, interacts with the MQTT Broker, and publishes necessary information. This level establishes a broad understanding of the system's structure and interactions.

Container Level (C2):
The container diagram, found at level C2, delves deeper into the architecture. It breaks down the main Erlang component into multiple containers, each with distinct responsibilities. The central Erlang container encompasses the Cowbow HTTP server and the developed endpoints. This container manages the HTTP requests, communicates with the Logic, and interacts with the lobby servers. Multiple Lobby Server containers handle validations, maintain player data, and manage individual game instances. The Mnesia component, acting as an embedded database, ensures data persistency. Additionally, the MQTT Messenger container facilitates communication between lobby servers and the MQTT broker. This level exposes the intricacies of container interaction and internal structure.

Component Level (C3):
At level C3, the component diagram showcases the inner components of the Lobby Server in its first version. It consists of three core components: Game Session, Session Heartbeat Monitor, and MQTT Messenger. The architecture leverages Erlang's lightweight execution threads and message-passing mechanism to create a server model. The Game Session component handles player states, game states, and host identification, ensuring the system's integrity. The Session Heartbeat Monitor monitors player connections to prevent disconnections. The Cowboy HTTP server employs multiple processes to handle HTTP requests, interacting with the Game Session component and responding with error or success messages. The desired solution combines HTTP for private updates and MQTT for broadcasted messages, utilizing Mnesia for data storage.

In conclusion, the C4 architecture provides a detailed depiction of the Erlang backend server's design. It encompasses interactions, components, and internal mechanisms. The architecture aligns with the system's goals of scalability, reusability, and efficient game management. By delineating the system's structure across different levels of abstraction, the architecture sets the foundation for a robust and adaptable backend solution.

Science4Pandemic's game needs an economic and easy to use solution to change from single player to multiplayer. This encompasses minimal code changes and a backend which is easy to understand and fast to adapt. The current architecture assumes the game application has already been in development for a while and the action logic will remain on the client side. This removes the need for developers to change the already developed code, only requiring the understanding of how publish subscribe models work. If the single player developers have no knowledge of how publish subscribe models works it will pose a challenge, however once this type of model is known, the single player solution must be structured in a way that the change from single player to multi player is easy to do. In Science4Pandemic's game, the single player logic which needed to be converted in multiplayer is present in a way that it can be changed from an ac-

tion to an event, triggered by a published message(using MQTT). From an economic standpoint, the architecture offers cost-efficient advantages that resonate with the project's financial considerations. By relying on Erlang's concurrency model and lightweight processes, the system can effectively manage concurrent requests with minimal hardware requirements. This translates to reduced infrastructure costs, as the architecture maximizes the utilization of available resources. Moreover, the use of MQTT for real-time communication minimizes the need for extensive bandwidth, optimizing data transfer costs while ensuring timely updates for players.

The architecture's modularity and reusability further enhance its economic adaptability. Components such as the Cowboy HTTP server, MQTT Messenger, and lobby servers can be seamlessly integrated into future projects, resulting in substantial time and cost savings. The ability to repurpose existing components reduces development cycles and minimizes the need for reinventing the already implemented code and logic, ultimately leading to cost-efficient software development practices.

Chapter 5

Implementation

The present Chapter documents the implementation of the solution proposed in the previous Chapter. The following sections contain a detailed description of the architecture's components and other tools needed to achieve the goal.

The proposed solution focuses on creating a stateless backend server, which is to be integrated into a game that has been developed as a single player game for the past year and a half. The implementation must focus on keeping the work of the previous developers to a minimum when it comes to changing their previous code.

In the implementation of the proposed architecture, a pivotal component is the dedicated process responsible for adeptly overseeing player interactions, game state updates, and the dynamic assignment of a designated host. This process assumes the role of a central coordinator, effectively managing player actions and holding the latest game state within the multiplayer environment. This orchestrator, distinct from the host player, plays a crucial role in maintaining synchronization and integrity among players' activities.

Parallelly, a sibling-like process assumes the responsibility of monitoring players' engagement. This process continuously dispatches messages, similar to a heartbeat, to ascertain the players' continued presence within the network. Upon receiving responsive acknowledgments from players, the primary orchestrator process validates their active participation, ensuring that the players whose state remained unchanged are considered disconnected.

Significantly, the architecture leverages a harmonious amalgamation of HTTP and MQTT protocols to fulfill distinct communication needs. HTTP facilitates discreet and private interactions between the orchestrator and individual players, enabling secure data exchange and sensitive updates. On the other hand, MQTT, functioning akin to a dynamic information conduit, is employed for broadcast communication. Game-changing events, initiated through HTTP requests, are swiftly propagated through MQTT topics, allowing all players to receive synchronized updates promptly.

Moreover, the ingenious integration of the dynamic process management model with the Cowboy HTTP server underpins the architecture's versatility in creat-

ing and managing multiple game instances simultaneously. By harnessing the power of Erlang's lightweight processes, the architecture achieves the ability to spawn and supervise distinct instances of the player interaction and game state management process. Each of these instances functions autonomously, dedicated to overseeing the intricacies of a specific game match.

Through this design, the architecture seamlessly scales to accommodate multiple game sessions in parallel. As new players join different matches, the system efficiently generates new process instances, ensuring that the gameplay experience remains uninterrupted and responsive. This modular approach to process instantiation and management optimizes resource utilization, as each instance focuses solely on its assigned match, avoiding bottlenecks or contention.

The Cowboy HTTP server acts as a seamless conduit, directing incoming requests to the appropriate process instance responsible for the corresponding game. This synergy of dynamic process generation and HTTP server orchestration ensures efficient load distribution and cohesive communication across numerous ongoing matches

5.1 Erlang Multiplayer Server

In the previous section, we delved into the intricacies of our proposed architecture, emphasizing the robust Publish-Subscribe model that underpins real-time communication. As we change our focus towards the endpoint configuration, the architecture interfaces with external entities, including web clients and user devices. The configuration of endpoints plays a pivotal role in defining how incoming HTTP requests are handled, processed, and responded to. In this section, we an explanation of the various endpoints integral to our architecture is presented. These endpoints serve as gateways for players to interact with the system, enabling critical functionalities such as game initiation, player actions, and data retrieval.

As stated during the architecture section, at the heart of our proposed architecture lies Cowboy, an essential and pivotal component that serves as the gateway to the entire system. For Science4Pandemics' game, Cowboy acts as the bridge between external user interactions and the multiplayer server. The game application will require the user's unique identifier to access the backend server. In the case of Science4Pandemics' game, this identifier is the account from Science4Pandemics' website. This guarantees that to access the backend server, the user must already have validated his credentials.

As described in the previous chapter, the Architecture's core processes are the lobby servers. Because processes in Erlang are lightweight and designed to be created and terminated frequently. They are not intended to persist indefinitely but rather to perform specific tasks and then gracefully terminate. In some cases(In the current architecture in Cowboy's case), a process may have a specific lifespan. For example, a short-lived process might be responsible for handling a single HTTP request or a single database query. Once its task is complete, the process can terminate naturally.

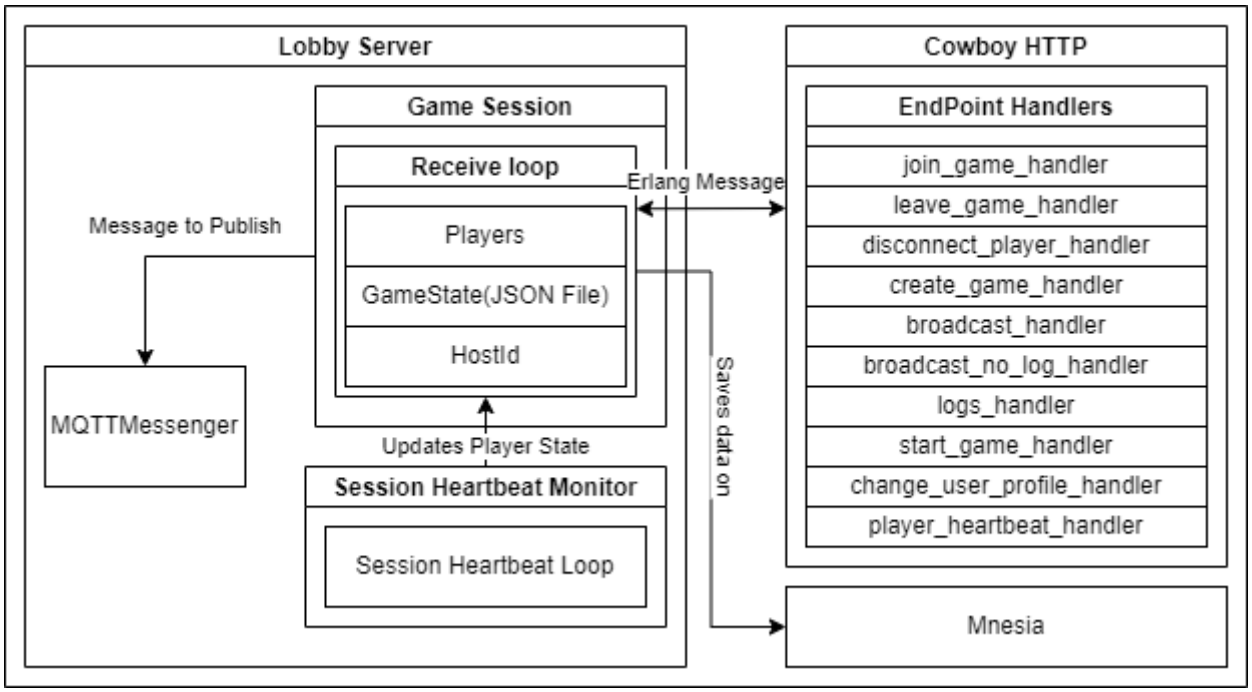

Figure 5.1: Lobby Server Architecture

Because Erlang follows the Actor Model, where computations are modeled as actors or processes that interact by sending messages to one another. Gen Servers are a form of Erlang processes designed to manage state and handle incoming messages in a concurrent environment. Gen Servers are used to manage and encapsulate state data. This state can be modified by handling incoming messages. To ensure the consistency and integrity of this state, Gen Servers typically need to maintain a loop that continuously processes messages.
In summary, an Erlang Gen Server uses a loop to continually process incoming messages, manage state, handle concurrency, ensure fault tolerance, and maintain a long lifespan. This loop is essential for the Gen Server to fulfill its role and to understand how Lobby Servers work.

The main loop of a Lobby Server is as following:

```
1:  function SESSION_HANDLER(PlayerDetails, Players, SessionId, PlayersOnline,
    HostId, GameState, OnGoing)
2:      receives message
3:      if (checks message Atom) then
4:          Runs respective Atom Logic
5:
6:          if (Server should shutdown) then
7:              Sends the shutdown Atom message to himself
8:
9:          else
10:             Calls Itself
11:         end if
12:     end if
13: end function
```

Its parameters paint a clear picture of its multifaceted responsibilities.

The **PlayerDetails** parameter, structured as a key-value store with PlayerId as the key and corresponding player nickname and avatars as values, acts as a repository of essential player information. Meanwhile, the **Players** parameter, designed as a key-value store with PlayerId as keys and connection states ("online," "waiting," or "offline") as values, tracks each player's real-time connection status.

SessionId, as the identifier for the current process, serves as a critical reference point for safe process termination when needed. **PlayersOnline**, an integer representing the count of currently active players, provides vital insights into the lobby's occupancy. **HostId**, holding the playerId of the current host, designates the player responsible for managing the ongoing game session.

The **GameState**, conveyed as a JSON string, encapsulates the latest game state, reflecting the dynamic evolution of gameplay. The **OnGoing** boolean parameter acts as a sentinel, guarding against unauthorized access to concluded game sessions, ensuring that players cannot join games that have already reached their conclusion.

Whenever a Lobby Server is created, a Lobby Server Monitor is also created. The only function of this Monitor process is to send messages on a fixed interval to its Lobby Server counterpart, for it to update it's player states accordingly. Its logic is as following:

```
1: function HEARBEAT_MONITOR(ServerPid)
2:     retrieves the interval defined by the user
3:     Sleeps for 500ms
4:     Try:
5:         Sends Message to ServerPid
6:         Sleeps for the interval duration
7:         Calls Itself
8:     Catch
9:         Safely terminates
10: end function
```

5.1.1 Creating and joining a Lobby Server

The unique identifier retrieved from Science4Pandemic's API is required for every HTTP request made to the server. If this identifier is not retrieved(the user does not login) a temporary user number is assinged instead(e.g. tmp#1111). Using the example of the first request to be made, each lobby is responsible for up to four players. To create a Lobby, the game application will send a request to the **create_game**, which responds with the new lobby ID. The request body should be something like:
{

```
    "clientId": "client1",
    "avatar": "client1_avatar_link_example",
    "nickname": "client1_nickname_example"
}
```


Receiving a 200 OK status response with the following JSON body:


```
{
    "Success": "Server_#"
}
```


After this first response, each client can subscribe to the MQTT Topic regarding the Server_#, in the context of this dissertation that would be /S4P/Server_#, in which # represents a number, and not the MQTT's wildcard.

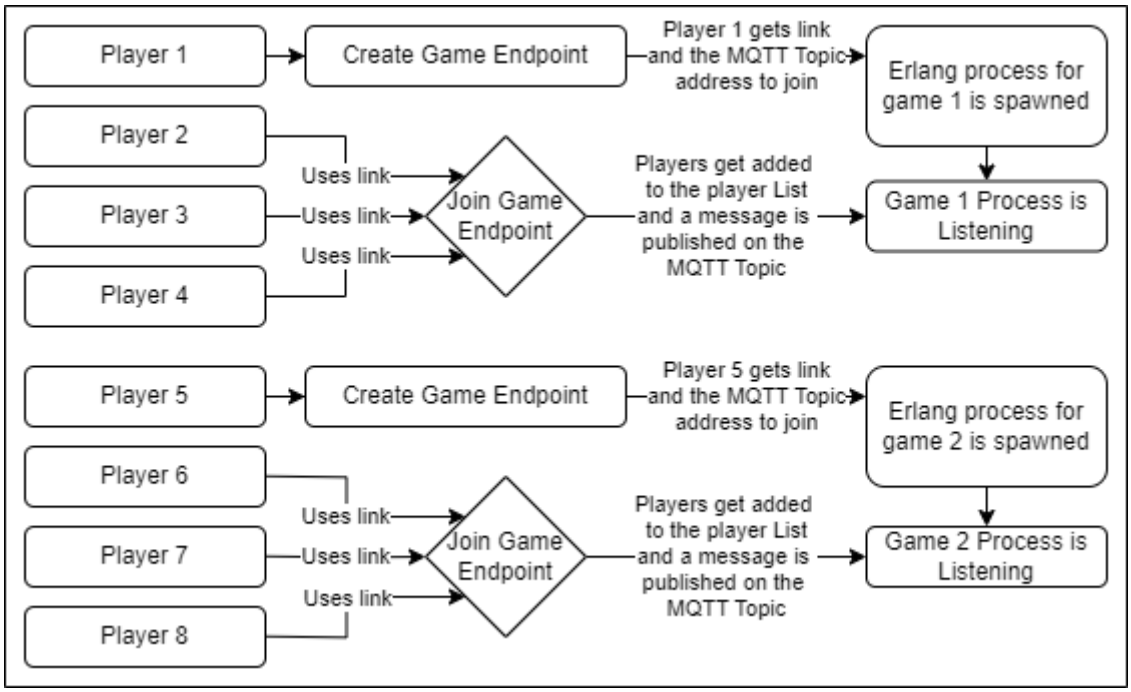

Figure 5.2: Create and Joining a game interaction diagram

While handling the response, Erlang creates the process Server_# and once the Host presses play button, a call to the endpoint **start_game** is made and the game is deemed on going and a Server_#_Heartbeat_Monitor process is spawned to manage the lobby server players and their connection states.

The lobby servers function as continuous looping processes, continuously monitoring incoming messages from other Erlang processes. Within its operational scope, the lobby server manages an array of critical data structures. These include a comprehensive player data repository containing player nicknames and avatars, a dynamically updated Player State dictionary, a unique HostId identifier, and the latest Game State. Notably, the lobby server does not actively manage the Game State but rather maintains it in memory, which eliminates concerns related to memory consumption. The Player State dictionary, a central component, undergoes periodic updates initiated by the Server_1_Monitor process at fixed intervals which can be defined by the user and is used to define the interval in which the server needs to receive HTTP requests from to ensure their connection is still working. To achieve this, the Server_#_Monitor, sends a message to the

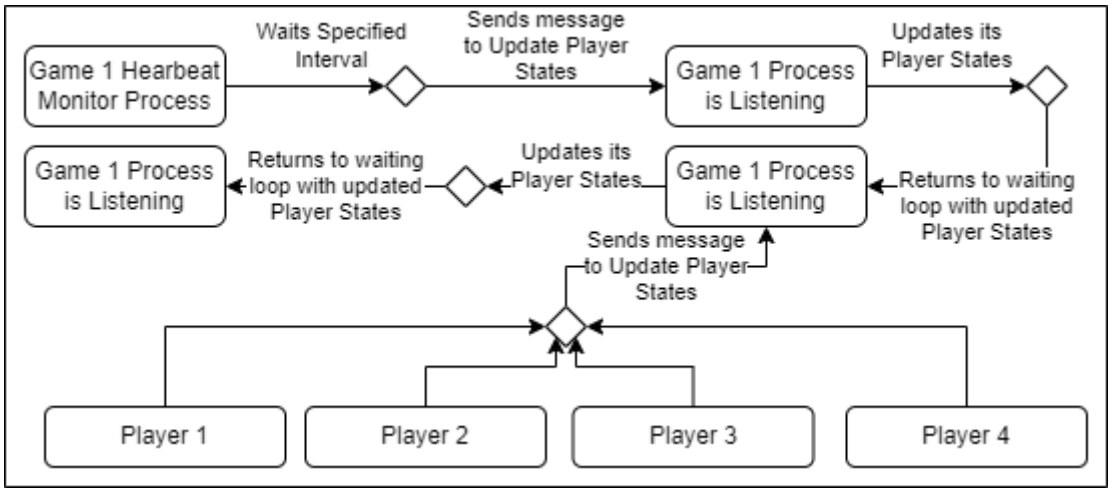

Figure 5.3: Heartbeat Monitor Process Logic

Server_# process, which makes every player with the value "online" change its value to "waiting" and every player with the value "waiting" change its value to "offline". To avoid being disconnected, the game application must send a request to the endpoint **heartbeat** so that the "waiting" player key has its value updated to "online". This ensures the host is updated if the current host loses his connection. It is a reliable way to ensure the game host is updated whenever the previous hosts stops communicating. This is essential due to the fact that, because the clients are self-updating, the host is the only one sending game updates unrelated to player actions(e.g. a timed event or something that would occur without player interactions).

5.1.2 Processing Player Actions

The proposed solution aims to reduce the amount of code previously developed for a single-player that needs to be changed. To achieve this goal, the actions developed for the single-player version of the game can still be used for the same purposes, but now they must be triggered by the message published on the MQTT broker. These messages are validated and redistributed by the multiplayer server.

In Science4Pandemics' game the players have the option to do impactful actions(e.g. start researching Vaccines) changing the game State of every player connected to the game as well as non-impactful ones (e.g. clicking bubbles which were created to keep the player engaged). The impactful actions need to be propagated and logged for further analysis, while the non-impactful ones do not. To propagate an action, the client sends requests to the **broadcast** or **broadcast_without_log** endpoints residing on the Cowboy HTTP server inside the Erlang multiplayer server. The message body must follow the format:

```
{
    "serverId": "Server_#",
    "clientId": "client1",
    "region": "example_region",
```

"simValues": "[]",
"actionCode": 0,
"messageBody": "Broadcast Example"
}

There is no need for the game application to handle a response from the server on these requests. If the message is correctly validated (meaning the player requesting it is a valid player in a valid game), then the message is published on the MQTT's topic for all players to execute as following:
"0 | example_region | client1 | Broadcast Example"
The subscribed clients will then receive the message and self-update accordingly. The action code is used so the MQTT message can be parsed and quickly identified.

5.1.3 Action Logging and Mnesia

The Server_# process also sends messages to the process in charge of storing data on Mnesia with the action logs, which can be fetched using the endpoint **get_logs**. The Mnesia access is located in the process **localDB**, which starts whenever the backend server is starting. The process is in charge of handling messages regarding Mnesia access. Data is saved in two distinct tables. One of the tables is responsible for the game actions, and the other is responsible for the Game State. The table responsible for game actions has the following columns:

playerId	sessionId	region	actionId	messageBody	creationDay	creationHour	simValues

- playerId - String. The user's unique identifier(e.g. temp#1927).
- sessionId - String. The session's identifier(e.g. Server_1).
- region - String. The region where the action occurred(e.g. Lauswen).
- actionId - Integer. The action code for a specific action(e.g. 4).
- messageBody - String. Whatever the developers want to pass around as a message(e.g. in Science4Pandemics' case it is used to propagate messages for the ingame chat).
- creationDay - String. The date the entry was created.
- creationHour - String. The hour,minutes and seconds the entry was created.
- simValues - String. The simulation values the developers choose to store, it's stored as a JSON String.

These are, respectively, the player's unique identifier, the lobby server where the action took place, the values of the current region, the current region's name, the ID of the action that was performed, the message to be displayed (The game has its chat system and this column can be used to communicate between clients,

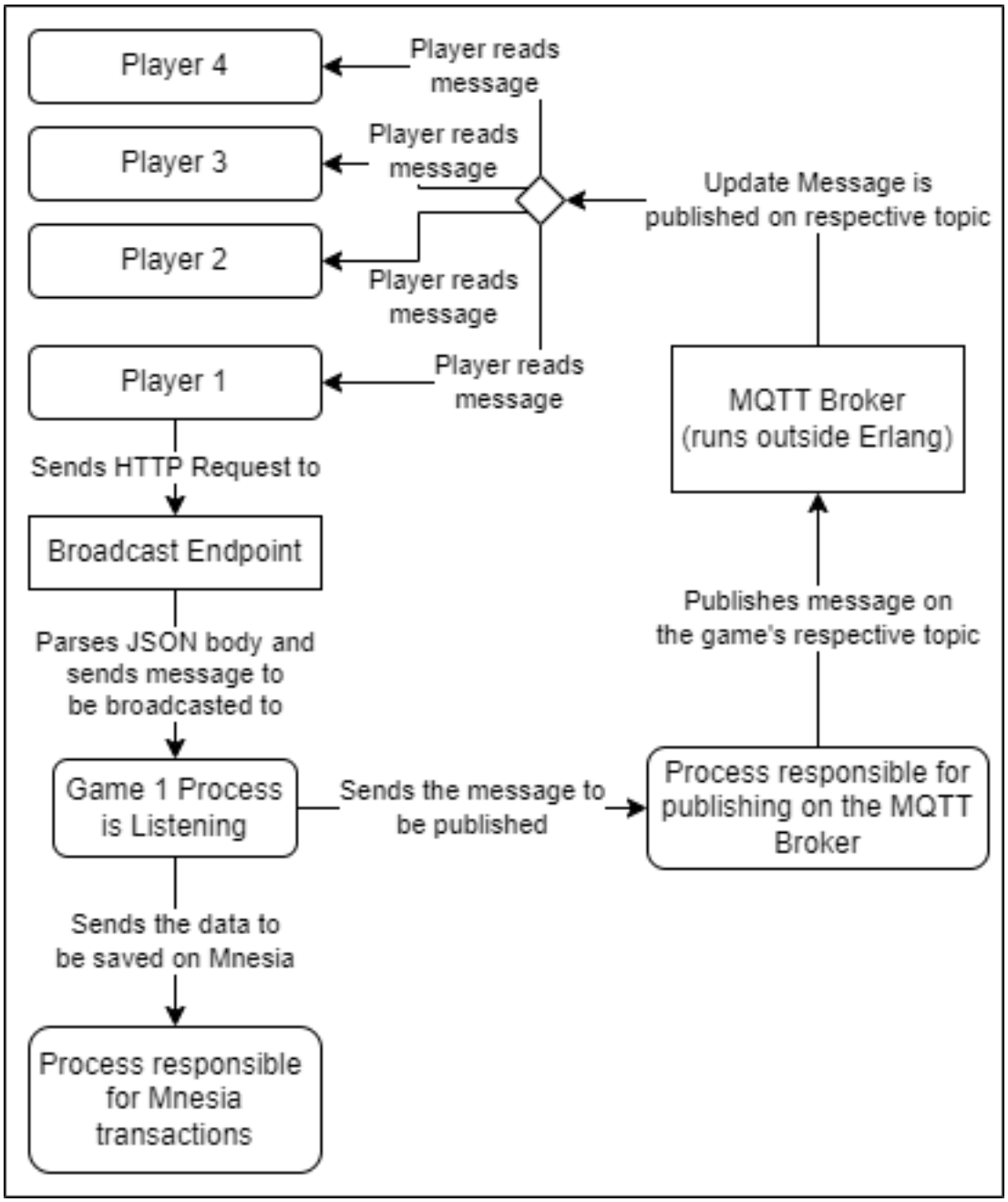

Figure 5.4: Action Broadcasting Logic

from the previous broadcast example this would be "Broadcast Example") and the date and hour the action took place.

The Game State table has the columns:

sessionId	created_on	player1	player2	player3	player4	ended	GameSessionFile

- sessionId - String. The session's identifier(e.g. Server_1).
- created_on - String. The date the entry was created.
- player1 - String. The user's unique identifier(e.g. temp#1927).

- player2 - String. The user's unique identifier(e.g. temp#1943), or null in case the player doesn't exist.
- player3 - String. The user's unique identifier(e.g. temp#2846), or null in case the player doesn't exist.
- player4 - String. The user's unique identifier(e.g. temp#1462), or null in case the player doesn't exist.
- ended - Boolean. True if the game came to an end, False if the game hasn't ended.
- GameSessionFile - String. The latest updated Game State. it's stored as a JSON String. It's named Game Session so it's similar with the already developed code from the singleplayer version of the game.

Which holds the lobby server to which the Game State belongs, the Game State JSON file, the players that were on that game (so that they can restore their last game session if wanted), if the game ended or not and the date of the save. The **localDB** process also contains the action to delete specific or all logs on either of the tables to get a specific log, and the Game State managing options such as save and load, each having their validations to check if the player requesting them is a member of the game or not.

5.1.4 Player States

As described before, the player state dictionary is managed by Server_#, the Server_#_Monitor is responsible for updating it in a specified time interval, and the game application is responsible for keeping the server informed that the client it holds is still connected. However, whenever a heartbeat is sent to the server, the heartbeat also sends the game state JSON file, so even if the last player disconnects, there is a game state in which the server can fall back.

```
{
    "clientId": "client1",
    "serverId": "Server_#",
    "gameFile": "{Ëxample_Json_File}̈",
}
```

The **disconnect** endpoint also uses this request body format. However, the disconnect endpoint is only used when a user wants to leave an ongoing game.

The player state has been divided into two parts:

- During the lobby phase, the players states are not managed, meaning that if they decide to leave (**leave_game** endpoint), they are removed from the player state and player dictionaries
- During a game, leaving using the **disconnect** endpoint (used during an ongoing game) will update the player state to offline

By separating these two things, a lobby-like system can be achieved where players can always have access to their previous game since as mentioned in the previous subsection, the Game State table holds which players are in which game. This will keep Mnesia from being overrun with pointless rows since one Mnesia table can only hold up to four gigabytes.

5.1.5 Fault Tolerance

As stated before, Erlang suggests a way of dealing with fault tolerance. Supervisors are used to spawn their children processes whenever they fail and crash. To achieve this in our system, a supervisor is in charge of managing **localDB** and the cowboy HTTP while there's no need for supervisors for games. Having no supervisors for individual games ensures that if a problem arises during a lobby phase, the player can restart the game, and the problem solves itself. Meanwhile, because there is no supervisor for each game, the lobby server processes and their monitors can freely exit and free up unused memory, guaranteeing scalability.

5.1.6 HTTPS

The singleplayer game was previously being served with https using apache. This means that Erlang's cowboy server must be compatible with it. However because during the development of the backend server the cipher-suite was not known, cowboy could not be https. Cowboy uses cipher-suites to manage its requests, meaning that if the same cipher-suite is not used, the request would be blocked. Due to this, the Erlang server was reverted to its original http, and left running on a port which is blocked on IP Tables. By doing there's a guarantee that there's no direct requests from the outside to the desired port, for which we can instead use apache to redirect requests on an open port to the Erlang Server.

5.1.7 Load distribuiton

Because of Erlang's process oriented nature, the load balancing is easy to manage as long as it is distributed throughout many processes. During requests, cowboy spawns a process to handle each request, making it highly scalable for as long as the machine running the multiplayer server has cores to handle the processes. However, to increase scalability, MQTT is used. By using MQTT, the need for blocking while waiting for responses disappears, reducing the overall load on the http port.
To achieve this, the following logic was implemented:

1: **function** HANDLER(Req0, State)
2: Runs all the validations on the request
3: $Req1 \leftarrow$ processBody$(PostBody, Req0)$
4: **return** ok, Req5, State
5: **end function**
6: **function** PROCESSBODY(PostBody, Req0)

```
7:      Parses mapped JSON contents
8:      Sends the message to be published to the process responsible for the specified game
9:      return
10: end function
```

where the code does the following:

- The request headers are verified, and the type of request(CRUD) is also verified, if valid the processBody function is called and the handler process finishes its execution.
- Process body parses the message and saves the message contents on their respective variables verified if the process responsible for this game is valid and sends the to it(this process is the serverId value from the JSON body) before exiting.
- The process responsible for the game, receives the message, verifies if this is a valid client in this game, and sends the message to the process responsible for the MQTT.
- The process responsible for the MQTT publishes the received message and exits, freeing memory.

By handling the requests like this, the process can send whatever needs to be published to the Lobby server and exit. The lobby server is then responsible for validating if the user is a valid user or not, and publish the response as a validated action message on the MQTT Topic. This is, however, only feasible for requests which need to be broadcasted, since individual requests still need a response, meaning that there are differences in the load distribution for individual replies and updates during ongoing games, making it less scalable for individual replies.

5.2 Publish-Subscribe Channels

The backend development was structured around having MQTT handle zoning. MQTT Topics are structured in a hierarchy similar to folders and files in a file system using the forward-slash (/) as a delimiter. They also have two wildcard characters that can be used. They are:

- **# (hash character)** – multi level wildcard
- $+$ **(plus character)** - single level wildcard

These can only be used to denote a level or multi-levels e.g /vehicle/# and not as part of the name to denote multiple characters e.g. vehic# is not valid

Topic naming Examples:

Single topic subscriptions:

- /
- vehicle
- vehicle/car/indicators
- vehicle/car/motor
- vehicle/car/oil

When using a wildcard e.g. vehicle/# we can cover:

- vehicle
- vehicle/car/indicators
- vehicle/car/motor
- vehicle/car/oil

In the case of Science4Pandemic's game Topics are used to segregate players amongst the different servers, and because the players need to always keep the updated zones because there's a constant simulation of the virus being updated on the background, it is impossible to use this to separate the zones. Erlang's validation on the current players allows up to four players to be connect to a game and therefore up to four players to receive the specific topic they should be listening to (Server_X). To provide an extra layer of security, mosquitto allows the users to specify their configuration files, in which IP's can be blocked from subscribing or from publishing. By only allowing the server's IP to publish, the server can validate the request and filter what should and what should not be published, preventing cheating.

Chapter 6

Evaluation of the Proposed solution

The present chapter documents the methods used to evaluate the proposed solution, in order to assess if it can hold multiple connections at once (scalability) and if it is easy to implement for single player games which have been in development for a long time.

6.1 Scalability

To test the scalability of the backend server, multiple requests will be sent at once, increasing exponentially as 10^n. This will be done through a Wifi connection using a machine with the following hardware:

CPU	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz (8CPUs)
RAM	8GB
GPU	Intel(R) UHD Graphics 620
OS	Windows 10

The following results were made using Postman's recent performance tests (which only go up to 100 virtual users) on the **create_game** endpoint, which requires a response:

Virtual Users	Total Requests	Requests/s	Resp. Time(avg ms)	Error %
1	49	0.56	50	0
10	473	5.78	71	0.63
100	1240	13.10	831	4.19

From this table we can stipulate that the average Response Time will be higher than a minute for 1000 concurrent requests and above. This number however, represents the number of hosts. Since Science4Pandemics' game in specific has

up to four users per game while only needing one host, the game could hold four times the amount of requests. Because this results are only for endpoints which need to process the request and provide a response back to the client they do not fully represent the backend server. However because there's no real way of testing the delay between a request being made and a MQTT publish we can only compare it with the systems implemented which are similar to the implementation described. In Peer-to-Peer the updates requires every single peer to receive the update, in a network with n amount of peers, this can cause a delay up to n*highest_delay since the peer with the highest delay needs to receive the information from every other peer. In the proposed solution, because the subscription to an MQTT topic does not influence individual clients, the highest delay will be of the person with the highest delay, since any client can read any subscribed topic's new publishes once they're published.

6.2 Reusability

In software development, the pursuit of efficiency and innovation often leads to the creation of components, modules, and methodologies that offer the potential for reusability. Reusability, a fundamental principle in software engineering, holds the promise of accelerating development processes, enhancing consistency, and promoting best practices across projects. As part of this research, the exploration of reusability extends beyond the mere creation of software artifacts; it delves into assessing how effectively these artifacts can transcend their original contexts and be successfully applied in diverse scenarios.

The Reusability of software components goes beyond technical considerations—it encompasses adaptability, usability, performance, and the ability to seamlessly integrate into new projects. This section explores how the backend is adaptative for implementation on already developed games. During the integration phase for Science4Pandemics' game, another thesis project required a backend server. However the developer was developing the game on his own and did not have the time to develop a backend server neither to adapt all of his previous code to be multiplayer-like. This other game was being developed in Godot and the thesis was the continuation of a previous one so the game was already far in development. To achieve Reusability the project needed some changes for which the documentation was created, and a configuration file was added to make the integration more dynamic. The configuration file contains the following changeable settings:

- HTTP port
- Max amount of players per game
- MQTT broker IP
- MQTT broker port
- MQTT Topic Root

- Heartbeat Interval
- Log File Path

The MQTT settings can be changed to a public broker, but MQTT Topic will be used in case the user wants to run their own private broker for the game. With this settings and the documentation, the author of this new game was able to implement the multiplayer without changing most of his code, and is currently developing the game in multiplayer mode.

By using MQTT the user can also format the messages, as stated previously the message body is:

```
{
    "serverId": "Server_1",
    "clientId": "client1",
    "region": "example_region",
    "simValues": "[]",
    "actionCode": 0,
    "messageBody": "Broadcast Example"
}
```

This allows the users to use regions or messageBody to alter the messages to their own liking. Non-related to this thesis in particular it is worth mentioning the reusability of the MQTT broker by itself. Because the topics works like folders directories, if each application has a root topic the broker can be used for virtually anything.

Chapter 7

Conclusion

This chapter delivers a concise overview of the primary contributions and discoveries made in this work. It also provides some investigation that can be done to extend the work from this thesis.

7.1 Contributions and Findings

In summary, this dissertation project embarked on a mission to craft a scalable and reusable backend server architecture tailored for real-time multiplayer online gaming, with a strong emphasis on economic scalability. Throughout this dissertation, the unique capabilities of Erlang, MQTT, and Cowboy, were combined to create a versatile system that thrives in the ever-evolving landscape of online gaming. From the findings I highlight the economic adaptability of this architecture, making it well-suited for scenarios where resource optimization and cost-effectiveness are the top priority:

- Versatility for Future Expansion: The system's readiness to transition into multiplayer gaming scenarios positions it as a versatile solution capable of accommodating evolving needs.
- Economic Viability: By minimizing memory overhead and optimizing resource allocation, the architecture proved economically adaptable, making it a cost-effective solution.

7.2 Learning Outcomes

In embarking on my journey to master Erlang, I found myself delving into a programming language that was both unfamiliar and captivating. Notably, Erlang became my first experience with functional programming languages, marking a significant pivot in my programming education. As I delved deeper into learning Erlang, I learned about its exceptional capabilities in managing concurrent processes and its ability to distribute workloads seamlessly. This sparked my interest

in load distribution, a concept of utmost importance in the domains of distributed systems and also in the universe of gaming. During this thesis, to understand the intricacies of load distribution in gaming scenarios, where the seamless coordination of resources could make or break player experiences, I experimented combining Erlang's robust concurrency with MQTT (Message Queuing Telemetry Transport) as a means to orchestrate the load distribution. MQTT's lightweight, publish-subscribe messaging protocol proved to be an elegant complement to Erlang's capabilities, offering an efficient means of coordinating game servers and managing player interactions. Concurrently, my exploration extended to fat client architectures, where the potential of powerful client-side processing in enhancing user experiences and system performance was used. These explorations, marked by the foundational acquisition of Erlang and the innovative usage of MQTT, have forged a unique path in my programming journey, blending functional programming paradigms with the complex intricacies of game development, distributed system management, and real-time communication.

7.3 Implementation and Goals

The dissertation project not only achieved its desired goals on the Science4Pandemic's game but did so in a shorter period than originally anticipated. This efficiency not only demonstrates the developer's agility and adaptability but also underscores the depth of expertise and resources invested. Such an accomplishment not only met but exceeded initial expectations. This success has enabled the project to be re-designed into a more generic solution which stands as a testament to the potential for broader applications. With this change the project was re-designed as a generic solution, featuring a github page [Calhau] with documentation that allows the backend server to be adapted towards other projects.

7.4 Future Works

As this thesis project attempts to create a way for scalable and reusable backend server architectures, it also beckons the exploration of intriguing avenues for future research and development. One promising direction lies in further enhancing the architecture's adaptability for both peer-to-peer and client-server hybrid architectures. By extending the current system, researchers and developers can delve into the intricacies of transitioning from a primarily client-server model to a more peer-to-peer-centric design, harnessing Erlang's inherent support for distributed systems. Such an endeavor would entail not only augmenting the architecture's existing capabilities but also exploring mechanisms for seamlessly balancing client-server interactions with peer-to-peer communication. In doing so, the architecture stands poised to redefine the boundaries of online gaming infrastructure, offering a harmonious blend of client-server and peer-to-peer paradigms. This promising future work holds the potential to enrich multiplayer gaming experiences by optimizing resource utilization, enhancing fault tolerance, and ensuring uninterrupted gameplay in diverse network environments.

References

Dewan Ahmed and Shervin Shirmohammadi. A fault tolerance procedure for p2p online games. In *10th International Conference on Information Science, Signal Processing and their Applications (ISSPA 2010)*, pages 614–617, 2010. doi: 10.1109/ISSPA.2010.5605426.

Marios Assiotis and Velin Tzanov. A distributed architecture for mmorpg. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, page 4–es, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595935894. doi: 10.1145/1230040.1230067. URL https://doi.org/10.1145/1230040.1230067.

Nathaniel Baughman and Brian Levine. Cheat-proof playout for centralized and distributed online games. volume 1, pages 104 – 113 vol.1, 02 2001. ISBN 0-7803-7016-3. doi: 10.1109/INFCOM.2001.916692.

Simon Brown. The c4 model for visualising software architecture. https://c4model.com/, (accessed: 14.1.2023).

João Calhau. Erlang backend server. https://github.com/ximaxer/erlang_backend, (accessed: 14.9.2023).

Sergio Caltagirone, Matthew Keys, Bryan Schlief, and Mary Jane Willshire. Architecture for a massively multiplayer online role playing game engine. *J. Comput. Sci. Coll.*, 18(2):105–116, dec 2002. ISSN 1937-4771.

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). 2003. URL https://api.semanticscholar.org/CorpusID:1280992.

Eya Dhib, Nawel Zangar, and Nabil Tabbane. Virtual machines placement problem based on a look-ahead workload window over distributed cloud gaming infrastructure, 11 2019.

Arlen Nipper Dr Andy Stanford. Official mqtt website. https://mqtt.org/, (accessed: 5.8.2023).

Stefan Fiedler, Michael Wallner, and Michael Weber. A communication architecture for massive multiplayer games. pages 14–22, 04 2002. doi: 10.1145/566500.566503.

Amazon Games. New world. https://www.newworld.com/en-gb, (accessed: 23.8.2023), a.

Blizzard Games. World of warcraft. https://worldofwarcraft.blizzard.com/en-gb/, (accessed: 23.8.2023), b.

Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr. Low latency and cheat-proof event ordering for peer-to-peer games. pages 134–139, 06 2004a. doi: 10.1145/1005847.1005877.

Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *Proceedings of the 14th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '04, page 134–139, New York, NY, USA, 2004b. Association for Computing Machinery. ISBN 1581138016. doi: 10.1145/1005847.1005877. URL https://doi.org/10.1145/1005847.1005877.

Tsun-Yu Hsiao and Shyan-Ming Yuan. Practical middleware for massively multiplayer online games. *IEEE Internet Computing*, 9(5):47–54, 2005. doi: 10.1109/MIC.2005.106.

Allan Jon. The development of mmorpg culture and the guild. *Australian Folklore: A Yearly Journal of Folklore Studies*, 25:97–112, 01 2010.

Yugo Kaneda, Hitomi Takahashi, Masato Saito, Hiroto Aida, and Hideyuki Tokuda. A challenge for reusing multiplayer online games without modifying binaries. In *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '05, page 1–9, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595931562. doi: 10.1145/1103599.1103612. URL https://doi.org/10.1145/1103599.1103612.

B. Knutsson, Honghui Lu, Wei Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE INFOCOM 2004*, volume 1, page 107, 2004. doi: 10.1109/INFCOM.2004.1354485.

Ericsson Computer Science Laboratory. Erlang otp. https://www.erlang.org/, (accessed: 3.3.2023).

Emmanuel Lety Laurent Gautier, Christophe Diot. Mimaze game. https://www-sop.inria.fr/rodeo/MiMaze/, (accessed: 28.07.2023).

Kyung-Seob Moon, Vallipuram Muthukkumarasamy, and A. Nguyen. Reducing network latency on consistency maintenance algorithms in distributed network games. 07 2023.

Shah Nawaz and Yiting Xu. A comparison of architectures in massive multiplayer online games. 10 2014.

Planetarium. Nine chronicles gitbook. https://planetarium.gitbook.io/nine-chronicles-1/, (accessed: 24.07.2022).

Swapna Naik Rutvji Joshi, Dharmik Patel. Implementation of peer-to-peer architecture in mmorpgs. https://www.ijsr.net/get_count.php?paper_id=ART20162499, (accessed: 28.07.2023).

Amazon Games Smilegate. Lost ark. https://www.playlostark.com/en-gb, (accessed: 23.8.2023).

Brian Storti. The actor model in 10 minutes. https://www.brianstorti.com/the-actor-model/, (accessed: 2.8.2023).

Ensemble Studios. Official age of empires website. https://www.ageofempires.com/, (accessed: 7.8.2023).

Bruno Van Den Bossche, Tom Verdickt, Bart De Vleeschauwer, Stein Desmet, Stijn De Mulder, Filip De Turck, Bart Dhoedt, and Piet Demeester. A platform for dynamic microcell redeployment in massively multiplayer online games. In *Proceedings of the 2006 International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '06, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595932852. doi: 10.1145/1378191.1378195. URL https://doi.org/10.1145/1378191.1378195.