1 2 9 0

UNIVERSIDADE Ð
COIMBRA

Miguel dos Santos Lopes

MULTIPLAYER SERVICE FRAMEWORK FOR MARINE POLLUTION CONTROL SIMULATOR (MPCS)

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

September 2023

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Miguel dos Santos Lopes

Multiplayer Service Framework for Marine Pollution Control Simulator (MPCS)

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

September 2023

Acknowledgements

First, I would like to thank my family for all their efforts during this academic stage and all the support during this period. I also thank my girlfriend for all her support and motivation.

I also want to thank my advisors, Dr. Licínio Roque and Dr. Jorge Cardoso, for their continuous help during this stage. I really appreciate all the attention and help they gave me at all times, solving my doubts and giving me suggestions during the development of this work.

Finally, I want to thank my friends for the good friendship we made through these two years and the help they gave me during my master's degree.

This work was partially funded by MPCS project 101048546 under UCPM-2021-PP-MARIPOL, programme UCPM2027 and by the FCT - Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), within the scope of CISUC RD Unit - UIDB/00326/2020 or project code UIDP/00326/2020

Abstract

When a oil spill occurs, there is significant financial and environmental harm. These spills may occur for various factors, as the most usual are: lack of vigilance or a boat/ship accident. So, the removal should happen as quick as possible. For this to happen, all people involved must be in coordination in all times which requires rigorous training.

Marine Pollution Control Simulator (MPCS) is a simulator whose main objective is to improve coordination training for all people involved in removing a pollutant in the sea, without the need for all persons be in the same place or neither to use real equipments. MPCS contains several components, such as a VR-component (VR-interfaces, in experimental version), and an User Interface (UI) Generation (generation of interfaces) component, an Game Editor (configuration of the exercises and entities involved), and the Multiplayer Service Framework. The objective of this thesis was the implementation of the Multiplayer Service framework on MPCS.

This module consists of developing a service where players can join a specific session/exercise. Then, they can perform various actions, from sending messages to other players to interact with objects in the game and perform an action that will influence the pollutant and its behavior. Also, this module is responsible for the emission and storage of the logs done in each game session.

To achieve this, a plan of the tasks to be made was done. The first phase contains the study on the state-of-the-art, the identification of a methodology to use, and the identification of risks. After that, the development of the architecture (and its documentation) and Data Model. The final phase consists in the development of the proper framework, and tests on the performance of the solution found.

Keywords

Pollution Simulator, Game-Based Learning, Networked game, Multiplayer Service Framework, HC Pollution, MPCS.

Resumo

Quando ocorre um derrame de petróleo, ocorrem danos significativos tanto financeiros como ambientais. Estes derrames podem acontecer por diversos motivos, sendo os mais comuns a falta de vigilância ou um acidente de algum barco. Portanto, a remoção deve acontecer o mais rápido possível. Para que isso aconteça, todas as pessoas envolvidas devem estar em coordenação o tempo todo, o que requer muito treino.O MPCS é um simulador cujo principal objetivo é melhorar o treino de coordenação para todas as pessoas envolvidas na remoção de poluentes no mar, sem a necessidade de todas as pessoas estarem no mesmo local ou de usarem equipamentos reais.

O Marine Pollution Control Simulator (MPCS) possui vários componentes, tais como, um componente de Realidade Virtual (interfaces RV, em versão experimental) e um componente de Geração de Interfaces de User Interfaces, um Editor de Jogo (configuração dos exercícios e entidades envolvidas) e uma Multiplayer Service Framework. O objetivo desta tese consiste na implementação desta framework no MPCS.

Este módulo consiste no desenvolvimento de um serviço onde os jogadores podem participar numa sessão/exercício específica/o. Depois, estes podem realizar várias ações, desde o envio de mensagens para outros jogadores, à interação com objetos no jogo e até à realização de ações que influenciarão o poluente e o seu comportamento. Além disso, este módulo é responsável pela emissão e armazenamento dos registos feitos em cada sessão de jogo (action log).

Para atingir este objetivo final, foi elaborado um plano das tarefas a serem realizadas. A primeira fase inclui o estudo do estado da arte, a identificação de uma metodologia a ser utilizada e a identificação de riscos. Após isso, o desenvolvimento da arquitetura (e sua documentação) e do Modelo de Dados. A fase final consiste no desenvolvimento da Multiplayer Service Framework e na realização de testes de desempenho da solução encontrada.

Palavras-Chave

Simulador de poluição, Game-Based Learning, MPCS, Multiplayer Service Framework, Poluição de HC.

Contents

Acronyms

API	Application programming interface.
AWS	Amazon Web Services.
CSN	CleanSeaNet.
DCPM	Directorate for Combating Pollution of the Sea.
DGAM	Central Services of the Directorate General of Maritime Authority.
EMSA	European Maritime Safety Agency.
EVM	Ecosistemas Virtuales y Modulares S.L.
HC	Hydrocarbon.
IPTL	Instituto Profissional de Transportes e Logística.
JAR	Java Archive.
JPA	Java Persistence API.
MMOG	Massive Multiplayer Online Game.
MP	Marine Police.
MPCS	Marine Pollution Control Simulator.
NAVER	Networked and Augmented Virtual Environment aRchitecture.
P2P	Peer-to-Peer.
PC	Port Captains.
POLREP	Pollution Reporting System.
RTF	Real-Time-Framework.
TCP	Transmission Control Protocol.
UI	User Interface.

List of Figures

List of Tables

Chapter 1

Introduction

When a oil spill occurs, there is significant financial and environmental harm. These spills may occur for various factors, as the most usual are: lack of vigilance (the owners clean their boat in the ocean for faster results, in remote places so as not to be caught) or an accident occurs (the boat sinks, the spill falls from the boat). Monitoring is a critical task, controlling all the ships that might commit fraud and monitoring when spills occur (the faster, the better). Sometimes it can be complicated and tricky (by weather-related problems and the difficulty in accessing/controlling some areas).

The identification and removal of a spill need to be done in the shortest time. There are several steps after identifying a spill (explained in detail in the 2.1 section) to combat/remove it from its local (sea or coast). Hence, communication must be as quick and effective as possible (for less damage).

This removal must include many individuals, from firefighters, air commanders, marine police, to even a person responsible for public relations (to handle the communications to the media). In other words, these processes require rigorous operational coordination training for all parties involved to be as efficient and successful as possible due to their complexity.

After training, each person should be able to know better what to do in each specific situation and perform tasks (mainly coordination/communication tasks with other people involved) with better quality and promptness. In general, these persons should have a better and greater perception of how the whole process in the removal of a pollutant occurs and what actions the person (depending on the role) is responsible for.

Nowadays, it is challenging to teach all participants because it is complex and difficult to get everyone involved in the process together at one location and time. Beyond that is also tough to recreate real-world or comparable events. Also, some other errors that can occur in these removals of the pollutant (not counting with communication errors between entities) are individuals forgetting their personal equipment, forgetting water/food or to rest properly (and doing something bad because of lack of energy) and equipment that is not well preserved (because they have not been used for a while).

With this in mind, the organizations (Ecosistemas Virtuales y Modulares S.L (EVM), Qualiseg, Instituto Profissional de Transportes e Logística (IPTL), Central Services of the Directorate General of Maritime Authority (DGAM) and University of Coimbra) involved intend to develop a simulator called Marine Pollution Control Simulator (MPCS). This simulator needs to act as a form of learning/training for everyone involved, thus avoiding all the constraints (reality, cost, and organization constraints) necessary to train everyone involved in real life.

In this simulator, players need to interact with each other (and with objects) for faster oil spill removal, so a multiplayer component is required. Adding a multiplayer layer makes it possible for different players to share the same game session on a different network. However, some essential services for this type of game are needed for this to happen. Some examples are servers for the game to be online (and players join the game), databases (to store the data of all entities and sessions), communication channels (for the player to communicate with the server and even with the game itself), among other services.

1.1 Motivation

This section first presents the motivation behind the existence of this project. After that, the motivation regarding the multiplayer module is also presented.

As mentioned before, removing the spill as soon as possible is essential, and there is a need for experienced and trained participants. However, it is difficult to train all people involved in the removal of a spill, mainly because of three aspects:

- Time and location constraints, it is tough to gather all individuals involved in this process in a single location and time (many people involved, all with their calendars and occupations);
- Money constraints, it is costly to get everyone together in a specific place and even to reunite all the vehicles/objects required for training;
- Reality constraints, it is challenging to simulate (in real-life) a spill (this one is impossible without causing damage to the environment) or even to preview the behavior of the sea. So, because of all these aspects, there is a real need to improve this situation.

So, MPCS tries to combat these constraints. MPCS is a simulator that combines real-life scenarios for the combat team (the team responsible for the removal of the spill) to train and improve their coordination and overall performance of the removal without the need to all be in the same place.

With the use of Marine Pollution Control Simulator (MPCS), time constraints (every person can be anywhere), money constraints (i.e., no costs regarding the movement of persons or equipment), and also reality constraints (using MOHID and this simulator, a virtual spill can be placed with all of its properties based on his type and behaviour), are all avoided. With that, after the utilization of

MPCS, an individual who has played it should be more capable of making decisions and be more aware of what steps he should do in a real situation of an oil spill, without the need for several real training sessions.

MPCS has four major components (more detail in 1.2): VR-Component (not integrated with the others, since it is an experimental version), an Interface component (UI Generation), a Game Editor, a Digital Twin (simulation of spill based on weather/water conditions) and a Multiplayer Service Framework.

Regarding this specific component (multiplayer), it is essential that players interact/cooperate with each other and with objects (vehicles, bombs) to remove the spill (as in real life) entirely, so there is a need for a multiplayer service in the game. Coordinating the multiple participants in the set of activities defined for each game (the exercise of operational coordination) requires interactions for the game to be as realistic/accurate/timely as possible, so the multiplayer component needs to be efficient and also it is needed that the simulator as a good variety of actions.

1.2 Scope

This internship focuses mainly on the Multiplayer Service architecture, including some elements such as configuring the server and the database(s) in the server, determining how the players will interact and affect the right game state, and also, the definition and implementation of actions that can occur in the game.

As a result, all other specifications and goals, such as VR-Operations or the User Interface, are outside the scope of this thesis or internship. Instead, they are the responsibility of my colleagues working on this project. Doing a quick overview of the simulator, it contains the other following components/features:

- A VR-Component: Virtual reality-based interfaces for placing the oil spill and the other elements involved in this process (such as boats or equipment). This will also interact with the Digital Twin to retrieve information about the behaviour of the HC spill and weather conditions. This component is only for research purposes, so there is no interaction with the other components (for now);

- Digital Twin: The simulator should be capable of retrieving weather and meteorological conditions (predict how the spill will evolve). A Digital Twin is used to model the conditions on the ground. A MOHID simulator will be substituting for the real scenario as if users were interacting with the physical scenario;

- Game Editor and Simulation Modelling: The overall simulation model (defining the game for each operational coordination exercise). Also, the development of an MPCS manager role. This role should be able to define exercises (and their actions). This component is responsible for the overall simulation model (defining the game for each operational coordination exercise). Also, the MPCS managers should also be capable of assigning players to different sessions;

- User Generation Interface: Development of User Interface (UI) using automatic UI generation. This component generates role-specific user interfaces for players according to what they have at their disposal. Also, this module is responsible for the player movement;
- Multiplayer Service Framework: Responsible for managing the multiple players' UI connection to the game world and developing all core actions in the game (i.e., messages and interaction with objects). Detailed in 1.3.

1.3 Objectives

The Multiplayer Service framework covers features such as how to transport and share data with all players, manage communications with players in the same game session, handle instances of the game, and set the game online. Another important objective of this internship is developing and implementing the actions possible during a game. With that in mind, the main objectives of this module are:

- To define and document the overall system architecture, mainly covering these aspects:
  - Services responsible for managing the multiple players' UI connection to the game state;
  - Database to store different types of data like photos, reports of the performance, live-tracking data, ...;
  - communication channels to players be able to connect to the server or even with each other;
- The Implementation of actions in the game, taking into account conditions required for a specific action and its impact in the session state (except for the movement action, which is done by the person responsible for the interface component);
- The storage of the log of actions done by the players. With this, the MPCS Manager should be capable of seeing (in an interface of the simulator) what actions were done during a certain exercise and who did it;
- To implement demonstration interfaces (Application programming interface (API)) that validate the system's multiplayer core architecture and performance;
- To develop a reference implementation and documentation so that the system can be further developed in the future;
- To assess the proposed architecture performance.

Chapter 2

State of the art

To be able to implement the required Multiplayer Service framework, a study of the different concepts, approaches, and technologies is required to understand what is needed in these types of games and even to understand the difficulties (and how to combat them) that game developers find when developing these services.

First, a study regarding the MPCS is done. It shows the current state of the domain, the problems faced, and other components related to this project.

Second, some critical concepts are done, including the different network topologies and the factors that can influence multiplayer experiences (section 2.2 and 2.3). After that, several multiplayer games are presented, their general architecture and how they handle the game world and game state (section 2.5).

There is also a study on different frameworks/approaches in multi-user contexts (section 2.6). Lastly, it shows some key components of a multiplayer architecture (libraries, databases, and hosting/cloud services) and their main players in the market (section 2.7).

These concepts were studied to learn more about multiplayer architecture and components and how to adopt actions in a game to be more capable and responsive to the user (e.g., messaging the actions to other users).

2.1 Background on problem domain

There are many entities involved in the combat of this type of pollution. The Central Services of the Directorate General of Maritime Authority (DGAM) and the Directorate for Combating Pollution of the Sea (DCPM) are the main ones responsible for controlling and monitoring the ocean/coast. They have many sources to be aware of sea pollution, including the Naval Command, the Air Force Operational Command, Port Captains (PC), Civil Aviation, the Merchant Navy and the Recreational Navy, and CleanSeaNet (CSN), all under contract from European Maritime Safety Agency (EMSA).

These entities must be coordinated to remove the spill quickly and efficiently. Hence, there is a guide on how to act after the identification of an accident. There are mainly four steps (process) after the identification of a spill and to combat/remove it from its local (sea or coast) [Sampaio and Roque]:

1. Initial actions (notify the accident, confirm, and evacuate all civilians): In this step, when the accident is identified, it is needed that the people responsible give an alert of the accident (notify). After this, the Marine Police (MP) or PC should confirm the alert. If there are persons needing rescue, the rescue starts immediately;

2. Information gathering (description of the spill): The second step is to gather the maximum information possible about the accident (causes, pollutant, font,...). Some examples are the type of pollutant (heavy, light, dangerous substances, quantity of the spill), the conditions of the environment and affected areas (geography, accessibility, support infrastructures), weather conditions (wind, temperature,...), and sensible areas (points of interest, i.e., natural reserves, protected areas). In this step, a notification is sent to the Pollution Reporting System (POLREP) (via email with a PDF or MMHS message). Tools to support decision-making are used, i.e., a simulator to predict the HC's behaviour (with the help of the DGAM-DCPM);

3. Complementary actions: Some complementary actions are done, such as interdiction of the areas (access-only to authorized persons and vehicles), collection of a sample of the pollutant (and sent to a proper lab), activation of the response (level 3), contact and inform the entities, communication to the media, and emission of a warning to the population (local, restricted areas and precautions to be taken).

4. Operations of combat: This is where the actual combat of the HC is done. First, there is an intervention by each entity's captains. It is defined as a plan of action to remove the HC (based on conditions and available means). After this, the removal and storage of the oil spill is done. Also, an acknowledgment of the situation (evolution of the incident and possible actions) is done, and there is a registration of the actions and all events (who and how).

As we can see, some complexity is involved in performing all these steps successfully. This complexity is further increased because, for each accident, there are several scenarios and approaches (more than one scenario can be identified in the same accident), all with their particularities. It is only mentioned two scenarios because the other ones are similar (the main objective is to remove and store the pollutant (if possible) and protect sensitive areas and/or biodiversity) [Marinha, 2022]:

- Containment and removal in the sea: The collection is still done at sea level, containing the pollutant in the sea first and then removing it. Two boats are used, which hold a barrier (through connecting cables) to contain the pollutant. After this, proper removal equipment is placed between the barriers

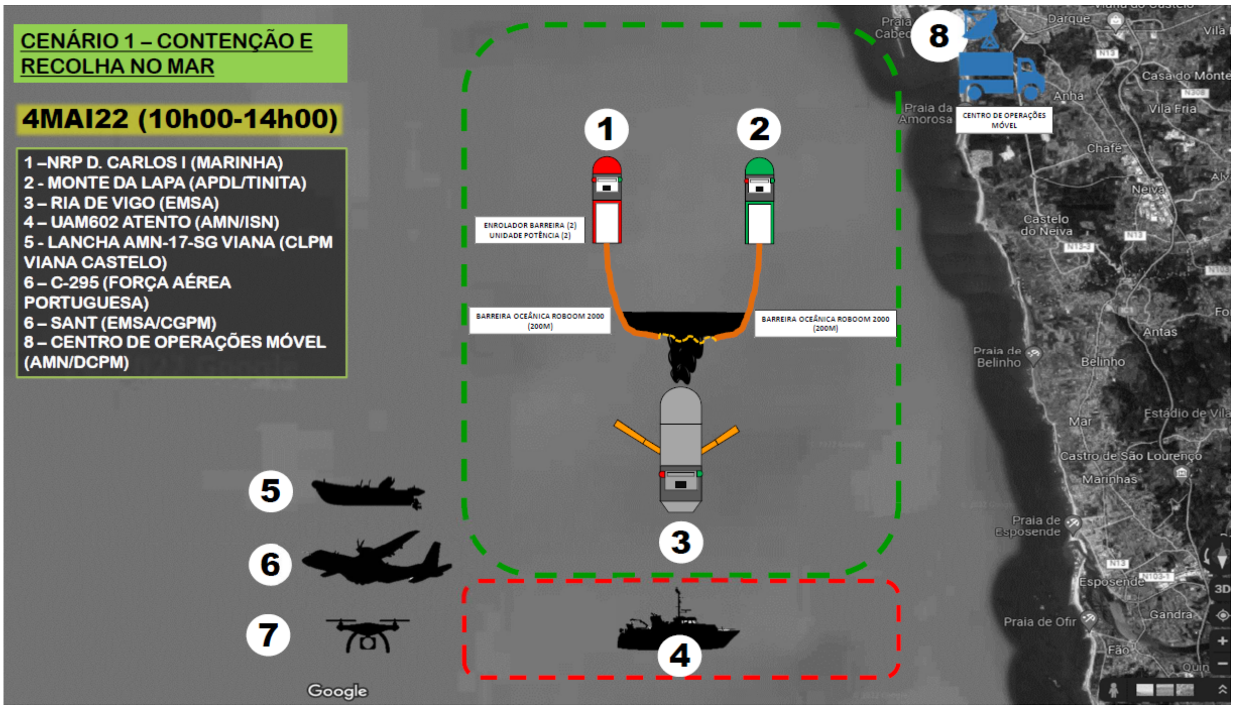

CENÁRIO 1 – CONTENÇÃO E RECOLHA NO MAR
4MAI22 (10h00-14h00)
1 –NRP D. CARLOS I (MARINHA)
2 - MONTE DA LAPA (APDL/TINITA)
3 – RIA DE VIGO (EMSA)
4 – UAM602 ATENTO (AMN/ISN)
5 - LANCHA AMN-17-SG VIANA (CLPM VIANA CASTELO)
6 – C-295 (FORÇA AÉREA PORTUGUESA)
6 – SANT (EMSA/CGPM)
8 – CENTRO DE OPERAÇÕES MÓVEL (AMN/DCPM)
ENROLADOR BARREIRA (2) UNIDADE POTÊNCIA (2)
BARREIRA OCEÂNICA ROBOOM 2000 (200M)
BARREIRA OCEÂNICA ROBOOM 2000 (200M)
CENTRO DE OPERAÇÕES MÓVEL
1
2
3
4
5
6
7
8
Google

Figure 2.1: Containment and removal on sea

and removes the pollutant. An example of contamination in the sea and how the accident is approached is presented in Figure 2.1;

- Containment in the coast: When the spill occurs/reaches the coast. In this scenario, a lot more processes are needed. For example, a definition of different areas is done (clean area, decontamination area, and dirty area), and depending on the area (decontamination and dirty), only a particular type of persons or equipment are allowed. For cleaning, several processes are made by different teams, such as a team for the cleaning of the rocks, one for the cleaning of the beach/coast, a team in the decontamination area, a team by the sea working as a support (responsible, i.e., for placing a barrier in the sea so that contamination does not spread and does not escalate) and a team responsible for the logistic of the scene (ambulances, tents, transport vehicles, catering, sanitation for the teams, among others). There is also a need for equipment to store all the pollutants. Figure 2.2 represents a containment and its removal on the coast.

Other scenarios are containment in a river, near a shipyard/docks, and protection of agricultural terrain. As we see (in Figure 2.1, i.e.), many teams and equipment are involved in this situation, so there is crucial to have all teams in coordination with each other to achieve the best performance possible. With this in mind, the persons involved must be capable of doing their required functions, so much coordination training is required.

The MPCS [Sampaio et al., 2022] is a multilingual environment (currently only Portuguese and English) based on a Platform as a Service for teaching, training, and performance evaluation, individually and in teams, of actions to combat maritime pollution. The training (simulation) will occur in a multi-user environment

Figure 2.2: Containment and removal on land

on the web. This simulator can be used via a smartphone, tablet, or computer. Each user assumes the role of a real professional on his or her specific device. Also, the simulation runs on a given real geography (with a map and Digital Twin component).

Currently, three are types of three fundamental actions in the simulator: the configuration of an exercise, where the MPCS Manager can configure an exercise, the running of an exercise (where the game is played), and a review of the exercise (via action log).

In the future, it is supposed to the MPCS to be a learning environment where users, through explanatory videos and PowerPoints, learn the different concepts and ways of acting and an evaluation environment, where the simulator provides an automatic evaluation (in some cases) and visualization of the performance and results (positive/negative, emitting a report) of the individual and the team.

This simulator is designed as a Game-based learning game. These types of games purposefully teach the person while playing the game, redesigning it as a motivating and engaging activity, i.e., combining a game environment with the rules and characteristics of real life. This simulator combines cases of real life, i.e., the spill in the ocean, and by players interacting inside the game with each other (like in real life), they can control the spill or not. This will be beneficial because it reduces the high cost of reuniting all people in one place and is more efficient in simulating coordination exercises.

The MPCS should be able to simulate a realistic spill of a pollutant. The spill behaviour will be done through the Digital Twin components (MOHID) with the help of a mathematical model (already acquired by the Marine) that will calculate the expected behaviour of a spill based on the weather conditions and the advance of time. After this spill, the simulator should be able to simulate the combat (as realistically as possible). That is, the participant's character should have their own physics (with life/energy, which will change according to the user's actions) and should interact with other participants as well as equipment (which also has its own physics and specific characteristics). During each exercise, action logs are issued detailing the events and actions of each player (action log).

The definition and modeling (in the simulator) of entities and their characteristics are not the main focus of this internship, but the definition of the database that will store this information and the use of the database in running a game are. Hence, the study of these entities is also important to choose a database that can meet the requirements of the simulator. With this, some examples of entities required in the simulator [Barata and Roque, 2022] are:

- Participants: Players with characteristics such as name, height, weight, gender, and function);
- Spill: Location, type of pollutant, size, weight, volatility,...;
- Equipment: Barriers (booms), Skimmers, Vehicles, basic equipment (phone)..;
- Vehicles: A specific type of equipment. It includes cars, trucks, drones, boats, and so on. These also have characteristics such as required driver's license, capacity, weight, type of vehicle (land, air, sea),...

Since this involves many components with all having their own problems and difficulties, a difficulty of this project is that it requires cooperation, interaction, and integration of all required components to have a successful and playable simulator. Also, another difficulty is that MPCS is an innovative simulator, so there is no other simulator or game similar that aims to what MPCS wants to accomplish with his development (NIMSIMPro is the simulator that is closest to what MPCS wants - which is referred in 2.4.2).

2.2 Overview of network topologies in multiplayer games

When defining the type of topology of a network, it is essential for every network to work correctly to grant stability, quality in a connection, performance, and security. Multiplayer games are no exception, so I reunited the three main types and looked for examples of each approach to determine the best from different points of view (e.g., if more security is required or if performance is more important than security).

The Client-server architecture (Dedicated Game Server) is this topic's most well-known architecture. In this architecture [Yahyavi and Kemme, 2013]), the server holds the master copies of all mutable objects and avatars and maintains global knowledge of the game world. The players must connect to the server to receive the required data about the game world and state. The player's interactions in the game world are done by sending information to the server, and then the server replicates to other players. This architecture's biggest drawback is scalability because it cannot hold as many players as Peer-to-Peer (P2P). To fight this drawback, we can implement multiple servers, but it will have major costs (one example is Quake II, an FPS Massive Multiplayer Online Game (MMOG), which can only support a few hundred players when using a traditional client-server architecture with only one server). Figure 2.3 retracts an example of a Client-server architecture.

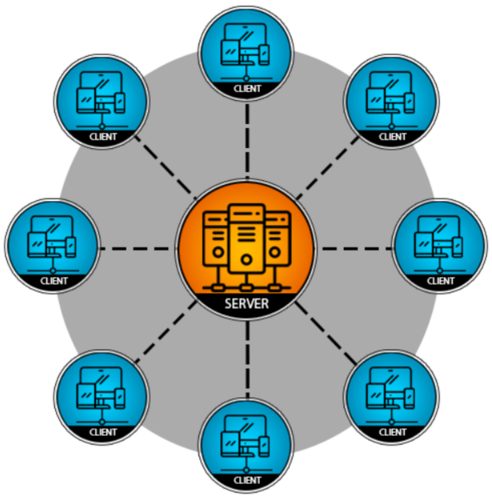


Figure 2.3: Client-Server (Dedicated Game Server) Architecture [Unity2022]

In games, Peer-to-Peer (P2P) (described in[Bauwens et al., 2019]) works the same way as in normal P2P networks. This means that each node (client) could become responsible for maintaining copies of some of the game objects and/or updating information for other clients, so, for example, whenever a player joins a session, he adds resources to the game. There are two different types of using P2P, with a host-player (the player acts as the server of that particular game) or with P2P directly (all players communicate with each other). Figure 2.4 shows an example of a P2P architecture.

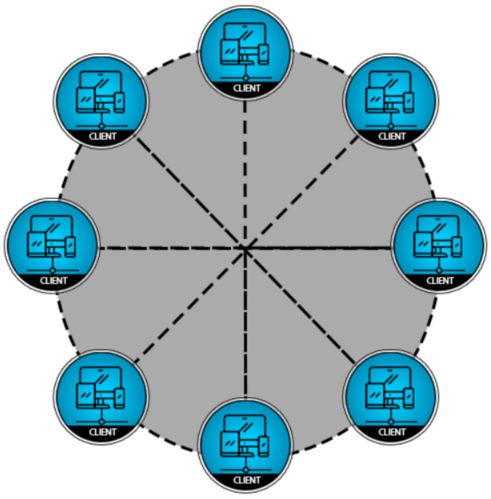


Figure 2.4: P2P Architecture [Unity2022]

As mentioned in [Jardine and Zappala, 2008], a Hybrid approach combines the best aspects of P2P with a Client-server approach. Sometimes it combines more core things related to P2P compared to Client-Server or the opposite.

Seeing some practical approaches, we can see the proposed Hybrid Architecture that Jardine and Zappala used, where the server only takes care of the critical events of the game. With this, they can lower the bandwidth (causing some relief to the server) and can prevent cheating.

Another example of this is various servers worldwide where they communicate with each other, but in each session, the players communicate with each other (in the same session). More different types of Hybrid architectures, more complex and more detailed, are presented in [Yahyavi and Kemme, 2013]. An example of a Hybrid Architecture is shown in Figure 2.5.

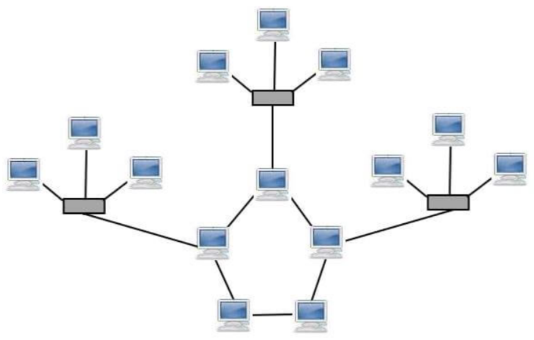


Figure 2.5: Hybrid Architecture [Javatpoint]

Combining the articles that were mentioned before and even with the article done by Barri et al., some advantages and disadvantages of each topology are shown in table 3.1

Topology	Advantages	Disadvantages
Peer-to-Peer	• Resources are added without any cost for the game provider; • Less cost in maintainability, due to no existing dedicated servers; • Provides scalability; • Low latency, because there's no need to send data to a server, it sends directly to (interested) players.	• If not implemented properly, a network failure in a client can compromise the game; • Cheating is easier, because there's no server; • Harder to manage because it's harder (or impossible) to the game provider have total control of the game; • Less consistency.
Client-Server	• More control over the game world and game state (total); • More manageability (updates and patches are much easier to implement); • Easier to prevent cheating, compared to a pure P2P approach; • Design is much simpler.	• More latency (all the data needs to be send to the server); • Less Scalability; • To combat scalability it is needed a greater amount of resources than in P2P.
Hybrid	• More Reliable, fault detection is easier; • More Scalable, comparing to Client-Server; • Easier to prevent cheating, comparing to a pure P2P approach; • More flexible; • More effective, in the way that we can "choose" the components that fits the best to our needs.	• Complexity in design; • Cost of the Hubs, that connect the different networks; • Costly Infrastructure, because of being Hybrid, it will probably be bigger in terms of scale.

Table 2.1: Comparison of topologies in games

2.3 Relevant factors impacting multiplayer game experience

This section presents an overview of factors affecting the experience in multiplayer games. Compared to sections 2.2, and 2.5, this section is more from a practical point of view. Since this thesis is a practical thesis where there is a need to implement a multiplayer system and not so much research is needed from articles, I think it is particularly interesting and important to mention these factors from a more practical perspective (from websites and books that really know the industry) and not so much from research articles (although, a study on research articles is made further in the document).

Regarding multiplayer games, these can be networked, LAN (despite being networked, there is a need to have all people in the same physical location) or couch co-op. Being MPCS, a networked multiplayer game, the focus will be on this multiplayer. By its own definition [Armitage et al., 2006], a networked game must involve a network, i.e., a digital connection between two or more computers. Multiplayer games are often networked games in that the players are physically separated, and the machines, whether PCs or consoles are connected via a network.

As Armitage et al. said, the realism of online gameplay depends on how well the underlying network allows game participants to communicate in a timely and predictable manner. These synchronization issues between the clients and servers are also commonly called *netcode*. Regarding *netcode*, a definition of some key concepts regarding the experience of a networked multiplayer game (focusing on the *netcode*) is presented:

- Latency: The time it takes for a packet of data to be transported from its source to its destination. There are two types of Latency [Armstrong, a]:
  - Non-network latency: input lag due to GPU, VSync, Render pipeline delay,..;
  - Network latency: Latency due to network constraints. Some examples are, *processing delay* (the time that the router needs to read the packet header and to know who is the next machine that should receive the packet), *transmission delay* (the time to push the packet bits onto the physical link), *queuing delay* (a router can only process a limited number of packets at a time), and *propagation delay* (time signal spends traveling through the physical medium). Although all can be optimized by implementing more capable servers, *propagation delay* is easily improved by placing servers near the player's region. Regarding networking, it is important to consider the combination of network latency (commonly called *ping* or RTT) factors;
- RTT (Round Trip Time): The time it takes for a packet to travel from one host to another and then for a response packet to travel back. The deviation of RTT in the communication of two hosts is called *jitter*;

- Jitter: Variation in terms of latency (ping) regarding one packet to another (the next one). It can affect RTT mitigation and also make packets arrive out of order if the server needs to send the packets to different routes.
- Packet loss: When a packet is entirely loss. If a determined packet loss affects the data transported during the game it can cause major damages to the *playability* of a game (high delay).

With these concepts in mind, some strategies can be used to mitigate network latency, for example, [Armstrong, b] [Valve] [Armitage et al., 2006]:

- Client prediction: The client can forecast the server's answer, allowing the game client to respond to user input and render player actions before receiving the server's official response. However, using prediction will result in a difference between the game state on the client and the server (as well as the status on other client computers);

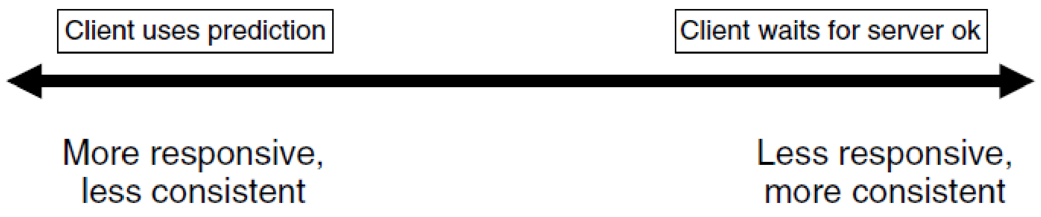

Client uses prediction
Client waits for server ok
More responsive, less consistent
Less responsive, more consistent

Figure 2.6: The trade-off between consistency and responsiveness using client prediction

- Entity Interpolation: The server sends updates with the locations of all entities (other players). The client interpolates between each update while waiting for a few before transferring the object;
- Lag Compensation: The concept of the server using a player's latency to rewind time while processing a user command in order to view what the player saw when the command was sent. However, this can provide new opportunities for cheating;
- Time Delay: Instead of immediately processing client commands, the server pauses them for a short period of time, enabling a client located further away (in terms of network latency) to respond to the game situation. As seen in Figure 2.7, the server waits for the second client to process both commands at the same time

There are more types of strategies, such as selecting a non-authoritative server, where the client can do his own updates. Still, it can provoke data inconsistencies and less reliability (for example, using client prediction in Figure 2.6). Selecting a network topology (detailed in section 2.2) and taking into account what matters most in the game (scalability vs security) is also an important detail to consider.

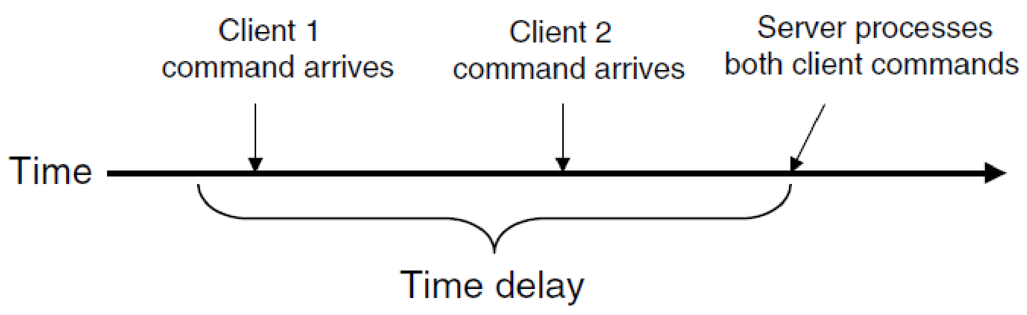


Figure 2.7: Server processing client 1 and client 2 at the same time due to time delay

The game developer should consider a large number of factors regarding these aspects and the possible trade-offs of choosing a specific strategy or component. Some factors have a significant impact on the game performance (*netcode*). Some features found with the aforementioned references and also [Unity2023] are:

- Latency tolerance: Network latency can be different depending on the distance between any of the performance from the networks along the transmission of data. In this factor, game developers should consider how quickly the information and data must be synchronized with all players (level of latency tolerance). In some cases, game developers can "hide" latency (and choose a higher level of latency tolerance) using prediction and reconciliation;

- Players per session: The players that the game will need per session. How many players will the game handle at the same time. With the increase of players, more power for synchronizing data is needed;

- Scale of the synchronized data: The parts that must be synchronized with all players. Game developers should define what parts are essential to be synchronized with all players or not. This can improve the network's performance since the resources for processing and synchronizing data are limited. For example, using a server authoritative strategy can be slower, but it will prevent discrepancies in data.

- Precision: The precision needed relating to the calculation of the world state and its entities. Some games need to be extremely precise but for others, using an approximation (prediction) of the results is enough. Higher precision requires more resources.

- Cost: The cost of the game. If, for example, server capacity and capability are needed, the game will be more expensive. This can be influenced based on concurrent users, players (and their distribution around the globe) per session, among other factors;

- Developer complexity: The complexity of the solution found. The developer should consider the solution's complexity before the actual implementation. Using solutions with good documentation, widely used and easy tutorials reduces the complexity of the project;

- Security: If the game needs to be secure with data protection or cheat prevention. The second one is not particularly important regarding MPCS, since it is not a game-competitive simulator.
- Lock-in: How easily the solution found can be reproduced or changed. If something happens to a certain component, how easily it is to replace him.

2.4 Simulation-based training

In this subsection, we will review two cases of simulators for training, even if the information available is scarce for the purpose of our project. These examples are interesting since their main objective is the same as MPCS, to train and improve individuals to real-life scenarios. In the case of *Sidh* it is a game to train firefighters, and *NIMSpro* is a simulator to train emergency responders from all disciplines in response (e.g., hostage criminal situations or natural catastrophes). This study made it possible to be more aware of what training-based simulators aim for and some of their particularities. Another familiar genre of simulation-based training not presented in this document is driving simulators.

2.4.1 Sidh

Sidh[Backlund et al., 2007][Williams-Bell et al., 2015] is a firefighter training simulator developed in collaboration between the University of Skövde and the Swedish Rescue Services Agency. *Sidh* stands apart by integrating computer game hardware and software into an innovative interaction model tailored for firefighter training. The game's objective is to have the player scan each required zone/area and, if needed, rescue the victims. After scanning one area, the player moves to another one. A player is successful if it scans all the available areas within an optimal time.

The simulator environment is in a specialized *Cave* environment (Cave Automatic Virtual Environment or Digital Cave), where players engage using gesture-based steering and physical movement facilitated by a specific model of a lantern (called *FogFigher nozzle* in the article) and *accelerometer*-equipped boots. This simulator simulates both physical and psychological stressors encountered in real-life firefighting situations. With participants wearing firefighting gear, including a breathing mask and boots, and dealing with elevated temperatures from equipment, the simulator tries to replicate authentic conditions.

Sidh uses the Half-Life 2 game engine with the required modifications to support a Cave environment. In each session, five game instances are running on separate computers. One is the primary instance (server), and the others are spectators to the server with the camera at the same point but with different angles. The sensor values are provided to the game through a joystick API, and specialized algorithms have been developed to map sensor readings to game actions. With a game engine based on a commercial game, the simulator has a wider range of tool support. Also, it is easier to develop more missions because it is based on a proper Editor already configured.

Figure 2.8: *Sidh* environment

Although the training type differs from MPCS and is a single-player game, the study of this game was valuable since it is also a training simulator for emergency disasters (fire). This study provided some information on how different technologies can be used in a simulator game (e.g., the innovation of using a *FozzyFighter* to control the movement of players).

2.4.2 NIMSPro

NIMSPro [C3] is a 3D game engine designed to simulate a first-person point-of-view experience within a virtual real-world environment (real scenarios are designed in the game) developed by C3 Pathways. It allows users to navigate in true-scale 3D models of real locations and even enter buildings. It is a multiplayer simulation system, enabling multiple players to interact within the 3D environment, engaging with each other, the surroundings, and various elements such as victims, vehicles, and even adversaries like criminals.

It is used to train professional emergency responders across various disciplines for response, mitigation, and recovery from significant emergency incidents. This simulator encompasses various scenarios, from natural disasters to terrorism simulations, active assailant scenarios, fire incidents, and transportation accidents.

C3 Pathways' approach combines the *NIMSPro* 3D simulator with expert instructors to ensure effective training experiences that improve performance. This approach involves creating dynamic emergency scenarios for responders to manage in real time, facilitating practical learning through repetitive exercises. With this, this simulator is not only done by developers, but it has the expertise of real professionals in the area, which leads to more effective training simulations.

Also, *NIMSPro* is extensible, which means if a client wants different scenarios for his needs, there is no obligation to do a whole new simulator. Instead, the development team can add the required functionalities and scenarios to the game more easily and faster.

NIMSPro3™ screenshots from Univ of N Florida Active Assailant Training

Figure 2.9: NIMSPro screenshots [C3]

Unfortunately, no information was available publicly about the architecture and how they approach some key questions regarding multiplayer or simulators, such as the handling of the game world and state. Although this does not have the type of simulation that MPCS aims for (coordination training for an oil spill situation), it was interesting studying it since this a sophisticated example of what MPCS wants to achieve with his simulator (the same purpose which is to train for real-life scenarios).

2.5 Multiplayer games and their architecture

The multiplayer industry has evolved, and more and more money is involved. With this, getting real and concrete information about architectures, approaches, and components of famous multiplayer games (with rare exceptions) becomes difficult. Therefore, the games mentioned in this section are more for a literature review (none of these games are known except for Fortnite) point of view, where the objective of this study is mainly to understand better the different components that can exist in a multiplayer game and its architectures and how they handled data affecting game world and state. This section describes the following games: Mammoth, Immersive Deck, Adventure 2, and Fortnite.

2.5.1 Mammoth

Mammoth [Kienzle et al., 2009] is a massively multiplayer game research framework designed for experimentation in an academic environment. Mammoth provides a modular architecture where different components, such as the network engine, the replication engine, or interest management, can easily be replaced. Mammoth also offers a modular and flexible infrastructure for defining non-player characters with behavior controlled by complex artificial intelligence algorithms.

The main objective of Mammoth is to provide an implementation platform for academic research related to multiplayer and massively multiplayer games in the fields of distributed systems, fault tolerance, databases, networking, and concurrency. Mammoth was built due to the lack of existing software documentation for implementing a multiplayer game or the knowledge and hardware needed for testing different types of components in an architecture.

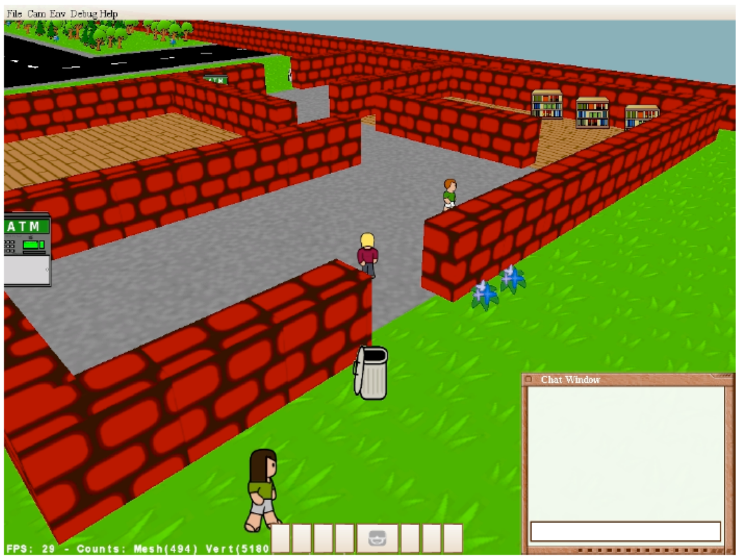


Figure 2.10: Mammoth 3D Client Interface

Figure 2.10 shows an example of an interface presented in the Mammoth game. As we can see, there are no spectacular graphics, but, of course, this wasn't the goal of Mammoth. They were only concerned with showing a pleasant one.

Now, relating to the technical aspects of the game, there are various complex characteristics. For example, we can use different types of network topologies in this game depending on what we want (defined in the config file of our game). In Figure 2.11, we can see an example of that.

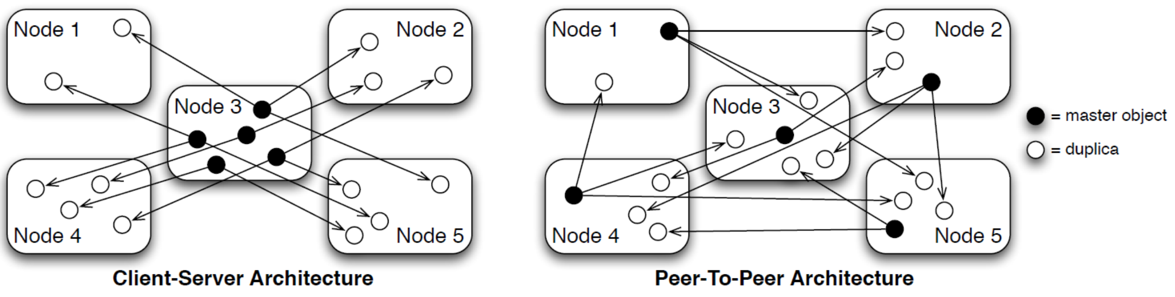


Figure 2.11: Mammoth with different Network topologies by migrating Master Objects

In this approach, the game objects are duplicated, encapsulating the state of the game objects that need to be distributed to players. Every machine that needs access to the game state creates a new local instance of the object, called *"a duplica"*. To assure consistency during the game, one of the copies of the duplicated objects created by these instances is assumed as the "duplication master".

In Figure 2.11, we can see an example of how that can be used to manipulate the different network topologies. In the Client-Server architecture, all the master objects are assigned to the server, and with P2P, the master objects are uniformly distributed over all clients.

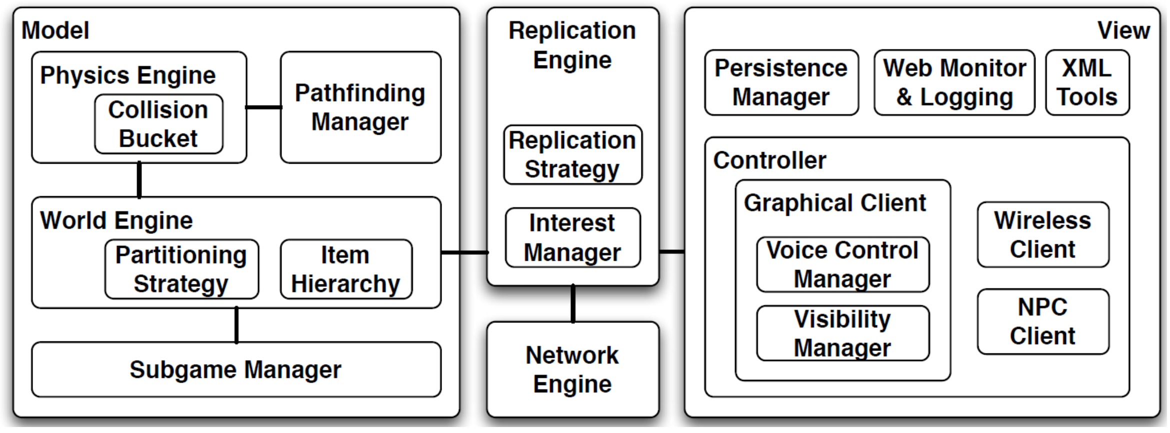


Figure 2.12: Components of the Mammoth Framework

In 2.12, we can see the different components of the framework used in the Mammoth game. In this game, the components communicate with each other via interfaces, so it is easy to remove and add/replace components in the game. For example, the Object factories and their respective configuration files define the instantiation (instance) of a given implementation, so a researcher/user can specify what component he wants to use (in the configuration file) and test it by launching the game. The game will initiate, and the appropriate engine will instruct the managers on how to act. By doing this, Mammoth takes off the developer's responsibility for knowing all the details of a component.

Giving more emphasis to the Network component (it is the most relevant one considering the scope of this thesis), this component makes communication possible in the framework. It allows communication of duplicated objects via direct asynchronous messaging (via remote calls) and publishes/subscribes broadcast support. Currently, the Mammoth framework has three network engines: *Stern* (communication is routed through a central hub), *Toile* (fully connected network), and *Postina* (a self-organizing peer-to-peer network engine using tree-based broadcast). In addition, a *Fake* network engine is provided, which uses shared memory and emulated serialization to route messages across components. *Fake* is mainly used when executing units on components that depend on a network engine.

Considering the game itself, in each session, the number of players is created a-priori (fixed number), so if a player joins a session, we will take over an inactive instance of a player of a determined session. If the player leaves, the (instance of a) player will become inactive again ("frozen").

Although this game is not similar to what MPCS pretends since it is not the genre type and it can not be implemented and used, it was interesting studying it because of some aspects of the game and framework:

- The capacity to be a manageable architecture where we can easily replace components;

- The important components (in particular network engine) and how they interact with each other;
- The possibility of having two different network topologies based on our configurations;
- The game itself (it is multiplayer), seems fun the way we can interact with each other and even we can create sub-games inside of the real game;
- In the article on this game, the authors give a good overview of the existing problems in creating/implementing architectures and how they tried to combat them.

2.5.2 Immersive Deck

Immersive Deck [Podkosova et al., 2016] is a system where several users can be simultaneously immersed in a virtual environment (VE), explore VEs by real walking, and interact with each other naturally and intuitively. This system is built from hardware (not available to the common public) or available prototypes and is easy to set up. The multi-user system architecture includes global position tracking, full-body motion tracking, user communication, interaction, and object tracking. Figure 2.13 shows how the players are in real-life and how they can see each other in the VR environment.

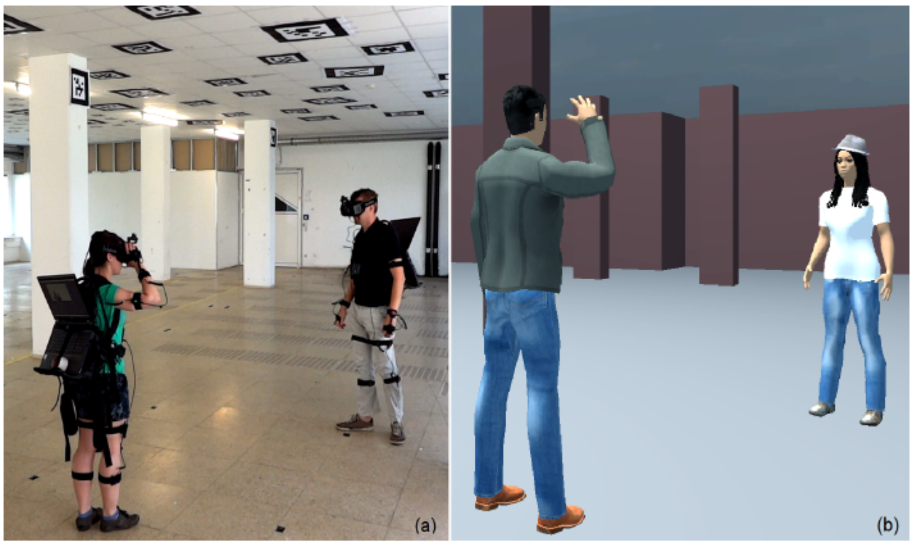


Figure 2.13: Immersive Deck real-life and in-game scenarios

One negative point of this article is that the authors are not explicit about what technologies/architecture they use. Regarding the technical aspects of the game, the article's authors are only explicit on data transportation and how data is handled when a player possesses an object of the environment (and how other players see and (cannot) interact with the same object).

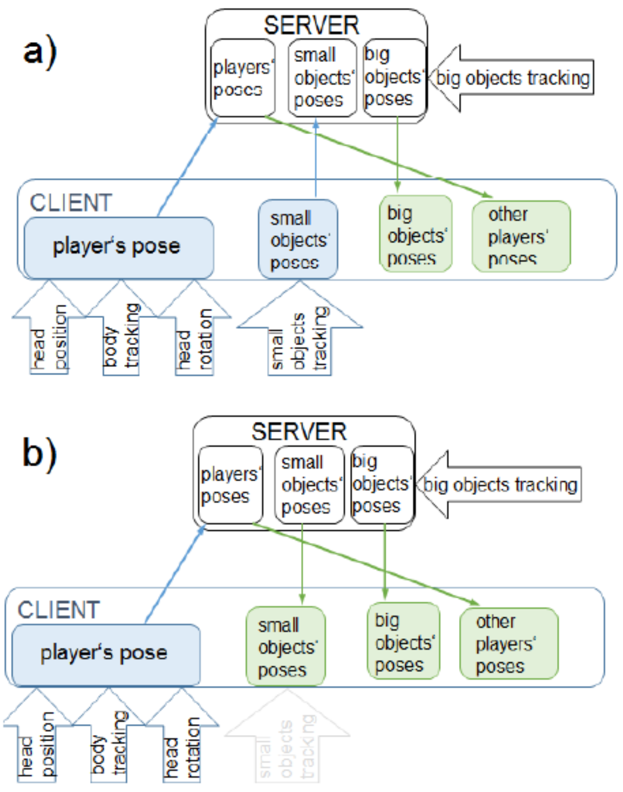


Figure 2.14: Data exchange between client and server

In Figure 2.14 we can see the data exchange between the server and a client. In a) it is when the mobile tracking of small objects is active on the client (when the client is holding a small object). In this case, the client application sets up poses of corresponding *GameObjects*. The poses of the server copies of these *GameObjects* are synchronized. In b) it is when the client application is not tracking small *GameObjects*. All synchronization is done via UDP.

This system uses the network built-in in Unity3D as the main component of its network implementation. In this architecture, each client only communicates with the server and not with other users (directly). In this case, for example, when a player does something that changes a common object, first, the change is made on the client side and only then goes to the server (non-authoritative).

The biggest drawback of this game compared to what MPCS needs to be is that this game was only used and built to be used in a pre-defined context (in a closed environment of about 200m2). In contrast, in the MPCS project, the environment cannot have a range limit because the users can be anywhere on the globe while playing the simulator.

The most relevant aspects of this game are:

- The genre itself is precisely what we want on the project, to players interact with each other and even with the objects in the field of action;
- The data flow and how the game authors approach the different state of an object (where is in use by another player, i.e.);

2.5.3 Adventure 2

Adventure 2 [Agostinho, 2013] is a platform for the creation of multiplayer pervasive games for mobile devices. It makes use of augmented reality, location and direction (GPS and compass) and of QR markers. It allows for the easy construction and orchestration of this type of games. It also allows for elements of narrative backgrounds with a logical sequencing of events, cooperation and competition, and a ranking/achievements system. This game uses Petri Nets as the constructor of this games by using generalization functions (e.g. "Point and Click, "Listen and Click", "QRCollect"). In Figure 2.15 we can see a screenshot of the mobile interface of the game. Adventure 2 [Agostinho, 2013] is a version 2.0 of Adventure with some adjustments and upgrades.

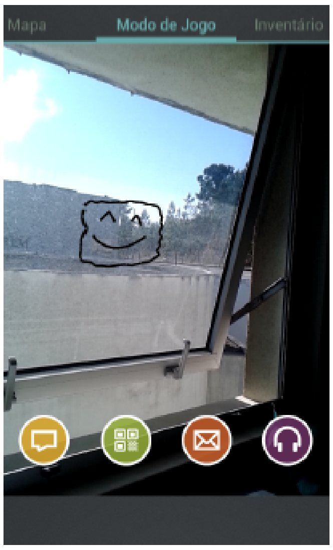


Figure 2.15: Adventure 2 mobile interface (Camera View)

Agostinho tried to improve the overall system, and his main objectives were solving escalating issues, portability, designing a new player interface, and designing a new editor for the game. As shown in Figure 2.16, this architecture has reused (and remade) some components from the previous version of Adventure with the addition of new components, like GCM (Google Cloud Messaging) for push notifications from the cloud for less latency and more scalability.

The architecture of Adventure 2, as shown in Figure 2.16, has several components. I will only elaborate on the main ones and on the server side. First, the Android client (presented in Green) has a *GameViewPager* that creates five distinct classes, which are the five screens of the game, including the list of all the actions, the map (simple map with self-location), a camera view where the action actually occurs (executable actions with augmented reality), an inventory of the items and achievements, and a ranking for the player to see his points.

The Google Cloud Messaging System (GCM) (in purple) receives a push message (through HTTP request) from the game instance with the new possible actions in XML whenever a player modifies the game's state. The client's GCM Receiver class is activated when a message is received. After receiving the content, the

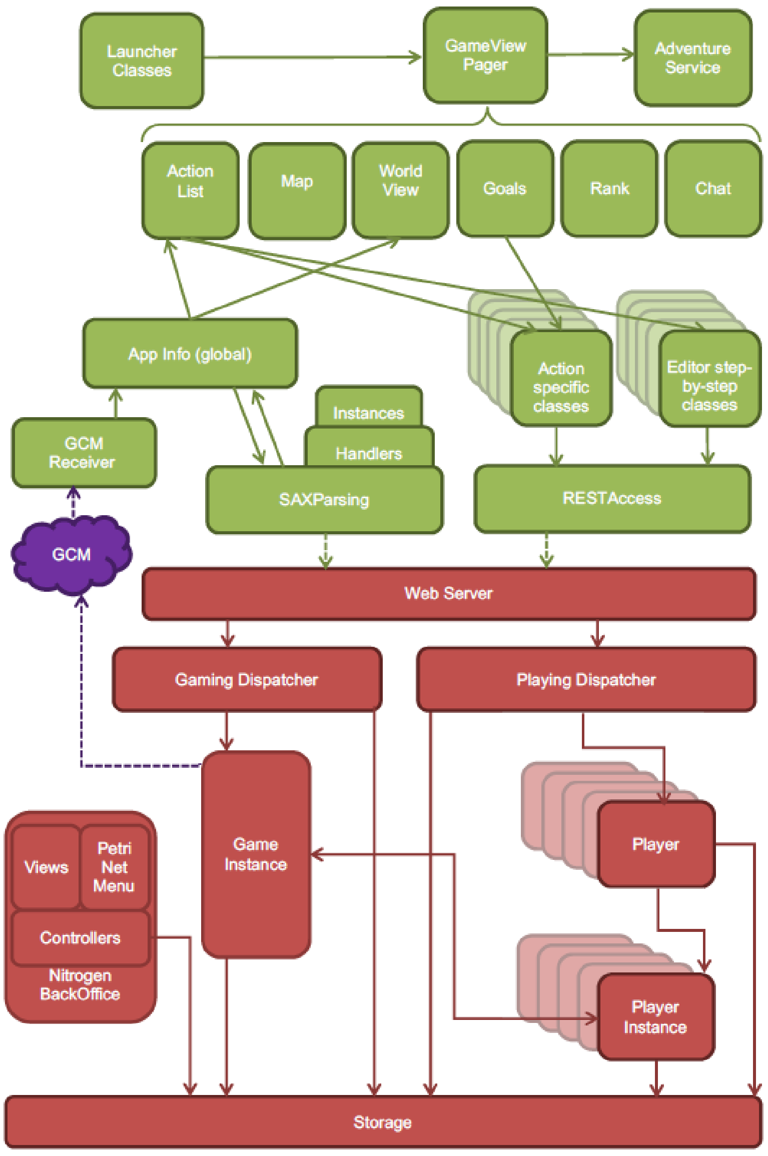


Figure 2.16: Adventure 2 architecture

AppInfo then parses the XML (using SAX Parsing) and changes the *ActionList* and Map to include the new actions (Action Specific Classes). Polling to the server is done when, i.e., the player does a specific menu switch or when an action is finished.

Regarding the Server side, the Erlang server receives requests on its *WebServer* process. A message is then sent to the *Playing Dispatcher* (if it is private – concerning just the player's state) or the *Game Dispatcher* (if it is of a shared nature – affecting the game state itself). Adventure Web was ported into Erlang to be launched along with the *Gaming Server*. The Back-Office runs on the Nitrogen platform (which uses Erlang and the same web server from previous version - *Mochiweb*). Therefore, it encompasses the views for each game state element (Game, Team, Action, Resource, Goal, and Bridges), the view for managing game creators (people that create the games), and the menu with the visual Petri Net interface.

When a particular game element is created, edited, or deleted, it creates a call into the controllers. Then, the controllers make a call to Storage (via RPC call), which acts as the data model of the website. Games have two states: running (playable) or stopped (can only be edited). If a creator makes some actions in the mobile editor, they will also be visible on the website for further editions.

The most relevant aspects of this game are:

- The genre itself is similar to what we want on the project, to players interact with each other and even with the objects in the same space/zone;
- The use of Petri Nets for architecting different games using generalized actions;
- The architecture is similar to what we intended to use in our project. It seems simple, explicit, and well-documented;

2.5.4 Fortnite

Fortnite [Epic] is a very famous free battle-royal game developed by Epic Games, whose main objective of this game is to be the last man (or team) alive. Doing a simple description of the game, in Fortnite, players drop into a map, either on their own (solo) or in team (duos or squad), alongside ninety-nine (more or less) other players. After landing, the players need to collect as many items and weapons as possible (players can pick up a limited number of items/weapons). There is also the possibility of do some constructions by using the materials presented in game. By killing the other players and also avoiding the storm in order to stay alive, the last player (or team) alive wins the game. Figure 2.17 shows an interface of Fortnite.

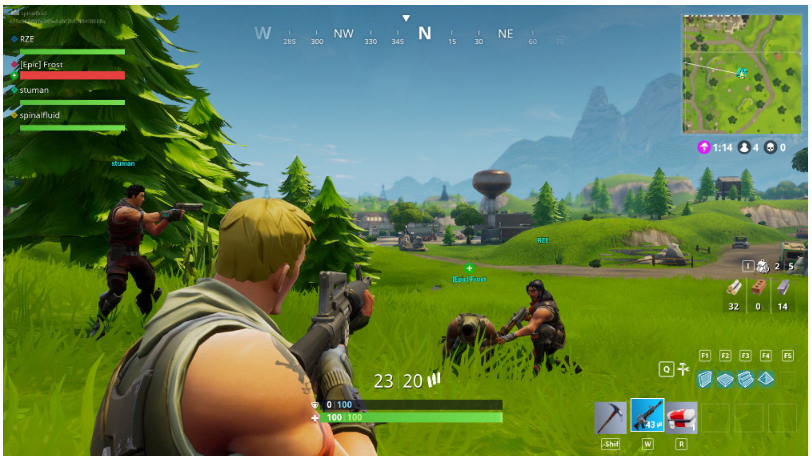


Figure 2.17: Fortnite interface in-game

The case study of this game showing his features and the characteristics of its multiplayer service was shown in a conference done by Amazon (*AWS re:Invent 2018*) [Amazon2018]. There are not much information about and that is the reason it is the last game mentioned in this chapter, because, although it is not a much deeper study, it is particular interesting to describe an architecture (even at a high-level perspective) of a very popular game.

As demonstrated by Chris Dyl (Director of Platform at Epic Games), Fortnite processes 92 million events per minute, which represents nearly 62 billions of events per day, also, at the peak of a day they process 40GB of data, so availability and scalability it is essential in this game. For this to be achieved, Fortnite, runs nearly entirely on AWS (Amazon Web Services), including its servers, Back End services, databases, websites, and analytics pipeline and processing systems.

For example, they use *Amazon S3* for storage. With *S3* it is possible to achieve great levels of data availability, security, scalability, and performance. They also use *Amazon Kinesis* to process and digest all the information and data in real-time. In Fortnite they have two types of pipelines, a *near Real-Time Pipeline*, used to process and store temporary data (this data is stored in *Amazon DynamoDB*). This temporary data can be related to Grafana, Scoreboards API or Limited Raw Data. The other type of pipeline is a *Batch Pipeline*, where almost every data is stored by using *Amazon S3* (data related to API's, several services embedded in Fortnite, among others).

Epic Games uses many other services offered by AWS, including *Amazon EKS* (for micro-services management), *Amazon Guardduty* (for threat detection), *Amazon Neptune* (for social network and anti-fraud).

Since it does not contain much detail, the architecture presented in *AWS re:Invent 2018* [Amazon2018] is shown in Appendix A, containing some details about it (services used, different main components and analytics).

Despite, the aspects mentioned in this topic, it is hard to obtain more detailed aspects of the architecture (as it is for other famous games) for obvious reasons (especially financial).

The most relevant aspects of this game are:

- It is a famous game;
- How AWS provides scalability;
- It contains a various number of player in a single session;
- Players can interact with items and with each other.

2.6 Frameworks used in online context

This section describes three frameworks (RTF, NAVER and 2Simulate) that are used in an online context. Since the frameworks mentioned in this section are more for a literature review (and not for real implementation), the objective of this study is mainly to understand better the different approaches (and concepts) that can be used regarding a multiplayer context (how data synchronization is assured, e.g.).

2.6.1 RTF

Real-Time-Framework (RTF) [Glinka et al., 2007] is a middleware system that aims to support the development of modern real-time online games. The RTF automatically distributes the game state among participating servers, supports parallel state update computations, and ensures efficient communication and synchronization between game servers and clients. The users of this framework can access the RTF via an interface with a high level of abstraction that improves issues related to time-consuming and error-prone programming.

The RTF supports three distribution concepts for real-time multiplayer games within a multi-server architecture: *zoning*, *instancing*, and *replication*. One feature of the RTF is that it allows for combining these three concepts within one game design. In Figure 2.18, explained in the following, shows how the distribution concepts can be combined to adapt to the needs of a particular game.

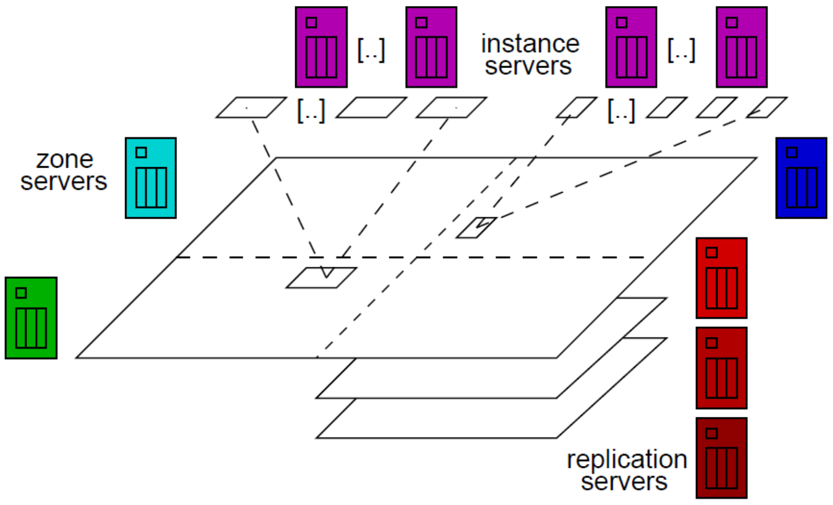


Figure 2.18: RTF, servers with combined concepts of *replication*, *zoning* and *instancing*

Zoning separates the world into different zones (in this case, four zones shown in Figure 2.18). Clients can move between zones, but no events occur in inter-zones. Also, the calculations in each zone are independent of the others.

Instancing is used to distribute the computational load by creating multiple copies of the more frequented subareas of the world. Each copy is processed entirely independently of the others. With this, the developer can specify different instance areas in the game world that will create new instances in the servers and change the clients and entities to these zones.

Replication distributes the entities among all servers. Each server has a list of the active entities presented in his session and a list of the entities that are part of the other servers (in these entities, the server only has read-only access).

It is common to use portals to move players and entities between zones. With RTF, the developer can use three portals: *SpaceToPointPortal*, *SpaceToSpacePortal*, and *BidirectionalPortal*. I will not go deeper on this because it is not especially important regarding this study since I cannot use this framework because there is no valid and proper documentation (beyond the paper that I studied).

The responsible server for determined clients must ensure that all the information the clients need is sent (accessing the proper game state of the game). This area is usually called the "client's area of interest". Determining this area is done by the developer, known as *Interest Management*. In RTF, this technique is a publish-subscribe method. So, the developer needs to define the proper topics the client must subscribe to or publish to an entity of the game. RTF also has an Euclidean distance algorithm to help the developer in the implementation. When a client is subscribed to an entity, the server transfers the required data, and a callback from RTF is done to inform the client about the new entity and its properties (displayed on the screen, i.e.). The opposite occurs when an unsubscribe method is done.

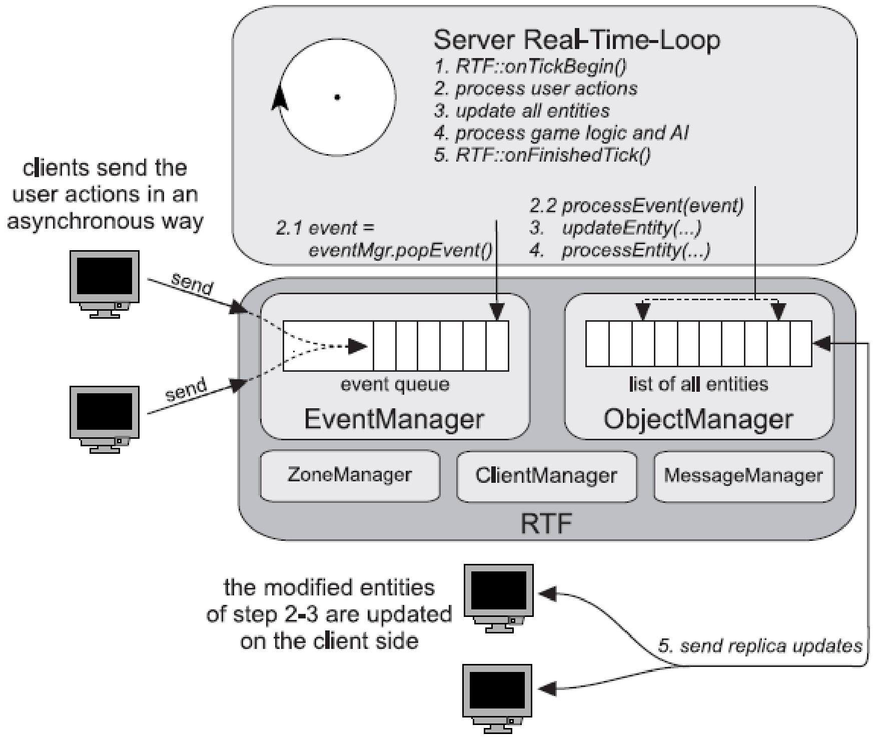


Figure 2.19: Data flow combining RTF and a server with areal-time-loop

In Figure 2.19, it is possible to see how this framework processes the game state (data across all entities involved) and check how client data is managed in RTF. The processing of the game occurs at the server level (real-time-loop), and it is asy to integrate RTF on this loop. The developer needs to take the actions done by the users (execute the game logic) and define the events done by the RTF (new connections, when the client changes his zone, . . .). The *ObjectManager* provides access to the list of Entities and the game world, and it is also where the new entities are recorded. The *EventManager* gives access to all incoming events from clients or servers, which are enqueued by the RTF in each loop iteration. The main objectives of the *ZoneManager*, *ClientManager*, and *MessageManager* are to retrieve information about the current distribution of the game among the different servers.

In this approach (combining real-time-loop with RTF), the clients asynchronously send their server their actions in the form of events. It is necessary to have access to the ObjectManager's entity list in order to process these events because some actions can affect the other entities presented in the game. Following this process, the game's world is and state are update as well as the appropriate game logic is done. When using the function "onFinishedTick()", the loop is finished, and the information from the updated entities is sent asynchronously to any interested clients or servers.

With the study of this article and framework and on this specific part, I tried to understand better the concepts of *Instancing* and *Replication* (very commonly used in Multiplayer Games) and *Zoning* and how these concepts can be implemented in a game. I also found this approach interesting because of how they integrate RTF with a real-time loop and how the data is processed (using publish and subscribe methods). The biggest drawback of this approach is the need for endless loops (the real-time loops) because it spends memory without any need (if nothing happens in the game, the real-time loop is made with no purpose).

2.6.2 NAVER

Networked and Augmented Virtual Environment aRchitecture (NAVER) [Park et al., 2003] is designed based on a distributed system consisting of multiple hosts on the network to integrate 3D virtual space with various interfaces and applications. This framework makes the system extensible, reconfigurable and scalable. NAVER was designed to support Gyeongju virtual reality (VR) Theater. The objective of designing and building the VR Theater was to construct a flexible public demonstration place for VR technology as a new medium for interactive storytelling of diverse artistic expression and virtual interactions with the public.

With this approach, providing a script will specify the virtual space and system integration (explained in more detail further in the document). The XML-based script enables a description of the exchange of events between virtual space and specific applications or interfaces in the same context.

Therefore, the VR Theater can extend new functions or interfaces by adding EMs (external modules). The system can be specified for determined contents by reorganizing the EMs, and more performance can be achieved by increasing the number of server applications. Figure 2.20 represents NAVER framework.

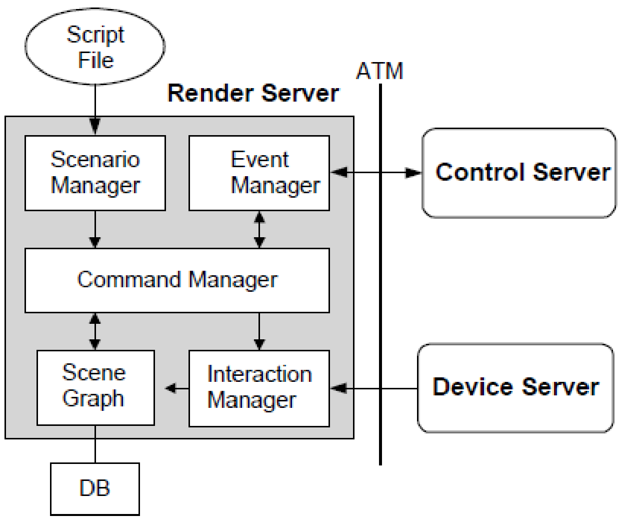


Figure 2.20: NAVER framework

The function of Scenario Manager is to validate and interpret the script files send by the user, which describes virtual environments, EMs, and how they are connected. Scenario Manager is compatible with script files in the extensible markup language (XML). The NAVER scripting language makes it easier to initialize virtual environment variables intuitively, build scene graphs hierarchically, and define connections between virtual world objects and EMs, which stand in for applications or human-computer interfaces.

The *Event Manager* handles the asynchronous message passing-based event transmission to and from the Control Server. In order to control the EMs controlled by the *Control Server*, the *Event Manager* enables the Render Server to transmit an event that includes an action.

Additionally, the *Control Server* can send a command and an event to the *Render Server* via the *Event Manager*. For example, the *Render Server* can activate the control server-connected beam projector. Users on the *Control Server* may also control the spotlight in a virtual space via a GUI. In the *Render Server*, the *Command Manager* processes receiving events in the queue every frame. The relationship between the number of events and real-time performance is vital. The size of the communication and the available network capacity must be considered. The Command Manager receives the command lists produced by the Scenario Manager and the Event Manager. The Command Manager then analyzes transmitted command lists and performs an appropriate action.

The *Interaction Manager* connects the user input from the *Device Server* to the *scene graph*, an abstract data structure representing a virtual space in the *Render Server*. The *Interaction Manager* allows real-time interaction, such as navigation or manipulation in the virtual space. For example, a user on the *Device Server* can navigate a virtual space using a force-feedback joystick, which vibrates when a collision occurs.

The *Sync Manager* provides multiple synchronous channels through a cluster of

PCs, making this channels as if a single computer manages them. For example, multiple channels are needed when a graphic display uses more than one image.

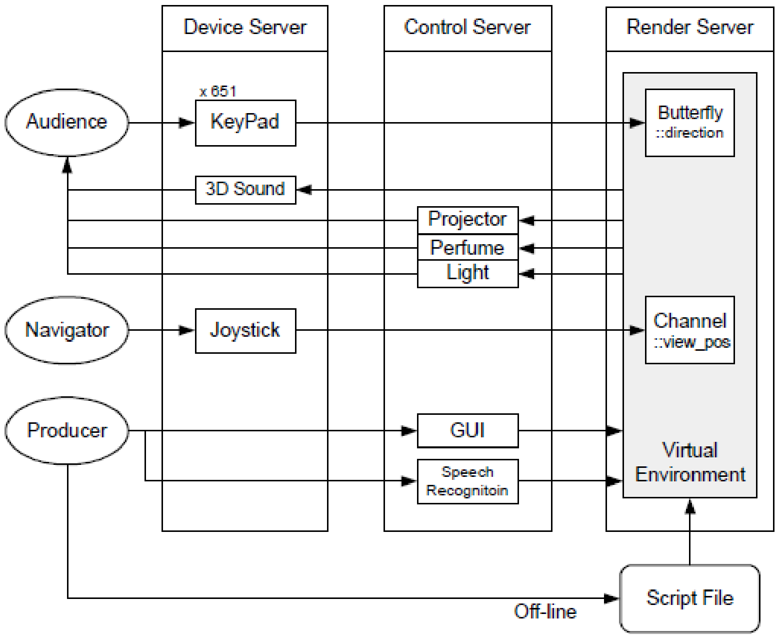


Figure 2.21: System integration of the Gyeongju VR Theater

In Figure 2.21, we see the different components interacting with the framework. In example, the Audience sees different components (presented in *Control Server*) in the theater, such as Projector and Light. These components on the *Control Server* are previously rendered in a virtual environment by the *Render Server* (this is accomplished based on the configuration files done previously by the *Producer*).

2.6.3 2Simulate

2Simulate [Gotschlich et al., 2014] is a simulation framework to make the integration of a wide range of models and simulation components like data recorders or image generators easier. Simulating large-scale complex real-time systems requires specific attention on the infrastructure that enables the real-time co-simulation of various sub-systems, so, it is important to have tools and infrastructures that make a system reliable and extensible. In this section, it will be presented how 2Simulate can do this.

The framework 2Simulate is written in C++. It mainly consists of the 2Simulate Real-Time Framework (2SimRT), the 2Simulate Control Center (2SimCC), and the 2Simulate Model Control (2SimMC) (2SimMC). The corresponding Component Diagram of this framework is shown in Figure 2.22.

The main module of 2Simulate is 2SimRT. It offers deterministic task scheduling and control in real time. It is delivered as API header files and Windows or QNX images (Libraries). When the 2SimRT does an application simulation, it is named Target. The 2SimRT API implements numerous real-time activities executed by

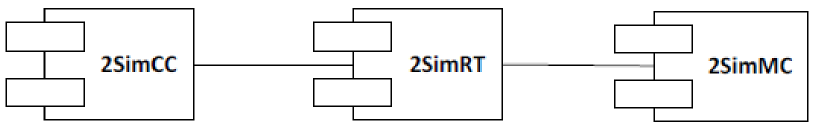


Figure 2.22: Components of the 2Simulate Framework

each Target. These real-time tasks comprise a variety of data connections to external devices or components and model control tasks. To control the data flowing through the internal and external interfaces, 2SimRT additionally uses a Common Database.

2SimMC is the component that abstracts model interfaces for 2SimRT. It supports models created in native C++, Advantage Framework, and MATLAB/Simulink. Targets may co-simulate across 2SimMC with multiple models. QNX (a real-time operating system), and Windows are both supported. Users may utilize 2SimMC by creating their models utilizing its API to generate native C++ models. 2SimMC is automatically incorporated into MATLAB/Simulink and Advantage Framework models during the code creation process.

The component 2SimCC configures the Control Center to meet specific requirements. It is a Windows executable that may be modified using 2SimCC project files, which are configuration files. Various Targets can be run, paused, or stopped by Control Center. It also uses the Target Data Dictionaries, a data access method that permits runtime change or presentation of Target data.

The simulation architecture of 2Simulate has a Control Center that can control several Targets that may co-simulate various Models and interact with various external systems.

The component architecture of 2Simulate is presented in Figure 2.23. 2SimRT provides many schedulable task templates and a shared database. The most common tasks are described in the figure. Doing an overview, SimpleTask is the most straightforward task type with no extra functionality. The user can modify it for his needs. With a UDPTask, one can schedule a UDP communication, and with TCPTask, a TCP communication. Using IPCTask, it is possible to do inter-process communication with other applications on the same machine. IOTask, enables users to connect to I/O interfaces such as switches or onboard computers, while Model-Task enables the execution of models created to be incorporated into 2Simulate. There are more task types whose properties and functions are mostly inherited from the significant tasks represented in the figure (check the article for more information and detail about this).

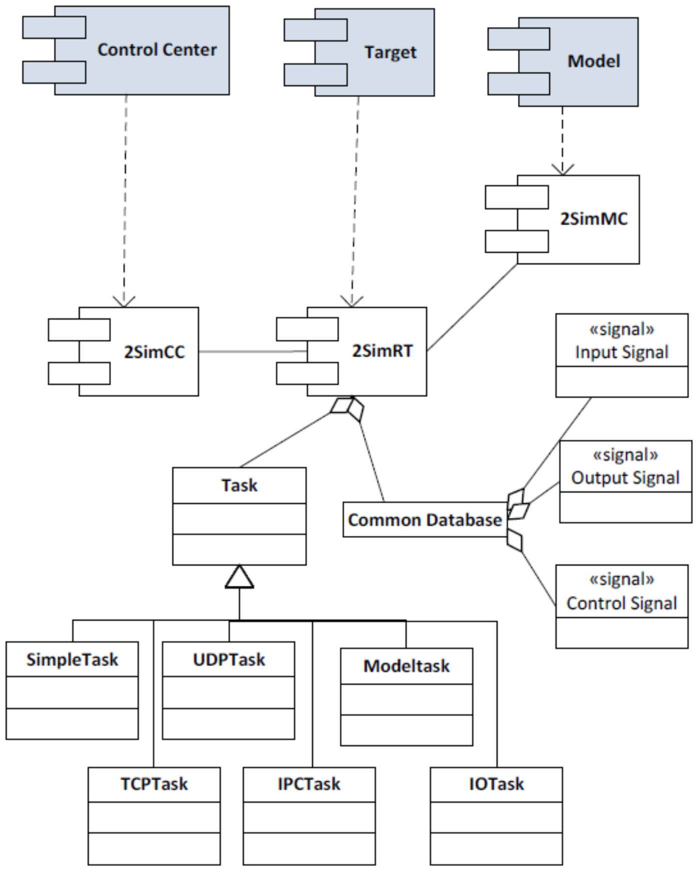


Figure 2.23: 2Simulate Component Architecture

2.7 Technologies in a multiplayer game

This chapter has the purpose of showing some tools existing in the market respectively for each component needed (main ones) in a multiplayer game (framework, hosting/cloud services, and databases). This study intends to demonstrate the various tools and services currently used, not considering the ones that best suit the project. In some tools, there was some difficulty in finding comparative terms between the others (since the company that developed them highlights the features they consider most important and there is no information about other issues afterwards).

2.7.1 Multiplayer frameworks (libraries)

This section discusses the different libraries that can be implemented to make a multiplayer networked game. This study aims to report on the main frameworks with their significant characteristics and some relevant characteristics, despite the fact that MPCS will not incorporate a game engine. Consequently, it will not use a specific multiplayer framework. However, it is essential to study them to learn how some aspects and issues related to multiplayer are handled (e.g., message system). Also, the comparative terms (e.g., Client-Server Architecture) were related to similarities to MPCS or its possibility to be used (e.g., if it has good documentation). Table 2.2 shows the different frameworks studied. Dark Rift2 [DarkRift2] was taken into account but not added to the table since it is more difficult to use since it has poor documentation/tutorials because it is not that popular (although it has an active community, it is free and contains powerful modifiable logging options).

	Netcode for GameObjects [Netcode]	PUN [PUN]	Photon Fusion [Fusion]	Fish Networking [Fish-Net]
Developer	Unity	Photon	Photon	Fish-Net
State	Early Development	Deprecated	Stable	Stable
Features	• Messaging System; • Game-Object oriented; • Data/Objects can be sent all at once; • High-Level networking	• Flexibility; • Integration with Unity; • Cross-platform; • Integration with Photon Cloud	• Tick-based Simulation; • Lag Compensation; • Client-side prediction; • Replication algorithms	• Lag Compensation (pro version); • Client-side prediction; • Local Remote Calls; • Server Authoritative
Embedded Solution	Yes, using Unity	Third-party	Third-party	Third-party
Client-Server	No (mainly Host-Server)	Yes	Yes (both)	Yes (both)
Scalability	Difficult	Easy	Easy	Medium
Documentation	Solid	Solid	Solid	Questionable
Feedback	Insufficient	Good	Good	Insufficient
Learning curve	Low	Low	Medium	Not found
Examples	Boss Room, Galactic Kittens	Real Table Tennis, Cars Battle	Fusion Kart, Dragon HuntersVR	Plethora

Table 2.2: Frameworks in multiplayer games

For example, in *Netcode for GameObjects*, one key aspect was the possibility to send Game objects and world data across a multiplayer session at once, so one message can be easily sent to every player by once. The main problems found for this framework regarding MPCS were that it mainly uses a P2P connection and is in early development. *PUN* is a library widely used because of his simplicity and the existent information and tutorials, but unfortunately it will be deprecated. *Photon Fusion* is a more capable and complete version of *PUN* (intention to replace *PUN*) but also a more complex and expensive approach. Lastly, Fish Networking is a server-authoritative framework that allows the use of dedicated servers but can also have a P2P architecture. One big drawback of this framework is the poor feedback and lack of guides on how to use it (with proper examples).

2.7.2 Cloud (or Hosting services)

Hosting is a vital part of a multiplayer game. Through it, it is possible for different players to access the game from any network (makes the game online) and even, if necessary, to have the database communicating with the game. With this, some tools are shown in table 2.3 as well as their characteristics. In this topic, it was not possible to make comparative terms (with pros and cons) due to the difficulty mentioned at the beginning of this chapter.

Name	Features	Price
Game-Server Hosting [Unity-Hosting]	• Integrated matchmaking; • Locates the optimum region for connectivity; • Automated orchestration; • SDK support; • Easy integration with Unity Engine; Examples: Among Us	Free for testing and an 800$ bill, pay-as-you-go after that
Amazon GameLift [Amazon]	• Deploy, operate, and scale dedicated servers, • Low-cost servers in the cloud; • Ideal for session-based multiplayer games; • Custom multiplayer game servers; • Auto-scaling; • Integrated matchmaking (FlexMatch); • Examples: Fortnite, Ubisoft	Pay-as-you-go
Google Cloud Game Servers [Google]	• Simplified management; • Flexible and extensible; • Global reach; • Open Source First (Agones); • Single Control Plane; • Customized Auto-scaling; Examples: Apex Legends	Depends on what is needed
Photon Cloud [Photon]	• Easy integration with Photon Fusion and PUN; • Scalable; • Global; Examples: Golf Clash, The Forest	Free up to 20 CCU, pay-as-you-go after that
HostHavoc [HostHavoc]	• Servers for popular games or our games; • 99,9% uptime guarantee; • 24/7/365 support; • Examples: Minecraft, Arma III Examples: Apex Legends	Depending on type of server

Table 2.3: Cloud (or Hosting) options for multiplayer games

Analyzing and comparing these hosting/cloud services, the advantages of using *Game-Server Hosting* are a region feature, which enables players to connect to the server nearer the region of the connections and the automated orchestration. *Amazon GameLift* provides easy scaling of a game (pay-as-you-go or auto-scaling options) and has the region configuration possibility. *Google Cloud Game Servers* is an easy, simple, and flexible way to manage servers and also it is Open Source First (using *Agones*). *Photon Cloud* is a good possibility if already using *Photon* technology (e.g. PUN) as well as favorable scalable options (can be tested freely). For last, regarding *HostHavoc* is a good hosting server for already existing games (not used for created games, but for game worlds, i.e., an *ARK* server) and it provides 24/7/365 support and a very high uptime rate.

2.7.3 Databases

By using Database services, it is possible to store and send the various data that a game uses, therefore, the data is not lost once it is transmitted. Since this project it is very important to save the data (either the logs to evaluate performance, or the various entities) it is vital to check some examples and their characteristics. In this study, some tools have been gathered that can be useful for this project, shown below in figure 2.4. Note that in this topic it was not possible to make comparative terms (with pros and cons) due to the difficulty mentioned at the beginning of this chapter.

Name	Features	Price
Oracle RDBMS [Oracle]	• Detect, and prevent security threats; • Single database for all data types; • Scalable; Examples: World of Warcraft	Pay-as-you-go
Firebase [GoogleFirebase]	• Cloud-hosted database; • Data is stored as JSON; • Synchronized in realtime; • Scaling.	Free up to 1GB, pay-as-you-go after
Amazon Aurora [AmazonAurora]	• Scalability; • Manageability; • Disponilibilty; • Uses MySQL or PostgreSQL; Examples: Fortnite, CAMPCOM	25 GB free, pay-as-you-go after
Microsoft Azure [Microsoft]	• Azure for MySQL or PostgreSQL; • Database integrated with Azure server; • Auto-scaling; • Cloud embedded; • Flexibility of data.	Pay-as-you-go

Table 2.4: Popular databases used in multiplayer games

The aforementioned database services have their own advantages and drawbacks, and it is a matter of the developer requirements. For example, *Oracle RDBMS* is a good selection to store all data types in only one database and it is the owner of *MySQL* (currently being used in MPCS), so there is easier implementation if needed. The Firebase service is a competent option if the developer wants to store data in *JSON*. Related to *Amazon Aurora*, it is a positive choice if using *MySQL* and also is scalable and manageable. Lastly, Azure has compatibility with *MySQL* or *PostgreSQL*. It is cloud-embedded and can have different types of data.

Chapter 3

Methodology and Work Plan

3.1 Objectives

This work aims to develop a simulator to train all parties involved in removing these pollutants. This simulator can do the actual training (exercises) for the members involved.

For this to be accomplished, several objectives should be achieved through the design and development of the multiplayer component of the MPCS project:

- Design the architecture of the Multiplayer Service framework, which contains the different technologies, services, databases, and communications channels and defines how the various components that are part of the MPCS system interact with each other (what information is sent/received, and in what way);

- Define the game state information model and develop the services responsible for managing the multiple players' UI connection to the game state;

- Develop the communication messaging channels for players to be able to connect to the server, which will enable the player to log into a session, play the game, and perform the various player actions;

- Designing and implementing all the actions that a player can perform in the game. Define and verify their prerequisites and process their impact on the game state, (except for the player movement and actions that directly depended on the digital twin (+MOHID) component);

- Design and development of an action log sub-system. This system should be responsible for keeping track of the player's actions. The MPCS manager should be capable of inspecting what actions a player has done in a session. The log report's evaluation and analysis for assessing the learning purposes are not in this thesis's work scope.

3.2 Approach

For this project to be successful, not only is cooperation between the mentioned organizations necessary but mainly collaboration within the development team, since for the project to be successful, all parties involved must meet the proposed expectations and goals, ensuring the interoperability of each component. As seen in Figure 3.1, different entities are involved in this process, so, as said, efficient communication is crucial.

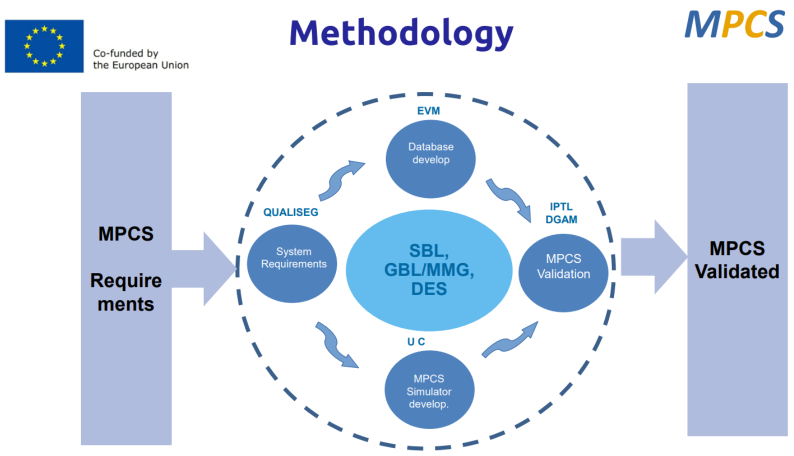


Figure 3.1: Methodology and parts involved in the development of the MPCS ([Sampaio et al., 2022])

A Scrum methodology was used with the UC development team, with weekly meetings between all participants. In Scrum [da Costa Ferraz, 2016], the development team collaborates with each other to evolve the software in small steps, also called sprints. Each sprint lasted between 2 weeks, thus involving all team members in the development area and decreasing the risk of failure to meet requirements and goals.

In Scrum [da Costa Ferraz, 2016], there are three significant roles: *Scrum master* (the person responsible for ensuring that the project is carried out according to practices and that it evolves as planned), *Product Owner* (stakeholders), and *development team* (the team who is responsible for developing the product). All the tasks and features that need to be implemented are called backlog. In the case of this project, the backlog (in a high-level perspective) is all the components presented in the simulator (Multiplayer Service, Digital Twin, UI Interface Generation, and the Game Editor). The backlog for the component related to this internship is shown in B.1.

Through the sprints, it was possible to assess and clearly understand what software elements needed to be implemented and in what time frame. Also, the participants better perceived what was being developed by team members responsible for the different components.

At the beginning of the development, I (and we) did not use sprints, only the weekly meetings. This is because each developer was doing independent work (regarding its component). However, date deadlines were used for each task (and week deadline).

3.3 Work Plan

In this first semester, the main priorities were:

- To understand what MPCS is all about;
- To study different approaches related to multiplayer contexts, such as games, frameworks, and technologies;
- To propose an initial architecture of the MPCS system;
- Write the intermediate report.

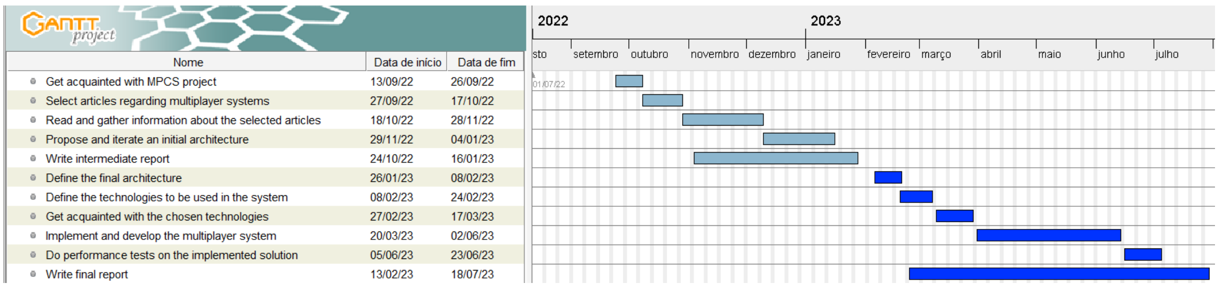
Regarding the second phase of this internship (second semester), the planned main goals to be reached were (defined at the time of February):

- Before sprints:
  - Define the final architecture and the Data Model;
  - Define the frameworks and technologies to be used.
- Sprint 1:
  - Learn the technologies chosen and validation tests of the technologies;
  - Migration to the server.
- Sprint 1-6:
  - Develop the multiplayer system. This contains the handling of the game world and state across the session and also the implementation of the action system. These actions should have a defined impact in the game state, and also, some actions should have their pre-conditions (simulating real life);
  - Integration with the modules.
- Sprint 5:
  - The action log system.
- Sprint 7:
  - Access performance of the implemented solution.
- During all semester:
  - Write the final report;

- Write the architecture report (deliverable to the consortium).

More detailed tasks and objectives of each sprint are presented in B.1.

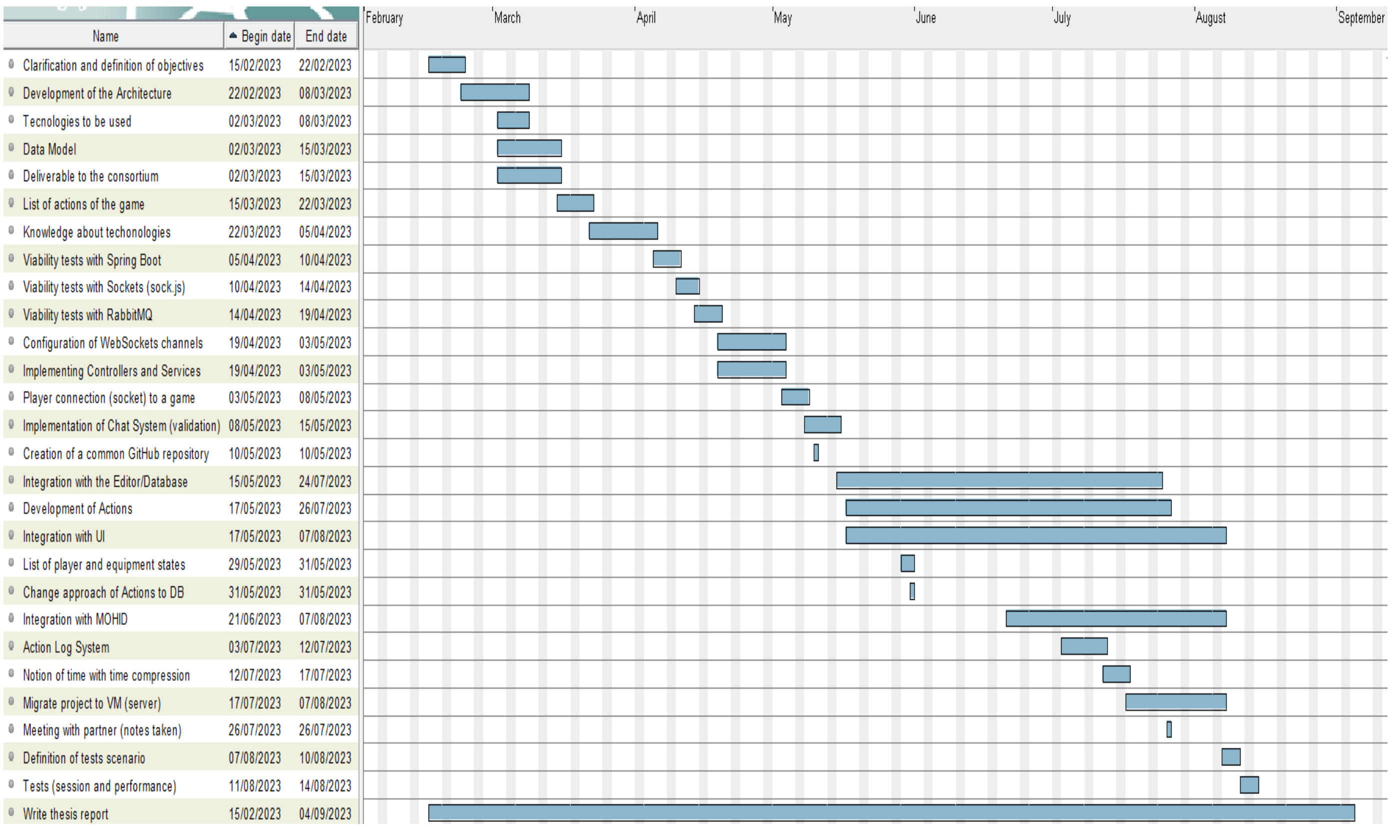
In Figure 3.2, it is possible to see the different tasks and dates that were done in the first semester (in light blue) and also the planned dates to do the various tasks needed (second semester, in blue). This plan was developed before the 2º semester.



Nome	Data de início	Data de fim
Get acquainted with MPCS project	13/09/22	26/09/22
Select articles regarding multiplayer systems	27/09/22	17/10/22
Read and gather information about the selected articles	18/10/22	28/11/22
Propose and iterate an initial architecture	29/11/22	04/01/23
Write intermediate report	24/10/22	16/01/23
Define the final architecture	26/01/23	08/02/23
Define the technologies to be used in the system	08/02/23	24/02/23
Get acquainted with the chosen technologies	27/02/23	17/03/23
Implement and develop the multiplayer system	20/03/23	02/06/23
Do performance tests on the implemented solution	05/06/23	23/06/23
Write final report	13/02/23	18/07/23

Figure 3.2: Gantt Chart of the tasks regarding the first (in light blue) and second (in blue) semester

Figure 3.3 shows all the effective processes during this semester. It was done at the beginning of the 2º semester.



Name	Begin date	End date
Clarification and definition of objectives	15/02/2023	22/02/2023
Development of the Architecture	22/02/2023	08/03/2023
Tecnologies to be used	02/03/2023	08/03/2023
Data Model	02/03/2023	15/03/2023
Deliverable to the consortium	02/03/2023	15/03/2023
List of actions of the game	15/03/2023	22/03/2023
Knowledge about techonologies	22/03/2023	05/04/2023
Viability tests with Spring Boot	05/04/2023	10/04/2023
Viability tests with Sockets (sock.js)	10/04/2023	14/04/2023
Viability tests with RabbitMQ	14/04/2023	19/04/2023
Configuration of WebSockets channels	19/04/2023	03/05/2023
Implementing Controllers and Services	19/04/2023	03/05/2023
Player connection (socket) to a game	03/05/2023	08/05/2023
Implementation of Chat System (validation)	08/05/2023	15/05/2023
Creation of a common GitHub repository	10/05/2023	10/05/2023
Integration with the Editor/Database	15/05/2023	24/07/2023
Development of Actions	17/05/2023	26/07/2023
Integration with UI	17/05/2023	07/08/2023
List of player and equipment states	29/05/2023	31/05/2023
Change approach of Actions to DB	31/05/2023	31/05/2023
Integration with MOHID	21/06/2023	07/08/2023
Action Log System	03/07/2023	12/07/2023
Notion of time with time compression	12/07/2023	17/07/2023
Migrate project to VM (server)	17/07/2023	07/08/2023
Meeting with partner (notes taken)	26/07/2023	26/07/2023
Definition of tests scenario	07/08/2023	10/08/2023
Tests (session and performance)	11/08/2023	14/08/2023
Write thesis report	15/02/2023	04/09/2023

Figure 3.3: Effective Gantt of the 2º semester

Compared to the planned process (B.1), it is possible to check that some actions were delayed during the semester (e.g., server migration). Regarding the plan done in the 1º semester, the obvious difference is the delivery delay of the thesis (September instead of July).

The use of the Sprint technique started later than expected because a document related to the architecture needed to be delivered to the consortium (partner). The sprints should be started during the initial of April and were started in the initial of May. Although the delivery was in mid-March, some tasks that should have happened before were delayed until after the delivery of the document.

In the initial of the 2º semester, we decided to postpone the final delivery date of the thesis to September. This has given us more time to develop a better product and thesis report. So, the tasks were assigned to each sprint after that decision.

However, except for these drawbacks, the work was (in general) well-planned, and no big differences were made during the semester related to tasks assigned to each sprint. The migration to the server task was the biggest difference, and it was because it was supposed to be done earlier to ensure that the technologies worked in the production server in an early stage, but due to time, that was not possible.

3.4 Risks

Risks are inherent in any software project. And this one is no different. Therefore, a collection and definition of the possible risks that can happen is vital. After defining these risks, a mitigation plan for each risk is also essential. Hence, some risks found (and the corresponding mitigation plan) are presented in table 3.1.

ID	Risk	Impact	Mitigation Plan
R#1	Initial Requirements need revision	High	• Collect all possible data and information regarding the simulator and its characteristics; • Periodic meetings with the owners.
R#2	Deadlines that are too short for the complexity of the project	High	• Calculate realistic deadlines for each task; • Define clear tasks (and how they can be done);
R#3	Inter-dependency between all components. One component can provoke delay to another	High	• Weekly meetings between all developers; • Verify if all developers are completing their tasks. • Each component should have a level of independence.

R#4	A member quits	High	• Meetings and Team Building sessions to ensure that no one loses focus or confidence; • Another member will have to take over.
R#5	Lack of communication (the responsible for one component does not understand what others are doing)	Medium	• Weekly meetings between all developers; • In meetings, each developer explains to the others what they have done and what they are doing.

Table 3.1: Risks and Mitigation plan

The interdependence between components (R#3) is may be what most affects and impacts the development of this internship. There must be a good level of independence between the multiplayer component and the others. Hence, cases where dependencies with each component are necessary, and strategies to mitigate this dependency were defined.

3.4.1 Module Interdependencies

- Multiplayer Service - Game Editor:
  - Use of Entities (e.g., equipment, players, or sessions): Create sample classes that allow simulating the classes in the game interaction without connecting to the Database (they only exist in memory in the application). These classes are only created with the attributes that are strictly necessary to optimize the time spent;
  - Calls to the Database: Create simple examples/tables in the Database if more severe cases require the Database connection.
- Multiplayer Service - Interface UÎ Generation:
  - Implementation and testing of actions: Use simple interfaces that allow the test of the various actions and features that need to be implemented (an example is shown later, as this is precisely what I did in the initial sprints);
  - Player connections to a session: Use simple interfaces that simulate a player joining a session.
- Multiplayer Service - Digital Twin:
  - Obtaining the spill: Add sample data related to the spill to the Database to be able to simulate it in the simulator (although this is more related to the interface component);

  – Updating the spill, based on actions taken by players: No strategy is used since these actions would only make more sense if the request were sent to Digital Twin. One strategy considered was the change of coordinates of the spill with sample data, but this would only take up more time, and it was an alternative that has nothing to do with interacting with Digital Twin as well.

These strategies take up more time in product development, but they are crucial for obtaining results (and testing a specific feature) when integration with other components is impossible.

Chapter 4

Requirements

This chapter presents the requirements considered during the project's development. First, the requirements related to the architecture. The second section concerns the actions that should be implemented in the game. Last, the proper non-functional requirements (quality attributes) are presented (although some architecture requirements could also be qualified). The functional requirements are not presented since these requirements (regarding this internship) are the actions possible during a game (presented in the section 4.2), the action log system, and the connections to a game (i.e., the authentication system are part of the Game Editor).

4.1 Architecture Requirements

MPCS should be well-performed since it will involve multiple players in each session. Another example is that the database should be scalable because of the possibility to store all the reports and logs regarding each game (actions of each player and consequences of each action). Due to the requirements list by Sampaio et al., some points must be taken into account (when designing the architecture and when defining the technologies and tools):

- REQ1: MPCS should be a multilingual environment (Portuguese, Spanish, and English), based on a Platform as a Service, for teaching, training, and performance evaluation, and in teams, of actions to combat maritime pollution;

- REQ2: The training (simulation) will take place in a multi-user environment in the Cloud, where each user will assume the role of a real professional belonging to the real professional (...);

- REQ3: The simulation will run on a given real geography, based on a Maps platform, on real weather conditions recorded from the relatively recent past (MOHID);

- REQ4: Users will be able, and should, interact with each other, interact with the equipment, and interact with the oil spill to achieve the objective of the exercise;
- REQ5: Players can participate in the exercise via smartphone, tablet or personal computer;
- REQ6: The MPCS Manager (Admin) should be capable of creating a complete exercise (with parametized data);
- REQ7: MPCS should use databases (SQL and/or NoSQL depending on the typology of data to be stored and performance decisions);
- REQ8: Exercises respective history of actions performed should be recorded;
- REQ9: Hydrodynamics and meteorology should be represented (with the help of MOHID);
- RE10: There is no provision for a maximum number of stored exercises, so the cloud infrastructure must provide the necessary space for this purpose;
- REQ11: Storage of unstructured data (reports, photos) may be more variable. Consider the possibility of configuring the location of unstructured files (e.g., independent cloud), which may differ from the database storage location;
- REQ12: Encryption of unstructured files containing accurate information on participants or operations should be foreseen. The same applies to personal data, etc. (…);
- REQ13: Unless otherwise specified, no personal data should be stored in this system, preferring consultation, using an API to external Systems, for example;
- REQ14: Selection of a scalable server to the needs of the simulation;
- REQ15: The development of the MPCS solution will take place on a specific server to be created in the data center of the Computer Engineering Department of the University of Coimbra;

4.2 Action Requirements

After carefully reading the document provided by Sampaio et al., a list of actions was elaborated and presented in Appendix C.1. The Appendix also presents the current state of the action (Done, Partially, or Not Done).

These actions were separated into different types to understand better the types of actions needed. The different types are:

- Basic Actions (Rest, Speak, Move). Actions that do not require equipment;

- Actions with basic equipment. Actions related to equipment that are not combat equipment (send/receive messages);
- Actions with Consumable. What a player can do with a Consumable;
- Actions with Combat Equipment. Actions interacting with combat equipment;
- Actions with Vehicles. Actions performed with a Vehicle;
- Actions with Facilities. Actions interacting with a Facility;
- Events. Events occurring during a game (health update or equipment degradation).

Regarding these different requirements (architecture and actions), a planned list of all the global requirements for this internship (2º semester) can be seen in the B.1. This list contains the requirements, the planned sprint for each requirement, its prioritization, and its state (implemented or not). All the "Development of Actions" in detail are in Appendix C.1.

The storage of the spill action was not taken into account when developing. This action was only to be considered if there was time at the end (what did not happen) since it needed other functions (skimmer functions, e.g.) and some effort to make (because of its particularities), so it was too ambitious to make in time for this thesis.

4.3 Non-functional Requirements

A non-functional requirement delineates a system's operational attributes and constraints. Unlike functional requirements that specify what the system should do, non-functional requirements pertain to how the system should perform, encompassing aspects such as performance, security, reliability, and user experience. They establish the parameters that ensure the system's effectiveness, efficiency, and quality.

So, the non-functional requirements of the project are [Roque et al., 2023]:

- NFREQ1: Security:
  - Encryption of unstructured files that may contain real information on participants or operations should be foreseen and as encryption of personal data stored in any platform;
  - No personal data should be stored in the system, using, if necessary, an API to external systems that keep the data updated and its history.
- NFREQ2: Performance:
  - Run multiplayer simulation exercises with 25 simultaneous users;

  - The game should give an answer to every action of the user within 5 seconds (it can be an awaiting message).

- NFREQ3: Usability:

  - Enable adequate time to learn (1 hour from first contact) and efficient use of role-specific interfaces for users familiar with browser-enabled devices (mobile and desktop).

- NFREQ4: Maintainability:

  - Maintenance to the MPCS server will depend on well-documented operations. Although the impact may be low, no guarantees can be given regarding maintenance involving changes such as updates to the operating system, database engine, and web server, software configurations (e.g., web server), and data removal operations without updated backups.

Chapter 5

Architecture Design

MPCS is a web-based application. It runs on a browser, and it uses a M(Model)-V(iew)-C(ontroller) approach. With that, it is possible to interact with all the components presented in the MPCS System, separating the View (UI interface) from the system's data (Model), with the use of Controllers (intermediary between Model and View).

5.1 Global Architecture Design

I was mainly responsible for developing the global architecture of the MPCS project [Roque et al., 2023], represented in figure 5.1. It should be noted that each component's specific classes and particularities were the responsibility of the person in charge of that respective component. The complete detailed architecture is shown in Appendix D.1.

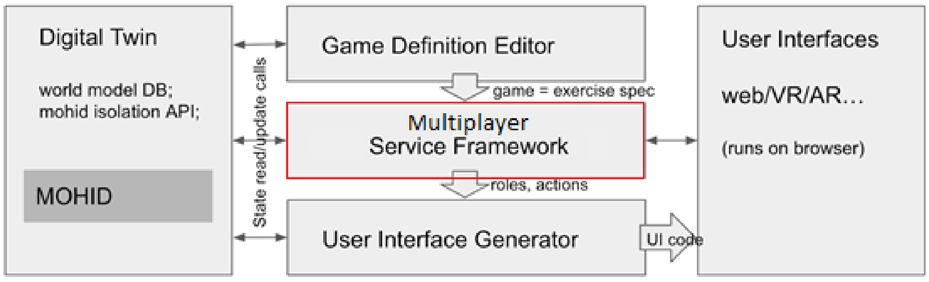


Figure 5.1: MPCS Architecture Components

A multi-tier client-server architecture is used to enable the separation of concerns between definitions of one core game simulation infrastructure accessible to enable access from multiple Internet and web-browser endpoints (desktop and mobile). Several interfaces must be served according to specific equipment, participant roles, and player state.

These components shown in the Figure are the components where specific development is required. For example, the Database component, common to all components (besides Digital Twin, which has its own Database), is not shown in this general design (it is a simplified view).

The connections and interactions between components in the simulator that occur outside a session/exercise and are managed using only controllers, services, and GET/POST requests (with the help of Spring Boot). During a game, the controllers will also be used with the addition of WebSockets. These technologies are detailed in 6.1).

In the next sections, each architecture component is explained, focusing on its part in the context of global architecture (D.1).

5.1.1 Game Editor

The Game Editor component (or Game Definition Editor) [Roque et al., 2023] is responsible for the configuration of all the exercises, including all the entities (i.e., assign participants/players to a determined user) and steps involved in the creation of a game as well as the creation of the interfaces before the start of a game (everything that occurs in the simulator before a player joins a game/exercise). These configuration interfaces are only available to the MPCS manager (described in [Sampaio et al., 2022]). Also, the Login and Authentication section is in the scope of the Editor, as well as the population of the Database Module with the required data. Figure 5.2 shows the Game Editor component in the context of the global architecture.

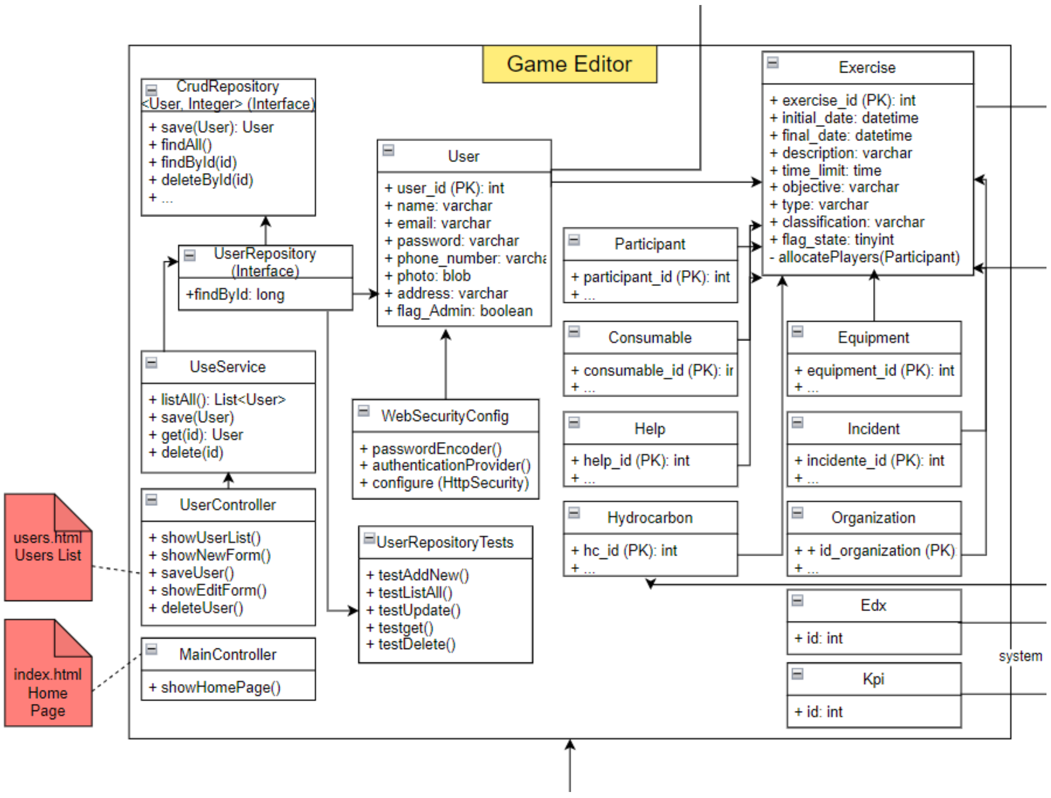


Figure 5.2: Game Editor component design

The Class Exercise is an important entity of a larger system as it has instances of several other classes (relations in Database), including User, Participant, Consumable, Help, Hydrocarbon, Equipment, Incident, and Organization. Each of these classes can represent agents participating in the game (Participant), resources (Equipment, Consumable, or Help), and elements in the game (Incident, Organization, Hydrocarbon). They all play a crucial role in allowing the class Exercise to work properly. These classes are interconnected. For example, a Participant needs to have a facility that he belongs to, and a Facility needs an Organization to which she belongs). An instance (in an exercise) of a Participant needs a User (who plays that role in the game). The class Exercise can only proceed when properly configured and integrated.

One important attribute of the exercise, is his state. If the exercise has the state *starting*, users can join an exercise (commonly called a game session). Otherwise, it is impossible (i.e., state *finished*).

This component communicates with the Multiplayer component, giving him information about the participants (adding it to the session attributes) and the remaining objects of an exercise.

5.1.2 UI Generation

The Interface/UI Generation component is in charge of the generation of all the interfaces that a player will see during a game. For instance, if a player is in a different location (vehicle, facility, or outside), interacting with different equipment, or performing some action (interacting with the Multiplayer component). In collaboration with the other components, it presents the possible actions according to the player's and equipment's states. In coordination with the Digital Twin, the interface should present the scenario of the incident (the spill). Figure 5.3 shows the key sub-components of the UI Generation component.

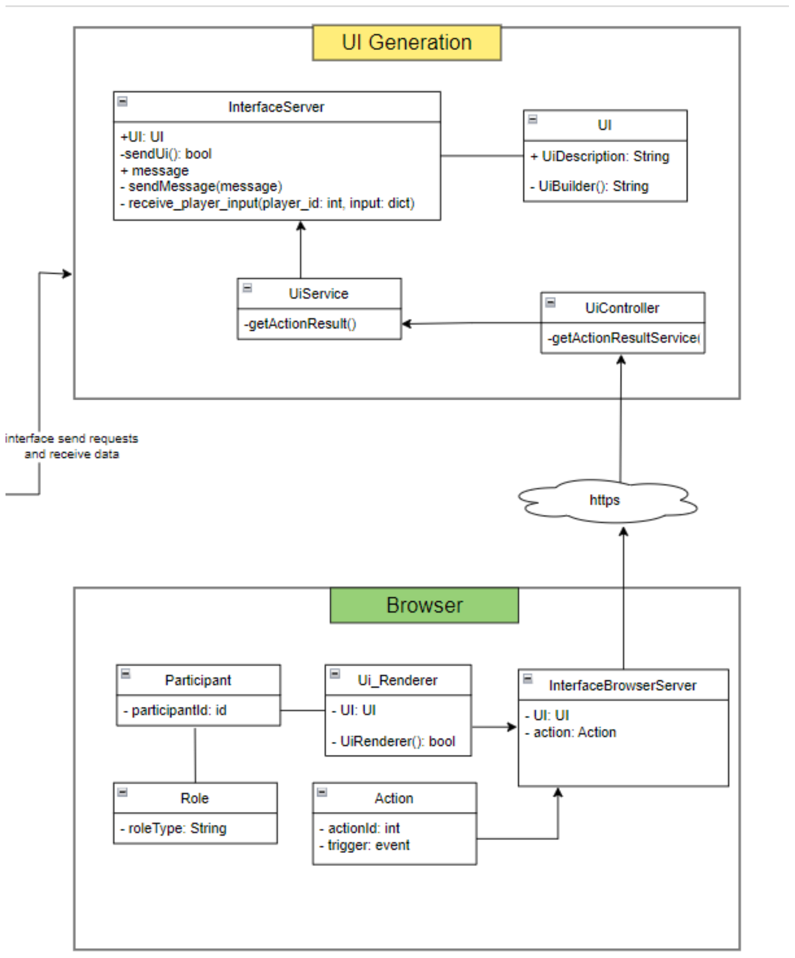


Figure 5.3: UI Generation component design

Also, it interacts with a WebSocket service during a game to send real-time messages to the server and receive the impact of those messages (with bidirectional communication). With that, the player can receive messages from other users without refreshing the page.

5.1.3 Digital Twin

The Digital Twin component is related to the simulation of the spill and its behaviour during the simulation and exercise time based on weather conditions and the type of spill. Also, actions related directly to MOHID (if a player wants to see how the spill evolves for the next three days or a drone view of the incident with the weather conditions). This component interacts with the others (especially Database, Multiplayer Service Framework, and UI Generation) to send or receive (if a player performs an action that affects the spill) information about the spill behaviour.

The Digital Twin interacts with an external server (API MOHID, which calculates the behaviour of a spill and simulates weather conditions) to send and retrieve information about the spill. This component serves as a bridge to other components since it processes and analyses the information by MOHID and sends it

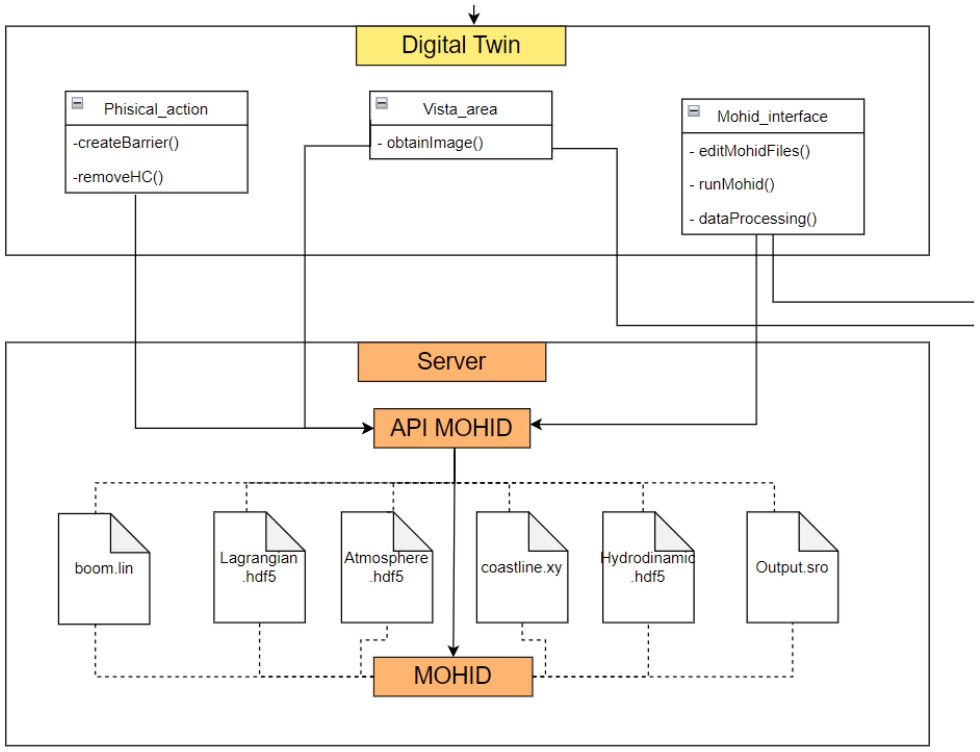


Figure 5.4: Digital Twin component design

(when necessary) to other components.

5.1.4 Database

The Database is where all the information of the simulator is stored. This component contains the configuration of the simulator, the state of the elements in a game, the storage of communication during a game (messages), and the log of all actions.

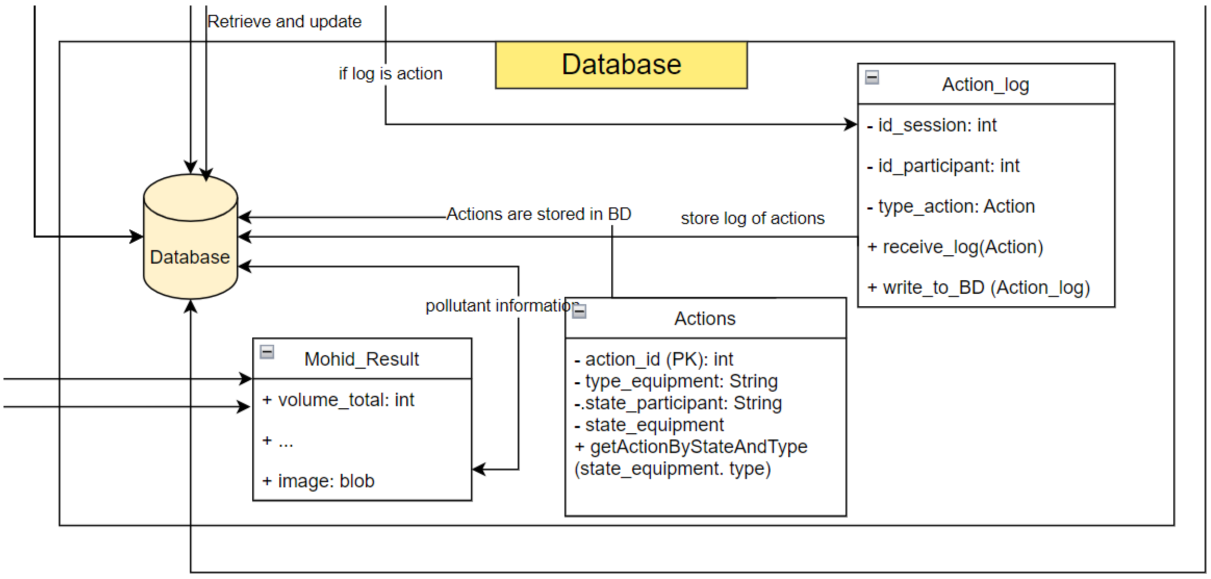


Figure 5.5: Database component design

5.2 Multiplayer Service Framework

The component related to this thesis. The objectives for this component of the architecture are mainly:

- To support the connection between players and the game world;
- To support the communication between players; To implement all player actions and their validation;
- To log all player activities in the exercise.

Figure 5.6 represents a detailed view of the Multiplayer Service Framework. This component handles the players' connections to a Game Session. This abstract class simulates a player's connection to an exercise and shows different functions the simulator can perform. For example, *connect()* occurs when the player joins a game/exercise, and *receive/update_game_state()* happens when some action impacts the game world (detailed further in the document).

No specific game session class (abstract class in the architecture) exists in the game. What happens is that a participant joins an exercise whose state is *starting*.

In each *Game Session*(Exercise), Participants will do actions (provided by the input in the Interface Module) that have consequences, thus impacting the game world and state (i.e., the spill or equipment and players). These actions have conditions that need to be validated. Also, for each session, a log report is issued. This report can have periodic logs provided by the system, or actions log (when a certain player does an action, this action and its possible impact are recorded).

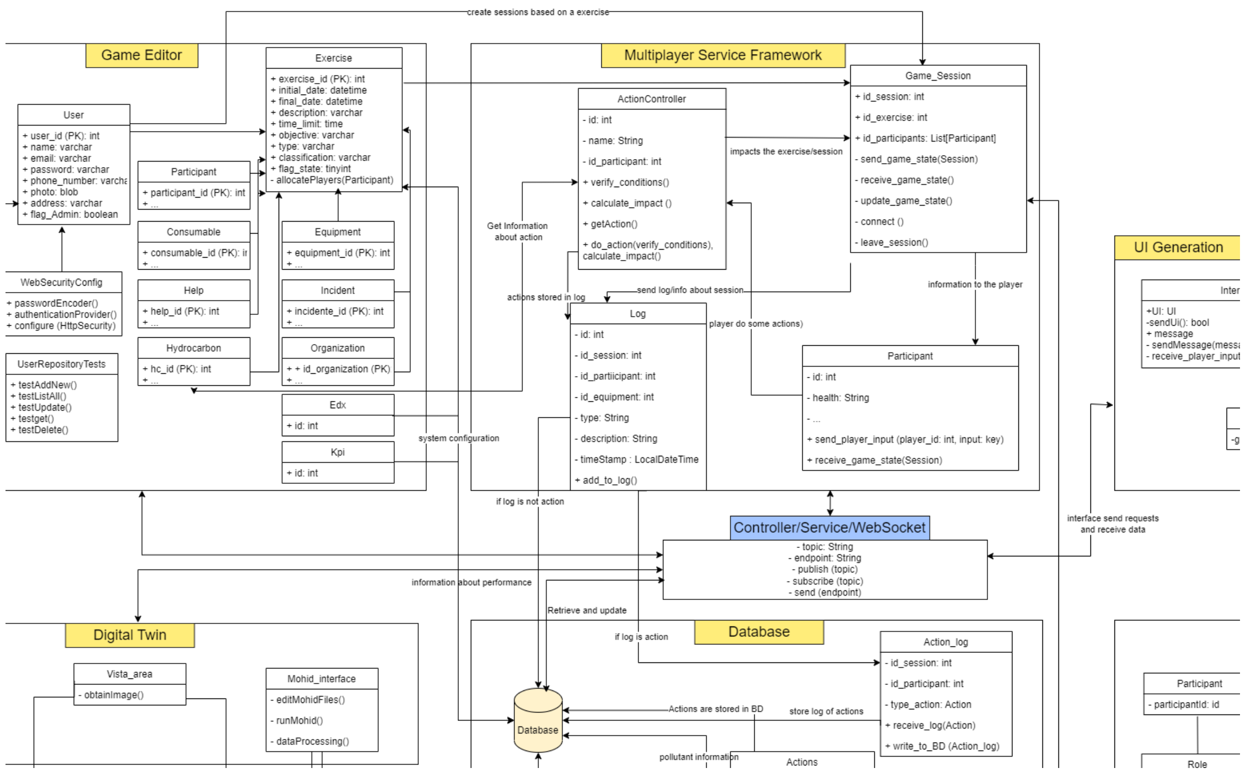


Figure 5.6: Multiplayer Service Framework component design

The *ActionController* class is an abstract class (to make the architecture design more user-friendly) that represents what an action does (the code of an action). The Database retrieves actions related to interactions with equipment when a player interacts with the equipment (checking the player's and equipment's state).

5.2.1 Client-Server interaction

This section shows how the connection with the WebSocket is made and how the client receives messages based on the subscription of topics.

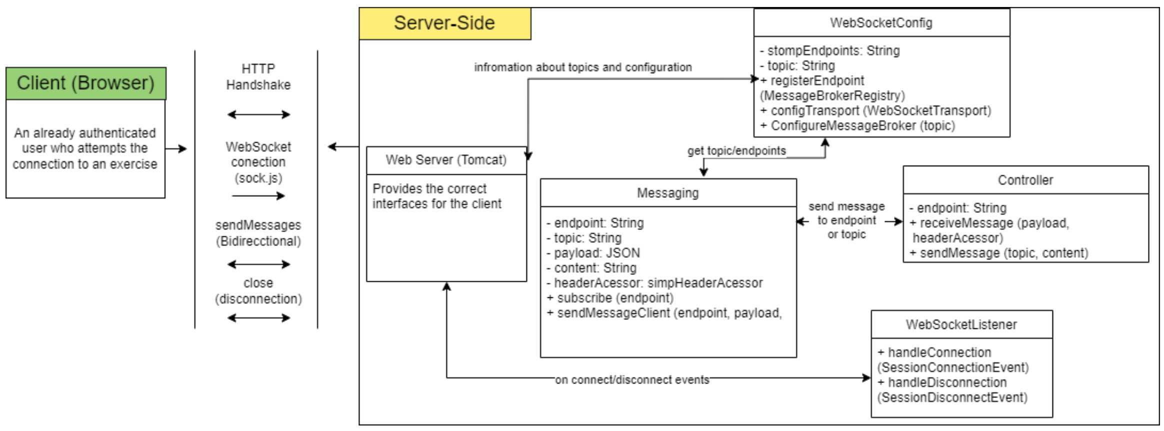


Figure 5.7: Client-Server interaction design

Figure 5.7 shows the possible outcomes of the WebSocket interaction. The configuration of the WebSocket is made on the server side. It is where the topics (subscribed by the players) are defined. Also, it is where transport capacity is configured, i.e., to send more data than the pre-defined one (sending PDFs).

Another function of The Server class is controlling what should be done when the player connects or disconnects from the socket.

For this connection to be possible, first, the player sends an HTTP request to the server to attempt a WebSocket connection. After the *handshake*, a TCP connection is established, and then, the client subscribes to the topics he needs.

For example, *"/topic/exercise/participant"* corresponds to the topic named "topic". These *urls* of the topic are created dynamically for each game (global actions) and player (actions only related to the participant).

Chapter 6

Infrastucture and Development

This chapter aims to show the different technologies used in the simulator and the project development process (and thesis). The process was separated into different sprints to show what was done in each phase, mentioning the objectives, tasks done, and problems found in each sprint.

6.1 Development Technologies

MPCS User Interfaces are accessible through a web browser, using web technologies such as HTML (structure and context), CSS (style and aesthetic appeal), and JavaScript (interactions). With a browser-based approach for developing the User Interfaces, the MPCS will be more accessible, allowing its exploration in different contexts and scenarios.

MPCS web application will run on a Tomcat server. It is a Java-based web server and servlet container that offers an adaptable approach for delivering Java-based web applications. Another reason for using Tomcat is because it is the Spring Boot framework's embedded server technology. As mentioned earlier, this server will take place on a specific server created in the data center of the Informatics Engineering Department of the University of Coimbra.

The MPCS backend is developed with the Spring Boot framework. It optimizes the building and deployment of web applications, providing a flexible (and automatic) way of building applications by using annotations (i.e., @Controllers and @Services). Spring Boot also provides a range of built-in features and libraries, including Spring Data for database access, Spring Security for authentication and authorization, and Spring MVC for building RESTful web services, which helps integrate the different components.

Spring Boot uses Controllers to map the different *urls* of the simulator and Services to perform certain actions (mainly related to accessing the Repositories). Entities are created using specific annotations and a JPA connection to a Database. Also, it has a simple interaction and integration with WebSockets technology. Also, it uses Hibernate for mapping Java objects to database tables and managing database interactions in Java applications. It provides a way to work with relational databases in an object-oriented manner.

MPCS is using MySQL technology. MySQL is a popular open-source relational database management system that provides a flexible way to store, manage, and retrieve data, supporting a wide range of data types, including text, numeric, date/time, and binary data to guarantee data integrity.

MPCS uses a WebSocket approach. WebSockets make bidirectional communication between the server and the client possible, so the client does not need to refresh the page to see changes in the interface. The client-side uses sock.js library, a javascript library that provides a way to implement Websockets.

The project uses a Maven architecture to manage the several dependencies in a project. It uses a single file (pom.xml), which contains all the dependencies. With Maven, it is also possible to build a JAR (Java Archive) file in a simplified way.

6.1.1 Player on-action event

When a player establishes a connection to the WebSocket, some actions can produce an event. The socket will send these events via TCP to the Server, processing and treating the received message accordingly. For example, a *connect* event represents a player's connection. In this *connect*, the server will request information from the Database. After getting this request, the server will send a message with all information needed to the topic, which will be received by that particular player (participant).

Events with equipment are similar with a particular difference. When the player does the interaction, a message is sent to the server, which then checks the player's and equipment's state (from the Database) and shows it in an equipment pop-up (interface generation) with the possible actions that the player can do. After that, if the player does an important action, it is sent to the rest of the game's players. With this approach, if an equipment needs two players to carry, when the player interacts with the equipment, it will know immediately when another player "joins" the equipment.

When an action is done and sent by the client to the server, it sends a message to an endpoint (an *url* in the backend, using Spring Boot). When the server sends the message, it is sent to a topic, also an *url*. After this, subscribed players to the topic will receive the processed message.

6.2 Development Activities

This section presents the project and thesis development process. As mentioned, a Scrum methodology was used with weekly meetings and two-week sprints. As a result, the following topics present the sprint's objective, what was done, and the problems found. This methodology was utilized in the project's development phase because it is crucial to have everyone in coordination. Before that, we (project members) were having weekly meetings as well but without sprints since we were doing more independent tasks where synchronization was not essential (only the final deadline where we chose the date to start the development (although the concepts of meetings and a backlog of a Scrum methodology were used).

The next sub-sections will show the development process separated by chronological marks and sprints. Also, I divided all the tasks into different topics (sub-components) to give a clear view of what has been developed in the respective sprint. The sub-components are:

- Simulator/Project Definition. Definition of several aspects such as data model or architecture;
- Technologies configuration. Also, it contains the configuration of the players' connections;
- Integration with other modules;
- Development of the game. Development of the actions and events possible in the simulator. It also contains the action log system;
- Migration to server. Migration of the simulator application to the production server;
- Tests. Tests regarding the simulator.

6.2.1 Initial Stage (Mid-February to Initial-May)

Regarding the 2º semester, the initial project (and thesis) development stage was mainly focused on Simulator/Project Definition. So, the tasks done concerning this sub-component were:

- Clarification and definition of the objectives of the project and thesis (meetings with professors and weekly iterations based on meetings);
- Detailed list of backlogs with prioritization and what to do in each sprint (Appendix B.1). The detailed actions to be implemented can be seen in Appendix C.1;
- Development of the global architecture and respective components. This process was continuous since sometimes the actual simulator development required it;
- Technologies to be used;
- Data Model. Architecture with all the entities to be created in the simulator. This was updated during development. Presented in D.2. It was done in collaboration between all internships related to this project (but mainly me and the Game Editor responsible);

- Deliverable to the consortium (partners). This deliverable contained a data model, the simulator's architecture, and the simulator's quality requirements. The simulator structure design;
- Creation of a list of actions that the game should have based on a checklist sent by the partners (guidelines on the process of a spill removal). How to translate the checklist into actions in the game (and the conditions implied for each).

Also, in this stage, the focus on the development started, and with this, the "Technologies configuration". The tasks were:

- Obtain knowledge about technologies and perform viability tests with chosen technologies;
- Start the development of the simulator itself. This part contains the use of the Controllers and Services features provided by Spring Boot;
- Configuration of the WebSockets channels (name of topics, processing disconnections/connections);

This stage was longer than expected for three main reasons. The first one was the deliverable to the consortium. We did not expect to present it at this early stage (although some changes were made during the actual development of the simulator. The other reason was that sometimes the responses from the partners were late.

Lastly, another reason was that the initial plan was to use RabbitMQ (a message queue technology) to communicate and interact with the different components (mainly because of the Digital Twin). However, due to time and complexity constraints, that was not implemented.

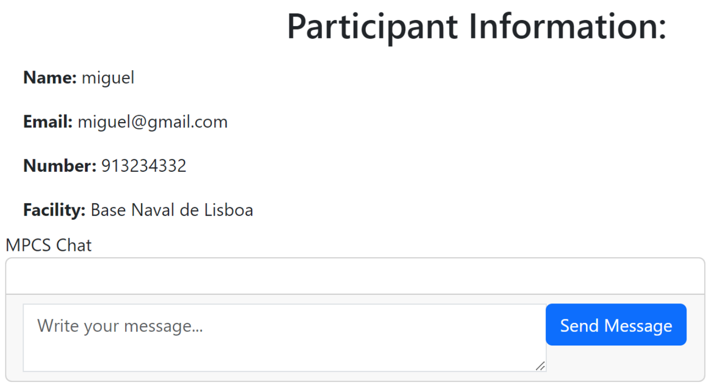
In the next phase, the main goals were to implement a proper WebSocket connection (Technologies configuration), development of the game (basic actions such as chat messages), and integration with the Editor component (Login system). Also, as we started the simulator development, we used sprints for more concrete deadlines and synchrony between members.

6.2.2 Sprint 1 (Initial of May to Mid-May)

In sprint 1, the main focus was the “Integration with other modules” between the Game Editor module. This integration was done earlier to enable an authenticated user to enter an exercise that he is linked to (as a participant/player). This means the connection to an Exercise (using a socket) by an authenticated user (participant). Another task of this sub-component was the creation of a synchronized Git Hub for all developers.

This sprint was the start of the "Development of the game" by implementing the first actions (basic actions, such as the chat system). This system allows the player to have the possibility to send messages to a desired receiver everyone or specific participants by email or phone number). This was not integrated with the Interface component, so the receiver input was the same for email or phone numbers.

All photos of the first interfaces were lost, so, unfortunately, it is impossible to show the first "interfaces" with the actions and player's information retrieved by the Database, and, consequently, all the figures are the current state of *MPCS*. Figure 6.1 shows the chat lobby closest to the first chat interface (only missing the input to the receiver and PDF file). Figure 6.2 shows the chat interface (email) after integration with the UI component.


Participant Information:

Name: miguel

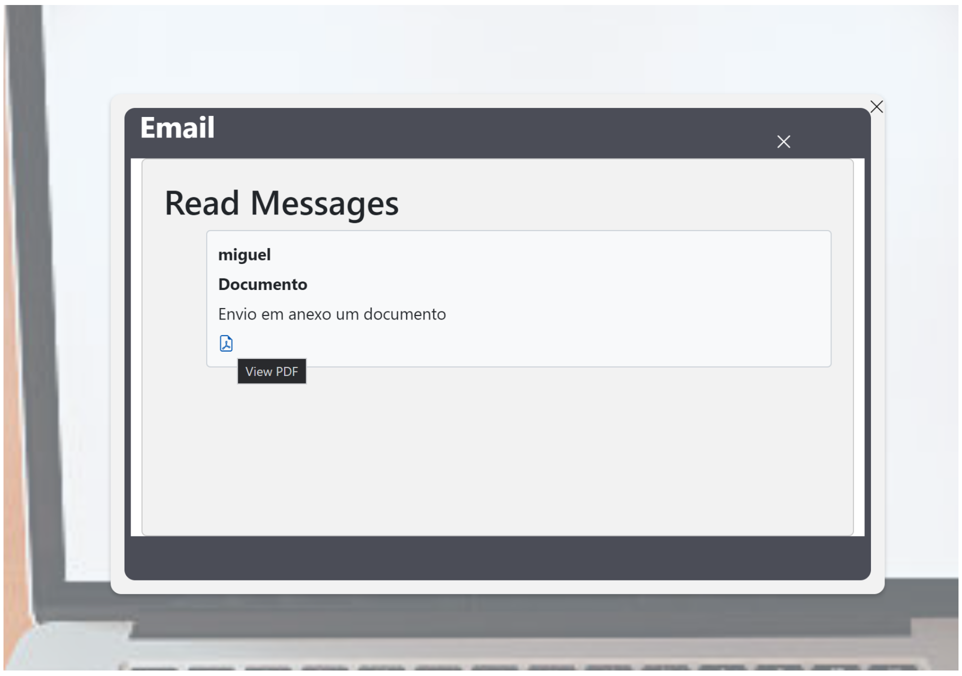
Email: miguel@gmail.com

Number: 913234332

Facility: Base Naval de Lisboa

MPCS Chat

Figure 6.1: Chat System (in lobby)


Email

Read Messages

miguel

Documento

Envio em anexo um documento

Figure 6.2: Chat System (Received messages in Email)

Regarding "Technologies configuration", it was in this sprint that a proper WebSocket connection (with a subscription to desired topics) from a player/participant was made.

Tests (*debug*) were done during all the sprints to validate if an implemented action or feature was properly working. This *debug* process consisted of unitary and integration tests (integration with other components). This was done periodically to avoid unwanted surprises.

Besides an error encoding the PDF and showing it afterward in the browser (Development of the game), no relevant issues were found, as all the planned objectives were accomplished and on time. For the next sprint, it was important to start developing actions with interaction between players and equipment. Also, integrating the UI component and Editor (for the equipment's configuration) was a major goal.

6.2.3 Sprint 2 (Mid-May to End of May)

This phase was marked by the "Integration with other modules". The first integration with the UI component was made. This integration consists of using the proper screens (interfaces) but also the definition of how to send the messages from the server to the client and vice-versa.

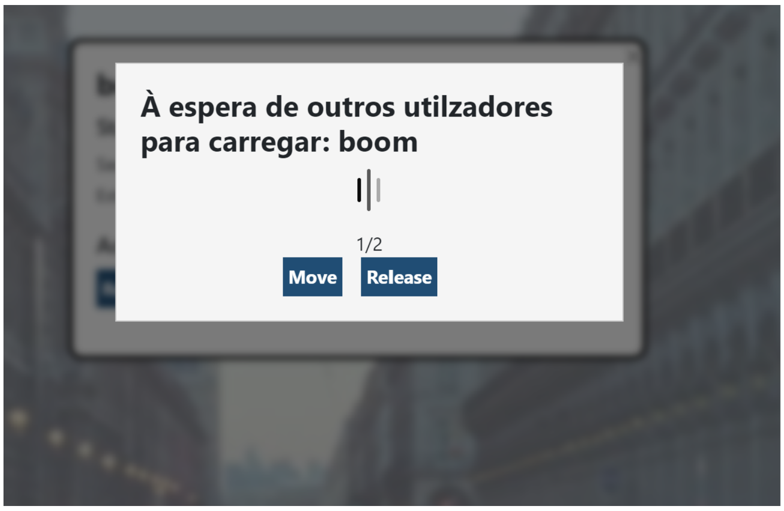
For this to happen, it needed to be decided the correspondent endpoints (client/interface sending the message to server) on the server (e.g., Send messages is the endpoint *SendMessage* and the location to send the message to the interface (e.g., Chat message has the type "CHAT" in message properties). During development, more type messages were added. By sending the type of message, the interface can perform the appropriate behaviour when receiving a message.

Regarding the "Development of the game", another goal of this sprint was to start implementing actions on equipment. These actions were the "Carry" (to carry equipment) and "Release" (to release equipment). In these actions, the other players (if interacting with the equipment) can see when a player performs one of these actions. First, the equipment was only a pre-defined class since the Editor had not finished the configuration for this entity (identified risk). However, at the end of this sprint, it was partially integrated.

Figure 6.3 shows the interface for a player carrying equipment that needs one more person to be moved (pre-condition).

In this sprint, due to integration tests, some problems were encountered:

- The first was WebSockets' process method to receive/send messages. The same messages (when involving objects) were read differently by the server and client. To fix this, these messages were sent using JSON in the server (client-side already sent JSON messages);
- It was in this sprint that we encountered some issues related to the player and equipment state (missing in the Equipment entity and an error in the Participant Entity), so the player could do everything, and his state was not taken into account (Integration with modules);
- The definition of state was still not used. The actions of carry and release are purely hard-coded, and when a player clicks on one button, it disappears, and the other shows up.


À espera de outros utilzadores para carregar: boom

1/2

Figure 6.3: Carrying equipment

Due to these errors, one goal that was delayed to the next sprint was a more complete integration with the Game Editor and Database (some attributes were only pre-defined).

With that, in the next sprint, the goals were to continue the "Development of the game" by implementing the state and the possibility to change the state of the objects, and consequently, integration with the Editor and Database, as well as the development of more actions.

6.2.4 Sprint 3 (End of May to Mid-June)

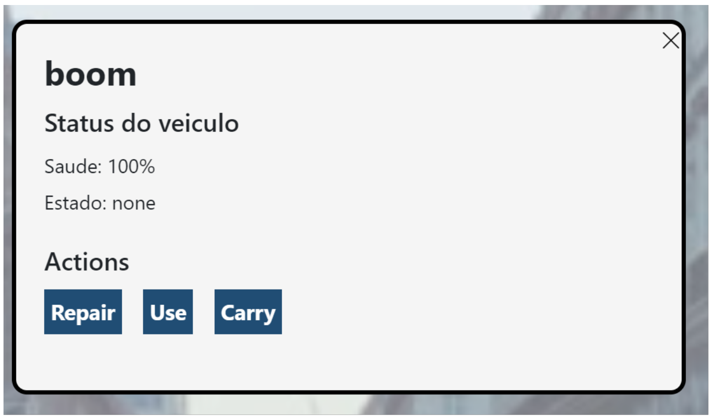
Here, we tried to integrate modules (from now on, it was a common objective in every sprint) on a weekly basis for the integration to be easier (simulator complexity was increasing). This approach also made finding the root of some errors/integration problems faster.

In this sprint, the tasks successfully done regarding the sub-components were:

- Simulator Definition: list of states a player or equipment can have in a game. Some actions can change the state of the objects;
- Integration with other modules:
  - Equipment entity fully retrieved by the Database;
  - Integration of the message system in the Interface (now, players can only see messages via email, in a computed, or phone equipment);
- Development of the game:
  - Adding equipment (basic equipment) and consumables to a player inventory was made possible. Also, it was possible to move inside a vehicle (in integration with the Interface, since it was responsible for the move action);

  – Development of a proximity radius in collaboration with the UI (because of the map coordinates and interface). With this, players can only perform actions to equipment (or other players) if the object is in a defined-meter radius (currently is 100 meters, but it is easily changeable);
  – Storage of actions in Database. The actions that were not with equipment were not stored. Also, this was only to store the generalization of the action. When the actions are shown in the interface, when clicked, they are sent to the proper controller (endpoint), where the endpoint's name is the same as the name of the action.

The last objective was not planned, but it provided a simple, dynamic way of retrieving the player's possible actions based on the state of the objects and the type of equipment. This approach was also utilized so the Interface component does not need to develop interfaces for all the actions (although some still require special interfaces). Figure 6.4 shows an example of possible actions.


boom

Status do veiculo

Saude: 100%

Estado: none

Actions

Figure 6.4: Possible actions interacting with equipment

In this sprint were found two main issues:

- The interface did not look user-friendly when placing all the equipment (too many objects). In the next sprint, this issue was corrected by only showing the combat equipment (not the basic ones such as phones or protection kits) and vehicles;
- Another issue was the storage of the actions. Since the basic equipment does not have a type, actions over this equipment were not retrieved by the Database. The same happens for the Consumables entity since it is not an equipment. This issue was only found later, and, due to time and simplicity constraints (only three actions), were manually retrieved by the client side.

6.2.5 Sprint 4 (Mid-June to End of June)

The goals planned for these two weeks were to continue the development of the game and perform integration with other modules.

With that, the tasks done during this sprint related to the "Development of the game" were:

- Now, players could carry equipment to a vehicle (if pre-conditions are met) or to the map (with UI integration);
- Actions on a vehicle. A time duration attribute was added to the actions presented in the Database to achieve this. Action such as "Fuel" a vehicle requires, e.g., twenty seconds (time compression). With this, the move action was fully integrated with the Multiplayer Service framework;
- Health update on participants. Now, players lose some health with the passage of time;
- Initial development of the actions related to combat equipment (Skimmer or a boom/barrier).

Regarding the "Integration" sub-component, full integration with the move action was made (UI Generation module). Also, the first tests of integration with Digital Twin were done. Due to this component's late development, only a simple test integration was made. For example, the spill shown in the interface was pre-defined in the exercise configuration and not updated accordingly to its behaviour.

No major issues were found during this sprint. The only drawback was the still weak integration with the Digital Twin. As this integration was not done, more time was available for other tasks, such as the health update and the start of implementation of some actions. The plan for the next phase was to develop the remaining actions, create the action log system, and try one more iteration of the integration with the Digital Twin. Another concern at this time was the migration of the simulator to the server.

6.2.6 Sprint 5 (End of June to Mid-July)

Although the objectives were mentioned above, one meeting with a partner to show the state of the game was scheduled for the last week of July. With that, the migration to the server happened in this phase. So, the steps done in this phase regarding "Development of the game":

- Implementation of the equipment combat actions. For example, it was possible to put a boom in a desired place. Although the actions related to a skimmer were developed, it was not implemented in time with the interface (due to the interface was not ready to receive the performed action);
- Action log system.

Integration with the Digital Twin module and the first try of migration to the server was also done in this sprint.

Although the integration with Digital Twin occurred (when putting a bomb, it sends a new state to the Digital Twin to re-run the simulation), the spill was still not updated in the interface. Initially, it was because the function that retrieved the spill was not in the same format as the coordinates sent to the Database by the Digital Twin. After implementing this, several major errors were identified in interaction with Digital Twin (MOHID). Sometimes, the MOHID does not calculate the spill's impact in time, so the database population is empty when asking for an update. This was corrected using the initial point of the incident as a default answer when the Database had no data for the request.

Another mistake was that in the Game Editor, the spill was sent to the exercise in a specific entity. Still, Digital Twin used the Incident entity to send the updated coordinates of the spill.

When migrating the project to the server, dealing with some errors consumed some of this sprint and the next. These errors were related to the project's build, where the Java Archive (JAR) does not recognize the dependencies and entities (because of Java Persistence API (JPA)). This happened because the JAR file was initially built using the Spring Boot embedded build process. However, it should be built using a Maven build configuration (mainly because of the dependencies), using the command: *install package spring-boot:run*.

Another error was related to the version of some dependencies being incompatible with other dependencies (which consumed some time to figure out). Also, small errors, such as case-sensitive HTML pages, were not working (that were working in *localhost*).

For the next two weeks, the main focus was to complete the project migration and add the notion of time in the game (with time compression). No more was planned because some abrupt change could compromise the simulator running on the server.

6.2.7 Sprint 6 (Mid-July to End of July)

The finish of this sprint was marked by the meeting with the partner from the Marine who was responsible for checking on the project. To make this meeting possible, the tasks were:

- Fix the aforementioned errors when migrating to the server;
- Notion of time added to the game. With a *Scheduler* annotation of Spring Boot in a function, occasionally, the server sends the current time to the player. This time is 10x faster than the real-time (Development of the game);
- Application added and tested in the server;
- Tests. Elaboration of a guideline to perform to the partner. Demonstration to the partner.

After these tasks, the meeting occurred, and, in general, the partner was satisfied with what he saw (some bugs were encountered in some actions). This demonstration with the partner was controlled since the developers performed all the actions to show to the partner. Also, notes were taken based on the feedback received in the meeting.

Unfortunately, one feature was impossible to add (it was already considered to be implemented). It consisted of a visibility radius, where the player could only see equipment and people in a certain radius. Although the proximity radius worked, the visibility was more complex to implement. It was causing some errors during the game (so it was commented in the code for the simulator to work properly).

So, the next sprint (and the last one) was related to correcting the errors found, implementing degradation on equipment, migrating a newer version to the server, and doing a round of tests with users without prior experience in the simulator.

6.2.8 Sprint 7 (End of July to Mid-August)

This sprint was the last one before closing the dissertation work since it was the end of the development of the product in integration with other members. The final steps were:

- Development of the game. Correction of errors found on the test previously done and implementation of degradation in equipment;
- Integration with Digital Twin;
- Migration of a newer version of the simulator to the server;
- Tests:
  - Creation of guidelines and a questionnaire for the users to perform in test session;
  - Test session (shown in chapter Testing and Evaluation);
  - Performance Tests (shown in chapter Testing and Evaluation);
  - Notes on issues caused on test session.

Two main issues were found in these steps. First, when implementing the degradation of equipment, a bug related to the persistence of objects was found. The degradation event uses a loop that checks all the equipment in all the exercises running. Still, when really degrading that equipment, it can be using an old version of that equipment (if a user performs an action on the equipment changing his state, i.e.). So, although some actions related to equipment improving its health were done (repair), this degradation update is not currently being used (there was no time to fix this error).

The same problem occurs when trying to update the spill. When requesting to the Digital Twin, it sends the exercise, but at that time, some changes could occur in the exercise (due to this component being late and consequently the integration being made in a later stage, this was not fixed).

Another reason for these errors (when updating) is related to the Database configuration (Game Editor component) and how the Database retrieves data (related to Hibernate, *force collection loading* error). This error is not fully related to this internship, so I did not look much into this. Despite that, some errors related to Database configuration consumed some of my time (fixed for the sake of the global project). However, I had no time to fix this specific error.

The notes and process used in the test session are detailed in the next section 7.

The remaining time was focused only on the writing of this thesis.

6.3 MPCS current state and Future Work

Currently, MPCS is allocated in a Department of Informatics Engineering production server and can be run and played there. For this, the build of the JAR file inside the server needs to be done. To access the simulator, the user needs to be connected to the DEI VPN, and search for the *193.137.203.28:8080* address.

A screenshot of a running exercise can be seen in Figure 6.5. It shows the map of the game when a boom equipment is already placed near the spill.

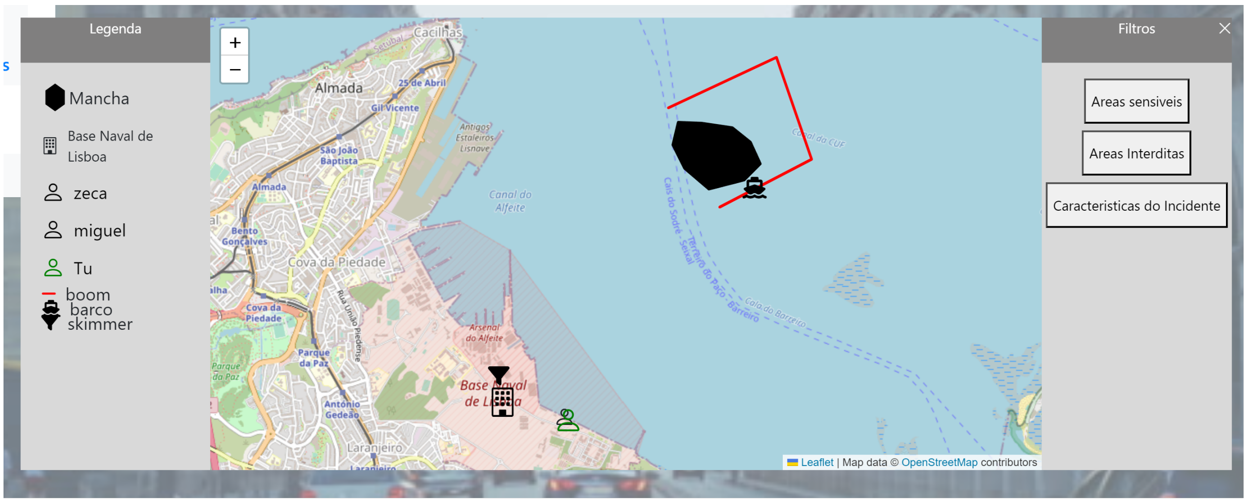


Figure 6.5: Map view of the simulator

All the actions and features implemented during a game are in C.1.

With that, some features that could be done in future work and errors to be fixed (related to this internship) are:

- Better approach on the degradation of the equipment;
- Better integration with the Digital Twin;
- Full implementation of action with Skimmer equipment (from the interface side);

- Implementation of the storage actions (equipment that stores the spill);
- Implementation of consumption in combat equipment. This was not done due to lack of time (involved complexity when knowing the time that the equipment was powered on) and a misconfiguration in the Database (not specifying the consumption rate of the equipment);
- Fix performance errors. Sometimes, many requests to DB are made simultaneously, provoking, sometimes, delay in the game. This delay causes the user not to know immediately if the action was executed. A cache/memory technique could be implemented in the future. This is also related to the Game Editor and the way the component saves and performs relations between entities in the Database.

Chapter 7

Evaluation

This chapter is related to the tests performed and the evaluation and analysis of results. The first three sections (7.1, 7.2, and 7.3) are related to a test session with participants testing the proper gaming part of the simulator. Section 7.4 is related to stress tests performed on the global simulator.

7.1 Objectives

The objectives of these tests were mainly to assure the simulator's performance (interface responsiveness in a short time), to evaluate if the actions are well simulated (close to real-life), and if the users can see an impact in the game world because of their actions. To do this, I wanted to evaluate if the Multiplayer Service has the required performance in the context of the following evaluations:

- Evaluate if the participants can understand where they are and what they can do. Since the multiplayer components send the information about the exercise (game session) to the UI Generation component;
- Evaluate the consistency and interpretability of the data presented in the interface;
- Assess whether the response times of actions are acceptable;
- Evaluate the quality of information and the timeliness of feedback;
- Ensure that participants understand the exercise scenario;
- Assess whether participants can keep track of the current state of the ongoing exercise;
- Ensure that participants comprehend how to act in the simulation;
- Evaluate whether participants can interpret the impact of their actions;
- Assess whether participants can initiate and receive necessary coordination and communication actions.

7.2 Process and Data Collection

We conducted a simulated training exercise based on a predefined script involving four participants scheduled for the tests. Each participant was assigned a distinct role and a set of objectives to achieve. This preparatory exercise laid the groundwork for a usability and performance inspection, which enlisted three experts (the advisors of the project, although they did not know how to play) and one developer from a different component who was not entirely familiar with the potential scenarios during an exercise or game. During the session, I followed all members and answered the questions that could come up.

The rehearsal spanned 45 minutes, and the exercise utilized contained the following elements:

- The participants;
- Several pieces of equipment, including vehicles, a boom, and a skimmer;
- A designated facility where participants were allocated;
- An incident scenario involving an oil spill;
- Provision of consumables, such as water and food.

The procedure started with the participant registration and a briefing, ultimately culminating in the response and actions on the oil spill. Pertinent observations were recorded throughout the exercise. After the exercise, a concise debriefing interview was conducted, followed by the administration of a post-experience questionnaire.

The responses to the questionnaire were analyzed and grouped by themes (UI interface, Game Editor, and Multiplayer Service) to unearth issues necessitating resolution and identify improvement opportunities.

The questions regarding the Multiplayer Service framework were:

- "Was I able to connect and join the desired game?";
- "Did the server consistently provide prompt responses (within approximately 2/3 seconds)?";
- "When I disconnected, was it easy to reconnect?";
- "Were the action times acceptable considering the time compression (10x)?";
- "Did I understand most of the action constraints (e.g., distance, movement towards water, participant's life already at maximum)?";
- "Could I move easily (by vehicle or on foot)?";

- "Whenever I needed to, could I interact with other users through messages (email or mobile), knowing exactly where and how to perform these actions?";
- "Could I interact with other users when loading equipment that required more than one player?";
- "Do I believe this game has the potential to be an effective training tool for pollution control coordination?".

7.3 Results

Based on the recorded observations during the test session, the comments from the testers, and the responses to the survey, the following relevant positive points are presented as the outcomes of this exercise:

- The game has a good range of actions, and the actions simulate well what happens in real-life (with their specific constraints), with an acceptable action time;
- It is easy to interact with nearby elements (equipment or other players);
- The simulator effectively manages connections and disconnections;
- The simulator has the potential to be an efficient training tool;
- The interaction with other users was easy.

In terms of addressing the issues identified while running the simulated training exercise, they were categorized to facilitate a better understanding of which category experienced failures and to identify how to correct these errors more easily.

Regarding issues at the level of the interface (issues that can be related to Multiplayer Service), were:

- At times, players/participants were uncertain about their current state and available actions;
- The interface occasionally experienced delays in responding to participants, causing uncertainty regarding the completion of actions (no interface feedback when waiting for a server answer;
- Movement actions, particularly when using vehicles, were not well synchronized with the server. For example, the interface displayed the movement as completed, but in the server, it had not finished executing;
- The spill was not updating during time.

Issues relevant for the learning support towards operational coordination:

- No feedback or help to perform an action;
- Certain actions lacked clarity on where to perform them. The most obvious examples were to get a vehicle, the participants should go to the facility's warehouse, and to send an email, the participant should go to the facility's office;
- No feedback on what the participants should do in the exercise (if the coordinator fails to give the participants instructions). A participant without prior training in spill removal might not clearly understand their role or what they should do.

7.3.1 Implications of the Results and Opportunities for Improvement

The analysis of the results from the test session indicates that certain corrective measures need to be implemented. Regarding the multiplayer component, the most significant implications (and opportunities to improve) include:

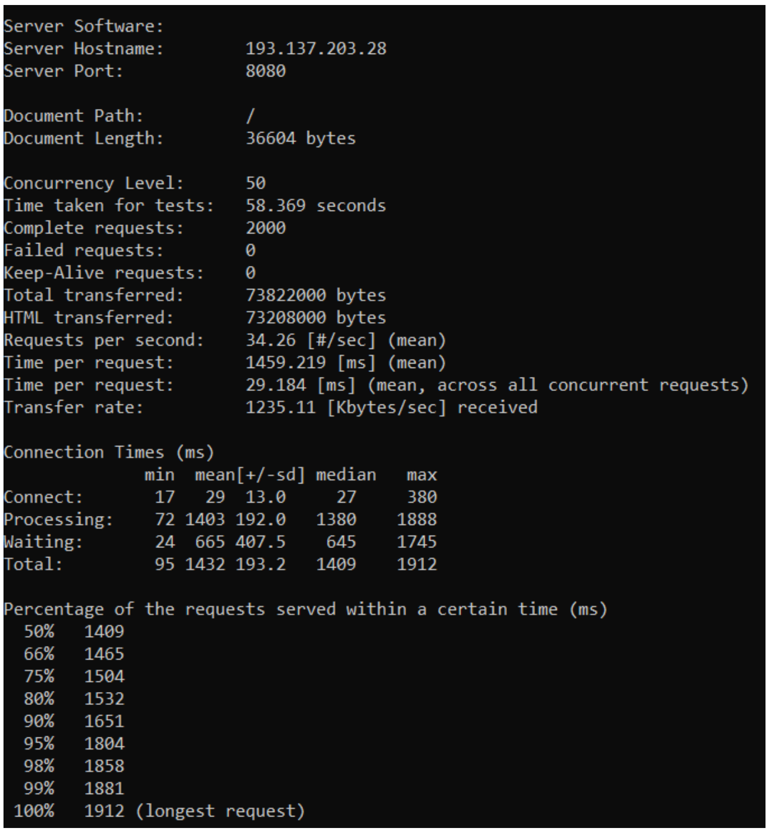
- Enhancing server response times for movement actions;
- Improving the delivery of action impact (map updates, which at times experience delays);
- Optimizing the mechanism for executing actions, as some actions impose significant demands on the database, resulting in exponential increases in requests with more participants;
- Addressing issues related to the Digital Twin dependency, specifically fixing update failures for the spill;
- Addressing dependencies with UI Generation, implementing a feedback mechanism regarding how and where to perform certain actions. Additionally, rectifying the lack of information related to player status and the visual representation of the surrounding environment;
- Ensuring greater consistency/data integrity in the game state and world. For instance, when two participants attempt to add an item to their inventory simultaneously, the player who did not acquire the equipment should receive appropriate feedback.

7.4 Stress Tests

This chapter is related to the stress tests made to the simulator, and not only the gaming part. These tests were aimed at the main page of the MPCS.

The tests were done using *Apache Bench*, a single-threaded command line computer program for testing HTTP web servers.

Figure 7.1 shows the results of the stressing test. The type of test shown in the figure was repeated 10 times to ensure that the result was not an outlier. It was verified that the tests (during different times of the day) were always close to each other.



```
Server Software:
Server Hostname:        193.137.203.28
Server Port:            8080

Document Path:          /
Document Length:        36604 bytes

Concurrency Level:      50
Time taken for tests:   58.369 seconds
Complete requests:      2000
Failed requests:        0
Keep-Alive requests:    0
Total transferred:      73822000 bytes
HTML transferred:       73208000 bytes
Requests per second:    34.26 [#/sec] (mean)
Time per request:       1459.219 [ms] (mean)
Time per request:       29.184 [ms] (mean, across all concurrent requests)
Transfer rate:          1235.11 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       17   29  13.0     27     380
Processing:    72 1403 192.0   1380    1888
Waiting:       24  665 407.5    645    1745
Total:         95 1432 193.2   1409    1912

Percentage of the requests served within a certain time (ms)
  50%   1409
  66%   1465
  75%   1504
  80%   1532
  90%   1651
  95%   1804
  98%   1858
  99%   1881
 100%   1912 (longest request)
```

Figure 7.1: Stress test using *ab* tool

The tests consisted of 2000 requests to the server, using 50 concurrency requests (multiple requests at a time) and with a keep-alive option (the connection of the request persists during the test). The command for the test was:

ab.exe -n 2000 -c 50 -k 193.137.203.28:8080

MPCS System showed acceptable results, taking 58.369 seconds to complete the test. So, an average response of 58.369 milliseconds per request and 29.184 requests per second. The request's processing took more time (1403 average seconds) than the others (connect or waiting).

So, in conclusion, the simulator can support at least 50 users simultaneously making requests (a requirement was to support 25 users) when testing for 2000 requests. As mentioned, this was tested for the global simulator and not for the gaming part.

Chapter 8

Conclusion

The thesis's overall objective consists of developing and implementing a multiplayer system architecture in the MPCS project. To achieve this, in this first part of the project, I studied (presented on Background and State of the art chapters) the different topics needed, such as the requirements and constraints of the MPCS project, the different tools, and technologies needed to develop a multiplayer system, games that already exist on the market and their architecture, and so on. The second one is the development of the

Overall, all objectives of the thesis were achieved. The development of reference documentation (the delivery to the consortium) with the architecture design and its services, the implementation of the multiplayer system and its communication channels, and an evaluation of the game's performance were done.

Despite the objectives being accomplished, some features could have a better implementation or approach. Related to integration with other modules, the integration with the Digital Twin is not 100% functional due to errors already mentioned.

Regarding the implementation of the multiplayer system and its actions, Improving the game (and player) state/world retrieval process for players is a task worth addressing in the future, especially considering that certain actions involve making database requests. This process can become resource-intensive for the game, particularly when dealing with a high volume of requests, impacting data integrity and system performance. For example, the degradation of the equipment is not implemented in this final version of the MPCS (although it is developed) for this reason and a *bug* in the Database configuration (mentioned earlier).

In the end, I am pleased with the final product of this thesis, as I feel that not only is MPCS a functional game where I was able to achieve the overall objectives, but I also gained valuable knowledge. This knowledge extends to how a multiplayer system should operate and its particularities. Additionally, I now feel more capable of being a part of a software team (as it was a collaborative project) and of implementing and utilizing more efficient software processes.

8.1 Deliverable list

During this internship, some deliverables were produced, such as:

- A JAR file of the simulator (and all the source code);
- A version of the JAR in the production server;
- A separate document that reports the system architecture structure;
- A guideline to perform tests, evaluation of the performance tests, and a questionnaire;
- The final report.

References

Tiago Agostinho. Augmented reality game and gamification, 2013.

Amazon. Dedicated game server hosting - amazon gamelift - aws. URL `https://aws.amazon.com/gamelift/`.

Amazon2018. Aws re:invent 2018: Epic games usa a aws para disponibilizar o fortnite a 200 milhões de jogadores. URL `https://aws.amazon.com/pt/solutions/case-studies/EPICGames/`.

AmazonAurora. Fully mysql and postgresql compatible managed database service | amazon aurora | aws. URL `https://aws.amazon.com/rds/aurora/`.

Grenville. Armitage, Mark. Claypool, and Philip. Branch. *Networking and online games : understanding and engineering multiplayer Internet games*. John Wiley Sons, 2006. ISBN 0470018577.

Larah Armstrong. Latency and packet loss | unity multiplayer networking, a. URL `https://docs-multiplayer.unity3d.com/netcode/1.0.0/learn/lagandpacketloss/index.html`.

Larah Armstrong. Tricks and patterns to deal with latency | unity multiplayer networking, b. URL `https://docs-multiplayer.unity3d.com/netcode/current/learn/dealing-with-latency/index.html`.

Per Backlund, Henrik Engström, Cecilia Hammar, Mikael Johannesson, and Mikael Lebram. *Sidh - a Game Based Firefighter Training Simulation*. 8 2007. ISBN 0-7695-2900-3. doi: 10.1109/IV.2007.100.

João Barata and Licínio Roque. Mpcs simulator operational requirements specification, 9 2022.

Ignasi Barri, Concepció Roig, Francesc Giné, I Barri, · C Roig, · F Giné, C Roig, and F Giné. Distributing game instances in a hybrid client-server/p2p system to support mmorpg playability. *Multimed Tools Appl*, 75:2005–2029, 2016. doi: 10.1007/s11042-014-2389-0.

Michel Bauwens, Vasilis Kostakis, and Alex Pazaitis. Peer to peer. *University of Westminster Press*, 3 2019. doi: 10.16997/BOOK33. URL `https://doi.org/10.16997/book14`.

Sandro Ricardo da Costa Ferraz. Recomendações para a adoção de práticas ágeis no desenvolvimento de software: estudo de casos, 2016. URL `https://hdl.handle.net/1822/46409`.

DarkRift2. Darkrift2 - open source powerful networking solution - darkrift2. URL `https://www.darkriftnetworking.com/`.

Epic. Fortnite | free-to-play cross-platform game - fortnite. URL `https://www.epicgames.com/fortnite/en-US/home`.

Fish-Net. Introduction - fish-net: Networking evolved. URL `https://fish-networking.gitbook.io/docs/`.

Fusion. Setting the benchmark for multiplayer games. | photon engine. URL `https://www.photonengine.com/en-us/fusion`.

Frank Glinka, Alexander Ploss, Jens Müller-Iden, and Sergei Gorlatch. Rtf: A real-time framework for developing scalable multiplayer online games. pages 81–86, 01 2007. doi: 10.1145/1326257.1326272.

Google. Gaming | google cloud. URL `https://cloud.google.com/solutions/gaming`.

GoogleFirebase. Firebase for games | supercharge your games with firebase. URL `https://firebase.google.com/games`.

Jürgen Gotschlich, Torsten Gerlach, and Umut Durak. 2simulate: A distributed real-time simulation framework. 1 2014. doi: 10.13140/2.1.4976.2081.

HostHavoc. Host havoc - game server, voice and web hosting provider. URL `https://hosthavoc.com/`.

Jared Jardine and Daniel Zappala. A hybrid architecture for massively multiplayer online games. 2008.

Javatpoint. What is hybrid topology - javatpoint. URL `https://www.javatpoint.com/what-is-hybrid-topology`.

Jörg Kienzle, Clark Verbrugge, Bettina Kemme, Alexandre Denault, and Michael Hawker. Mammoth: A massively multiplayer game research framework. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, page 308–315, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584379. doi: 10.1145/1536513.1536566. URL `https://doi.org/10.1145/1536513.1536566`.

Autoridade Nacional Marinha. Exercise to combat sea pollution (atlantic polex.pt 2022), 5 2022.

Microsoft. Serviços de informática em nuvem | microsoft azure. URL `https://azure.microsoft.com/pt-pt/`.

Netcode. Unity multiplayer what's new | unity multiplayer networking. URL `https://docs-multiplayer.unity3d.com/releases/introduction`.

Oracle. Database | oracle. URL https://www.oracle.com/database/.

ChangHoon Park, Heedong Ko, and Taiyun Kim. Naver: Networked and augmented virtual environment architecture; design and implementation of vr framework for gyeongju vr theater. *Computers Graphics*, 27(2):223–230, 2003. ISSN 0097-8493. doi: https://doi.org/10.1016/S0097-8493(02)00279-0. URL https://www.sciencedirect.com/science/article/pii/S0097849302002790.

Photon. Multiplayer game development made easy | photon engine. URL https://www.photonengine.com/.

Iana Podkosova, Khrystyna Vasylevska, Christian Schoenauer, Emanuel Vonach, Peter Fikar, Elisabeth Bronederk, and Hannes Kaufmann. Immersivedeck: a large-scale wireless vr system for multiple users. In *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 1–7, 2016. doi: 10.1109/SEARIS.2016.7551581.

PUN. Photon unity 3d networking framework sdks and game backend | photon engine. URL https://www.photonengine.com/pun.

Licinio Roque, Luís Pereira, Jorge Cardoso, Miguel Lopes, Ana Sobral, Fernando Barros, João Barata, and José Sobrinho. Simulator structure design report, 2023.

Rui Sampaio and Licinio Roque. Checklist com resposta a incidente de poluição.

Rui Sampaio, Manuel Carrasqueira, and José Daniel. Marine pollution control simulator functional requirements deliverable 2.2 version 1.0, 9 2022.

Unity-Hosting. Cloud game server hosting service (aka multiplay) | unity. URL https://unity.com/products/game-server-hosting.

Unity2022. Network topologies | unity multiplayer networking. URL https://docs-multiplayer.unity3d.com/netcode/0.1.0/reference/glossary/network-topologies/index.html.

Unity2023. 8 factors of multiplayer game development | unity. URL https://unity.com/how-to/multiplayer-game-development-factors#cheat-mitigation.

Valve. Source multiplayer networking - valve developer community. URL https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.

F. M. Williams-Bell, B. Kapralos, A. Hogue, B. M. Murphy, and E. J. Weckman. Using serious games and virtual simulation for training in the fire service: A review. *Fire Technology*, 51:553–584, 5 2015. ISSN 00152684. doi: 10.1007/S10694-014-0398-1/FIGURES/10. URL https://link.springer.com/article/10.1007/s10694-014-0398-1.

Amir Yahyavi and Bettina Kemme. Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Comput. Surv.*, 46, 7 2013. ISSN 0360-0300. doi: 10.1145/2522968.2522977. URL https://doi.org/10.1145/2522968.2522977.

Appendices

Appendix A

Epic's architecture at a glance [Amazon2018]

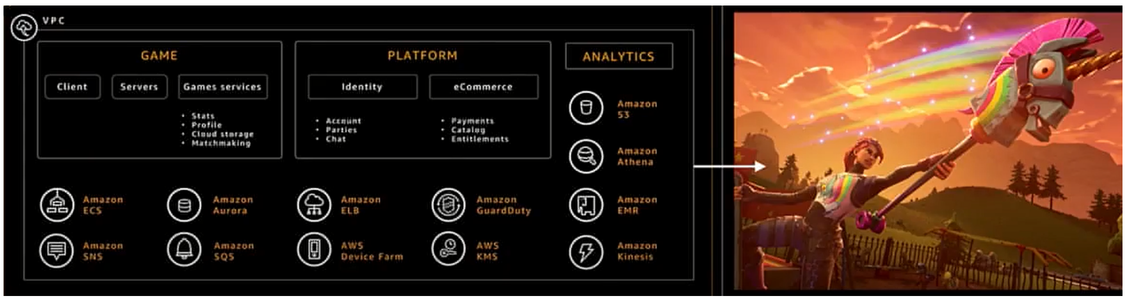


Figure A.1: Epic's architecture at a very high-level

Appendix B

Methodology

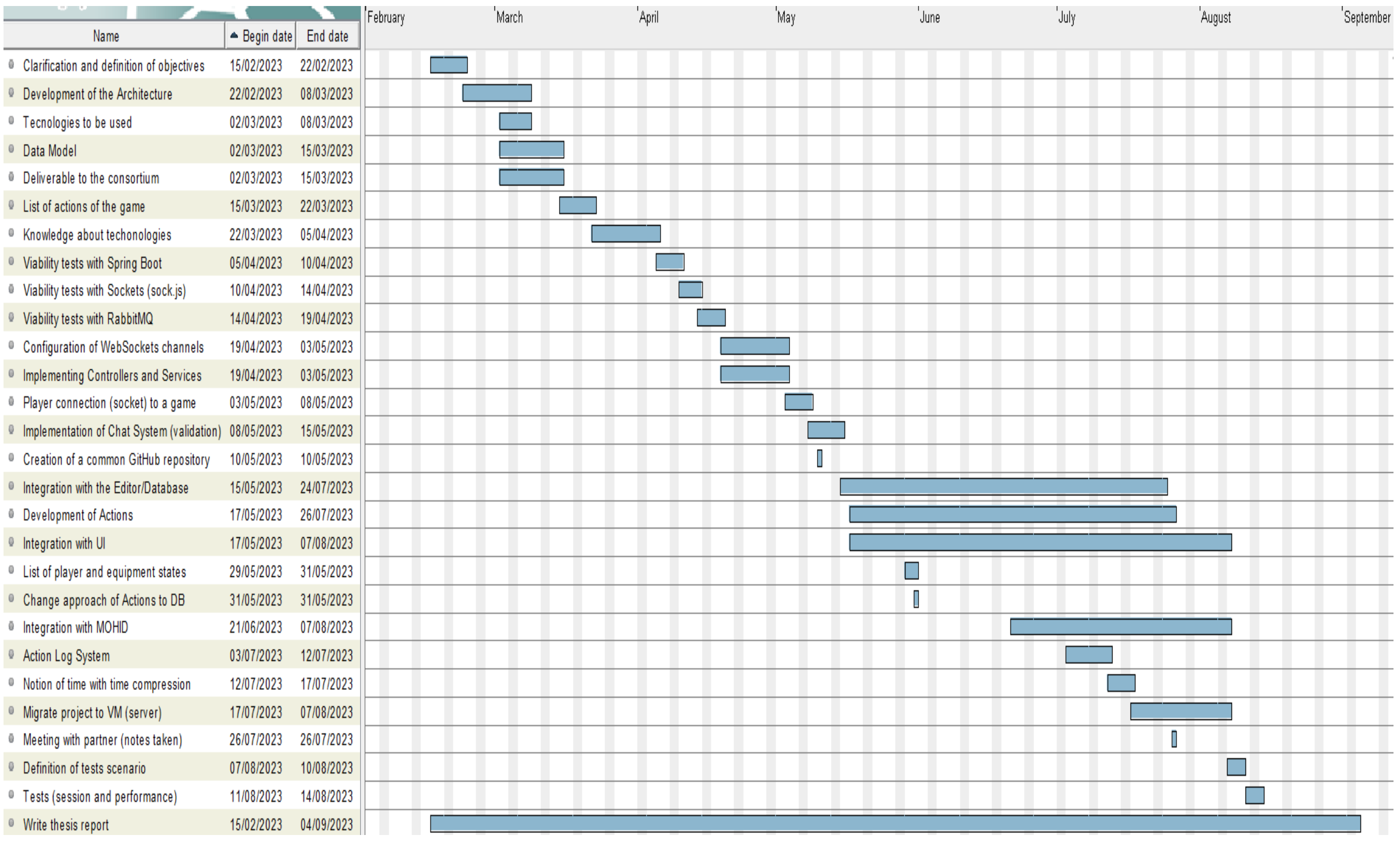
B.1 Panned Task List/Backlog

Planned List	**Planned time**	**Effetive time**	**Prioriti-zation**	**State**
Development of the Architecture	Before Sprints	Before Sprints	Must	Done
Data Model	Before Sprints	Before Sprints	Must	Done
Deliverable to the consortium	During semester	Before sprints	Must	Done
Use of Sprints Methodology	Initial of April	Initial of May	Must	Done
Use of backend technology (Spring)	1	1	Must	Done
Use of real-time messaging (Web-Sockets)	1	1	Must	Done
Use of a Message Queue (Rab-bitMQ)	1	-	Could	Not Done
Creation of a common GitHub repository	1	1	Must	Done
Migrate project to VM (server)	1	6	Must	Done
Player connection (socket) to a game	2	2	Must	Done
Implementation of Chat System	2	2	Must	Done
Integration with the Edi-tor/Database	1 to 6	1 to 6	Must	Done
Integration with UI	1 to 6	1 to 6	Must	Done
Development of Actions	1 to 5	1 to 7	Must	Par-tially
Integration with MOHID	2 to 6	2 to 7	Must	Par-tially

Notion of time with time compression	5	6	Should	Done
Action Log System	5	5	Must	Done
Test Sessions (with questionnaire and evaluation)	7	7	Must	Done
Performance Tests	7	7	Must	Done
Write of thesis report	During semester	During semester	Must	Done

Table B.1: Thesis backlog (plan)

B.2 Effective Gantt



Name	Begin date	End date
Clarification and definition of objectives	15/02/2023	22/02/2023
Development of the Architecture	22/02/2023	08/03/2023
Tecnologies to be used	02/03/2023	08/03/2023
Data Model	02/03/2023	15/03/2023
Deliverable to the consortium	02/03/2023	15/03/2023
List of actions of the game	15/03/2023	22/03/2023
Knowledge about techonologies	22/03/2023	05/04/2023
Viability tests with Spring Boot	05/04/2023	10/04/2023
Viability tests with Sockets (sock.js)	10/04/2023	14/04/2023
Viability tests with RabbitMQ	14/04/2023	19/04/2023
Configuration of WebSockets channels	19/04/2023	03/05/2023
Implementing Controllers and Services	19/04/2023	03/05/2023
Player connection (socket) to a game	03/05/2023	08/05/2023
Implementation of Chat System (validation)	08/05/2023	15/05/2023
Creation of a common GitHub repository	10/05/2023	10/05/2023
Integration with the Editor/Database	15/05/2023	24/07/2023
Development of Actions	17/05/2023	26/07/2023
Integration with UI	17/05/2023	07/08/2023
List of player and equipment states	29/05/2023	31/05/2023
Change approach of Actions to DB	31/05/2023	31/05/2023
Integration with MOHID	21/06/2023	07/08/2023
Action Log System	03/07/2023	12/07/2023
Notion of time with time compression	12/07/2023	17/07/2023
Migrate project to VM (server)	17/07/2023	07/08/2023
Meeting with partner (notes taken)	26/07/2023	26/07/2023
Definition of tests scenario	07/08/2023	10/08/2023
Tests (session and performance)	11/08/2023	14/08/2023
Write thesis report	15/02/2023	04/09/2023

Figure B.1: Effective Gantt of the 2º semester

Appendix C

Requirements

C.1 Action Requirements

Actions	**Developed in Sprint**	**State**
Basic Actions		
Send Message (Speak)	1, 3	Done
Move	2, 3	Done
Rest	4	Done
Actions with Basic Equipment		
Send Message (phone)	1, 3	Done
Add to Inventory	3	Done
Actions with Consumable		
Consume (Eat/Drink)	3	Done
Add to Inventory	3	Done
Actions with Combat Equipment		
Carry/Pick Up	2, 3	Done
Release/Drop	2, 3	Done
Move	4	Done
Unload from Vehicle	4	Done
Use Boom	5, 6	Done
Place Boom	5, 6	Done
Remove Boom	5, 6	Done
Use Skimmer	5, 6	Done
Place Skimmer	6	Partially
Remove Skimmer	5, 6	Partially
Power on/off Skimmer	4, 6	Partially
Take out from Facility	6	Done
Consumption	–	Not Done
Storage of Spill	–	Not Done
Actions with Vehicles		
Enter/Leave	3	Done
Move	3, 4	Done

Repair	4	Done
Fuel/Charge	4	Done
Consumption	4	Done
Take out from Facility	6	Done
Actions with Facilities		
Enter/Leave Office	5	Done
Send Message in Office (Email)	1, 5	Done
Enter/Leave Storage	5	Done
Enter/Leave Facility	5	Done
Events		
Get/Show objects in proximity	2, 3	Done
Heath Update	4	Done
Time Compression	6	Done
Equipment degradation	7	Partially
Spill Update	7	Partially

Table C.1: Actions required (defined) for MPCS

Appendix D

Architecture

D.1 Global Architecture

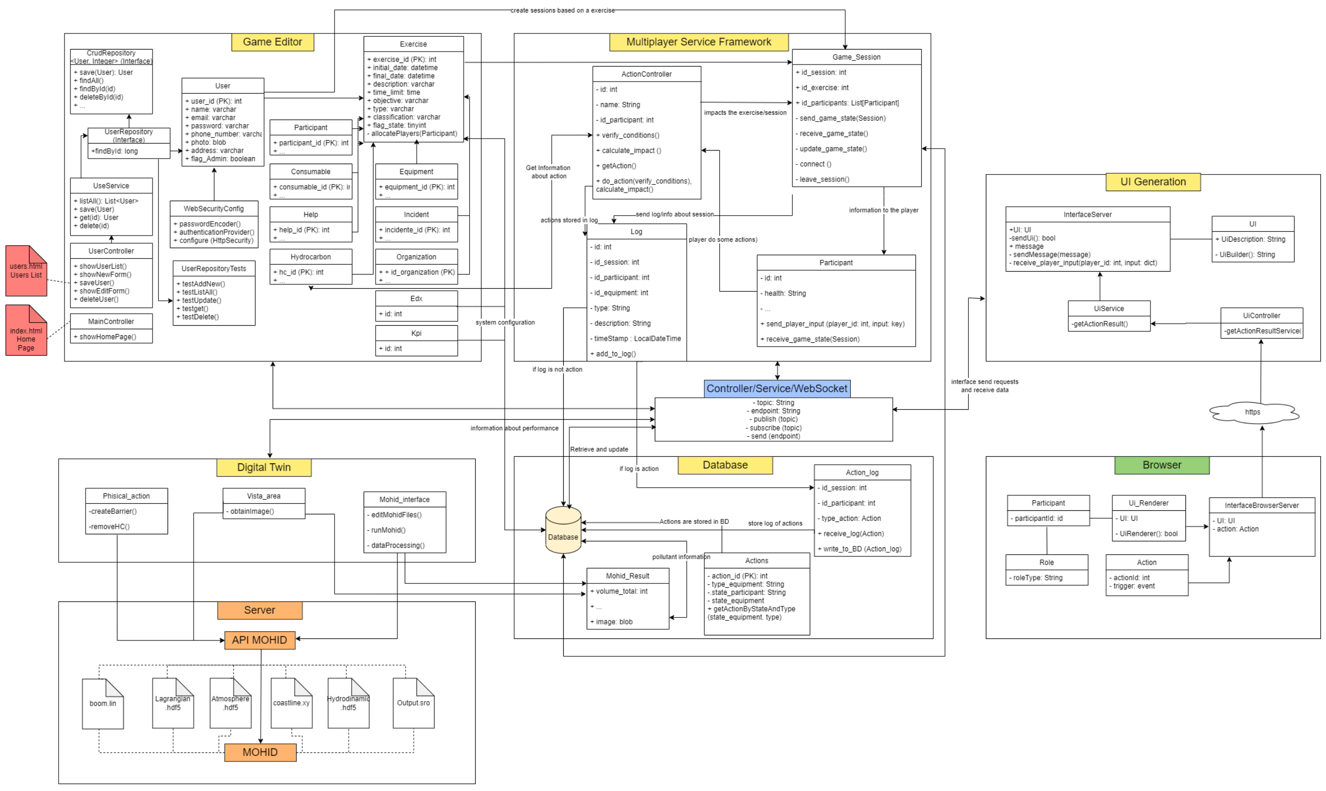


Figure D.1: MPCS Global Architecture

D.2 Data Model

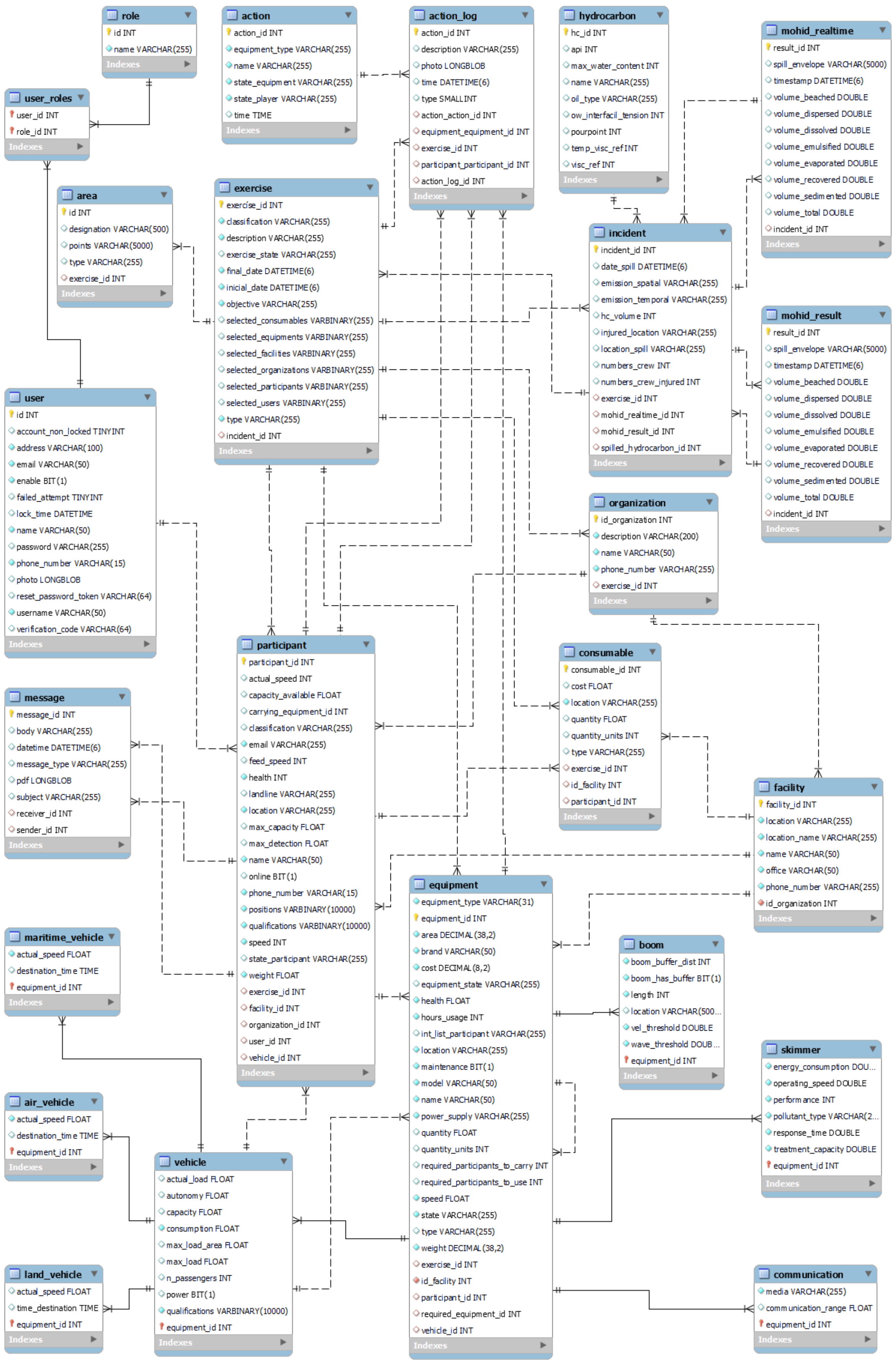


Figure D.2: MPCS Data Model