Salúquia Cristina Dias Norte Marreiros

Master of Computer Science

A Framework for Turn-Based Local Multiplayer Games

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Hervé Paulino, Associate Professor,
Faculty of Sciences and Technology,
NOVA University Lisbon

Examination Committee

Chair:	Associate Professor João Araújo, NOVA University Lisbon
Rapporteur:	Assistant Professor Eduardo Marques, University of Porto
Member:	Associate Professor Hervé Paulino, NOVA University Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

November, 2020

A Framework for Turn-Based Local Multiplayer Games

This document was created using the (pdf)LaTeX processor, based on the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my parents.

Acknowledgements

Firstly I'd like to thank my adviser, Hervé Paulino and João Silva, for the help and guidance throughout the elaboration of this dissertation and the underlying project.

I'd also like to thank my family and friends for believing in me and supporting me during my studies.

This thesis was developed in the context of research project DeDuCe (PTDC/CCI-COM/32166/2017) funded by Fundação para a Ciência e Tecnologia.

Abstract

Mobile devices are present in people's everyday lives and have gone from being a tool used purely to communicate. Currently they are also used as a means to entertain, by listening to music, watching videos or playing games. When it comes to games, these can be played alone (single player games) or with other people (multiplayer games), from strangers to family and friends. Local multiplayer games are a popular choice because they connect groups of physically close people to play and allow them to interact.

However, there are some concerns to address. Local multiplayer games connect devices but that alone isn't enough to ensure correct game play. These games need to distribute the game state between the devices and solve the issues that ensue from that. These involve matching players, managing game state (making sure players get the current state in a reasonable time frame, in order for the next moves to be performed), dealing with player inflow and outflow, among other problems.

To reliably handle the aforementioned issues, in this thesis we propose *Peppermint*, a framework and runtime system to program local multiplayer games on the mobile edge. It was developed on top of *Basil GardenBed*, a data storage and dissemination system for the mobile edge developed at NOVA LINCS, that provides communication between devices. On the other hand, the challenges stemming from the games' execution will be addressed by our framework, which are validated by the development and evaluation of one game according to a set of functional metrics.

The results obtained during testing of our framework, mostly in a simulated setting, show that the framework is able to create and store matches, letting players join, leave and play in them. It will also discard the generated data when the match ends, so that the network doesn't end up being cluttered with data that isn't being accessed anymore. These characteristics constitute a framework has a set of core features that can be expanded in future work.

Keywords: Mobile devices, edge computing, multiplayer games

Resumo

Os dispositivos móveis estão presentes no dia-a-dia das pessoas e deixaram de ser apenas utilizados para comunicar. Presentemente são também usados como meio de entretenimento, ao permitirem ouvir música, ver vídeos ou jogar jogos. Em relação a jogos, estes podem ser apenas para um jogador, ou podem ser jogados por várias pessoas (jogos multijogador), desde desconhecidos a família e amigos. Os jogos multijogador locais são uma escolha popular porque permitem que grupos de pessoas próximas fisicamente se juntem e interajam.

No entanto, existem problemas a resolver. Os jogos multijogador locais conectam dispositivos mas apenas isso não é suficiente para garantir a sua correcção. Os jogos necessitam de distribuir o seu estado entre os dispositivos e resolver as questões que decorrem disso. Estas envolvem agrupar jogadores, gerir o estado do jogo (ao garantir que os jogadores recebem o estado mais recente atempadamente, para que os próximos movimentos possam ser efectuados), lidar com o fluxo de jogadores, entre outros problemas.

Para resolver os problemas mencionados, nesta tese apresentamos *Peppermint*, uma infraestrutura e sistema de execução para implementar jogos multijogador locais em dispositivos ligados a uma rede na *mobile edge*. Foi desenvolvido sobre o sistema*Basil GardenBed*, um sistema de armazenamento e disseminação de dados na *mobile edge* desenvolvido no NOVA LINCS, que fornece comunicação entre dispositivos. Por outro lado, os desafios resultantes da execução dos jogos são endereçados pela nossa infraestrutura, validados pelo desenvolvimento e avaliação de um jogo de acordo com um conjunto de métricas relativas ao seu funcionamento.

Os resultados, predominantemente obtidos em ambiente simulado, mostram que a infraestrutura permite criar e armazenar partidas, deixando outros jogadores entrar, sair e jogar. Também elimina os dados criados quando estas terminam, para que a rede não fique preenchida com dados que já não serão acedidos. Tudo isto forma uma infraestrutura com um conjunto de características básicas que podem ser expandidas em trabalho futuro.

Palavras-chave: Dispositivos móveis, computação móvel, jogos multijogador

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

This thesis addresses the development of a framework to develop local multiplayer games on the mobile edge. As such, in this chapter, we will present the motivation behind the dissertation in Section 1.1, such as the reasoning behind the elaboration of such systems. Additionally, we present the definition of local multiplayer games in section 1.2, what are the problems that surface when creating a tool to build games in section 1.3 and what is the solution we propose in section 1.4. Additionally, we will also enumerate what contributions we made with this framework in section 1.5 and specify the remainder of the document in section 1.6

1.1 Motivation

Mobile devices are a part of people's lives and have long developed from being purely a communication means. Nowadays, they are also used as a means of entertainment, as they can be used to listen to music, watch videos or play games, among other things. In the particular case of games, multiplayer games. These are played by at least two people and require players to interact with each other and information may need to be sent between devices via wireless networks. Players play against each other or work together to reach a common goal, which requires communication between them.

These mobile devices are usually of reduced dimensions, so people can take them anywhere they go, which lets them gather in groups to play in locations like schools or while commuting.

According to a study [20] published by Newzoo in March of 2019, mobile games are in the top 3 of the most used application types during the week, in surveys done during two periods of time in 2018, alongside music. The same study shows that the top 10 of the most played games of the previous month for the same time period was comprised

of multiplayer games only. The website "Business of Apps" published an article [15] in 2019 that states that, in 2018, there were 2.2 billion mobile players worldwide, and the prevision for 2020 is that that number will increase to 2.7 billion. It also declared that, collectively, people spend 120 billion hours (over 13,5 million years) playing games every year. This said, it can be concluded that mobile gaming is still a growing trend, that will continue to exist in the years to come.

1.2 Local Multiplayer Games

Local multiplayer games are games played by players in physical proximity of one another. These can be played on a single, shared device or over multiple devices on a local Wi-Fi or P2P network. The devices involved in the game sessions are physically close and as such latency is low, as information does not need to transverse long distances to reach every device.

These games exist in a multitude of genres, such as Action, Adventure or Strategy (among others), and in the context of this framework we will study Turn-Based Games, or Turn-Based Strategy, a sub-genre of the Strategy genre. In these games, the game is played by one player at a time, in turns, and the next player can only play after the current player ends their turn.

1.3 Problem Statement

Typically, multiplayer mobile games are played using centralized servers to store data and coordinate devices. This can be costly due to latency and battery usage, which degrades the playing experience and diminishes engagement. This can be the most appropriate solution when players are in several locations, but it is unnecessary when players are in a shared location, where the games are executed in a local setting. When the setting is purely local, as a way to diminish traffic on the internet and have a better playing experience we can take advantage of resources close to where games are being played. This rationale meets a recent trend of placing data and processing close to the users, in what is called the “mobile edge” [18].

The mobile edge is formed by devices usually with reduced processing capabilities compared to the cloud, placed close to people’s devices. This allows to offload a network, promoting low latency in communications and lower battery usage on devices due to the fact that data is being exchanged near the users’ devices, which improves the playing experience. This allows people to be playing in places like public transportation or adjacent rooms, as can be seen in Figure 1.1.

However, to keep a group of devices coordinated when playing a turn-based game, it isn’t enough for them to be connected in a local setting, with support from the mobile edge. Additionally, there are also other issues to be solved, such as correctly matching players, disseminate a player’s turn information to the correct players within a period

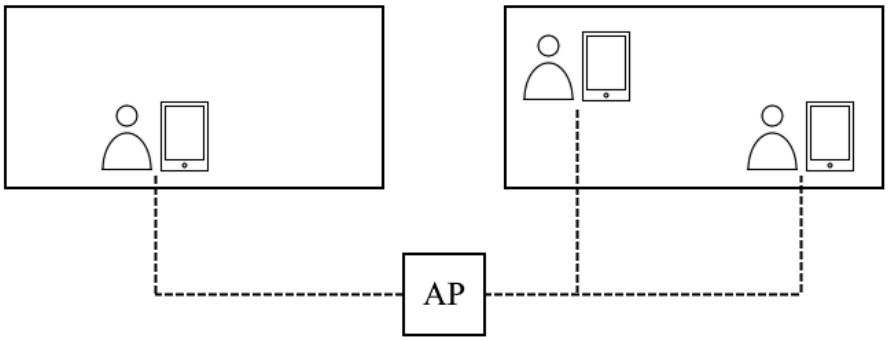

Figure 1.1: Devices connected by devices belonging to the mobile edge

of time so the match can proceed correctly (in case there are several matches occurring simultaneously), decide how to proceed when players enter and leave matches, keep and update the game score accurately, among others. To our knowledge, there isn't any system that solves all of these problems, instead only offering partial solutions to these issues.

1.4 Proposed Solution

In this thesis we propose the design and implementation of *Peppermint*, a framework for the development of turn-based local multiplayer games that does not require centralized services.

Peppermint will allow the implementation of games that allow players to play turn-based games, by creating matches or entering previously created ones, either by choosing from a list or letting a matchmaking algorithm choose the most appropriate. Each match must be able to coordinate several characteristics, such as players entering and leaving, its state or modifications made to the game board, while making sure every player is playing according to the most current state.

To achieve this, our framework will be developed on top of *Basil GardenBed* [1, 29, 31], a data storage and dissemination system for mobile edge networks that offers both a Key-Value storage and a P/S interaction model, and allows devices to subscribe to tags related to games and matches and publishing content to those same tags, notifying the respective players of changes. This system provides us with methods that allow players to create and join matches by choosing one from existing ones and make movements in turns, but there are features it does not address, either partially or completely, such as coordination between devices after turns, ordering of events or matchmaking. These are challenges we will need to overcome during the development of our framework.

1.5 Contributions

In this project we develop *Peppermint*, a framework to implement turn-based multiplayer games for devices in the mobile edge. *Peppermint* provides the base mechanics to develop

such games, such as match creation, joining or leaving a match, make moves, and end existing matches. The related methods are to be extended and built upon in order to create a game. All of this functions due to *Basil GardenBed*, which handles the communication between the devices involved in the matches.

Additionally, we also implement a case study, *Distributed Snake*, to verify how the framework and *Basil GardenBed* work in real life scenarios. In addition, we run several simulations on a computer to prove that the framework is operating as intended, via the testing of a number of case scenarios using the developed concepts.

1.6 Structure

The structure of this paper is as follows: chapter 2 enumerates a set of systems similar to this framework, alongside a set of games with analogous characteristics to the ones created with it, which are then analyzed. Moreover, chapter 3 introduces *Basil GardenBed* and explains its characteristics and API, demonstrating that it is an apt choice to build our framework on top of. Additionally, chapter 4 exposes and explains the methods and concepts that comprise the framework. Finally, chapter 5 details how to approach the necessary features, the evaluation criteria and plan and chapter 6 presents the reached conclusions and future work that can be developed.

Chapter 2

State Of The Art

Mobile multiplayer games attract groups of people to play them, because they can be played on the move, on devices players already own and are accustomed to. These can be played in cooperative or competitive fashion, connecting players towards a common goal. To achieve this, they must have a set of features which allows them to create matches, find, group and manage players and keep track of scores and state for each match. Besides this, each player can only make moves on the match they're playing, not being able to interfere with other matches that are occurring simultaneously.

This being said, in section 2.1 we present existing systems and games that will be compared to our framework and the games that are created with it, respectively. Following that, in section 2.2 three architectures are presented, in order to understand which one is the most similar to what has been implemented in *Thyme GardenBed* and how that may influence interactions between physically close devices. Finally, from section 2.3 to 2.7 the features required to correctly implement games will be specified and explained, and lastly, the conclusions regarding the presented material will be presented in section 2.8.

2.1 Local Multiplayer Games

In this section we are going to explore what features we need to take into consideration in order to implement our framework. To do this, firstly there is going to be an analysis of existing frameworks in subsection 2.1.1, which will be compared to our own, to understand which features are needed to implement local multiplayer games. Finally, in subsection 2.1.2, a group of games is going to be presented and analyzed to check how they compare to already existing games, and how our games would compare to already existing games.

2.1.1 Systems and Frameworks

Coglobe: a co-located multi-person FTVR experience [36] Coglobe is a system that features a round display used for virtual reality games and simulations, in which people interact with it directly or via smartphones, using shutter glasses and a head-tracking system to determine viewpoints using the participants' positions and projectors to produce the images onto the display.

Up to two people can use the shutter glasses, while other participants may only do so from smartphones connected to it. They can move freely around the display and interact directly with it or via the mobile devices. Every person interacting with this system may only do so in a closed room, and as such need to be in close proximity to the devices and each other in order to interact.

Transition-enabled event dissemination for pervasive mobile multiplayer games [24] This system takes advantage of physically close devices in order to spread event information. Communications are made using P2P interfaces such as Bluetooth or Wi-Fi Direct and a server on the cloud to process and maintain state. It uses the P/S paradigm to allow mobile devices to publish game data to the server, which in return delivers events regarding global state to devices via a cellular connection. On the other hand, devices disseminate UI events between themselves in ad-hoc P2P networks formed with co-located nodes. It was developed and tested using the game *TowerWorld*, in which players need to interact directly with each other, as it has cooperative and competitive features. In the test setup, instead of a server in the cloud, the server was in a laptop, which alongside the devices was connected to a Wi-Fi access point, and each device's location was obtained by scanning an NFC tag on a map, as the physical space it occurred on was small enough so that devices could be in range of each other and disseminate information when they weren't supposed to. Information is disseminated by the communication interface that is most appropriate according to the size and density of the group of devices detected, in order to decrease latency and allow for continuous game play. The players' locations, determined by scanning the NFC tags, would determine which devices should form networks. This way, even if a group of devices was in range of each other, unless they were located near adjacent NFC tags, they wouldn't disseminate events between each other.

Prime: a framework for co-located multi-device apps [9] Prime is a framework that allows multiple co-located devices to interact through the same application. A host device running the application can support several devices, providing a seamless experience with low latency and good performance. Only the host has to run the app, running several instances for the clients, that interact via a remote display app. Their interactions with the screen are sent to the host, and they are sent a stream regarding what's happening on the instance designated to them from the host. Devices are connected via Wi-Fi, in close physical proximity, which gives participants the possibility to interact directly with

each other. However, because networks are formed using short ranged communication interfaces, mobility is reduced, and people must remain close to each other to continue using the application.

Gameon: P2p gaming on public transport [35] GameOn is a system set up to allow players in public transportation to play multiplayer games using P2P communication, namely Wi-Fi Direct, between devices. Players are grouped to play games via a matchmaking server on the cloud, which allows for faster discovery of players than if it were to be done with P2P communication. This also allows to match them according to collected player data such as commuting times or individual skill levels, among other criteria, or performance data, related to how distant players are to each other. The number of participants depends on the games being played, as GameOn can be adapted to already existing games of any genre with minimal changes to their code, even if they were originally single player games. Players can interact directly with each other as they are in the same vehicle, communicating to discuss strategies or simply engaging in conversation.

Devices grouped by the matchmaking server form a P2P network, in which peers are able to be a server and a client. However, only one device is chosen to act simultaneously as such, communicating with the other peers to exchange game information. A network lasts for as long as the game occurs or the host has to leave, after which a game ends.

2.1.1.1 Comparative Analysis

Using the presented material the table 2.1 was constructed (to which was added a line relative to the framework proposed in this thesis, which is in the last row). The green rows highlight our system and the systems we found similar to ours. As for "communication between participants", "non-existent" refers to the fact that the game itself doesn't possess a means for players to communicate with each other when they aren't physically close.

From the research works presented, the ones not involving communication between devices or where people only communicate with each other can be set aside. As such, [36]'s features are not going to be further analyzed.

From the remaining systems, the ones most similar to the framework being developed are [9, 24, 35], where participants must be physically close, allowing their devices to form ad-hoc networks between them by using P2P connections. However, in [9, 35] players have reduced mobility, as P2P connections have a short range, while [24] and our framework also allows participants to move between APs and continue playing.

Regarding connecting players to play games, the article that resembles most what needs to be enforced is [35], where players need to contact a matchmaking server in order to get sorted into groups to play games. In the case of the framework being developed, devices are matched using matchmaker servers in each AP, instead of using a centralized server.

Table 2.1: Comparison table with the described systems

System	Type	Devices used	Number of simultaneous participants	Physical space where interactions occur	Communication between participants	Communication between mobile devices	Communication type	Mobility
[36]	Display	Display, projectors, shutter glasses, camera system and mobile devices	Multiple	Closed room	Direct	No	–	Reduced
[24]	System	Mobile devices, NFC tags and server	Multiple	Anywhere	Direct or non-existent	Yes	Bluetooth or Wi-Fi Direct and Wi-Fi	It depends
[9]	Framework	Mobile devices	Multiple	Anywhere	Direct	Yes	Wi-Fi	Reduced
[35]	System	Mobile devices with internet access	Multiple	Public transportation	Direct	Yes	Wi-Fi Direct and cellular data	Reduced
Peppermint	Framework	Mobile devices and access points	Multiple	Anywhere with Wi-Fi access	Direct or non-existent (for now)	Yes	Wi-Fi	It depends

Finally, to keep and manage the game's state, [24, 35]'s peers send information to a device that acts as a server and that may or may not be part of the P2P network (e.g., in [35], mobile devices send information to the peer which had the local server, while in [24] information is sent to a server on the cloud, outside of the P2P network), while in [9] one of the devices participating in the network is serving as the server. In the case of our framework, updates to the game's state are published in a region, later being cached by a server located in the AP, close to the mobile devices but outside of the network formed by them, and disseminated to other APs and all the interested nodes.

2.1.2 Games

In this subsection we present a set of games, including games used in studies, which we are going to compare to games produced with our framework.

Brick: A Synchronous Multiplayer Augmented Reality Game for Mobile Phones [5] Brick is a mobile multiplayer AR game for two players in room-scaled locations. Players use their mobile devices (that must be ARCore compatible) to scan and interact with the environment, by moving virtual objects (denominated "bricks") into a set of slots (denominated "wall"). They can communicate directly with each other to choose where to place the wall before starting a new game or collaborate to move the bricks. Once the wall has been placed, it is set in place and its location cannot change while the current game is occurring, and as such the players can only move in the room they're in. Devices are used to interact with the environment and connect to the ARCore Cloud Anchor service to store and synchronize progress and do not communicate directly with each other.

Game Changer: Designing Co-Located Games that Utilize Player Proximity [11] The Game Changer Suite presents a set of five VR multiplayer games played in a room, using only a head tracking system and a display. Players make moves using their bodies and interact directly with each other, working cooperatively or competitively towards the chosen game's goal, having also to be aware of their own location to avoid collisions with other players. Each player's actions are detected by the tracking system, being then mapped into the virtual world shown on the display. They are also limited to the room where the tracking system is mounted in, and as such can only be played in that physical space.

Designing for social play in co-located mobile games [14] This article presents a set of four local mobile multiplayer games (*Spaceteam*, *Bounden*, *Fingle* and *i-dentity*), that are played by people in the same physical space. *Bounden* and *Fingle* are played by two people in a single device, *Spaceteam* supports between two and four players and *i-dentity* can be played by multiple people simultaneously. The devices playing *Spaceteam* and *i-dentity* form local networks, with using *i-dentity* using Bluetooth to connect devices,

while for *Spaceteam* the communication means is not specified. These games are designed so that players have to interact directly with each other in the same space to succeed, due to game-specific conditions and to the fact that they are played in a single device or in networks formed by communication interfaces that have a short range.

DUAL [12] DUAL is a mobile platform game local multiplayer game for two players, communicating over Wi-Fi or Bluetooth, in which players play against each other or together against an AI in three different modes. They must stay in close proximity to one another to play, which allows them to interact directly.

NBA Jam [19] NBA Jam is a mobile sports game with local multiplayer support for two players and online multiplayer support for four players in teams of two, who play basketball with a relaxed set of rules. In the case of local multiplayer, players need to be in close proximity, and as such can interact directly. As for online multiplayer, players can be anywhere, as long as they have access to the internet.

2.1.2.1 Comparative analysis

Using the presented material the table 2.2 was constructed (to which was added a line relative to the games built with the framework proposed in this thesis, which is in the last row). The green rows highlight our games and the games we found similar to ours. As for "communication between participants", "non-existent" refers to the fact that the game itself doesn't possess a means for players to communicate with each other when they aren't physically close.

From the research works and games presented, the ones not involving communication between devices or where people only communicate with each other can be set aside. As such, [5, 11] and Bounden and Fingle's features are not going to be further analyzed.

Compared to existing games, the games created with the framework would be more similar to [12, 19], Spaceteam and i-dentity, as they all feature communication between mobile devices. Although Spaceteam's communication means isn't mentioned, and Fingle can only communicate over Bluetooth, the remaining can connect over Wi-Fi, although mobility is limited, as the devices cannot leave the range of their Wi-Fi connection and continue playing, and Fingle and Spaceteam's players can't step away from each other due to the short range of P2P networks. However, in our games, mobility is higher than in the other games due to the fact that players can move between APs and keep playing (although that may cause a temporary disconnection, if detected by the framework). Players can also communicate directly to each other if they can (by talking to each other), since in [19] and our games players may not always be able to do it, if they are scattered enough. Finally, the only major aspect these games and the ones created by our framework differ on is the fact that [12, 19] and Spaceteam are played by only up to four players, while

Table 2.2: Comparison table with the described games

Game	Type	Devices used	Number of simultaneous participants	Physical space where interactions occur	Communication between participants	Communication between mobile devices	Communication type	Mobility
[5]	AR game	Mobile devices, ARCore Cloud	Two	Room-scaled places	Direct	No	Wi-Fi or cellular communication	Reduced
[11]	Collection of game prototypes	Tracking system and display	Multiple	Closed room	Direct	No	–	Reduced
Bounden	Mobile game	Mobile devices	Two	Anywhere	Direct	No	–	Reduced
Fingle	Mobile game	Mobile devices	Two	Anywhere	Direct	No	–	Reduced
Spaceteam	Mobile game	Mobile devices	Between two and four	Anywhere	Direct	Yes	Unspecified P2P connection	Reduced
i-dentity	Mobile game	Mobile devices	Multiple	Anywhere	Direct	Yes	Bluetooth	Reduced
[12]	Mobile game	Mobile devices	Two	Anywhere	Direct	Yes	Bluetooth or Wi-Fi	Reduced
[19]	Mobile game	Mobile devices	Up to four	Anywhere	Direct or non-existent	Yes	Local Wi- Fi or Bluetooth, or Wi-Fi or cellular communication	It depends
Our games	Mobile game	Mobile devices and access points	Multiple	Anywhere with Wi-Fi access	Direct or non-existent (for now)	Yes	Wi-Fi	It depends

some of our games' matches can be played by an arbitrary number of players chosen at the start of the match, that can be set or change over time.

2.2 Architecture

Mobile multiplayer games can be implemented over different types of architectures, depending on where the network's processing units are. This influences how many points of failure there are and how long it takes to complete requests made by connected devices, and how scalable, secure and efficient they are.

2.2.1 Centralized

A centralized architecture has a device (denominated "server") serving a number of other devices (denominated "clients"). Most of the processing is performed by the server, which receives requests made by the clients, that can be highly mobile and have low battery life, which adds constraints to the traditional client-server model, in which the server assumes its clients have a set position and their connections are stable [16].

The server is easy to setup and maintain, as there is only one machine to manage. Besides processing, it also stores data and user information, keeping it all in one place.

However, because there is a single server, the whole system is compromised if it fails, as clients cannot have their requests fulfilled, which can have more or less serious consequences depending on what the server is capable of processing.

Additionally, the server could be physically distant from the clients, causing their requests to hop several routers until they reach it, which takes time and inserts potentially noticeable delays to the users.

This single point of processing can also become a bottleneck because there is a limited amount of simultaneous requests a server can fulfill at any given time, which may be inferior to the incoming requests made by the clients.

While this issue could be resolved by scaling the server, in this situation scalibility is limited, as the modifications required to increase its capabilities may be too costly, and therefore, not effective in the long run.

Another problem comes from the fact that all the information is kept in one place, which also raises security concerns, because if the server is taken over by someone with malicious intent, all the information can be stolen and used wrongfully.

Concerning our framework, in order to use this type of architecture, the devices in several regions would have to be connected to a server in the cloud, through Wi-Fi or cellular connections, which would introduce high latency and degrade the playing experience.

2.2.2 Fully Decentralized (P2P)

Fully decentralized networks are made up of a set of interconnected heterogeneous devices (denominated "peers" or "nodes") that have the same responsibilities and capabilities, and are directly connected to each other, without intermediary devices [27]. These collectively own the resources shared in the network, which are distributed between them. This increases tolerance to crashes, because even if a device is disabled, the others can still operate, and decreases security risks, because every device would need to be compromised for the network to fail.

There may also be peers that have increased capabilities (such as more storage space) and responsibilities in relation to other nodes, called "super nodes". These nodes are highly available and their IP address is publicly known, so they can be easily accessed. However, although these may make the P2P network more efficient, it may also make it more vulnerable to failures.

In this architecture, scalibility is possible and easy to perform, as the arrival of new devices is made by connecting to a group of already connected devices.

These networks may be structured, unstructured, or a combination of both. In unstructured networks, the peers use flooding to spread requests across the network. Once the requested content is found, its owner sends it back to the requester. This kind of search can be inefficient due to the fact that the whole network may need to be searched for content that may or may not exist.

On the other hand, in structured networks (such as *Basil GardenBed*), peers are ordered by unique identifiers in a circular network (where the peer with the first identifier connects to the last). When a peer searches for information, key-based routing is used. This type of routing has several algorithms but overall it allows information to be found in a number of hops that is logarithmic to the size of the network, making searches efficient.

However, there are also some negative aspects regarding P2P networks. It is difficult to maintain this type of network because the involved devices can have different characteristics, and as such there is the need to have maintenance traffic to ensure the connections between the peers.

There is also an issue regarding sent requests in unstructured networks. When a node wants to request a resource, it needs to send a request to all the peers it is connected to, asking if they have it (a concept designated "flooding"). If one of them does, it replies directly to the requester, otherwise, it forwards the request to the peers it is connected to (except the node where the request came from), and so on, until it reaches the node who owns the resource, that then sends it to the original requester.

Regarding *Basil GardenBed*, it can't be considered a fully decentralized network according to the presented features, due to the fact that there are two groups of devices with different responsibilities.

Finally, the following two systems are examples of architectures following the fully decentralized network structure.

VoroGame : A Hybrid P2P Architecture for Massively Multiplayer Games [6] VoroGame is a P2P architecture for MMGs. It operates with structured and unstructured P2P overlays, based on a DHT and a Voronoi diagram (used for overlay support and "virtual game world decomposition"), respectively. This is due to the fact that players and objects are in constant and quick change in the world of a MMG, and that can quickly hinder performance when data storage and management is done simultaneously. The hybrid solution solves that, by using the DHT to evenly distribute game data among players and the Voronoi diagram to disseminate updates to the relevant peers. Each peer stores the respective object mapped by the DHT and is responsible for a zone in the game world and determines the set of nodes affected by changes to its own state.

Peer-to-Peer-based Infrastructure Support for Massively Multiplayer Online Games [25] The system presented in this paper uses a structured P2P overlay to aid the infrastructure of MMOGs in order to improve their performance, in terms of availability, reliability and scalibility. It isn't simply a lookup service for the game it is enforced on, being able to also be used as a communication and application-specific structure. The system also complies with game features such as charging and accounting, player data storage and cheating prevention, among others. Its nodes are distributed over a map, divided in rectangular sections, and due to the fact that the network is structured, that means that all the nodes are connected in a circular overlay, allowing routing and load balancing.

2.2.3 Partially Centralized (Hierarchical P2P)

A partially centralized architecture features a set of devices (denominated "peers" or "ordinary nodes"), connected to each other and to a node that controls them [27]. Because of this as such, it is a combination of the fully centralized and the fully decentralized architectures, which in turn means it has the vantages and pitfalls of both architectures.

Partially centralized and fully decentralized P2P networks are rather similar, the main characteristic that sets them apart being the fact that partially centralized networks have a control node that overlooks the whole P2P network, while in decentralized P2P networks there is no dedicated node that controls the set of peers. At most, fully decentralized networks have nodes that have increased capabilities and responsibilities, but that do not manage the network.

On the other hand, the node overseeing the network in partially centralized networks manages the set of peers, by keeping user information or content indexes. This specialized node may affect scalibility and resistance to attacks however, due to the fact that it is a single node serving a number of peers, being a possible bottleneck and a single point of failure.

As for mobile games developed over a partially centralized network, they are usually implemented with or integrated with block chain technology. Companies such as ITAM

[34] and Equiti [13] made it possible for players to own and trade assets (in-game items such as skins or virtual cards) between each other. Assets are non-fungible (have a unique representation) which means they cannot be replaced or modified. These can be implemented with *Basil GardenBed* using immutable *DataItems*, that once created can no longer be modified, being only possible of being retrieved by other devices.

In the case of our framework, this architecture is enforced by *Basil GardenBed*, with the edge servers serving as control peers and collecting and disseminating the data generated in their region and disseminating it to other regions, while also facilitating interaction between devices, while the peers have the same responsibilities in relation to each other and the network formed among them is symmetrical.

Lastly, the following system is an example of a partially centralized architecture, that also has some fully decentralized features. However, it seems to relate closer to the partially centralized architecture as we took into consideration that, because this system independently modified the game to also act as a server, there should be more versions of the unmodified game in players' computers than modified. Hence, there should be more networks formed by modified and unmodified versions, which form a partially centralized network, rather than completely formed by computers running the modified versions, which form a fully decentralized network.

Colyseus: A Distributed Architecture for Online Multiplayer Games [4] Colyseus is a decentralized, distributed architecture for multiplayer games. It was built as an extension to the server-client model, with nodes being able to run as servers or as peers, and hence is a hybrid system. This system was tested using Quake II, and unmodified versions of the game can connect to the modified versions, that as such work as servers. Otherwise, if all the nodes connected are modified versions of the game, it functions as a P2P application, where each client runs a version of the server.

Each node runs an instance of the game, and is made of a local object store, an object placer, a replica manager and an object locator. In each node's local object store there is a primary copy of a global object (which are the mutable objects in a game, such as player characters or items), which is owned by just that node, alongside a number of replicas made from the primary copies in other nodes. These replicas are synchronized with their primary version by the replica manager. Additionally, the object placer decides where to place primary copies, which in context of the paper, are assumed to be placed on the nodes closest to the players controlling them. Finally, the object locator fetches objects according to subscriptions and publications made inside of an "area-of-interest", an area determined by a ranged query expressed according to a player's in-game character's position.

2.2.3.1 Analysis

From the presented architectures, the one used to implement games using the framework is the partially centralized one. Considering players can be playing together connected

to different internet networks, a fully decentralized architecture cannot connect all the devices, as they are not physically close and in range of each other, and therefore cannot create connections purely among themselves.

On the other hand, a centralized approach may require game data to perform a number of hops across the network until reaching the server, requiring more resources from the devices, decreasing their battery life and increasing latency, especially when the network connection isn't strong. This would also potentially flood the server, rendering it unable to attend all the requests in a timely fashion, injuring the gaming experience. Additionally, there is also difficulty in scaling up, which happens when players enter a running match. All factors considered, the server might not be able to handle every request sent by each device and therefore crash, abruptly ending a match. Therefore, a centralized architecture also isn't suited to implement such games. Contrary to this, in a partially centralized architecture, individual devices are connected to a device with management responsibilities that is placed close to them, and therefore requests are answered faster, saving resources and decreasing latency. Since requests are answered in less time, the device serving as the "server" for a set of devices has higher availability and can fulfill a larger number of requests, and as such, scalability is also possible. This further solidifies *Basil GardenBed* as an apt middleware to implement games on top of, as it is a set of partially centralized P2P networks, where the mobile devices under an AP are peers and the edge servers are the specialized nodes managing them.

Compared with the solutions presented in [4, 6, 25], *Basil GardenBed* distinguishes itself from two of these for using specialized nodes that deal with data storage and dissemination, while in [6, 25] there are no special nodes. On the other hand, the nodes in [4] can either be peers and servers, depending on which version the device connected to the network is running, while in *Basil GardenBed* the edge servers only fulfill one role.

2.3 Co-location

Co-location mechanics can be used to make a game more interesting, by allowing groups to interact, both in-game and directly with each other, in a shared physical environment.

Pokémon GO [7] players can engage in battles or Pokémon trading with each other if players are in the same space, but that restriction can be relaxed for battling with the existence of in-game "friends" and "friendship levels", each of them granting rewards and better conditions to the players involved [21].

Friends are other players, which can be added to a friends list via insertion of a 12-digit code or the scanning of the QR code associated to their account. Each player must give one of their codes to other players to be added to their lists and take advantage of the friendship levels.

Friendship levels are milestones that unlock rewards and benefits for both players involved. The milestones are "Friend", "Good Friend", "Great Friend", "Ultra Friend" and "Best Friend", "Friend" being the one with the least amount of perks and "Best Friend"

the most rewarding one. These can be increased by performing activities like battling (together against Pokémon or against each other in one versus one battles) or trading gifts or Pokémon, which results in the value of the friendship level to be increased by one. This can only be done once in every day players interact and until the maximum value is reached. Up until "Ultra Friend", exclusively, players may only battle if in close physical proximity, which allows them to interact directly with each other. On the other hand, trading is only ever done when both players are in the same physical space, where they can communicate directly and decide what to trade. In this case, friendship milestones unlock new types of Pokémon that can be traded and reduce resources spent on trades. Both of these claims are made assuming neither player is cheating (e.g., using means to move their avatar to a place the player actually isn't at).

In [35], players must be within approximately forty meters of the host in order for their devices to communicate and enable games to be played. When picking a game and game mode, a player can also choose not to play with people too close to them, as to avoid being recognized, while other players may choose to play with certain people (such as friends) or take the chance to meet new people, which will be taken into matchmaking, which is approached in section 2.4.

In [24], the players' proximity to each other is used to form P2P networks and disseminate local events. These are only formed if players are in locations marked by adjacent NFC tags placed on a map, which players can scan to establish their virtual position in the game.

In the case of our framework, player co-locality enables the possibility of matches where all the players' devices are connected to the same AP. In this case, the information related to the occurring match is only transmitted inside the AP and is not disseminated to other APs.

2.4 Matchmaking

Matchmaking is the act of joining players for online multiplayer matches based on their latency [2, 17]. Players can be matched based on several criteria defined by the game developers - physical proximity, skill level, previous matches history, relationships with other players, among others (such as in [35]). They can also decide what kind of matches they intend to play in, according to the game modes provided (e.g., play against a single player or a group of players, on a time constraint or other limitations). After choosing how they want to play, the matchmaking system processes that information and attempts to find a suitable match with other players also looking to play or matches occurring with the same features.

In [35], players are grouped by accessing a matchmaking server. This server isn't part of any of the P2P networks created after matches, being only contacted for player matching purposes via cellular communication. It collects data such as player location inside the public transportation vehicle, their skill levels or commuting times, among

other factors, to generate a weighed sum that corresponds to the match score of each player, clustering players that have similar scores in order to form networks to play the desired games.

Regarding our framework, *Basil GardenBed* doesn't have a matchmaking feature, which makes this a challenge that needs to be solved in order for players to play. We will try to place matching servers in each AP separate from the edge server in order to match players across regions, so that matches don't happen only within individual regions.

2.5 Distributed Coordination

In mobile multiplayer games, for players to play fairly they need similar conditions, which implies they act upon the same game state in any point in time. Given that data dissemination across a network is not instantaneous, it may take a period of time until every device presents the same game state, after which the next moves can be executed. This issue is especially problematic in fast paced games, where an incorrect game state may signify the end of a match because the next player to play, not having the current state, may still be making their moves based on outdated data.

In [35], all changes to the global state are made by the server, which periodically receives the moves made by each player, computes their effects on the current game state and sends the resulting state to its clients (itself included). Considering network latency is relatively low due to the use of Wi-Fi Direct, the small group size and the proximity of the players in a game, communication between clients and the server takes little time, which allows every device to quickly display the same state.

Concerning our framework, coordination between devices can be done resorting to the CRDTs and *DataItems*, both used in *Basil GardenBed*. In order for a match to proceed correctly, every player must have the same current state before the next move can be done. In cases when there is a global state to be kept updated, such as a match's score, a CRDT representing a shared counter can be used. Whenever a player finishes their turn, the update to the score is sent to the other players, which have their local state updated. Otherwise, in the case of moves, that only need to trigger local changes, such as the next player to play or changes to the game board, a *DataItem* can be used. There is a problem to be solved concerning the latest turn made however, which isn't solved by *Basil GardenBed*. While disseminating the latest turn's state, if there are devices that don't receive it, the existence of this state possibly isn't known by the affected devices after the next turn takes place. This is due to the fact that the *Basil GardenBed* only retrieves missing messages if it detects a gap in the latest messages received by the device (e.g., the device received all the messages up to five, and then receives message seven. This means message six is missing and needs to be requested). If the device doesn't receive the latest message, the framework can't detect it is missing and request it until a new message arrives. This can cause a match to end abruptly and injure the playing experience.

Another problem that arises is the ordering of events happening during a turn. For instance, if a group of players is playing a match of Distributed Snake and during a player's turn a new obstacle spawns in the board, that information needs to be distributed to all the other players so it can be spawned in their devices as well. However, if the player ends their turn and sends the snake to another player, which arrives before the data about the obstacle reaches all the players, the next player may play their turn without the obstacle, which isn't fair. This is an issue that also isn't addressed by *Basil GardenBed* and needs to be solved to ensure fair matches.

Finally, there is also an issue concerning subscriptions, as when disseminating content across the network, the current implementation sends data to all the devices subscribed to any of the tags used in subscription, although theoretically it is possible to send it to the users subscribed to the conjunction of those tags. For example, if there are two chess matches with two players each, one subscribed to the tags "chess" and "easy", and the other subscribed to the tags "chess" and "hard", all four players will receive updates for the two matches, as "chess" is a common tag in both games, which can ruin both games as they may be receiving data that doesn't concern them.

2.6 Execution Context

There are certain aspects to be considered to maintain a match with a group of players, who are possibly spread across several locations, and as such it is needed to address latency, battery and load division concerns. As mentioned in section 2.5, communication between devices isn't instantaneous and depends on the communication interface, and hence it may take time until every device gets the same information. Besides this, mobile devices have a battery with a relatively short uptime, especially if their users use more battery consuming applications or are using their devices to access the internet. Other factor to consider is justice in dividing the load between devices, so a particular device or set of devices isn't consuming more resources than the others to keep the games running. These three aspects are interconnected and can greatly affect a game's performance and the time users spend playing them.

In [35], latency in communications depends on where client devices are relatively to the game host. When playing on trains, client devices placed at about twenty meters from the host had relatively low and consistent latency values, while client devices at about fifty meters suffered from latency spikes, especially when the train stopped to let people in and out. Regarding battery usage, both the host an the clients seem to spend approximately the same amount of battery, which itself was similar to the battery usage of the unmodified games. Devices communicate using Wi-Fi Direct instead of Bluetooth. This was chosen because of its overall better performance compared to Bluetooth, when it comes to battery consumption, network latency, peer discovery and establishing connections between peers. Wi-Fi Direct allows a device to discover and connect to peers reliably until around forty meters, while Bluetooth only allows connections to be formed

until around twenty five meters; on a lab setting, it was tested that out of cellular connection, Wi-Fi Direct and Bluetooth, Wi-Fi Direct spends less battery when pinging an in-site server and its ping values are considerably lower when compared with the other interfaces. However, when connecting to the matchmaking server, communications are made via a cellular connection, because it requires previously recorded data about players to match them, and as such it's not a part of the P2P networks formed, due to their short lived connections, which would discard that information when the connections were severed. Finally, concerning load division, there was a single device who served as a game server for all the clients, including itself, as this device was both a server and a client.

In [24], the cellular or Wi-Fi communication is allied to the P2P connections, improving latency in communication. Global updates, computed by the server and concerning all devices, are sent by the server via a cellular connection or Wi-Fi, while local events, which only interest to specific groups, depend on their location and do not interfere with the global state, are sent via P2P connections, which have low latency and offload the server's work. Devices only activate P2P mechanisms when they detect other devices nearby, choosing the most adequate one according to group size and density, reducing overall battery usage.

In our case, latency is rather low since devices communicate with APs, which are close to them, and is ensured by *Basil GardenBed*. The system allows devices to communicate through several interfaces, although the preferred to use in this framework is Wi-Fi for interactions between all elements. To further keep latency low, instead of having devices fetch the match's data from other devices, it can be placed inside the notification and retrieved directly. This can counteract the load division policy, in which the most popular cells get more load, which can hurt the system's performance. As for battery consumption, it depends on the game being developed but *Basil GardenBed* itself consumes little battery, being needed hundreds of operations to deplete 1% of a device's battery.

2.7 Fault Tolerance

Games can be implemented to be played by a fix number of players or allow players to entry and leave a match. In the first case, if a player leaves, the game can either immediately end or there may be a period of time where the match tries to find a replacement for the vacant place, while in the second, players can freely enter or leave the match at any time, which may cause some changes to ensure it continues without problems (e.g., in a turn-based game, if a player leaves and there were other players connected to them, existing players may need to be reallocated to fill in that gap). In either of these cases, as long as the player exiting isn't the only host, the match has a chance to continue.

In [35], each game has a single host, which is chosen based on their commuting time, to ensure a game doesn't end abruptly. On the other hand, clients can join and leave a game after it has started without that causing issues.

In [24], the game is perpetually happening and the game state is processed and stored in a server in the cloud, hence client devices can join and leave freely. P2P networks can also be easily formed and dissolved because every device receives the events that affect global state from the server, while events affecting UI elements are disseminated between peers.

Given that games implemented using our framework can either have a fix or dynamic number of players, both of the strategies specified above can be used. Additionally, due to replication (either active or passive), there isn't a single device that computes and maintains a match's global state, allowing any device to leave a game without compromising the remaining devices. This also allows a new device to receive the current global state from any of those devices, specifically the closest to it.

2.8 Conclusions

In this chapter we compared what we want to implement in our framework with already existing systems and games, and could conclude that it is a feasible concept. We could also verify that although it doesn't create new features, there doesn't seem to exist a framework that addresses all of them simultaneously. We could also define that the partially centralized architecture is best for the needs of our framework, which solidified that *Basil GardenBed* is a appropriate choice to build on top of. Finally, each of the features needed to implement the game was explored and we could see how our approach differs from the existing materials.

Chapter 3

Basil GardenBed

In this chapter, we introduce *Basil GardenBed* and its features, which will be analyzed, in order to understand why this system is appropriate to implement our framework. As such, in section 3.1 we present the system and its characteristics in sections 3.2 throughout 3.4. Finally, in section 3.5 we conclude the chapter with the explanation of why and how it can be used to implement turn-based games.

3.1 Overview

Basil GardenBed [1, 31] is a distributed system that comprises of two sets of nodes, one being stationary and placed at the edge of a network, and the other being distributed and mobile. The stationary nodes (denominated "servers") are connected among themselves and can be accessed from wireless APs in a 1-to-N relationship. These are responsible for managing network regions (areas affected by an AP, to which mobile devices can connect to) and can either run on the APs (if these possess the resources to do so) or in simple devices connected to them (such as cloudlets [28], small scaled datacenters located and used by devices in edge networks). Besides this, they can also contact outside entities (like cloud services) to perform data analysis or other actions. As for the mobile nodes (denominated "clients"), each one can only belong to a region, even if it is in range of several APs at once. However, to optimize communication, clients may group themselves via D2D connections (such as Bluetooth or WiFi-Direct [26, 33]), as long as some of them have the ability to directly contact their region's server.

The aforementioned servers run the *GardenBed* [31] server to form a distributed system at edge of the network, that offers a "cross-region topic-based P/S abstraction" and Key-Value datastore. As for the clients, they run a system that allows for content to be shared and stored in other clients contained within a region. In *Basil GardenBed*, clients

run *Basil* [1], a key-value datastore build atop of *Thyme* [8, 29, 30], a reactive storage system for networks of mobile devices. Basil inherits Thyme's publish/subscribe programming interface, complementing it with the usual key-value store operations.

When published, besides persisting in the publishing device, data is only accessible by other clients in the same region. Periodically, servers gather and cache data published in their regions to serve two purposes: share it with other servers, and therefore, other regions; and relieve some of the burden of sharing data of the mobile nodes in its region. As such, mobile nodes can fetch data generated and stored in other regions, as long as these are connected. On the other hand, not all the data generated is guaranteed to be shared, as it may or may not be relevant. This is determined by a programmable ranking algorithm executing in the server that is used to define popularity metrics.

Being a P/S system that allows storage of data, *Basil GardenBed*'s interface provides to its users operations to publish and download data, subscribe and unsubscribe to topics within a time frame and look up the existing tags in the system, thus not requiring them to know what tags to subscribe to beforehand. Because the published data is stored, publications are visible to subscribers during the period of time given upon subscription. This also allows new users or users facing connection issues to retrieve data published before their arrival or during their period of disconnection, respectively. When new data is published, subscribers receive a notification containing the data's metadata, including its location, which may be in one or multiple devices, or maybe even a server, if that data is in the edge. The published data is then retrieved via the download operation. Besides this, the P/S topics are linked to namespaces, which allows for several applications using the same topics to coexist or for an application to contain several namespaces.

3.2 Clients

In this section we will introduce and explain the structure needed to allow the clients to interact between each other and the servers. In Subsection 3.2.1 and following subsection there is an explanation of the overall functionality of *Thyme*. The system's features will be presented in the remaining subsections 3.2.1.1 through 3.2.1.3. Finally, in Subsection 3.2.2 and following subsection there is an explanation of the overall functionality of *Basil*.

3.2.1 Thyme

Thyme is a distributed and persistent data storage system for mobile edge networks, implemented following the P/S paradigm, which allows the collection and dissemination of data between mobile devices connected to each other.

It is the first P/S system designed for mobile devices that features time-aware subscriptions. Users subscribe to a set of topics over a time period, and as such can retrieve content that has been or is being published or notified when future content matching the

specified keywords is published, as long as it is published during the subscription's time frame.

The devices do not have specialized roles and instead share the same responsibilities, being able to publish, subscribe or do both actions, meaning there are no centralized components.

Published content is stored in a cell-based DHT, divided in a number of cells within the existing regions, in which a region may contain several cells. Each cell may contain a set of devices, which are treated as a single virtual node. Hence, when messages are sent to a cell, they aren't sent to specific devices inside it but to the virtual node. As such, this cell-based DHT serves two purposes: cells are used to store publications and to verify if subscriptions match published content.

The publications originate data objects, which can be replicated via active or passive replication. Active replication happens upon publication, when all the nodes in the publishing device's cell receive a copy of the object being published. On the other hand, passive replication happens upon data retrieval operations from another device, allowing data to be present in different regions and potentially decreasing the latency of future data retrievals.

Knowing *Thyme* is designed to work in mobile networks and considers time as a first order dimension, it is an appropriate system to build turn-based games on top of, as devices perform better when data is placed closer to them, matches do not last indefinitely and players who enter a match need to receive the current score (built from moves that have already happened) in order to play correctly.

3.2.1.1 *Thyme*'s interface

Considering Thyme is a topic-based P/S system with storage capabilities, its interface presents operations available to both features, namely the ability to publish and subscribe, and data insertion, replication and retrieval procedures.

Moreover, because time is taken into consideration in this system, some of the mentioned operations, such as subscribe, were designed with that in mind. The subscribe operation is altered further to allow the user to specify more than a single topic, by using a set of tags in a logic formula in the disjunctive normal form.

Besides this, at subscription time, a subscriber also provides handlers that instruct the system on how to proceed upon success or failure of the subscription action or arrival of new notifications.

As for publications, each data object is accompanied by a set of metadata such as its tags, a short description (such as a thumbnail) and a function indicating the status of the publication (published or not). This set of information may be completed by several more attributes regarding extra information about the object, whether it should be kept in storage, if its tags' counter is to be tracked or if it is going to be replicated between the publisher's cell neighbors.

Data publication The operations of publication and insertion of a data object in the system have been merged into a single operation, due to the fact that it is a P/S system with storing features. Data objects are the basic units of work and have a set of metadata associated. The information relative to the publication of a data object contains at least the tuple <*data*, *tags*, *description*, *handler*>, in which:

- *data* is the object being published;
- *tags* is the set of tags, defined in a logic formula in the disjunctive normal form;
- *description* is a short description of the object, such as a thumbnail;
- *handler* is a callback function that indicates whether the object was successfully published or not.

The publication tuple may also contain some or all of the following parameters:

- *extra* is a byte array with extra information relative to the data object;
- *activeReplication* is a boolean value to specify whether the object should be replicated among the publishing device's peers upon insertion or not;
- *shouldBeStored* is a boolean value to specify whether the published object should be stored after its publication;
- *keepTagCounter* is a boolean value that specifies whether the data object's tag counter is to be tracked.

When a data object is inserted, the formula containing the tags is parsed to retrieve the individual tags, and each of them is hashed and indexed in the corresponding cell of the DHT, sending just the object's metadata across the network. The cells to which the tags are sent are then responsible of verifying if there are subscriptions matching the new object, notifying them in that case. Other behaviours incur from the additional parameters given during publication: the *extra* array is stored in the object's metadata; the object itself can be replicated and stored in every device in the same cell as the source, if *activeReplication* and *shouldBeStored* are set to *true*; and the *keepTagCounter* value is used on object storage and replication. In either case, the object is always kept in the device where it originated from.

User subscription When a user performs a subscription, a message containing the tuple <*tags*, *startTime*, *endTime*, *notHandler*, *opHandler*> is sent, in which:

- *tags* is the set of tags the user wants to subscribe to, defined in a logic formula in the disjunctive normal form;

- *startTime* is the beginning of the subscription period, which can be anterior to current time, or even 0, if the user wants to retrieve content matching the provided tags since it started being published on *Thyme*;
- *endingTime* is the end of the subscription period, which can have the value infinity, if the user wants to be notified of content yet to be published on that set of tags;
- *notHandler* is a handler that is executed when a new notification arrives (e.g., to download the object or not);
- *opHandler* is a handler that executes an operation depending whether the subscription was successful or not.

After a subscription's been made, the formula containing the tags is parsed to retrieve the individual tags. Each one is hashed and that value corresponds to a cell of the DHT, that is also responsible for managing subscriptions with the same tag. When a new object is inserted, the corresponding tags' cells are responsible to match it against existing subscriptions and notifying the ones interested in the content.

A user can unsubscribe to a set of tags (in our case, when voluntarily leaving a match), which will cause the system to stop sending new content to that user.

Notifications Upon publication of a new object, interested nodes are notified and sent the metadata related to it, and it may let nodes download the object or not. The default option is that nodes can download content, in which case, the object corresponding to the metadata can be retrieved by sending a data retrieval message to the nodes in the object's replication list. This message is firstly sent to the node closest to the requester. If it receives a negative reply, it tries the next location, until it either exhausts all the options or reaches a predetermined number of retries. In a last effort, the message is sent to the cell that is replicating the object, if it hasn't been tried yet, as it grants a change of higher success.

However, if the object is small enough, it can be directly sent in the notification, being readily available to be downloaded and optimizing the retrieval process, as there is no need to iterate through the object's replication list to get it, decreasing overall latency and allowing clients to increase their battery uptime.

3.2.1.2 Mutable data

Thyme supports two kinds of data types: immutable and mutable data. Immutable data can't be changed after published, while mutable data can be modified and updated with information generated by other users in the network. The system originally only supported immutable objects of type *DataItem* or that extend *DataType*, but has been extended to also support mutable data objects of type *MutableDataItem*, a generic interface for CRDT objects [3, 22, 23].

CRDT is a set of data types for highly available systems, that allows replicas to be independently updated, there being no need to coordinate and avoiding conflicts and roll-backs. These data types can be an appropriate solution for eventual convergence due to their asynchronous characteristics, all the while being "responsive, available and scalable", even with high latency or network failures.

In *Thyme*, this is made possible with the use of P/S-CRDTs, which adapt CRDTs to dynamic environments, by combining them with the P/S paradigm to disseminate data. These are designed to achieve eventual consistency in collaborative applications while handling network related issues, such as poor connectivity, which can make users connect and disconnect, either voluntarily or involuntarily.

Additionally, this CRDT type has been implemented to contain a *save* operation, which allows nodes to share their updates with others in the network. These updates are automatically published to the network by a CRDT manager which performs periodic checks (defined by the programmer) to data. The updated states then reach subscribed nodes, that can request to download the object from the publishing node's cell. To access and receive these updates, devices must subscribe to an object identifier provided upon the publication of the CRDT or subscription to the tag where the CRDT is inserted.

Thyme provides two types of P/S-CRDTs, them being LState-based and Operation-based. In LState-based CRDTs each node sends the modifications made to its local state to the other nodes in the network to be merged with their copies of the global state. This CRDT type stems from the State-based CRDT, in which the entire global state is sent to the other nodes each time there is a modification done to it. However, this is not practical in mobile networks, where data size should be taken in consideration when disseminating information across the network.

On the other hand, in Operation-based CRDTs each node sends a queue of operations made to its local state since the last update to the other nodes, so they can apply them to their version of the global state.

Thyme currently offers both of these CRDTs' set and counter implementations, and more can be created by writing the necessary logic and implementing a *merge* function.

3.2.1.3 Data replication

Thyme was thought of as a system for dynamic and quickly changing environments, and as such, data is at risk of disappearing or becoming inaccessible. In order to avoid that, two replication strategies have been defined: active and passive replication.

In active replication, upon the publication of a data object every node inside the cell the publisher is in receives it as well. This allows them to reply to requests for data retrieval in the future and guarantees the object is still available even if the original owner leaves the cell or the network entirely. This feature can be enabled or disabled.

On the other hand, passive replication relies on the nodes that have asked for the retrieval of a data object, meaning there are more replicas stored across the network, in

different cells. This increases availability and offers more locations to retrieve them from, which may decrease latency as a future requester may be closer to one of the replicas relatively to the original publisher.

For either of these strategies to work, the system needs to know the locations of each replica, by placing each object's replicas' locations in a list in its metadata, called replication lists. These are formed by lists of pairs containing the node's id and the node's cell. The lists have a limited size, and as such only contain the most recently made replicas.

Of both strategies, the most appropriate to implement turn-based games is passive replication, as a publishing node's neighbors that are not participating in a game do not have to be burdened with data that doesn't concern them, while on the other hand requesting devices should keep a replica of data published by the original device, in case of a network failure. That way, even if a device fails, the others can continue playing the game.

3.2.2 Basil

Basil is a Key-Value datastore designed to work on mobile devices connected to an edge network. This system is used as an interface between *Thyme* and the application, which allows it to use Thyme's mechanisms to implement its own.

In this system, data can be stored in "hierarchical namespaces", making it easier to build applications that have structures that bundle groups of elements under common names, such as a file system.

Additionally, *Thyme*'s *tags* have been altered into *keys* and besides the adaptation of the mechanisms offered by *Thyme*, it also provides methods that allows users to list the keys already present in the system or to link additional keys to published objects or unlink them.

Finally, considering these characteristics, and the fact that *Thyme* itself is a fit choice to implement multiplayer games with, *Basil* can also be considered a solid option to achieve that goal.

3.2.2.1 *Basil*'s interface

Considering that *Basil* was built as a layer between *Thyme* and the application, its interface presents the same operations as *Thyme*, although adapted to a Key-Value datastore setting. Additionally, it also provides processes that allow it to list all the existing keys in the system or link or unlink published objects and keys.

Users requiring to retrieve data from the system do not need to know its keys beforehand, as the listing mechanism allows them to verify which keys are present in the system before attempting the retrieval.

However, the retrieval operations do not take time-awareness into consideration, and these actions only return objects published until the retrieval time.

3.3 Edge servers

Each server is formed by the implementation of a set of modules that allow communication and dissemination of data between regions. The *Cluster-head Lookup* algorithm determines which node in a region is to be contacted when operations are to be performed on a specified data item; the *Matching Logic* algorithm verifies if new publications match with existing subscriptions; a programmable *Ranking* Algorithm that determines which are the items that must be uploaded to the edge; and the *Notification Priority Policy* algorithm determines whether notifications should be sent by the server, the clients or both. Of the aforementioned, the only module relevant for the development of this framework is the Ranking Algorithm.

A server and its clients communicate bidirectionally, with most of the communications being sent to the head of the group. Clients inform their respective server of operations that occurred within the region and can request data only available in other regions. This is placed in a set of information, which contains nodes' subscriptions and statistical data concerning the number of downloads of each data item and can be used to order items according to their popularity, and sent to the server with the objective of being sent to other servers, so their requests can be fulfilled.

Servers communicate with clients for three reasons: to notify them about newly published items in other regions; to notify them of changes to be made on metadata of items managed by clients (e.g., because that item was removed from the server's caches); to download a data item and place it in the server's caches or fulfill a download request from a client in another region.

These servers could be set to store the highest score for each game in their caches. However, currently, data is only cached according to its popularity, which means that after an uncertain period of time it stops being relevant and is removed from the caches. This has to be changed to allow high scores to stay indefinitely or only get removed from the caches after a period of inactivity of the game and to also free up space in the servers, or be permanently moved to a server in the cloud so they can be safely removed from the system.

3.3.1 The ranking algorithm

Each server periodically gathers and caches data from its region according to a programmable ranking algorithm, which allows to control what can or can't be cached. This algorithm chooses the most popular items in a region and sends a request to the relevant clients to download the objects' data and metadata, storing them in a cache. From then on, they are placed "on the edge" and for each item, the head of the nodes responsible for managing the item's metadata is contacted, so that information can reach all the nodes in the concerned groups. The fact this is a periodic event allows the cache to be updated to match the needs and tendencies of the clients.

3.4 Node mobility and churn

Considering *Thyme GardenBed* was developed for networks on the mobile edge, where devices can enter and leave APs at will (in a process denominated of "churn" [10]), it is important to address the challenges related to a node's absence. Absences are noticed when other clients attempt to contact absent nodes regarding data notifications or retrievals. In the case of data notifications, that node's subscriptions are discarded, and the abandoned region is no longer contacted to process them. On the other hand, its published data continues to be handled by the system, if it is stored in other nodes.

However, if the node's absence is temporary and goes undetected, the existing system doesn't change. Else, its return is treated as if the node has entered another region. In that case, its subscriptions are automatically reinstalled in that region when a server is found. To avoid receiving repeated notifications, the subscription starting time is updated to the timestamp of the latest publication received.

When applied to a match, an unintentional, temporary disconnection that goes undetected by the network allows the player to keep playing with no indication it happened. Else, the game explicitly states there has been a disconnection and stops the player from proceeding all together until the AP detects the player's mobile device once again and reconnects it to the occurring match. Regarding the remaining players, if the match has a set number of players, it will be put on hold until either that or another player enters it, or it ends immediately or after a timeout.

Additionally, nodes are allowed to freely roam within the range of the AP they're connected to, and continue to receive notifications regarding their subscriptions.

3.5 Conclusions

Regarding turn-based games, where there may be a state or a set of movements to maintain and be kept consistent among every player involved, a P/S system is an apt choice, as whenever a player ends their turn, they publish their decision and the remaining players are informed of the game state change. The changes made to the global state due to the results of these decisions are stored in *MutableDataItems*, which can be modified, and publish only the necessary data to update match states through the network to subscribed players; on the other hand, the decisions, such as movements, are published as *DataItems*, which are immutable, and sent to other players. Either of these interactions causes changes in the other players' local states, which are updated. This approach is executed in an attempt to keep latency low and impact the mobile nodes' battery usage as little as possible.

Furthermore, the potential mobility of players (for games that allow a dynamic number of participants) is also addressed, ensuring the game doesn't abruptly end due to temporary connection issues or withdrawals from the game all together. Instead, the remaining players continue playing and returning or new players receive the current

state of the game upon reentering or joining the match, respectively, ensuring its correct progression.

Additionally, because a single game can have several simultaneous matches happening at once, the subscription mechanism ensures that players are only getting and publishing states to the match they are playing, being isolated from the other matches and ensuring each individual match is not disrupted by data that doesn't concern it. Moreover, the existence of a programmable ranking algorithm enables games occurring specifically on a single AP to not send their data to the edge servers. Contrarily, games with players connected to multiple APs need the game state to be distributed among several servers so the game can proceed correctly.

Finally, after analyzing both *Basil* and *Thyme*, it should be noted that they're both apt to implement multiplayer games on top of. However, due to the fact that *Basil* allows its users to access the keys already published in the system before subscribing, it is a better choice over using *Thyme* alone, which requires players to know the tags prior to subscribing. Despite this, while using *Basil*'s put methods or *Thyme*'s publish methods is trivial, *Basil*'s get method only retrieves content published until the time of retrieval, and so we chose to use *Thyme*'s subscribe methods, as they allow users to receive content posted after subscribing.

Chapter 4

Peppermint

In this chapter we present our framework, *Peppermint*, alongside its features. As such, in Section 4.1 we present an overview of the system, followed by the definition of the framework and its core classes and features from Section 4.2 onwards, until Section 4.13, where the conclusions regarding the material are drawn.

4.1 Overview

During the span of this document, a *game* is defined as a set of properties and mechanics, while a *match* is merely an instance of a game, which is played by a group of players.

Peppermint is an event-oriented framework designed to create turn-based games and manage their matches on devices located in mobile edge networks. Such is accomplished by using *Basil GardenBed*, which makes use of the P/S and storage functionalities offered by *Basil* and *Thyme*.

By using *Basil GardenBed*, the game's data is managed close to the devices, causing low latency values during communications, causing the devices' batteries to last longer and allowing for a better gaming experience. *Basil*'s P/S features also let players easily join or leave matches as they wish by subscribing to and unsubscribing from the relevant tags. Additionally, the use of distributed state and structures of small dimensions aids in keeping latency values low, due to the publishing of data that can be directly and incrementally downloaded from notifications to the players' devices.

Peppermint provides mechanisms that allow players to create and manage matches, while managing the distributed state generated during the course of a match and keeping it consistent across all the players involved in it. The aforementioned mechanisms are provided by classes that must be extended by game developers to fulfill the needs of the games created with the framework, which are connected to the *Peppermint* class, that

handles the interactions with *Basil GardenBed*.

Each match is created by a player (denominated “host"), who defines the match's characteristics and can choose to begin or end a match as they see fit. After creating a match, other players (denominated "guests") can join it. Both host and guests can leave the match or play in it for as long as it is viable (the match is still within the maximum number of turns or can go on indeterminately, or the match has enough players to keep being played), but if the host leaves the match immediately ends to every other player as well. The actions taken by the players will trigger the publishing of events, which are sent to every other player in the match, letting them keep track of what is occurring. The framework also lets players be both hosts or guests (a match is hosted by a single player, but players can be hosts or guests in several matches), but the ability of letting players switch between matches is left to the applications themselves, i.e., whether they allow players to exit the application or keep it in the background, or they require players to keep the application running on the foreground at all times.

This being said, a *Peppermint* game makes use of a three layer software stack running on top of a Java virtual machine, composed of three layers: the application running the game's logic and interface, which uses *Peppermint*, and *Basil GardenBed*, as can be seen on Figure 4.1. The application interacts with *Peppermint* via actions, which communicates in return with events, while *Peppermint* makes use of *Basil GardenBed*'s methods to publish data items and CRDTs, download data items or updates to CRDTs and list and subscribe to the tags related to published content, content which may get returned as a response to handlers set up at the time of publishing or subscribing. Peppermint also uses Basil's namespace abstraction to provide hierarchical namespaces, which allows for the co-existence of several games and matches belonging to those games, and facilitates the retrieval of existing matches when a player wishes to join a match.

4.2 The framework

The `Peppermint` framework offers methods to create games and manage their matches, by extending provided abstract classes to correspond to the game being developed, besides the titular class and an interface to represent a player.

In order to implement a game, each of the three abstract classes must be expanded with the necessary properties and methods that define said game:

`MatchManager` class that gathers the matches of a game a player is playing, letting them act upon said matches.

`Match` class that stores all the properties of a match, and provides the methods to manage the match state after it has been created.

`Play` class used to create a move to be made on a match.

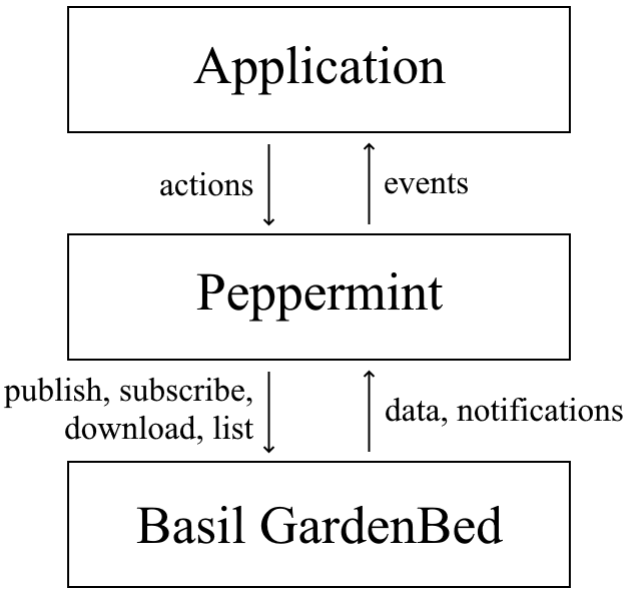

Figure 4.1: The layers that comprise a game made with the framework

Finally, there is also an interface and class to represent a player, `IPlayer` and `Player`, respectively, which have fields and methods to get a player's username and matches they played in.

The class `Peppermint` is then used by these classes to make the connection between the application and *Basil GardenBed*.

4.3 Match overview

When creating a match, and depending on the game, a host is required to provide a subset of the following set of properties, among other fields developers deem necessary to implement their games:

name match name, used in the list of active matches for other players to find and join;

numPlayers the number of players that can play (which can be a set value or vary between a minimum and a maximum values given by the host);

staticNumberOfPlayers if there can be variations on the number of players in the match;

autoStart if it can start as soon as the number of players is met (or the minimum number of players if the number of players can vary);

maxTurns how many turns can go by before the match ends (or -1 for an "infinite" match).

Besides these properties, the match's state includes a UUID, which is automatically generated upon creation, the application's context and two values to be used on timers, one to limit how long a turn can take and the other to limit how long it takes for a player

to join the match after someone leaves and it stops (both in seconds), and can be changed by the game developers.

After creating a match, it becomes available for guests to join, after which it is considered active, i.e., it has been created and published and has not ended yet.

During a match's life cycle, there are several values it can take for its status, which are now presented:

"LOBBY" set at the time of creation, when the match is awaiting guests to start;

"PLAYING" set after the match starts and the players can make moves;

"STOPPED" set after the match has started and needs a certain number of players in order to be played, but some have left (e.g., a chess match needs exactly two players to be played, if one leaves it can't continue);

"OVER" set after the match has ended because it has reached the maximum number of turns;

"CANCELLED" set after the match has ended due to player interference (i.e., all of the players have left the match or a player has decided to end it).

The relations between these statuses are presented in Figure 4.2.

When joining an active match, the guest is placed in the player list and waits until there are enough players to start. While this condition is not met, other guests can choose to enter or leave the lobby. This will be reflected on every player's device in the match, which will display events related to these arrivals and departures. If the match has enough players to start, the host verifies if it is within the given interval given upon creating the match, due to the possibility of concurrent joins. If the list has more players than allowed, the host sorts it and removes the extra players. This update causes the guests who have been removed from the list to be removed from the match. On the other hand, the remaining players get sorted into their positions, depending on the game being played (e.g., if the game being played is chess, the ordering is not complex, while if it is a game that places its players on a two dimensional map, ordering those players requires a more complex algorithm). Finally, an event is published by the host to alert the guests that the match is about to begin and the game board is drawn.

After this process, each player plays in their turn, which will change who plays for each turn and trigger an event to be sent to the remaining players regarding the next turn. `Peppermint` retrieves the play, stores it in the match and draws it on each player's device, preparing them for the next turn. It also receives and updates the next player object's value. As for the event itself, `Peppermint` checks if it is the player's turn, allowing them to play if so, or alerting them of the contrary otherwise. Each move is timed by the host, from the moment they receive an event related to a new turn or the match can be played, and stops whenever they receive a new play or the match has to be stopped or ended

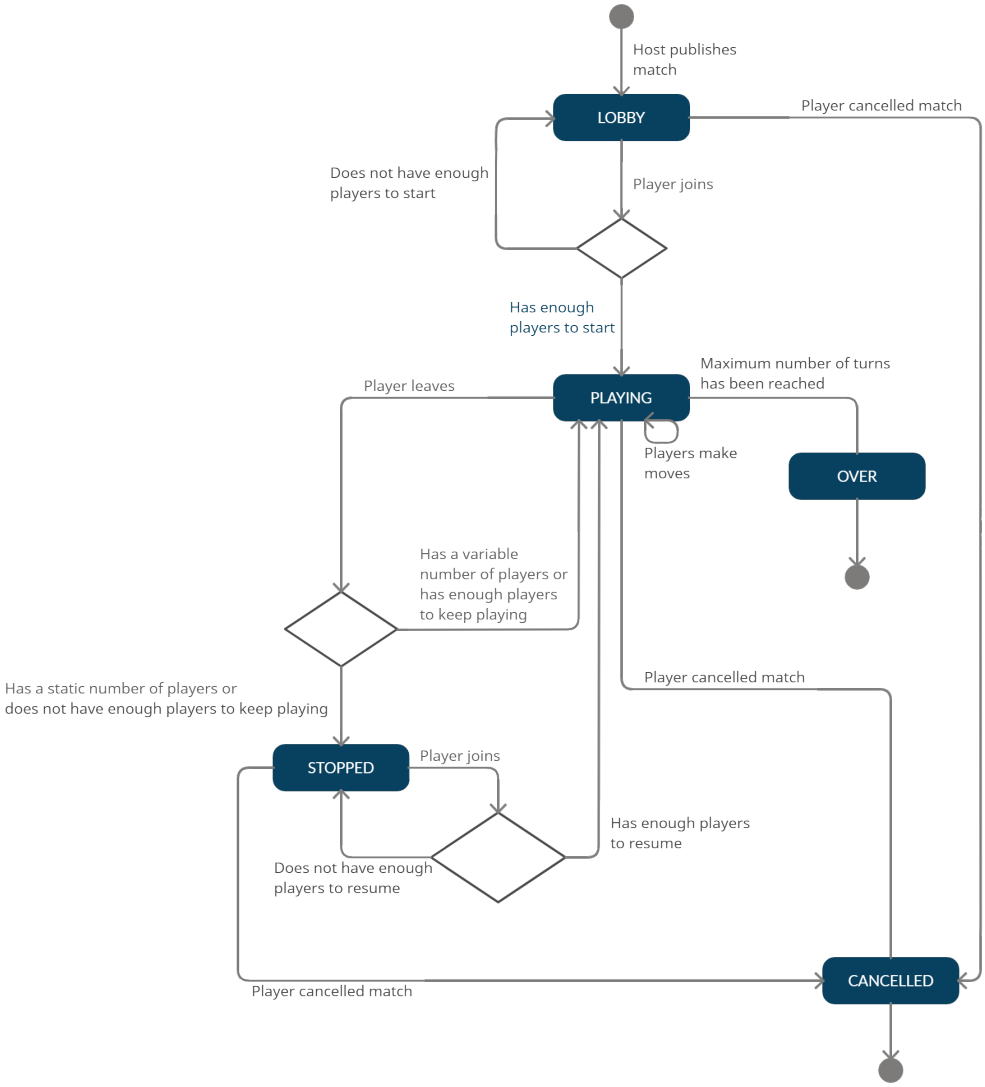

Figure 4.2: State diagram representing the relations between the statuses a match can take

(either naturally or because it has been cancelled). If the next player fails to play, they are expelled from the match, if after a number of attempts to try to get them to play they do not do so. However, if the player being removed is the host, the match is cancelled. If the match stops because a player leaves, another timer is started to time how long it takes for someone else to join the match (or that player rejoins). If no other player joins or rejoins and the host doesn't end it themselves, the match is cancelled, as it can't be kept waiting indefinitely, and an event indicating the end of the match is published. Otherwise, the timer is stopped and the match restarts from the point it was interrupted. If the next player is still in the match, they play the next turn. Otherwise, the player who just played plays again.

This process can either last until the *maxTurns* value is reached (if not configured to

go on infinitely) or until the match has no players in it, neither guests nor host. In case the match naturally reaches its end, the match is now declared as *over*, otherwise it is considered *cancelled*. The host can also choose to end the match at any point, either while it is in the lobby or already being played, which will cause it to be deemed *cancelled*. In any case, an event signaling the end of the match is sent to the remaining players' devices to conclude the match, after which the host deletes the published content generated during the course of the match and unsubscribes from the subscribed tags. As for the guests, they unsubscribe from the subscribed tags as well.

4.3.1 `MatchManager`

`MatchManager` is the class used within an application to let players manage the matches they wish to play in, by allowing them to create and publish them, or list existing active matches and join one of them, via interactions with `Peppermint`. Creating or joining a match will return the respective `Match` object, so that each player can act upon it (leave, play or even end it). A `MatchManager` object is created every time the application is loaded, and the instance is used until the application is closed. This object is used for a single type of game, so even if there are several games running simultaneously in a single device, each of those application instances will have their own `MatchManager` object (e.g., a snake game uses a `SnakeMatchManager` and a chess game uses a `ChessMatchManager`, and these are two separate objects). Additionally, this object is not stored in the `Peppermint` class, and simply interacts with it.

Each `MatchManager` object gathers the current active matches for guests to join. These can only be joined as long as there are vacancies, and the list of matches shown to guests only contains those that meet both criteria. To manually join one of the matches listed, guests select it from the list and are inserted into the player list alongside a random number, which is used later if need be to decide which players can play if the player list has more players than the maximum number of players allowed to join the match.

In order to create an extension of a `MatchManager` class, game developers must implement the methods that allow players to create a new match instance and to join matches (which can be joined manually or by letting a matchmaking algorithm decide, which is defined by the programmer) and the handlers to the events associated with those actions.

When using the method to create a match, it calls the method `onCreate(T match)` to signal that the match has been successfully created, being then returned to the host to be acted upon.

Similarly, when either of the methods to join a match is used, it calls the method `onNewPlayer()` to create the handler used when a player joins a match, and has a method that alerts the other players in the match that someone has joined it, or informs the player that attempted to join that match that it can be joined, displaying the corresponding reason. A guest can join matches by listing them and choosing to join one manually from the list of active matches, or they can let the game sort them into one via a matchmaking

```
abstract class MatchManager<T extends Match> {
  abstract T newMatchInstance(Object... args);

  abstract CreateHandler onCreate(T match);

  abstract OnNewPlayerHandler onNewPlayer();

  abstract T joinMatch(String matchID, IPlayer player);

  abstract T joinMatchWithMatchmaking(IMatchmaking algorithm, IPlayer player,
  ↪  int tries);

  //other methods
}
```

Listing 1: `MatchManager` class and methods that need to be implemented

feature. This feature automatically places them in a match based on a set of characteristics deemed relevant, such as, but not limited to, skill level or location relative to other players.

Finally, due to the fact that matches may not be able to be joined, there is a method that needs to be implemented to inform the players of those failures.

These methods are shown in Listing 1.

4.3.2 `Match`

`Match` is the class used to represent a match of a given game, with methods to leave or end it and make moves. Besides this, it also has get methods for its fields and event handlers. These handlers are to be used whenever guests leave the match, when the host decides to start it, when there is a new turn or when the match is being ended.

In order to extend this class, the developer must implement the handlers of the events triggered after the players' actions. Each action a player makes causes an event to be created and published, in order to inform the other players in the match that something has happened. For example, if a guest's device executes the action to join a match, this will create and publish an event to inform the remaining players that the match has a new player. Additionally, if the developer intends to use an algorithm to sort the order map or to add a new player to it that is different from the ones offered by the framework, they must implement it themselves and use it in the generation method.

The methods that require events to be implemented can be seen in Table 4.1.

Furthermore, there are two methods associated with the timers, *onJoiningTimerEvent()* and *onPlayingTimerEvent()* that modify the playing board in the player's device according to which timer timed out. Additionally, given that each game is different, the method used to display each move (`processLatestMove()`) also needs to be implemented, alongside

Table 4.1: Match's events that need to have their handlers implemented

Event	Cause
OnPlayerLeavingEvent	A player leaves a match
StartEvent	The match is about to start
NewTurnEvent	There is a new turn
MatchEndEvent	The match has ended
TimerEvent	A timer has timed out

```
abstract class Match {
  abstract LeaveHandler onLeaveHandler();

  abstract StartHandler onStartHandler();

  abstract NewTurnHandler onNewTurnHandler();

  abstract MatchEndHandler onMatchEndHandler();

  abstract void onJoiningTimerEvent(TimerEvent timerEvent);

  abstract void onPlayingTimerEvent(TimerEvent timerEvent);

  abstract void processLatestPlay();

  //other methods
}
```

Listing 2: Match class and methods that need to be implemented

with the method to generate the order map and add individual players to that map. These methods are listed in Listing 2.

Regarding the presented handlers, while *LeaveHandler*, *StartHandler* and *MatchHandler* contain one method each regarding their namesake, the *NewTurnHandler* handler deserves a more detailed explanation.

Whenever a play is published and the value of the next player object changes in *Basil GardenBed*, each player needs to know if it is their turn to play next. As such, the state of game board in their devices should then be updated in order to reflect that (e.g. only the current player can make actions on their device, while for the other players the playing actions are "locked" until their turn). In order to do this, the developer needs to implement the handler for the NewTurnEvent event, which changes the game board to correspond to the next turn's state.

4.3.3 Play

Finally, `Play` is the class used to represent a play made by a player in a match, and is composed of a set of basic characteristics:

turn number of the turn;

currentPlayer player who made the play;

score score generated during the turn via the player's actions;

attempt number of the attempt to be made;

direction value that determines the next player to play;

These features are used to draw the next game board and prepare it for the next turn. Each play is immutable and can't be edited after being published, and as such, `Play`s are *DataItem*s.

The direction field can take the values "UP", "DOWN", "LEFT", or "RIGHT", and is used by `Peppermint` to determine which who is the next player to play. By default this value is set to "RIGHT" upon the creation of a directionless (i.e., matches with only two players, like chess or checkers) match, or a match whose next player changes follow the order of the order map for the duration of the match (e.g. sueca). The changes to the next player according to the direction taken are as follows:

"LEFT" in one dimensional order maps, the next player changes according to the descending order of the order map; in two dimensional order maps, the next player changes according to the player "to the left" the current one on the order map;

"RIGHT" in one dimensional order maps, the next player changes according to the ascending order of the order map; in two dimensional order maps, the next player changes according to the player "to the right" of the current one on the order map;

"UP" in one dimensional order maps, this value would need to be changed to one of the previous ones, if it ever got to be used; in two dimensional order maps, the next player changes according to the player "above" the current one on the order map;

"DOWN" in one dimensional order maps, this value would need to be changed to one of the previous ones, if it ever got to be used; in two dimensional order maps, the next player changes according to to the player "below" the current one on the order map;

Although these values are set for a two dimensional map, or a one dimensional map where directions matter (i.e., the player makes a move for the player to their left or right), they can be overridden to fit whatever game a developer attempts to make (e.g. the next and previous players may be identified by those same terms, "NEXT"and "PREVIOUS", respectively).

```
abstract class Play extends DataItem {
  abstract byte[] toByteArray();

  //other methods
}
```

Listing 3: Play class and method that needs to be implemented

The class Play has a single method to be implemented (as can be seen in Listing 3), *toByteArray*, that is inherited from *DataItem*, and used when listeners download plays to the players' devices, to rebuild the play object and draw it on the game board. Besides this, the game developers need to add the fields that define a play in their game and methods related to those characteristics. For example, to make a move in a cards game, it is needed to at least know its suit and value.

In conclusion, the framework lets players create, join and leave matches, start, end and play in them. A player is able to host a match or simply be a guest, and depending on the implementation of games, they can even play several games at once.

4.4 Distributed state and stateless publications

A match's complete state is comprised of the set of the states of the objects that are involved in the match: data, player list, current data, player flow (entries and exits), status, player order map, start event, the sets of plays and new turn events, the next player and its end event.

The match's data is the information set by the host when the match is created (as listed in Subsection 4.3.2), and once the match is published it can no longer be changed, and hence it is immutable.

The match's player list is a list containing all of the players that are in the match. Players can enter or leave the match so this object is mutable.

The match's current data is a set of information composed from values from the match's data and the player list, gathering the match's name, its maximum number of players, and how many players are in the match. Due to the fact that the number of players in the match is subject to change, this object is mutable.

The match's player flow is a set of events that are published whenever there's an entry or a departure from the match. These events are independent from each other and hence immutable.

The match's status is an object that pinpoints where the match is in its lifespan, to determine what is to be drawn or done.

The match's status changes along the course of the match and as such this object is mutable.

The match's player order map is a map containing the players in the match and their positioning in the game board. Since players can enter and leave at any time, this object needs to change alongside the player list, and as such it is mutable.

The match's start event is the event that causes the players to transition from the "LOBBY" to the "PLAYING" status, effectively starting the match. As this event is used to begin the match and that only happens once during the match's course, this is an immutable object.

The match's plays is the set of moves made by the players in the match. Plays are independent from each other and only made once and are immutable objects.

The match's new turn events is the set of events generated upon the publication of a play. Since plays are independent from each other, so are these events, and hence these are immutable objects.

The match's next player is the player to play next. This value is changed every turn and hence this is object is mutable.

And finally, the match's end event is an event sent to the players whenever the match is ending, either because it reached its number of maximum turns or because it was cancelled. This event is sent one at the end of the match to signal its end, and hence it is immutable.

The instance of the class`Peppermint` running on each device keeps a copy of some or all of these objects as they are created or downloaded, alongside the identifiers generated for them by *Basil GardenBed*. All of this data is stored in a map, which uses the match's name as a key, in order to facilitate fetching and storing operations.

Out of all of these, the match's player list, status, player order map and next player are *CRDT*s, while the remaining are *DataItem*s. This can be visualized in Figure 4.3. The tags used to publish and subscribe to these items follow the pattern *<gameName>/<matchName>/<objectTypeName>*.

match's data (immutable) *<gameName>/<matchName>/MATCH*;

match's player list (mutable) *<gameName>/<matchName>/PLAYERS*;

match's player flow (immutable) *<gameName>/<matchName>/PLAYER_FLOW*;

match's status (mutable) *<gameName>/<matchName>/STATUS*;

match's player order map (mutable) *<gameName>/<matchName>/ORDER_MAP*;

match's star event *<gameName>/<matchName>/START_EVENT*;

match's plays (immutable) *<gameName>/<matchName>/PLAY*;

match's next player (mutable) *<gameName>/<matchName>/NEXT_PLAYER*;

match's new turn events (immutable) *<gameName>/<matchName>/NEW_TURN_EVENT*;

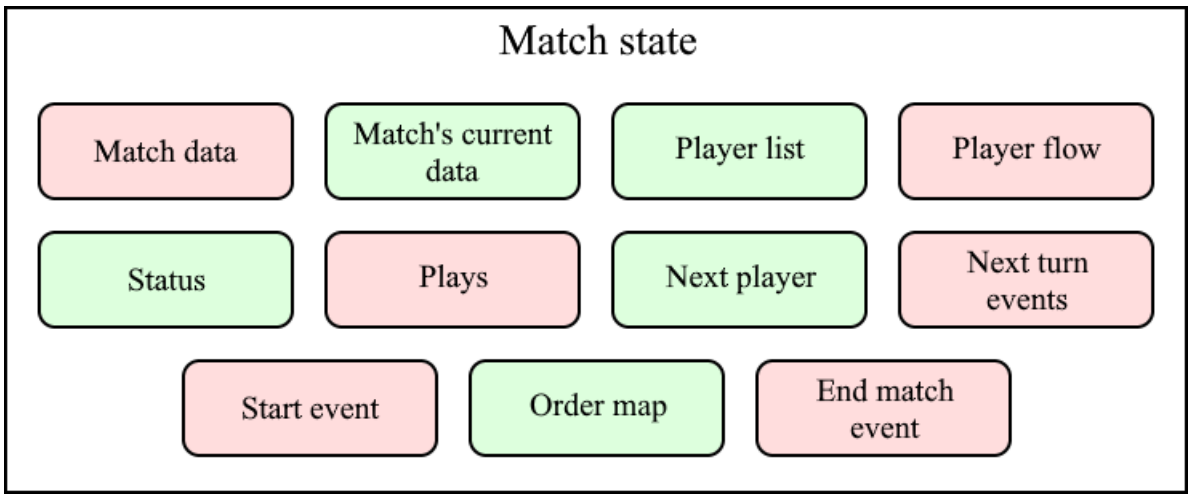

Figure 4.3: Complete match state. The green coloured containers represent *CRDT*s and the red coloured containers represent *DataItem*s

match's end event (immutable) *<gameName>/<matchName>/END_EVENT*;

Besides this, the match's current data (mutable) is published in the tag *<gameName>/<VACANCIES>* whenever the match has vacancies, and is removed from the tag when it is full.

At the time of creation, each match is comprised of three State-based *CRDT*s (as described in Section 3.2.1.2), the player list, the match's current data and the match's status, which are published into *Basil GardenBed*, alongside the *DataItem* containing the match's data. Each guest joining a match subscribes to those objects' respective tags, and both host and guests subscribe the tag that notifies them when players are entering or leaving the match. Each of the subscriptions has a handler associated, that downloads that data to each player's device. All of the mutable objects used are state *CRDT*s, which were used because their states are updated throughout the match's lifespan.

When an update to a *CRDT* is made and published, players only need to fetch it from the notification and merge it with their own copy. This way, only incremental updates are published each time, and the entire object is not published several times over a single match, allowing it to have low latency.

If the match reaches a state where it has enough players and can begin, the player order map is generated and published by the host, being retrieved and stored. Additionally, all players subscribe to the play, new turn event and next player tags, which set up the handlers to notify them of new plays and events and changes to the next player object, respectively, so they can retrieve them. The `Play` objects are stored in *DataItem*s and are published independently from one another. Players download them from the incoming notifications and store them in the match, then using the data contained within the extended `Play` object to draw the current play on the game board, getting it ready for the next turn. `NewTurnEvent` objects are also stored in *DataItem*s, and when downloaded are

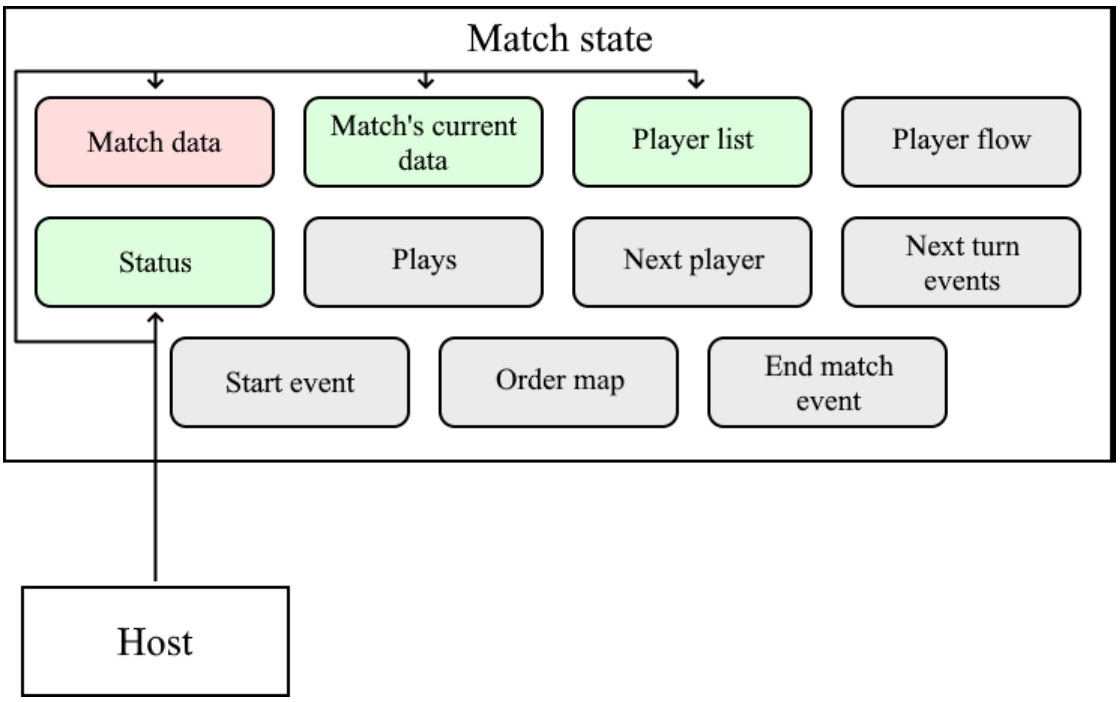

Figure 4.4: Creating a match. The grey coloured containers represent objects to be created and published at the time of creation of the match, while the continuous line represents a publish.

saved into `Peppermint`, so they can be fetched later on if need be, in case the match stops and returns to being played, with the framework then verifying if it that player's turn to play, and further changing the game board as necessary (e.g., unlocking the playing options for the next player - which is then the current player - and locking them for the player who just played).

4.5 Creating a match

The match's data is used to create the `Match` object, which is published into *Basil GardenBed*, alongside the player list, the match's status and the match's current data objects. The player list contains the host by the time of publication and the status object is initialized with the "LOBBY" status, to indicate the match is awaiting guests, and the match's current data, besides the match's name and maximum number of players, has its current players set to one due to the host being in the player list. The host's device also subscribes to a tag that notifies it when players enter or leave the match, and to tags related to the published *CRDT*s (player list and status and match's current data objects) in order to receive notifications about changes made to them. Those changes are extracted from the notification and used to trigger the *CRDT*s' local update.

Once the object relative to the match's basic characteristics (i.e. name, number of players, among others) is published into the system, it can no longer be edited, and the only way the host can "alter" those values is by leaving and starting a new one. This is, if

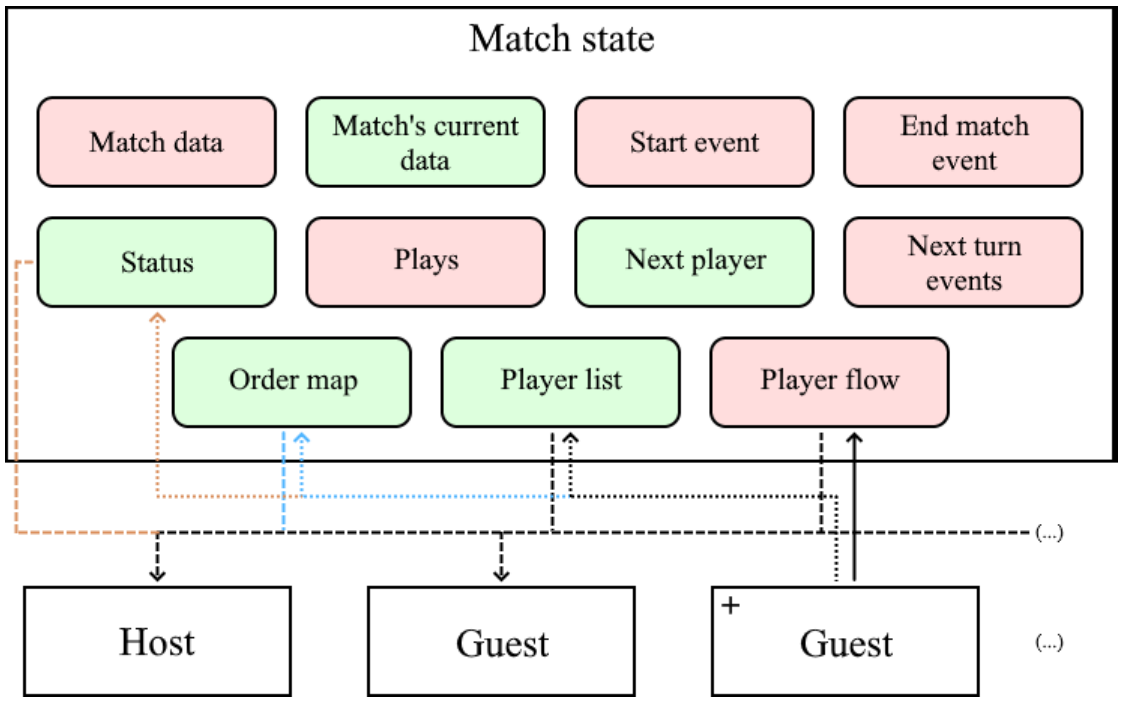

Figure 4.5: Joining a match. The player is inserted into the player list and an event is triggered to alert the other players there is a new player in the match. In case the match can start automatically, the status changes to "PLAYING", being the remaining players updated (which is signified by the orange arrows, being the dotted one a change made to published data and the dashed line an update delivered to the remaining players). However, if it has already started, the player has to be inserted into the order map, which is then updated for the remaining players (which is signified by the blue arrows)

the host want to change the match's name or number of players, they can't do it if it has already been published. Changes to these fields midway through a match could result in players getting kicked out if the number of players is lowered, or causing disparities in subscriptions because the match's name was changed, causing some players to stop receiving information (this would specially impact the match after it had started, since there would be players subscribed to the tags with the old match name, while new players would subscribe to the tags with the new name, and all of these depend on the match's name. In order to avoid these issues, the older players would need to have their subscriptions changed to the new name tag to avoid incompatibilities, not to mention *CRDT*s tags would have to be be altered to fit the new name because they are published in the tags with the old name), which could be detrimental to the playing experience.

After the match's publication, it is returned to the host to store locally and act upon it, being able to leave, start (if the conditions for that are fulfilled), play in or even end it.

The described process of creating a match can be visualized in Figure 4.4.

4.6 Joining a match

Players wishing to join a match can select it from a list of active matches with vacancies. Vacancies are checked by retrieving the matches published on the tag associated with vacancies for that game. Each match's host publishes its data there if it is lacking players and is active, and removes it when at least one of the conditions fails. In case there aren't any matches or the existing ones don't fulfill the players' requirements, they can refresh the list to obtain new data. However, they can also let a matchmaking algorithm sort them into one.

While the retrieval of the matches needing players could be done continuously, we considered it would be best to retrieve it on demand to limit the number of operations done by *Basil GardenBed*.

The list presented to the players only includes matches with vacancies and that are active at the moment of retrieval. However, as there can be players coming and leaving any match at any time, or the match may even end in the meantime, this list can get quickly outdated, and hence, they may not be able to join it.

When a suitable match is found, they first must verify if there are any vacancies (as other players may have filled the match between the local `MatchManager` object obtaining and displaying the list and the player choosing it) and if the match is in a compatible status (i.e., "LOBBY", "PLAYING" if it doesn't require a set number of players or "STOPPED" if otherwise).

If so and the status is "LOBBY", they join it and are put into the player list, subscribing to it to receive updates on its state as other guests join and leave. However, there may be concurrent joins, and more players attempt to join than the available vacancies. In that case, an algorithm is run to remove the extra players.

When joining, each player is inserted alongside a randomly generated value. In the case that the player list has more elements than the maximum value allowed, the host removes themselves from the list, sorts the remaining elements by numeric order using the randomly generated value, and then reinserts themselves. Afterwards, they remove *playerList.size() - (match.maxNumPlayers)* elements from the list, and change all of their values to *Integer.MIN_VALUE*. This is done to prevent that players who have already played to not be mistakenly removed from the list midway through a match.

In case that the status of the match is either "PLAYING" or "STOPPED", they are also sorted into the player order map, to which they also subscribe, so they and the other players in the match know when it is their turn to play. If the status is "STOPPED"and the required number of players to resume the match is met, the match is resumed.

The guest also subscribes to a tag related to the flow of players relating to the match so they know who joins and leaves. The action of joining the match itself will trigger an event to be published in that tag, which is broadcasted to all the other players, to let them know another player has joined the match.

The described process of joining a match can be visualized in Figure 4.5.

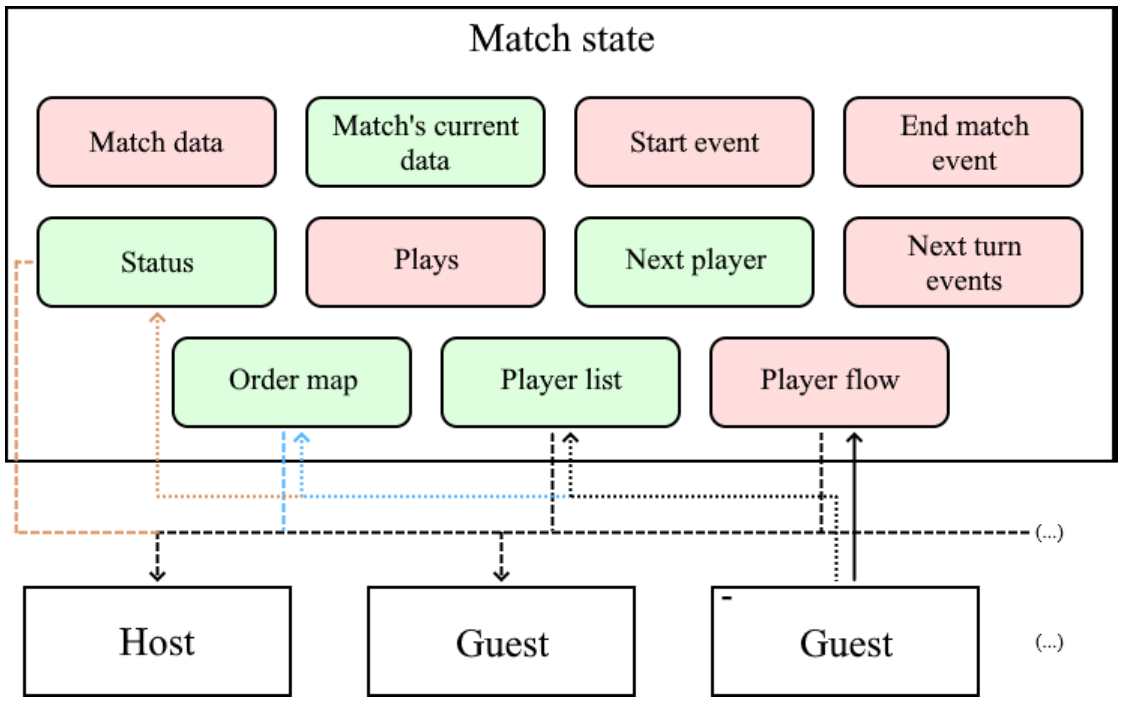

Figure 4.6: Leaving a match. The player is removed from the player list and an event is triggered to alert the other players there a player has left. In case the match has already started, the player has to be removed into the order map, which is then updated for the remaining players (which is signified by the blue arrow), and if the match needs a set number of players, its status changes to "STOPPED" (signified by the orange arrows)

4.7 Leaving a match

After entering a match, either it has started or not, a player may decide to leave. As such, the player is unsubscribed from all the tags they were subscribed to upon joining and if it has started, the match's status changes to "STOPPED" if the match requires a set number of players to be played, "CANCELLED" if they are the last player leaving, or it remains as "PLAYING" otherwise, and they're removed from the order map. After leaving the player doesn't receive any updates on the match, and is free to rejoin it or join another match altogether.

Leaving a match will also trigger an event to be published on the tag related to the player flow. That information is then broadcasted to all the other players, alerting them that a player has left the match. Upon receiving this event, the host will remove them from the order map, and if the match is being played by a set number of players, stop the match and the timer related to new turns, starting instead the one that waits for new players to join the match. If this timer times out, the match is cancelled.

However, if the player who decides to leave is the host, the match immediately ends, as the host manages the timers and the order map object (only they change it when there are entries of departures of guests on the match). If the host leaves, the remaining players do not have the capability to replace the host, and hence an event to indicate the match

is ending is published, the players unsubscribe from the subscribed tags, the host deletes the published content, and the match ends.

A match which status' has been changed to "STOPPED" may wait until another player joins, either on a timeout or until the host decides to end it or end immediately, depending on the game's implementation.

The described process of leaving a match can be visualized in Figure 4.6.

4.7.1 Managing involuntary departures

In the case that a player is forced to leave a match (e.g. due to the battery in their device running out or they leave the range of the Access Point) there are two possible outcomes: they leave and come back before the absence is noticed; or they leave and that absence is noticed by *Basil GardenBed*.

If they leave and get back to the match before *Basil GardenBed* can detect the absence, the match proceeds as intended.

Otherwise, the system detects they are gone when it is the player's turn and they timeout (due to a timeout provided by `Peppermint` and that is programmable by the game's developer), and acts as if the player left on their own volition. This outcome can unfold into several others, described below.

If the player comes back and can be placed back into the match, their subscriptions will be redone, although that may cause the match to relocate them to a different place on the order map if it has been taken by another player who joined before of them, causing their new plays to be made to different players. This return is seen by `Peppermint` as if a new player joins the game.

However, if the player comes back but someone else entered in their place they cannot rejoin the match unless someone else leaves it before their rejoin.

Additionally, if the match ends, they can't return to that it.

Finally, the player may not even come back, in which case no extra steps are done.

4.8 Starting a match

When a match reaches or is within the number of players required to start (and depending on whether it can start automatically or if the host has to start it themselves), it can do so. The host changes the status to "PLAYING" and the players are sorted into a map to generate the playing order, which is published in the respective tag, as is shown in Figure 4.7. Finally, after being notified that the game's status has changed, each player subscribes to the tags for plays, the next player and events generated on new turns, and an event is published into the start tag and triggered in each device to draw the playing board, after which the first move can be made.

Upon receiving these publications, the guests download them to their respective devices and update them as new notifications arrive from the respective tags.

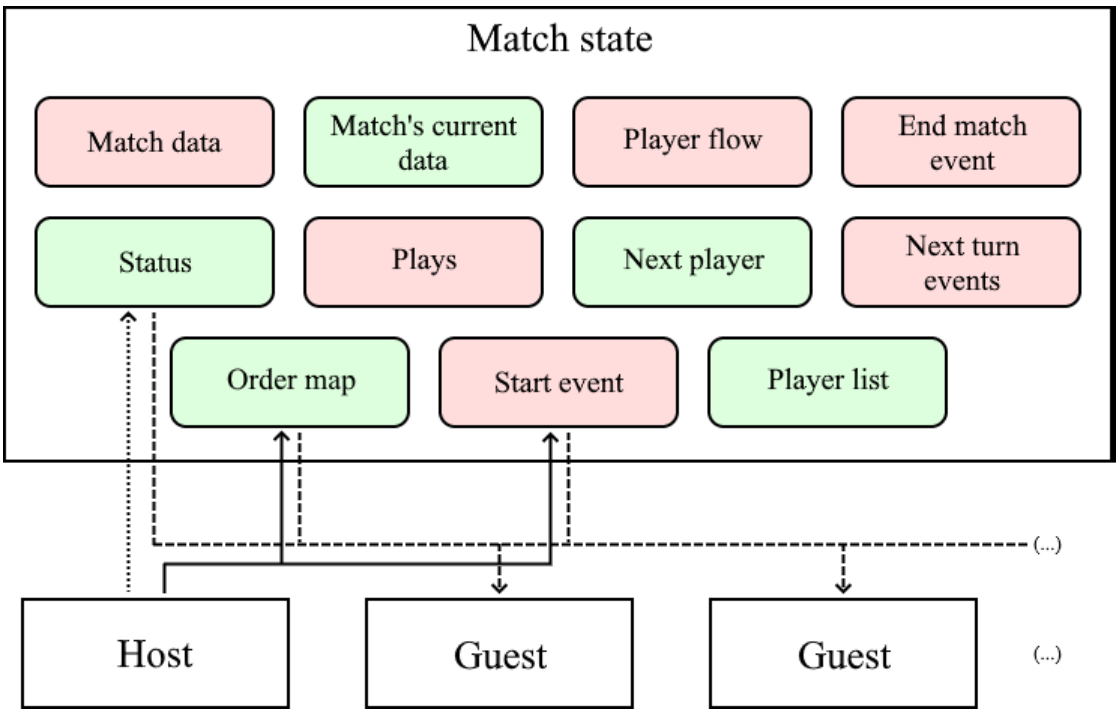

Figure 4.7: Starting a match. When a match starts, its status is changed, the order map is published and an event is published to the remaining players to indicate them of that, and this triggers the listeners setup on the guests' devices on subscribing to download this content.

The described process of starting a match can be visualized in Figure 4.7.

4.9 Communicating plays

By the time a match starts, each player's `Peppermint` instance is subscribed to a tag related to the plays, with a listener setup to notify them when a new play is published, which is then downloaded into their devices and used to update the game board. They are also subscribed to tags that notify them about the next player by `Peppermint` letting them know who played, and events that are published for them to get ready for the next turn.

When a play is made and published, it is downloaded by the other devices. Additionally, the next player's *CRDT* value is also changed according to the value of the play's direction field, causing it to be updated on the remaining devices. Finally, an event is published, which uses the most current move's contents to update the playing board in each player's device. Additionally, `Peppermint` verifies if it is their turn to play next by comparing the next player *CRDT*'s value with the username stored in each device's `Peppermint` instance. In case it is, they'll have the ability to make and publish their own play, while the remaining players wait to be notified about it, via the listener setup by the framework to receive and download the plays. This process continues until the match ends, either because it reached the maximum number of turns (if defined), it doesn't have

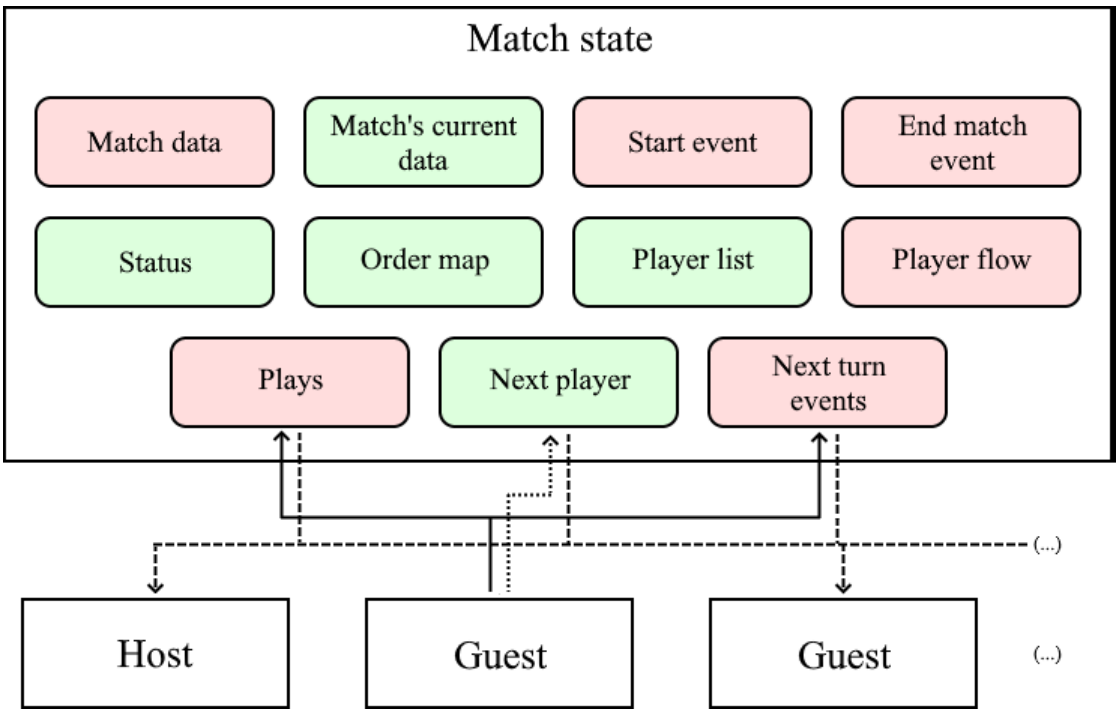

Figure 4.8: Playing in a match. When a play is about to be published, `Peppermint` changes the next player object's value. The play and an event regarding the new turn are then published in their respective tags, and the next player *CRDT* is updated, which prompts the listeners on the other devices to download that information. The dashed line is used to represent both the download and update from the immutable and mutable objects, respectively

enough players to continue or because it has been cancelled.

However, in case the next player doesn't play within the interval of time set upon the match's creation, that will eventually cause a timeout. In that case, the host removes the problematic player from the list, which can cause the match to change status to "STOPPED" if it plays with a set number of players. However, if the player being expelled is the host, the match ends.

The described process of playing in a match can be visualized in Figure 4.8.

4.10 Ending a match

When the players playing a match reach the maximum number of turns, one or all players leave (this depending on the implementation of the game), or the host decides to end the match, it is needed to finish it and make it unavailable to new players. As such, its status is changed to "OVER" if the number of turns reaches the maximum value or "CANCELLED" otherwise, Furthermore, if the player list is not empty, an event is published to the remaining players to let them know the match is ending, and the subscriptions for all players regarding that match are cancelled as they shouldn't be receiving more data about the match, nor should any more be published. Finally, the data is erased from the system,

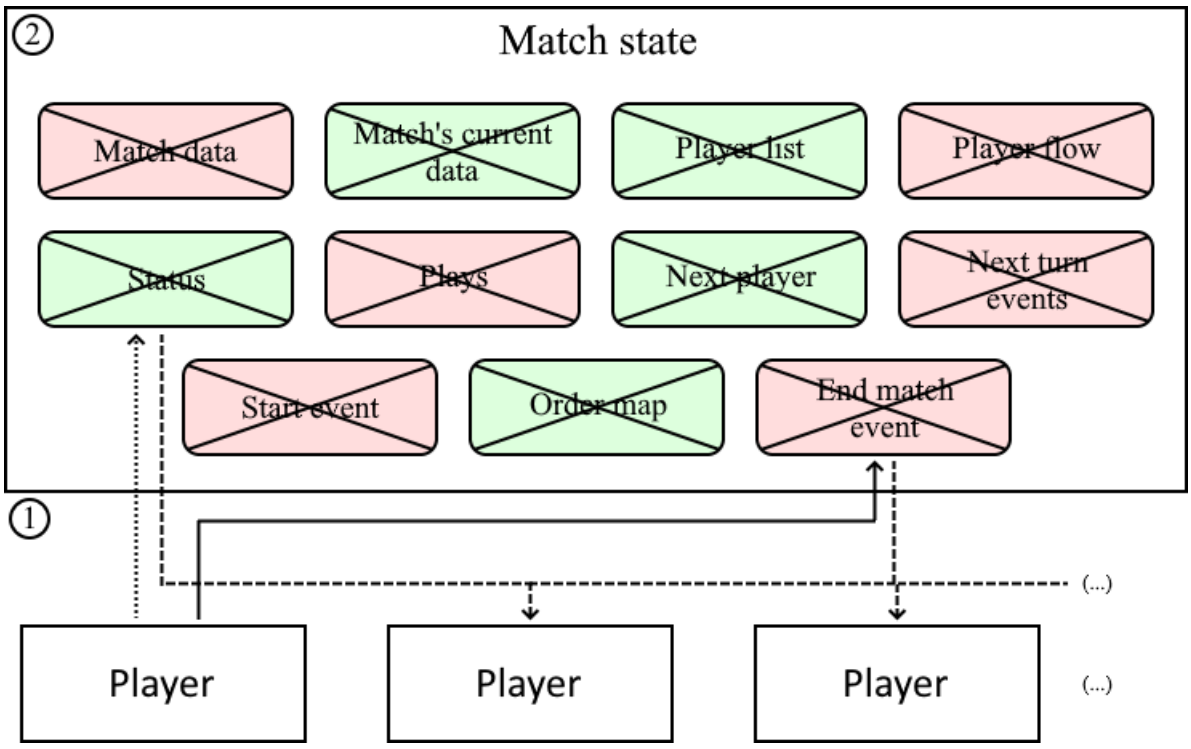

Figure 4.9: When a match ends, a player changes the match's status and sends an event to mark the end of the match, which notifies all the other players (1). Then, the match's data is deleted from *Basil GardenBed* and from `Peppermint` (2).

as it is now obsolete. While `Peppermint` deletes its local objects, this is not guaranteed to happen in *Basil GardenBed*. This could either be supported by `Peppermint` itself, or via a garbage collection mechanism implemented in *Basil* or *Thyme*.

In either case, it is also removed from each player's `MatchManager`, not only because it isn't needed anymore, but so they can't tamper with it, even if the changes they can do are purely local because they are no longer subscribed to the tags relative to the match.

The described process of ending a match can be visualized in Figure 4.9.

4.11 Matches with a set number of players

Games that require matches to be played with a set number of players do not tolerate variations to that value. Examples of such games are chess, sueca or checkers.

If after a match's start a player leaves (either voluntarily or not), and depending on the implementation of the game, it can either end or wait for another player to fill in the vacancy, after which they'll be sorted into the order map. In the case it waits for new players, this can happen in a set amount of time (provided by `Peppermint` and programmable by the game's developer), but the host can also decide if they wait until someone joins or end the match if no one seems to join it.

4.12 Matches with a varying number of players

On the other hand, there are games that allow their matches to allow players to join and leave. The order map is altered if needed to suit the new playing conditions, and in the case the match has a minimum number of players, as long as the flow of players doesn't cause the number of current players to lower that value, the status remains unchanged, and the ones in-game keep playing.

4.13 Conclusions

In this chapter we introduced and explained the core mechanics to implement turn-based multiplayer games for devices in mobile edge networks. These mechanisms allow players to create matches, join, leave and play in them if it gets the necessary amount of players. Moreover, they also have the ability to end one, either by one of the players' choice (depending on the game), because it reached the last turn or because there are no players in it to play it. The matches also display different behaviours to players leaving the match after it started depending on whether it has a set number of players or if that value can vary. These departures may be voluntary or involuntary, being both cases handled by the framework and the underlying system, *Basil GardenBed*. Finally, in order to implement their games, developers who use the framework must extend the given classes (`Match`, `MatchManager` and `Play`) with their own concepts, which allows for the creation of several games.

Chapter 5

Case study: Distributed Snake

In this chapter, we present the evaluation made to Peppermint. In Sections 5.1 and 5.2 we present the evaluation and methodology, in Section 5.3 the case study is presented, in Section 5.4 we present the simulated environment and in Section 5.5 the derived results are presented.

5.1 Evaluation

The evaluation of this framework was done in two parts. To understand which aspects a developer must take into account when creating a game with Peppermint, we developed and analyzed a study case. Additionally, we also tested that implementation in a simulated environment and physical devices, to conduct the empirical assessment of Peppermint's correctness.

5.2 Methodology

The framework was tested using functionality and usability criteria, such as the correctness of the playing order in deterministic settings or of the implemented handlers, or the time taken for matches to be available for guests to join after being created. We want to know if the framework is correctly performing the actions, while also correctly responding to the events developed in the study case in response to the pre-programmed actions.

Firstly, we implemented our study case, developing each of the required methods and handlers.

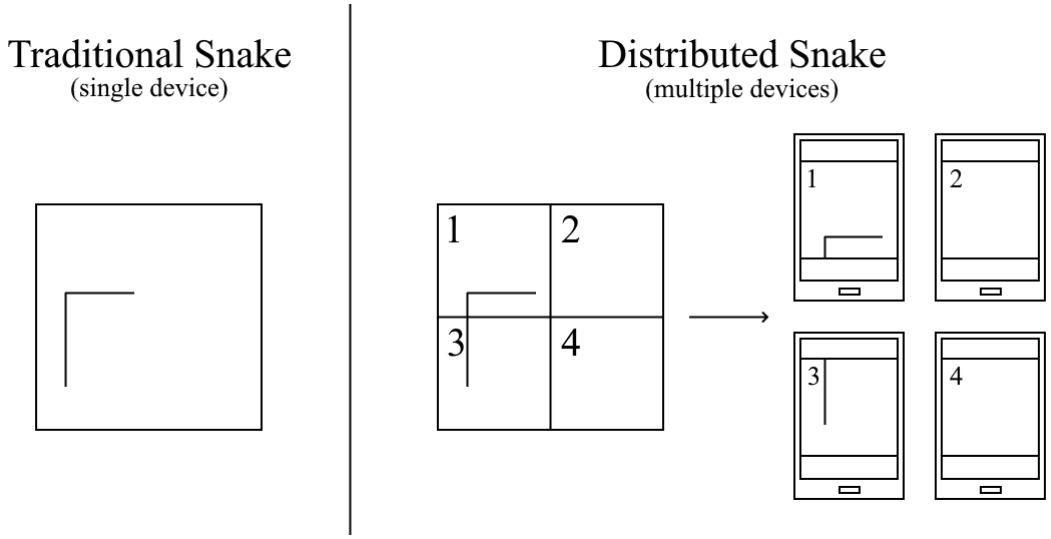

Figure 5.1: The difference between *"Traditional Snake"* and *"Distributed Snake"*. The numbers in the corner of each piece of the playing board are simply an identifier for the purpose of showing the concept and do not represent player order

Afterwards, we analyze the implemented study case, by evaluating what was necessary to program it, and if that implementation needed to take into consideration Peppermint's distributed state management characteristics.

Finally, we conducted experiments using the use case, by using simulated and real environments. In the simulated environment, we used a set of traces to emulate several situations in a deterministic way, which might have been hard or impossible to occur in the real environment, while the testing in the real environment was to be done using physical devices, in which determinism is not guaranteed. Unfortunately, testing on physical devices could not be done.

5.3 Implementing Snake

In order to evaluate *Peppermint*, we have developed one study case, a distributed version of *Snake* (which will hereon out be called *Distributed Snake*). Unlike traditional *Snake*, this game is to be played by at least two players, with the players' devices forming the game board, which can be seen in Figure 5.1, in which the snake circulates via the players' actions. In this game, players must control the snake to the edge of their device's screen in order to end their turn and hand over the control of the match to the next player.

To implement the game, the `Match`, `MatchManager` and `Play` classes need to be extended into the respective classes for the game, `SnakeMatch`, `SnakeMatchManager` and `SnakePlay`.

The class `SnakeMatch` presents the necessary event handlers to respond to the actions made during the match's runtime, alongside five methods, implemented as follows:

LeaveMatchEvent its handler has a method that alerts the other players in the match

about a player's departure;

StartEvent its handler has a method that receives the match and alerts the player it is ready to start, allowing the first player to make the first play;

OnNewTurnEvent its handler has one method that allows a player to make a move on their turn, and another method that alerts it is another player's turn;

EndMatchEvent its handler has two methods that alert guests that the match has ended, either because it reached the maximum number of turns or because it has been cancelled, respectively.

processLatestPlay() stores the play in the match, to be used when needed;

onJoiningTimerEvent(TimerEvent timerEvent) this method cancels the match when no new players join the match until the set time limit after it stopped;

onPlayingTimerEvent(TimerEvent timerEvent) removes the player who is supposed to play from the match when the timer reaches the time limit.

As for the `SnakeGameManager` class, the implemented methods are used to create and join matches. While the method to create matches simply creates a new object of type `SnakeMatch`, the join method that requires a single match returns the match if it was successfully joined or null if otherwise. As for the matchmaking method, it doesn't use a specific matchmaking algorithm, and hence matchmaking is left up to `Peppermint`. Furthermore, the required events and methods were implemented the following way:

newMatchInstance(Object[] args) creates and returns a new match of type `SnakeMatch`;

onCreate(T match) returns a *CreateHandler* that prints a message informing the host the match has been created;

onNewPlayer() returns a handler that has a method that prints a message alerting the remaining players in the match that a new player has joined it;

joinMatch(String matchID, IPlayer player) calls `Peppermint`'s *joinMatch(String matchID, IPlayer player, OnNewPlayerHandler joinHandler)* method, which verifies if the match can be joined and does so if it is and returns the match object, or returns null otherwise, printing the adequate error message;

joinMatchWithMatchMaking(IMatchmaking algorithm, IPlayer player) calls `Peppermint`'s *joinMatchWithMatchmaking(IMatchmaking algorithm, IPlayer player, int numberOfTries, OnNewPlayerHandler joinHandler)* method with a null *algorithm* argument;

Similar to the match `SnakeMatch`, it has a method that stores relevant information to the analysis into a file, that takes a String object as an argument.

Finally, unlike `SnakeMatch` and `SnakeGameManager`, `SnakePlay` doesn't have additional methods.

Our implementation of *Distributed Snake* changes depending on whether it is being used in the simulated environment or in real devices, when it comes to user interactions and play generation.

In the simulated environment, messages are printed on the console, while in the physical devices they are displayed on the device's screen. Furthermore, in the simulated environment, plays are automatically generated, either when after it has been deemed ready to start, and upon the receiving of a *NewTurnEvent* event when it is that player's turn to play, according to the order set by the match during instantiation. During instantiation, the match must define the order by which the next player is determined. To ensure a deterministic order, this value must be "*SEQUENTIAL*" or "*SEQUENTIAL_REVERSE*", depending on whether we want the match to go on ascending or descending order of the order map, respectively. Alternatively, we can choose "*RANDOM*" to randomly determine the next player, which is the player right before or right after the current player in every turn, which in turn will generate non-deterministic results. On the other hand, in physical devices the players themselves decide where to direct the snake to, and as such, a set order is not guaranteed.

Finally, the simulated version has methods in *SnakeMatch* and *SnakeMatchManager* to store relevant results generated during the simulation (i.e., publications and subscriptions of matches, plays and events), to be later analyzed to ensure their correctness.

5.4 The simulated environment

Due to the current pandemic situation, we elected to evaluate the framework's correctness purely on a simulated environment, using an in-house simulator created to evaluate *Thyme*, which was adapted to fit the framework. The *Thyme* and *Basil* code used for the simulations is the same code run by the physical devices, which can have latency values automatically generated by the simulator. The simulator uses traces to emulate the content that is used in the simulations and later analyzed. These traces simulate situations in which hosts create matches, guests attempt to join them, and players attempt to leave or play in them. Additionally, a match can also be cancelled before it reaches its end.

The traces used in these tests were generated by a trace generator implemented during the course of this work. Each trace is generated from a set of given characteristics:

numberOfMatches how many matches are to be created;

averageNumberOfPlayersPerMatch how many players each match has on average;

totalNumberOfPlayers how many players there are among all the matches;

averageNumberOfTurnsPerMatch how many turns a match has on average;

probabilityOfLeavingGame how probable is a player of leaving the match they are in (in a scale of 0-100);

churnAnalysisPeriod analysis of how often players enter and leave the match (in time units).

This information is then used to print a set of commands and their respective arguments, which are as follows:

NODE$|$<time>$|$<nodeId> create a node with ID <*nodeID*> in instance <*time*> (in seconds);

CREATE$|$<time>$|$<nodeId>$|$<match_name>$|$<num_players>$|$<max_turns> node *nodeId* creates match with name <*match_name*> in instance <time> (in seconds), to be played by <num_players> for a maximum of <*max_turns*> turns;

JOIN$|$<time>$|$<nodeId>$|$<match_name> node *nodeId* attempts to join the match with name <*match_name*> in instance <time> (in seconds);

LEAVE$|$<time>$|$<nodeId>$|$<match_name> node *nodeId* attempts to leave the match with name <*match_name*> in instance <*time*> (in seconds);

START$|$<time>$|$<nodeId>$|$<match_name> node *nodeId* attempts to start the match with name <*match_name*> in instance <*time*> (in seconds);

END$|$<time>$|$<nodeId>$|$<match_name> node *nodeId* attempts to end the match with name <*match_name*> in instance <*time*> (in seconds).

Hence, an example of a trace created with these commands and characteristics is presented in Listing 4.

Lines 1 through 4 allow for the creation of the nodes partaking in the simulation, which are created at a given instant (in this case 0) and have an ID associated. Following that, in lines 5 and 6, two matches are created (named *GAME0* and *GAME1*), which have as hosts the nodes with IDs 0 and 2, respectively, two players each and are to be played for thirty turns each.

The remaining fields of the match are instantiated the following way:

staticNumberOfPlayers set to "true";

autoStart set to "false"; The host must run the command *START* to begin the match.

order set to "SEQUENTIAL", the playing order starts at 0 and is incremented until the last player's order, after which it loops back around to 0;

context the context of the simulation; taken from the node's context;

```
1  NODE$|$0.000000$|$0
2  NODE$|$0.000000$|$1
3  NODE$|$0.000000$|$2
4  NODE$|$0.000000$|$3
5  CREATE$|$8.000000$|$0$|$Game0$|$2$|$30
6  CREATE$|$9.000000$|$2$|$Game1$|$2$|$30
7  JOIN$|$10.000000$|$1$|$Game0
8  JOIN$|$11.000000$|$3$|$Game1
9  START$|$12.000000$|$0$|$Game0
10 START$|$13.000000$|$2$|$Game1
11 LEAVE$|$35.000000$|$3$|$Game1
12 JOIN$|$37.000000$|$3$|$Game1
13 LEAVE$|$45.000000$|$3$|$Game1
14 JOIN$|$47.000000$|$3$|$Game1
15 END$|$180.000000$|$0$|$Game0
16 END$|$181.000000$|$2$|$Game1
```

Listing 4: Generated trace used in the simulations

Additionally, the values for the timers are 90 for the timer limiting the entry of new players, and 30 to limit the publishing of new plays.

Then, in lines 7 and 8, the other two nodes join the match, at instants 10 and 11. The method used to join the matches is the "manual" method, that requires a *matchID* to join a match.

Since the match has the required number of players to start (two), when the *START* command is run, they can do so (lines 9 and 10). Additionally, since the matches can only be played with a set number of players, any deviation to this value will impede the match from starting or continuing.

Running the *START* command causes the order map to be generated by the host and published to every other player, while also creating and publishing an event signaling the start of the match, containing the first play to be made by the host.

After this, each turn is automatically generated upon the fetching of a *NewTurnEvent* event. Additionally, since this version of *Distributed Snake* is being played in a one dimensional map, each play's direction is either "LEFT" or "RIGHT" being set to "RIGHT" if the match follows the "SEQUENTIAL" order or "LEFT" if it follows the "SEQUENTIAL_REVERSE" order. However, if it is set to "RANDOM" the set of each play's direction is a mixture of "LEFT" and "RIGHT" values.

During the course of the match, nodes may also decide to leave, which happens in lines 11 and 13, when node 3 leaves match GAME1. In this trace, it also comes back to the match, which is shown in lines 12 and 14.

Finally, if the matches haven't already been ended because they reached their maximum number of turns, they are cancelled by their respective hosts with the use of the

command *END* (which is shown in lines 15 and 16).

The results of these simulations are written in a file, to be later read by a program written that verifies if the the plays are in order according to the order map (for orders "SEQUENTIAL" and "SEQUENTIAL_REVERSE"), how many turns have been played, in what state it ended and how many publications and subscriptions a player has made. All of these criteria are sorted and grouped by the match's name, as can be seen in Listing 5.

5.5 Results and discussion

In this section, we present the results obtained after analyzing the study case implementation and the results derived from running tests on it, in Sections 5.5.1 and 5.5.2 and 5.5.3, respectively.

5.5.1 Framework results

By analyzing the framework's implementation, we can conclude that its implementation is mostly independent from the manner that `Peppermint` manages the matches' states. However, when defining the values to be used by the timers, it should be taken into consideration that the performed actions take some time to be published and disseminated to every player. Hence, the chosen values must not be too limiting, as to not cause unintentional timeouts and interfere with the match state prematurely. This leads us to conclude that the developer needs to be aware of the way `Peppermint` manages the state of its matches, and should carefully consider what values to choose.

Additionally, due to the relatively small number of handlers and methods to implement (four handlers and five methods from class `Match` and two handlers and three methods from class `MatchManager`) we deemed it not to be a very complex implementation, as the performed implementations were rather simple themselves (consisting of a single line or a small number of lines).

5.5.2 Simulation results

The experimental results show that the framework is able to create matches with a set number of players and store them, and that these can be listed and joined by players wanting to play. They also can handle matches being played simultaneously and voluntary player departures and returns to a match. If the match has already started, the entry of a new player or the rejoin of a player that has previously left is correctly handled, and the match continues from the turn where it stopped. Additionally, the match's data is removed from *Basil GardenBed* once it ends or is cancelled.

By inspecting the produced results from the analysis in Listing 5, we can see that when started, the matches are being played in order (indicated by the value "true"on the *next player* criteria, in lines 8 and 33), while the number of turns played varies depending on the trace, but is within the number set during the match's creation (as can be seen in lines

```
Match(es) analyzed:
 --- GAME1
   - Host: C
   - Order: SEQUENTIAL
   - Maximum number of turns: 30
  Keys:
   - 'Next player' (Correctness): true
   - 'Turn': 30
   - 'Subscribe'
    - B (1):
        - match data updates: 1
       - C (8):
        - match data updates: 1 | new turn: 1 | next player updates: 1 | order map updates: 1 |
        ↪  play: 1 | player flow: 1 | player list updates: 1 | status updates: 1
       - D (33):
        - end: 3 | match: 1 | match data updates: 1 | new turn: 3 | next player: 1 | next player
        ↪  updates: 3 | order map: 1 | order map updates: 3 | play: 3 | player flow: 3 | player
        ↪  list: 1 | player list updates: 3 | start: 3 | status: 1 | status updates: 3
   - 'Publish'
       - C (37):
        - match: 1 | match data: 1 | new turn event: 15 | next player: 1 | order map: 1 | play:
        ↪  15 | player list: 1 | start: 1 | status: 1
       - D (35):
        - new turn event: 15 | play: 15 | player flow: 5
 --- GAME0
  - Host: A
  - Order: SEQUENTIAL
  - Maximum number of turns: 30
  Keys:
   - 'Next player' (Correctness): true
   - 'Turn': 30
   - 'Subscribe'
       - A (8):
        - match data updates: 1 | new turn: 1 | next player updates: 1 | order map updates: 1 |
        ↪  play: 1 | player flow: 1 | player list updates: 1 | status updates: 1
    - B (15):
        - end: 1 | match: 1 | match data updates: 1 | new turn: 1 | next player: 1 | next player
        ↪  updates: 1 | order map: 1 | order map updates: 1 | play: 1 | player flow: 1 | player
        ↪  list: 1 | player list updates: 1 | start: 1 | status: 1 | status updates: 1
    - D (1):
        - match data updates: 1
   - 'Publish'
       - A (37):
        - match: 1 | match data: 1 | new turn event: 15 | next player: 1 | order map: 1 | play:
        ↪  15 | player list: 1 | start: 1 | status: 1
    - B (31):
        - new turn event: 15 | play: 15 | player flow: 1

-------------------------------------------------
- Vacancy subscriptions:
 - B: 1 | D: 3
```

Listing 5: Analysis made over the presented trace

Table 5.1: Mapping between actions and subscriptions made during each action

Action	Subscriptions
Creating a match	**4** (**1** subscription to the player flow tag + **3** subscriptions to *CRDT* updates)
Joining a match	Depends on the match's status at the moment of joining and if they have joined or not. If they join when the status is "LOBBY", **at least 10** subscriptions are made (**8** for match state tags and *CRDT* updates + **2** for the VACANCIES tag and respective match data + additional subscriptions per each existing match's current data); if the match has started, there are **6** additional subscriptions. Otherwise, if they re-join the match, they only subscribe to the *CRDT* update tags, DataItems tags and new inserted matches' tags and the vacancies tag, making it **at least 6** and **at least 10** subscriptions, respectively
Leaving a match	**0**
Starting a match	**4** or **6** (match state objects tags and updates to *CRDT*s needed to play the match); it depends on the player (host or guest, respectively)
Playing in a match	**0**
Ending a match	**0**

9 and 34 and comparing them with their match's maximum number of turns indicated in lines 5 and 30, respectively). This can be further confirmed by the status value (if the match ended prematurely, the number of turns is smaller than the number of maximum turns, and the status is "CANCELLED". Otherwise, it is "OVER". As can be seen in lines 7 and 32, both matches were "CANCELLED"). As for the publications and subscriptions, it is possible to check how many of each each player makes of each. As expected, the host makes a bulk of the publications, due to the publishing of the objects that compose the match, with the guests on the other hand doing the bulk of the subscriptions (as can be seen in lines 10-25 and 35-50, and can be seen checked back against the matches' respective hosts in lines 3 and 28). Additionally, there is a section separated from the remaining information regarding subscriptions to the *VACANCIES* tag in lines 53 and 54, due to the fact that these are not tied to a particular match, and hence, do not fit with the remaining data.

Furthermore, while analyzing the output generated by the simulations, we could see that the performed actions would trigger the creation and publication of the correct events, and that they reached every player on the match. We could also verify that the timers set by the host perform as intended when they timeout.

Additionally, for a better understanding of the publications and subscriptions made and how they correlate with the available actions and event handlers, the Tables 5.1 and 5.2 were created.

As examples, we selected a host and a guest to attest if these mappings and the results

Table 5.2: Mapping between actions and publications made during each action

Action	Publications
Creating a match	**4** (**3** match state objects + **1** object in VACANCIES tag)
Joining a match	**1** (*OnNewPlayerEvent* object)
Leaving a match	**1** (*OnPlayerLeavingEvent* object)
Starting a match	**5** (**1** order map + **1** next player object + **1** *StartEvent* object + **1** `Play` object + 1 *NewTurnEvent* object)
Playing in a match	2 (**1** `Play` object + **1** *OnNewTurnEvent* object)
Ending a match	**1** (*MatchEndEvent* object)

derived from the verification program are correct.

If we select host A from Listing 5, we know it created the match, and hence, according to Table 5.1 it made 4 subscriptions to the player flow tag, and *CRDT* updates to the objects it published (which will be discussed later). As the remaining nodes joined and the match became ready to start, it subscribed to the tags regarding plays, next player updates, new turn events and order map updates, to a total of 4 new subscriptions. When adding these values, it is determined that node A made 8 subscriptions.

Additionally, regarding this node's publications, when creating a match, it publishes the match's data, its current data, its player list and status object, for a total of 4 publications. Additionally, after the match gets the required number of players to start and the node runs the command to start it, it makes 5 additional publications. Moreover, because node A is the match's host and the match follows the "SEQUENTIAL" order, it is the first to play, and since the match was played for 8 turns, it plays for 14 additional turns (for a total of 15), it makes 28 more publications. Finally, because the match comes to its end, it makes a publication in the match's end tag. Adding these values, it can be concluded that node A made 37 publications during the match.

Otherwise, if we select guest D from Listing 5, and considering that it joined one of the matches, we can affirm from the information shown in Table 5.1 that it makes 1 subscription to the vacancies tag, and since two matches were created, it subscribes to both matches' current data, so at this point it made 3 subscriptions. Additionally, since it joined match GAME0 before it started, it also subscribes to the tags of the match object, player list and its updates, status and its updates, player flow events and the match's start and end events, which are 8 additional subscriptions, to a total of 11. Since the match could be started, there were 6 more subscriptions to the remaining tags (plays tag, next player object and updates, new turn events tag and order map and updates). Afterwards, node D decided to leave and rejoin the match after it started in two moments, which means it made 20 more subscriptions ((start, end, new turn, player flow, play and VACANCIES tags, next player, status, player list and order map *CRDT* updates) * 2). Hence, in the end, node D made 37 subscriptions, 33 regarding match GAME0, 3 for the VACANCIES tag, and an additional one to the GAME1's current match data. The subscription made to the other match's current data is due to the fact that when listing

the existing matches, every match's current data that is in the VACANCIES tag at the time of retrieval is fetched. While in the simulations this is irrelevant since the match name to join is given in the JOIN command, when using physical devices the device does not have that information when starting the game and the user must choose a match to join or let the matchmaking feature decide for them.

Furthermore, regarding publications, we can check from Table 5.2 that it made 1 publication on the player flow tag when it entered the match, and 2 others for each turn. Since the match GAME0 had a total of 8 turns, the order was "SEQUENTIAL", and node D was the last node to join, it played for 2 turns and made 4 publications. In total, this node made 5 publications.

Both of these analysis can be checked against the results presented in Listing 5, where it can be seen that the referenced objects and values match.

Finally, although there is some support for matches with a variable number of players, this has not been tested, and as such, no conclusions can be drawn from it.

5.5.3 Physical devices results

Unfortunately, given the current world situation, tests in real devices were not conducted, being the evaluation done purely via the simulated environment, where latency and battery depletion values cannot be measured accurately. However, even with these limitations, it is possible to extrapolate data from existing results to our case, to estimate these values and how they could impact gameplay and the playing experience. According to the results in paper [32], in which `Peppermint` is based on, that describes an image sharing application, a Nexus 9 publishing photos gets its battery depleted by 1% after ten minutes. Considering that a publication made by our framework is smaller than a photo, it would take a number of publications more to deplete the battery by 1%, and hence matches could last longer. As for latency values, they should be similar to the values presented by the referred paper, and be kept under 300 milliseconds, which is considered acceptable for applications that require content to be quickly available for all the devices, such as games.

Chapter 6

Conclusions

In this chapter we present the conclusions withdrawn from the elaboration of this thesis and what can be done in the future regarding the betterment of existing features, the addition of new ones and what games can be developed using the framework.

6.1 Conclusions

In this thesis, we present *Peppermint*, a framework that allows for the creation and management of turn-based multiplayer games on mobile edge networks. By addressing the existing technology regarding multiplayer games on the mobile edge, and by analyzing its features and the main features we included in our work, we can conclude that there is nothing quite like we developed. Hence, our work was developed to offer a more complete option to game development than the current frameworks and systems, by handling the features the others do not address.

The framework we developed allows hosts to create and publish matches for others to join, and guests to see what are the matches present in the system at any given time, which they can choose to join, and afterwards leave, or continue in to play. At the moment, matches can be played by a set number of players, given at the time of creation, and a change in that value will change the status of the match. The framework can also handle the cases where other guests join or not the match within a set time limit. Finally, when the match ends, its data is being correctly removed from *Basil GardenBed*.

Additionally, there is also a rudimentary matchmaking feature, that is only able to get a match from a list of randomly chosen matches.

Finally, regarding the planning for this thesis, it was divided in three main phases:

phase one the development of the core set of features (i.e., match creation and management and churn) for matches with a set number of players, and respective study

cases;

phase two the development of dynamic features (e.g., random generation of objects in a match's board in order to dynamically change its difficulty, and assuring those changes reach every player timely in order to keep the justice in the playing conditions) and matchmaking, alongside the adaptation of one of the study cases to use the dynamic features;

phase three testing to evaluate the framework, discover and correct existing issues and optimize the overall functioning of the framework and the developed case studies.

Out of the these, we implemented phase one's core features and a simple matchmaking algorithm, and conducted testing on the framework via simulations, using one case study.

6.2 Future work

In this section, we present the possible betterments to be made to the current work, such as framework improvements in Subsection 6.2.1 or other ideas in Subsection 6.2.2, and suggestions for future implementations using the framework, in Subsection 6.2.3.

6.2.1 Framework improvements

Bettering matchmaking At the moment, the matchmaking feature simply picks a random match from a list of matches for the player to join. In the future, this could be expanded to let developers provide an algorithm that picks a match based on a number of characteristics, such as the player's skill level, duration or number of players, to name a few.

Implementing matches with a varying number of players Although the framework can support matches with dynamic numbers of player, this has not been tested. Hence, further testing should be conducted in order to allow matches with a varying number of players to be created. These could still have a lower and upper limit of players, but a match would not necessarily stop or end after a player exited, being the remaining players able to continue playing.

Ordering players in a match Currently, the players are placed in the ordering map in the same order they're in the player list, making the player order a one dimensional match map, where each player can only reach the players that are positioned right before and right after them (and additionally the host and last player joining are connected to each other). In future work, there could be different algorithms to place the players in the match map, ensuring they are connected to each other and that there are not unreachable players.

Host leaving a match Currently, when the host leaves the match, this causes it to end, as the host controls the start and interruption of timers regarding playing and joining after the match has started. Moreover, they also change the order map whenever there are changes in the player list, so if they left, no other player would update it. This could be changed to delegate a new player as host when the host left, to allow the match to continue without its original host, instead of ending it while the remaining players could still play.

Implementing and testing the study case in real devices The evaluation presented was done for a simulated environment, where latency and battery depletion values cannot be accurately measured. As such, it would be ideal to test the study case in real devices, to measure these values and understand how much they impact the playing experience. However, according to the results in paper [0], which describes an image sharing application, a Nexus 9 publishing photos gets its battery depleted by 1% after ten minutes. Considering that a publication made by our framework is smaller than a photo, it would take a number of publications more to deplete the battery by 1%, and hence matches could last longer. As for latency values, they should be similar to the values presented by the referred paper, and be kept under 300 milliseconds, which is considered acceptable for applications that require content to be quickly available for all the devices, such as games. Additionally, this type of testing would allow us to understand what real users think of the games in terms of usability and enjoyment, and what could be improved in those regards.

6.2.2 Other features

Storing match state after it ends When a match ends, the data produced during its lifespan is simply deleted from *Basil GardenBed*. However, it could be store on a server outside of the system for safekeeping and to develop statistical data about past matches and aid the matchmaking algorithm to choose appropriate matches for future matchmaking attempts.

Random and dynamically generated objects in the drawing board In order to momentarily challenge players, the boards could have dynamically generated objects, spawned at random. For example, if a player touched them, they could be expelled from the match or have their time to play skipped for a number of turns. However, for all players to have a fair chance during the match, there cannot be some players getting these objects and others than do not get them. While challenging, the correct generation of these in-game obstacles could prove to make the game more fun to play.

6.2.3 Games

Being the focus of this thesis a framework to implement turn-based games, there are a number of games that could be implemented. Board games, such as *chess*, *checkers* or *monopoly*, are obvious choices. While *chess* and *checkers* are only to be played by two players, *monopoly* can be playing by more people, and once the match has started, players cannot leave, otherwise the match will reach its end.

Additionally, card games are another option, with games such as *Uno*, *poker* or *sueca*. Again, while *sueca* requires four players to play, both *Uno* and *poker* can be played by a varying amount of players.

Finally, other types of games, such as *Distributed Snake* (which itself is an adaptation of an existing game, *Snake*), can be created or adapted, such as *dominoes* or *tic tac toe*. Although many table games have the potential of being adapted into a mobile game using our framework, there are no limits as to what games can be created within the genre of turn-based games.

Bibliography

[1] J. Afonso. "Key-Value Storage for handling data in mobile devices." Master's thesis. Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa, 2019. Visited on the 30th of November of 2020.

[2] S. Agarwal and J. R. Lorch. "Matchmaking for online games and other latency-sensitive P2P systems." In: *Proceedings of the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Barcelona, Spain, August 16-21, 2009*. Ed. by P. Rodriguez, E. W. Biersack, K. Papagiannaki, and L. Rizzo. ACM, 2009, pp. 315–326. ISBN: 978-1-60558-594-9. DOI: 10.1145/1592568.1592605. URL: https://doi.org/10.1145/1592568.1592605. Visited on the 17th of February of 2020.

[3] A. Barreto, J. A. Silva, H. Paulino, and N. M. Preguiça. "CRDTs em Ambientes Dinâmicos." In: *Simpósio de Informática, INForum*. 2019. Visited on the 20th of February of 2020.

[4] A. R. Bharambe, J. Pang, and S. Seshan. "Colyseus: A Distributed Architecture for Online Multiplayer Games." In: *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings*. Ed. by L. L. Peterson and T. Roscoe. USENIX, 2006. URL: http://www.usenix.org/events/nsdi06/tech/bharambe.html. Visited on the 17th of February of 2020.

[5] P. Bhattacharyya, Y. Jo, K. Jadhav, R. Nath, and J. Hammer. "Brick: A Synchronous Multiplayer Augmented Reality Game for Mobile Phones." In: *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*. Ed. by R. L. Mandryk, S. A. Brewster, M. Hancock, G. Fitzpatrick, A. L. Cox, V. Kostakos, and M. Perry. ACM, 2019. ISBN: 978-1-4503-5971-9. DOI: 10.1145/3290607.3313257. URL: https://doi.org/10.1145/3290607.3313257. Visited on the 20th of January of 2020.

[6] E. Buyukkaya, M. Abdallah, and R. Cavagna. "VoroGame: A Hybrid P2P Architecture for Massively Multiplayer Games." In: *2009 6th IEEE Consumer Communications and Networking Conference*. 2009, pp. 1–5. DOI: 10.1109/CCNC.2009.4784788. Visited on the 17th of February of 2020.

[7] *Catch Pokémon in the Real World with Pokémon GO!* URL: https://www.pokemongo.com/. Visited on the 18th of January of 2020.

[8] F. Cerqueira, J. A. Silva, J. M. Lourenço, and H. Paulino. "Towards a persistent publish/subscribe system for networks of mobile devices." In: *Proceedings of the 2nd Workshop on Middleware for Edge Clouds & Cloudlets, MECC@Middleware 2017, Las Vegas, NV, USA, December 11 - 15, 2017*. ACM, 2017, 2:1–2:6. ISBN: 978-1-4503-5171-3. DOI: 10.1145/3152360.3152362. URL: https://doi.org/10.1145/3152360.3152362. Visited on the 29th of November of 2020.

[9] D. Chu, Z. Zhang, A. Wolman, and N. D. Lane. "Prime: a framework for co-located multi-device apps." In: *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp 2015, Osaka, Japan, September 7-11, 2015*. Ed. by K. Mase, M. Langheinrich, D. Gatica-Perez, H. Gellersen, T. Choudhury, and K. Yatani. ACM, 2015, pp. 203–214. ISBN: 978-1-4503-3574-4. DOI: 10.1145/2750858.2806062. URL: https://doi.org/10.1145/2750858.2806062. Visited on the 28th of January of 2020.

[10] *Churn*. URL: https://www.merriam-webster.com/dictionary/churn. Visited on the 16th of February of 2021.

[11] J. Diephuis, A. Friedl, G. Kostov, P. Piroozan, and D. Wilfinger. "Game Changer: Designing Co-Located Games that Utilize Player Proximity." In: *Proceedings of DiGRA 2015: Diversity of Play* (2015), pp. 1–26. Visited on the 20th of January of 2020.

[12] *DUAL! - Apps on Google Play*. URL: https://play.google.com/store/apps/details?id=com.Seabaa.Dual&hl=en_US. Visited on the 18th of January of 2020.

[13] *Equiti Games Decentralized Games Distribution Platform*. URL: https://equiti.io/. Visited on the 25th of November of 2020.

[14] W. Goddard, J. Garner, and M. M. Jensen. "Designing for social play in co-located mobile games." In: *Proceedings of the Australasian Computer Science Week Multiconference, Canberra, Australia, February 2-5, 2016*. ACM, 2016, p. 68. ISBN: 978-1-4503-4042-7. DOI: 10.1145/2843043.2843476. URL: https://doi.org/10.1145/2843043.2843476. Visited on the 20th of January of 2020.

[15] M. Iqbal. *App Download and Usage Statistics (2019)*. Business of Apps. 2019. URL: https://www.businessofapps.com/data/app-statistics/. Visited on the 05th of February of 2020.

[16] J. Jing, A. Helal, and A. K. Elmagarmid. "Client-Server Computing in Mobile Environments." In: *ACM Comput. Surv.* 31.2 (1999), pp. 117–157. DOI: 10.1145/319806.319814. URL: https://doi.org/10.1145/319806.319814. Visited on the 26th of January of 2020.

[17] J. Manweiler, S. Agarwal, M. Zhang, R. R. Choudhury, and P. Bahl. "Switchboard: a matchmaking system for multiplayer mobile games." In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011*. Ed. by A. K. Agrawala, M. D. Corner, and D. Wetherall. ACM, 2011, pp. 71–84. ISBN: 978-1-4503-0643-0. DOI: 10.1145/1999995.2000003. URL: https://doi.org/10.1145/1999995.2000003. Visited on the 18th of February of 2020.

[18] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief. "A Survey on Mobile Edge Computing: The Communication Perspective." In: *IEEE Commun. Surv. Tutorials* 19.4 (2017), pp. 2322–2358. DOI: 10.1109/COMST.2017.2745201. URL: https://doi.org/10.1109/COMST.2017.2745201. Visited on the 17th of February of 2020.

[19] *NBA JAM by EA SPORTS™ - Apps on Google Play*. URL: https://play.google.com/store/apps/details?id=com.eamobile.nbajam_na_wf&hl=en_US. Visited on the 18th of January of 2020.

[20] Newzoo. *Betting on Billions: Unlocking the Power of Mobile Players*. Mar. 2019. URL: https://cdn2.hubspot.net/hubfs/4963442/Whitepapers/ABM_Newzoo_Betting_on_Billions_Unlocking_Power_of_Mobile_Gamers_March2019.pdf. Visited on the 05th of February of 2020.

[21] *Pokémon GO - Friend List & Friendship Levels*. URL: https://niantic.helpshift.com/a/pokemon-go/?p=web&l=en&s=friends-gifting-trading&f=friend-list-friendship-levels. Visited on the 01st of February of 2020.

[22] N. M. Preguiça. "Conflict-free Replicated Data Types: An Overview." In: *CoRR* abs/1806.10254 (2018). arXiv: 1806.10254. URL: http://arxiv.org/abs/1806.10254. Visited on the 02nd of February of 2020.

[23] N. M. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia. "A Commutative Replicated Data Type for Cooperative Editing." In: *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 2009, pp. 395–403. ISBN: 978-0-7695-3659-0. DOI: 10.1109/ICDCS.2009.20. URL: https://doi.org/10.1109/ICDCS.2009.20. Visited on the 29th of November of 2020.

[24] B. Richerzhagen, M. Schiller, M. Lehn, D. Lapiner, and R. Steinmetz. "Transition-enabled event dissemination for pervasive mobile multiplayer games." In: *16th IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks, WoWMoM 2015, Boston, MA, USA, June 14-17, 2015*. Ed. by L. Bononi, G. Noubir, and V. Manfredi. IEEE Computer Society, 2015, pp. 1–3. DOI: 10.1109/WoWMoM.2015.7158179. URL: https://doi.org/10.1109/WoWMoM.2015.7158179. Visited on the 21th of January of 2020.

[25] S. Rieche, K. Wehrle, M. Fouquet, H. Niedermayer, L. Petrak, and G. Carle. "Peer-to-Peer-Based Infrastructure Support for Massively Multiplayer Online Games." In: *4th IEEE Consumer Communications and Networking Conference, CCNC 2007, Las Vegas, NV, USA, January 11-13, 2007*. IEEE, 2007, pp. 763–767. ISBN: 1-4244-0667-6. DOI: 10.1109/CCNC.2007.155. URL: https://doi.org/10.1109/CCNC.2007.155. Visited on the 17th of February of 2020.

[26] J. Rodrigues, E. R. B. Marques, L. M. B. Lopes, and F. M. A. Silva. "Towards a middleware for mobile edge-cloud applications." In: *Proceedings of the 2nd Workshop on Middleware for Edge Clouds & Cloudlets, MECC@Middleware 2017, Las Vegas, NV, USA, December 11 - 15, 2017*. 2017, 1:1–1:6. DOI: 10.1145/3152360.3152361. URL: https://doi.org/10.1145/3152360.3152361. Visited on the 29th of November of 2020.

[27] R. Rodrigues and P. Druschel. "Peer-to-peer systems." In: *Commun. ACM* 53.10 (2010), pp. 72–82. DOI: 10.1145/1831407.1831427. URL: https://doi.org/10.1145/1831407.1831427. Visited on the 26th of January of 2020.

[28] M. Rouse. *What is cloudlet? - Definition from WhatIs.com*. Nov. 2016. URL: https://searchcloudcomputing.techtarget.com/definition/cloudlet. Visited on the 11th of November of 2020.

[29] J. A. Silva, F. Cerqueira, H. Paulino, J. M. Lourenço, J. Leitão, and N. M. Preguiça. "It's about Thyme: On the design and implementation of a time-aware reactive storage system for pervasive edge computing environments." In: *Future Gener. Comput. Syst.* 118 (2021), pp. 14–36. DOI: 10.1016/j.future.2020.12.008. URL: https://doi.org/10.1016/j.future.2020.12.008. Visited on the 15th of January of 2020.

[30] J. A. Silva, H. Paulino, J. M. Lourenço, J. Leitão, and N. M. Preguiça. "Time-aware reactive storage in wireless edge environments." In: *MobiQuitous 2019, Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Houston, Texas, USA, November 12-14, 2019*. Ed. by H. V. Poor, Z. Han, D. Pompili, Z. Sun, and M. Pan. ACM, 2019, pp. 238–247. ISBN: 978-1-4503-7283-1. DOI: 10.1145/3360774.3360828. URL: https://doi.org/10.1145/3360774.3360828. Visited on the 16th of January of 2020.

[31] J. A. Silva, P. Vieira, and H. Paulino. "Data Storage and Sharing for Mobile Devices in Multi-region Edge Networks." In: *21st IEEE International Symposium on "A World of Wireless, Mobile and Multimedia Networks", WoWMoM 2020, Cork, Ireland, August 31 - September 3, 2020*. IEEE, 2020, pp. 40–49. ISBN: 978-1-7281-7374-0. DOI: 10.1109/WoWMoM49955.2020.00021. URL: https://doi.org/10.1109/WoWMoM49955.2020.00021. Visited on the 30th of November of 2020.

[32] J. A. Silva, F. Cerqueira, H. Paulino, J. M. Lourenço, J. Leitão, and N. Preguiça. "It's about Thyme: On the design and implementation of a time-aware reactive storage system for pervasive edge computing environments." In: *Future Generation Computer Systems* 118 (2021), pp. 14–36. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2020.12.008. URL: https://www.sciencedirect.com/science/article/pii/S0167739X20330703. Visited on the 15th of February of 2021.

[33] A. Teófilo, J. M. Lourenço, and H. Paulino. "RedMesh: A WiFi-Direct Network Formation Algorithm for Large-Scale Scenarios." In: *Proceedings of the 17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. ACM, 2020. Visited on the 30th of November of 2020.

[34] *The World's Leading Blockchain Mobile Gaming Platform*. URL: https://itam.games/. Visited on the 19th of February of 2020.

[35] N. Zhang, Y. Lee, M. Radhakrishnan, and R. K. Balan. "GameOn: p2p Gaming On Public Transport." In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2015, Florence, Italy, May 19-22, 2015*. Ed. by G. Borriello, G. Pau, M. Gruteser, and J. I. Hong. ACM, 2015, pp. 105–119. ISBN: 978-1-4503-3494-5. DOI: 10.1145/2742647.2742660. URL: https://doi.org/10.1145/2742647.2742660. Visited on the 20th of January of 2020.

[36] Q. Zhou, G. Hagemann, S. S. Fels, D. B. Fafard, A. J. Wagemakers, C. Chamberlain, and I. Stavness. "Coglobe: a co-located multi-person FTVR experience." In: *Special Interest Group on Computer Graphics and Interactive Techniques Conference, SIGGRAPH 2018, Vancouver, BC, Canada, August 12-16, 2018, Emerging Technologies*. ACM, 2018, 5:1–5:2. ISBN: 978-1-4503-5810-1. DOI: 10.1145/3214907.3214914. URL: https://doi.org/10.1145/3214907.3214914. Visited on the 20th of January of 2020.