Abstract

Massively multiplayer online role-playing game (MMORPG) is a game genre which allows thousands of people to connect and role-play in a virtual world. Typically, these games are experienced by controlling an in-game character which properties, such as name, race and skills can be customized. Hence, a big part of these games is about progression through quests and combat, because unlike FPS games, MMORPGs allows players to create characters which lasts between different gaming sessions. However, very little is known how these games are implemented in practice.

The objective of this master’s thesis is to study how MMORPG backend can be designed using the microservice architecture pattern? In this context, MMORPG backend means a server-side game system which needs to be in place for thousands of people to connect and role-play in a same or separate instances of a virtual world. On the contrary, an application based on the microservice architecture pattern means a distributed system which is composed by a set of services, each of them designed to perform separate and well-defined task.

The research was conducted with design science research methodology. Consequently, the search problem was approached by building a proof of concept implementation of an MMORPG backend which was developed according to principles of a microservice architecture pattern. These principles along the requirements of the system were derived from the existing literature because the research was not conducted in a company setting. The knowledge base of this research was the existing work towards designing MMORPG architectures, and the studies describing microservice architecture pattern in more detail. The developed system was rigorously evaluated by comparing the design against the principles, and the functionality against the requirements using three different test strategies: unit testing, service testing and end-to-end testing. From the results of the evaluation, and the process of developing the system, the research problem could be addressed.

The results showed that there are multiple steps in the process of designing an MMORPG backend using the microservice architecture pattern, but the requirements engineering is the most important. Having a detailed understanding about the game requirements makes it possible to construct a distributed system in which multiple microservices works together to serve player goals. Hence, the architectural design developed in this research helps game developers to construct their very own MMORPG backend.

Keywords
game networking, microservice architecture, mmorpg

Supervisor
Postdoctoral researcher at University of Oulu, Dr. Arto Lanamäki
Foreword

I would like to take this opportunity to thank following people for helping me in the development of this master’s thesis. First of all, I would like to thank Dr. Arto Lanamäki for his expertise and patience as the thesis supervisor; I think we can both agree that the process was long and painful but also rewarding and fun. Second, I would like to thank my former coworker Thomas O’Rourke for sharing his knowledge regarding microservice-based systems. And finally, I would like to thank my mom for the extra support.

Mauno Vähä

Oulu, November 27, 2017
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AoI</td>
<td>Area of Interest</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CMS</td>
<td>Client-Multi-Server</td>
</tr>
<tr>
<td>CS</td>
<td>Client-Server</td>
</tr>
<tr>
<td>DSL</td>
<td>Digital Subscriber Line</td>
</tr>
<tr>
<td>FPS</td>
<td>First-Person Shooter</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>MMORPG</td>
<td>Massively Multiplayer Online Role-Playing Game</td>
</tr>
<tr>
<td>MMOG</td>
<td>Massively Multiplayer Online Game</td>
</tr>
<tr>
<td>MOG</td>
<td>Multiplayer Online Game</td>
</tr>
<tr>
<td>NPC</td>
<td>Non-Player Character</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>PP-CA</td>
<td>Peer-to-Peer with Central Arbiter</td>
</tr>
<tr>
<td>REST</td>
<td>REpresentation State Transfer</td>
</tr>
<tr>
<td>RoR</td>
<td>Ruby on Rails</td>
</tr>
<tr>
<td>RPG</td>
<td>Role-Playing Game</td>
</tr>
<tr>
<td>RTS</td>
<td>Real-time Strategy</td>
</tr>
<tr>
<td>RTT</td>
<td>Round-Trip Time</td>
</tr>
</tbody>
</table>
## Contents

Abstract ........................................................................................................................................... 2  
Foreword ........................................................................................................................................... 3  
Abbreviations ................................................................................................................................... 4  
Contents ........................................................................................................................................... 5  
1. Introduction .................................................................................................................................. 6  
2. Prior research ............................................................................................................................... 8  
   2.1 The effect of the internet on multiplayer game development .................................................. 8  
      2.1.1 Network bandwidth ........................................................................................................... 9  
      2.1.2 Network latency ................................................................................................................... 9  
      2.1.3 Computational power ......................................................................................................... 10  
   2.2 What is a massively multiplayer online role-playing game? .................................................... 10  
   2.3 Popular multiplayer online game architectures ......................................................................... 11  
      2.3.1 Peer-to-Peer ....................................................................................................................... 12  
      2.3.2 Client-Server ....................................................................................................................... 13  
      2.3.3 Peer-to-Peer with Central Arbiter ..................................................................................... 14  
      2.3.4 Client-Multi-Server ............................................................................................................ 15  
   2.4 Application-level techniques for multiplayer online games .................................................... 16  
      2.4.1 Delta coding ....................................................................................................................... 16  
      2.4.2 Interest management .......................................................................................................... 17  
      2.4.3 Server discovery .................................................................................................................. 19  
   2.5 Microservice architecture vs monolithic architecture ............................................................. 20  
3. Research problem and methodology ............................................................................................. 25  
   3.1 Research method ...................................................................................................................... 25  
   3.2 Developing the artifact ............................................................................................................ 26  
   3.3 Literature review ....................................................................................................................... 27  
4. Requirements engineering .............................................................................................................. 29  
   4.1.1 Selecting high-level requirements from the literature ......................................................... 29  
   4.1.2 Developing the detailed requirements ................................................................................. 30  
   4.1.3 Splitting the detailed requirements to different microservices ........................................ 32  
5. Design ........................................................................................................................................... 34  
   5.1 Architecture overview .............................................................................................................. 34  
   5.2 API gateway ............................................................................................................................. 35  
   5.3 Discovery service ...................................................................................................................... 37  
   5.4 Game website ............................................................................................................................ 38  
   5.5 Account service ......................................................................................................................... 39  
   5.6 Realm service ............................................................................................................................ 40  
   5.7 Zone service .............................................................................................................................. 42  
   5.8 Logger service ............................................................................................................................ 42  
6. Evaluation ..................................................................................................................................... 44  
   6.1 Test strategies and known limitations ...................................................................................... 44  
   6.2 Results of testing the system ..................................................................................................... 45  
      6.2.1 Comparison against the requirements ............................................................................... 45  
      6.2.2 Comparison against the design principles ....................................................................... 50  
7. Discussion ...................................................................................................................................... 52  
8. Conclusions ................................................................................................................................. 55  
References ......................................................................................................................................... 56
1. Introduction

The internet changed the development of multiplayer computer games. In particular, the internet made everything bigger as it allows anyone, from anywhere, to connect and play the same game. Consequently, the internet introduces a wide range of problems which were not present in local area network (LAN) games. For instance, multiplayer online games (MOGs) are affected by cheating (Li, Ding, McCreary & Webb, 2004) and game bots (Gianvecchio, Wu, Xiem & Wang, 2009) because competing against players – other than friends – is tempting for misuses. In addition, usability needs to be looked from a different perspective as more people come along to a play (Pinelle, Wong, Stach & Gutwin, 2009). But most importantly, technology sets resource limitations: network bandwidth, network latency and computational power in which MOGs must operate (Sandeep, 1996, p. 5).

Over the past years the game industry has learned that the resource limitations are the biggest problem in MOG development. The reason is that they affect the development of MOGs from the very first day, whereas others – like game bots – appears only after the game under development gains more popularity. For instance, network latency may cause players to miss their opponents in first-person shooter (FPS) games (Beigbeder et al., 2004) and exploration of a map to be slower in real-time strategy (RTS) games (Sheldon, Girard, Borg, Claypool & Agu, 2003). For this reason, the majority of the existing research within the field focuses on the design of MOG architectures, because having the right design ensures that the game works despite the limits set by the technology (Chen & Maheswaran, 2004). Today, at least four popular MOG architectures can be identified: Peer-to-Peer (P2P), Client-Server (CS), Peer-to-Peer with Central-Arbitrator (PP-CA) and Client-Multi-Server (CMS) (Yang & Sutinrerk, 2007).

The objective of this master’s thesis is to study how MMORPG backend can be designed using the microservice architecture? In this context, MMORPG backend means a server-side game system which needs to be in place for thousands of people to connect and role-play in a virtual world (Chen & Muntz, 2006; Ferrara, Fiumara & Pagano, 2010). On the contrary, an application based on the microservice architecture means a distributed system which is composed by a set of services, each of them designed to perform separate and well-defined task (Newman, 2015).

The research method of this thesis is a design science methodology, and the research is conducted following the design science guidelines of Hevner, March, Park and Ram (2004). Consequently, the research problem is solved by building a proof of concept implementation of an MMORPG backend which is developed according to principles of a microservice architecture pattern. These principles along the requirements of the system are derived from the existing literature because the research is not conducted in a company setting. The knowledge base of this research is the existing work towards designing MMORPG architectures, and the studies describing microservice architecture in more detail. The developed system is rigorously evaluated by comparing the design against the principles, and the functionality against the requirements using three different test strategies: unit testing, service testing and end-to-end testing. From the results of the evaluation, and the process of developing the system, the research problem can be addressed.
Figure 1. A chart representing the popularity of a “microservice architecture” –keyword using the google search trends with 01.01.2014 – 01.01.2017 as a date range.

The motivation for the study comes from the fact that using microservice architecture pattern in a design of an MMORPG backend has not been previously studied. At the same time, the interest towards modelling systems using microservices has been increasing in a rapid fashion (see Fig. 1). The reason is twofold. First, some of the big companies – such as Netflix and Amazon – adopted the new style with good results (Newman, 2015). Second, using the architecture is said to offer multiple benefits for software companies, such as ease of systems scalability, deployment and possibility to form distributed teams which are responsible for developing individual microservices (Fowler & Lewis, 2014; Newman, 2015; Taibi, Lenarduzzi, Pahl & Janes, 2017). Consequently, studying microservices in the context of MMORPGs makes an interesting research topic.

The results shows that there are multiple steps in the process of designing an MMORPG backend using the microservice architecture pattern, but the requirements engineering is the most important. Having a detailed understanding about the game requirements makes it possible to construct a distributed system in which multiple microservices works together to serve player goals. Hence, the architectural design developed in this research helps game developers to construct their very own MMORPG backend.

The rest of this thesis is organized as follows: Chapter 2 describes the existing work towards designing MMORPG architectures and discusses microservice architecture in more detail. Chapter 3 presents the research method used in this thesis. Chapter 4 describes the process in which the requirements of a proof of concept implementation of an MMORPG backend were formed. Chapter 5 presents the design of the proof of concept implementation. Chapter 6 shows the results of evaluating the system. Chapter 7 answers to the research question of this thesis, followed by Chapter 8 to conclude the thesis with an indication on what work remains to be studied.
2. Prior research

This chapter is reserved for describing the existing work towards designing MMORPG architectures. The chapter begins with a short introduction to a problem space by explaining the effect of the internet on multiplayer game development. Next, some time is spent to define the MMORPG genre so that mutual understanding about the common features of the games could be formed. After that, the focus shifts to a solution space by giving a concise overview of popular MOG architectures and application-level techniques used in online games. Finally, the chapter concludes with an introduction to a microservice architecture pattern which has potential to support the development of MMORPGs.

2.1 The effect of the internet on multiplayer game development

The internet changed the development of multiplayer games. In particular, the internet made everything bigger as it allows anyone, from anywhere, to connect and play the same game. Consequently, the internet introduces a wide range of problems which were not present in LAN games. For instance, MOGs are affected by cheating (Li et al., 2004) and game bots (Gianvecchio et al., 2009) because competing against players – other than friends – is tempting for misuses. Also, usability needs to be looked from a different perspective as more people come along to a play (Pinelle et al., 2009). But most importantly, technology sets resource limitations: network bandwidth, network latency and computational power in which MOGs must operate (Sandeep, 1996, p. 5).

Over the past years the game industry has learned that the resource limitations are the biggest problem in MOG development. The reason is that they affect the development of MOGs from the very first day, whereas others – like game bots – appears only after the game under development gains more popularity. For instance, according to studies of Claypool, M. and Claypool, K. (2006; 2010), there is a measurable limit on how much network latency players are able to tolerate before the user experience of a MOG becomes too poor to be playable. The authors conclude that FPS/racing games may tolerate latencies around 100ms, role-playing games (RPGs) 500ms and RTS 1000ms. However, the actual delay tolerance of an individual MOGs needs to be tested as it is a sum of game genre and type of player actions. (Claypool, M. & Claypool, K., 2006; 2010.)

Maybe for these reasons, network latency, sometimes referred as ping, is often seen as the root cause for delays in MOGs. In truth, the actual delay experienced by the player is further increased because calculating the game state takes central processing unit (CPU) time (Brun, Safaei & Boustead, 2006). And there is a possibility that a large number of players exceeds the maximum connection capacity of a network (Kim et al., 2005). As a minimum, developers need to be aware that these limitations exist, otherwise it is impossible to make a MOG which meets the expectations of the players. For this reason, resource limitations are explored briefly.
2.1.1 Network bandwidth

Network bandwidth can be defined as a speed of transferring data over a network connection from one destination to another (Smed, Kaukoranta & Hakonen, 2002). In the context of MOGs, transferred data is the events occurring in the game. For instance, shooting a weapon in a FPS game creates an input event and calculating the result of a hit creates an updated game state. These states need to be transferred to the players so that everyone can see the progress of the game. (Chen & Maheswaran, 2004.) Hence, one could argue that the same principle applies to games of all kind. A player is a producer of inputs and a game is a producer outputs according to pre-defined game rules.

Today, players are employed with relatively fast internet connections. For instance, the author of this thesis is using a digital subscriber line (DSL) connection with a sending and receiving speed of 100/10 Mbps. On the contrary, back in 1990s, developing MOGs were remarkably much harder as players owned a so-called dial-up connection with a transferring speed of 56 Kbps (Pellegrino & Dovrolis, 2003). However, network bandwidth continues to be a resource limitation which is reached sooner or later. Mostly because MOGs are growing larger than they used to be. In fact, these games are growing so large that a term massively multiplayer today means very different size of game than those of the past. For example, back in 2000s, the author of this thesis used to play a PC game known as Dransik which claimed to be a MMORPG with a maximum player limit of ~200. By comparison, another MMORPG, known as World of Warcraft has over 11.5 million players worldwide (Pace, Bardzell, S., & Bardzell, J., 2010). Consequently, World of Warcraft is played by thousands – rather than hundreds – of simultaneous players and that increases bandwidth usage significantly (Kim et al., 2005).

Interestingly, the size of MMORPGs is not the only thing which has changed in recent years. Advances in the field of mobile phones makes it possible to develop MOGs for multiple platforms. For instance, Parallel Kingdom is a mobile MMORPG which has over 600,000 players (Patro, Rayanchu, Griepentrog, Ma & Banerjee, 2012). The finding is remarkable because mobile games are played over a wireless network which speed is typically slower than the one achieved with DSL. According to Soh and Tan (2008), maximum capacity of a wireless connection ranges from 2 to 100 Mbps.

2.1.2 Network latency

Network latency can be defined as a round-trip time (RTT) which it takes to a message to travel over a network connection from a sender to a receiver, and back (see Fig. 2) (Pantel & Wolf, 2002). The truth is that advances in technology cannot remove the network latency. The reason is that the latency is caused by physical constraints – like speed of light – and qualities of network cables (Smed et al., 2002). In addition, the delay is further increased as messages needs to be routed through the internet (Sandeep, 1996). The routing is especially tricky process because it may cause amount of latency to change (called as jitter) between individual transmissions (Beigbeder et al., 2004). Hence, there is also a chance that a message gets lost upon transmission. In these cases, the jitter is much higher if underlying transport protocol or a MOG requires retransmission of lost packets. (Brosh, Baset, Misra, Rubenstein & Schulzarinne, 2010.)
Figure 2. Network latency can be defined as a round-trip time which it takes to a message to travel over a network connection from a sender to a receiver, and back.

All things considered, network latency is a big problem for developers of MOGs because it affects the user experience. For example, network latency may cause players to miss their opponents in FPS games (Beigbeder et al., 2004) and exploration of a map to be slower in RTS games (Sheldon et al., 2003). In other words, even a professional player can be much worse on the internet if the latency exceeds a maximum tolerance of the game (Claypool, M. & Claypool, K., 2006; 2010). Also, there is a chance that MOG favors players with lowest latencies. For example, an unskilled player with low latency could win a professional player with high latency if her network messages travel faster through the internet. For these reasons, a wide range of application-level techniques have been invented to improve playability of MOGs. Hence, some of these techniques are explored briefly during Chapter 2.4.

2.1.3 Computational power

Computational power can be defined as a speed of a computer to perform given operations (Sandeep, 1996, pp. 7-8). In the context of MOGs, performant CPUs are needed for processing different parts of a game simulation. For example, a player needs to have a computer which is powerful enough to display the game on the screen. On the contrary, a computer acting as a game server needs to be powerful enough to calculate a game state and distribute it to the players in the network. (Chen & Maheswaran, 2004.) Unfortunately, as the number of players in a game increases, so does the demand for computational power and network bandwidth. According to Smed et al. (2002) interpretation of Sandeep (1996) “...simulation handling the network traffic of 100,000 entities (i.e., participating objects) can require up to 80 percent of the total CPU time on a 500 MHz processor.” (chap. 2) Hence, network traffic is not the only activity using CPU time. Namely, players interact with other and objects of a virtual world which may involve calculations of complex physics (Brun et al., 2006). For these reasons, designing an architecture for MMORPGs – such as World of Warcraft – is not an easy task; multiple computers need to share the load of processing the game and return the results in a timely manner.

2.2 What is a massively multiplayer online role-playing game?

MMORPG is a game genre which allows thousands of people to connect and role-play in a same or separate instances of a virtual world (Chen & Muntz, 2006; Ferrara et al., 2010). Typically, these games are experienced by controlling an in-game character which properties, such as name, race and skills can be customized. Hence, a big part of these games is about progression through quests and combat, because unlike FPS games, MMORPGs allows players to create characters which lasts between different gaming sessions (Achterbosch, Pierce & Simmon, 2008). For this reason, it is not surprising that many MMORPGs – such as Diablo 3 – provides permanent in-game stash for storing of goods and valuable items. Very often, however, according to Suznjevic, Dobrijevic and Matijasevic (2009), valuable items cannot be obtained without communicating with other players and forming of alliances. The reason is that
the monsters with best loots are often impossible to defeat alone and accessible only through special events, such as raids (see Fig. 3). (Suznjevic et al., 2009.) Consequently, a need for in-game communication between players have risen interesting studies over the years, like analysis of intimate experiences during an online play (Pace et al., 2010).

Figure 3. A screenshot from a popular MMORPG known as World of Warcraft. Here, a raid group is preparing to challenge a boss Ragnaros.

Today, there are probably more than hundreds of MMORPGs available, and new ones are being developed from time to time. Yet, it is very unlikely that big commercial hits like Ultima Online during the 1990s (Achterbosch et al., 2008) or World of Warcraft during the 2000s (Pace et al., 2010) are being developed anytime soon. However, there is an interesting shift towards making MMORPGs which designs are much more versatile than some of the older games. For instance, Eve Online is a MMORPG which allows players to build their own ships and explore a galaxy consisting of more than 7000 star systems (Eve Online, 2003). As a consequence, architectural requirements of Eve Online can be very much different than the likes of Ultima Online or even World of Warcraft, because the whole game takes place in a single-universe setting. But what is more interesting, is the fact that the entire game world is run using a single server cluster (Eve Online, 2003). On the contrary, World of Warcraft uses multiple servers, known as realms, and maintains its own copy of a game world on each of them. This means that the game might be easier to scale, but players residing at different realms cannot see each other in the game. (World of Warcraft, 2004.) For these reasons, one could argue that high-level requirements of MMORPGs should be defined early in the development process. Otherwise, it is very unlikely that a designed MMORPG architecture is able to meet the expectations of the designers nor the players.

2.3 Popular multiplayer online game architectures

The majority of the existing research within the field focuses on designing MOG architectures. In this thesis, MOG architecture refers to a fundamental structure of a game system which needs to be in place for players to interact in a same virtual world (Chen & Maheswaran, 2004). Today, at least four popular MOG architectures can be identified: P2P, CS, PP-CA and CMS (Yang & Sutinrerk, 2007).
In this chapter a concise overview of popular MOG architectures (see Fig. 4) is given. The concise overview means that the architectures are reviewed only with a level of necessary within the context of this thesis. Hence, the architectures are listed in no particular order of popularity, because the existing research is lacking the required information for doing so. The prior work also fails to make a clear statement on what is the best architecture for MMORPGs. Maybe because making such conclusion is theoretically impossible; multiple factors, such as scalability (Bauer, Iliadis, Rooney & Scotton, 2004), game requirements (Claypool, M. & Claypool, K., 2006; 2010), vulnerability to cheating (Knutsson, Lu, Xu & Hopkins, 2004; Yang & Sutinrerk, 2007), cost (Hampel, Bopp & Hinn, 2006) and even personal preferences affects the decision. Nevertheless, the author of this thesis attempts to review the architectures from the viewpoint of MMORPGs.

### 2.3.1 Peer-to-Peer

According to Smed et al. (2002), P2P is a non-authoritative MOG architecture in which player’s computer, called as peer, is directly connected to all the other peers in the network (see Fig. 4a). For this reason, the game state management is split between the players connected to the game rather than using a centralized server for doing the calculations. (Smed et al., 2002.) Hence, at least two variations of the architecture exist; every peer calculates the game state based on all the events happening in the game (Bauer et al., 2004; Yang & Sutinrerk, 2007) or using only subset of the events relevant to a player location in the game world (Bauer et al., 2004). Bauer et al. continues that the former approach is useful for small MOGs, whereas the latter needs to be used within a game which world is large enough to form separate groups of interaction (Bauer et al., 2004). That being said, many MMORPGs could benefit from the ideology of the latter approach.

All things considered, P2P model as described above is not an easy architecture to work with. First of all, according to Bettner and Terrano (2001), the solution requires that all peers must progress the game completely in sync. For this to work, developers need to implement a relatively complex game loop which is able to delay execution of events on all participants whenever some peer in the network cannot keep up with the game. In addition, the game needs to be designed such that given a set of input events every peer calculates the same game state. (Bettner & Terrano, 2001.) Another problem with P2P architecture is the fact that lack of centralized server increases the opportunities for cheating (Knutsson et al., 2004; Yang & Sutinrerk, 2007). The finding is interesting because Hampel et al. (2006) argues that P2P architecture is cheaper to maintain. However, the costs of cheat prevention should be considered as players are not going to play MOGs which is crowded with cheaters (Li et al., 2004).
From the viewpoint of resource limitations, P2P architecture is argued to have a low latency as the messages are transferred directly between the peers (Jardine & Zappala, 2008; Yahyavi & Kemme, 2013). But the author of this thesis reminds that P2P architecture alone cannot guarantee low latency as the physical distance between the players could still be thousands of kilometers. On the contrary, P2P model does have a potential on sharing network bandwidth and computational power usage between the peers. All in all, the P2P model is interesting solution and it is used successfully in RTS games, such as *Age of Empires* (Bettner & Terrano, 2001).

### 2.3.2 Client-Server

According to Pellegrino and Dovrolis (2003), CS is an authoritative MOG architecture in which player’s computer, called as *client*, is connected to a centralized computer, called as the *server* (see Fig. 4b). In this model, the server sets rules which clients’ needs to obey. In other words, the game state management is centralized to the server rather than using multiple clients – as in P2P model – to negotiate the progress of the game. By doing this, CS architecture achieves separation of concerns. A client is responsible for sending input events to the server whereas the server is responsible for calculating the game state and returning it back to the clients. (Pellegrino & Dovrolis, 2003.) The model of networking in CS is a security-wise solution. Because unlike P2P, the centralized server can either accept or deny malicious inputs of the players. For this reason, CS model is considered a better choice for the development of commercial MOGs (Jardine & Zappala, 2008; Kabus, Terpstra, Cilia & Buchmann, 2005; Pellegrino & Dovrolis, 2003). Hence, due to historical reasons, at least two variations of the architecture exist.

The first option with CS model is to implement the game client using a so-called *dumb client* approach. The dumb client means that the client cannot render the results of input events without receiving calculated states from the server. For example, a player with 300ms RTT to the server needs to wait 300ms between pressing a button and seeing the bullet. (Bernier, 2001.) For this reason, the dumb client approach is not very useful for MOGs which are sensitive to delays. Hence, FPS games in particular needs to respond within 100ms or the delay starts to affect the user experience of the game (Claypool, M. & Claypool, K., 2006; 2010).

According to Bernier (2001) the second option with CS model is to implement the game client using a so-called *client-side prediction* approach. A client-side prediction increases responsiveness of a MOG by predicting the results of input events ahead of the state calculated by the server (see Fig. 5). In more detail, client-side prediction works by implementing a game client which stores player inputs in-memory as well as sends them to the server. Hence, while the server is calculating the correct state of the game, the client predicts the result locally and updates the screen. Later on, when the server replies with correct state of the game, the client accepts the state, discards the input event which caused the state change, and predicts the current state by looping through rest of the in-memory events. The client-side prediction is interesting solution for MOGs because it trades consistency for responsiveness. Thus, at best, accepting states sent by the server is completely invisible to the player. At worst, frequent simulation differences between a client and the server causes a player to experience jumpiness. For this reason, client-side prediction should only be used with games which are deterministic enough to predict results of input events. By doing this, jumpiness is experienced only by the players which are trying to cheat, because the server remains authoritative over the game state calculation. (Bernier, 2001).
Figure 5. A client-side prediction increases responsiveness of a MOG by predicting the results of input events ahead of the state calculated by the server.

From the viewpoint of resource limitations, CS architecture is argued to have higher network latency than P2P, because seeing the results of input events takes full RTT (Smed et al, 2002). In addition, the architecture requires significant amount of network bandwidth and computational power from the server, but not from the clients (Cai, Xavier, Turner & Lee, 2002; Pellegrino & Dovrolis, 2003). That being said, CS is an architecture which scalability is limited to the processing power of a single server. For the same reason, the crash of the server causes every player to be dropped out of the game (Bharambe, Rao & Seshan, 2002). Nevertheless, CS model is a good solution for MOGs and it is used successfully in FPS games, such as QuakeWorld and Counter-Strike (Bernier, 2001).

2.3.3 Peer-to-Peer with Central Arbiter

PP-CA is a P2P architecture with improved security. The architecture improves the security by employing a centralized computer, called as arbiter, for detecting malicious inputs of the players. For this to work, peers need to send input events to each other – as in P2P – but also to the arbiter (see Fig. 4c). The role of the arbiter is to remain silent as long as everyone is playing by the rules, but once the inconsistency is detected, the arbiter repairs the game state and delivers it to the peers. (Yang & Sutinrerkerk, 2007.)

The PP-CA model as described above is interesting because it attempts to combine the best parts of P2P and CS architectures. However, the author of this thesis questions whether PP-CA architecture is as secure as it is intended to be. The reason is that existing studies (Pellegrino & Dovrolis, 2003; Yang & Sutinrerkerk, 2007) does not make very clear what would happen when some player in the network attempts to cheat by sending different inputs to the peers than those to the arbiter. In theory, this could result into a situation in which non-cheating players starts seeing inconsistent game states. In terms of FPS game, the cheater could send move right –event to the arbiter and move left –event to the peers, and therefore making herself impossible to hit. The author of this thesis argues that the problem could be partially circulated by making every peer to deliver inputs of all the players to the arbiter. By doing this, the arbiter could detect the inconsistencies by incorporating the inputs transferred between the peers to the game state calculation. Unfortunately, this would also require more upstream bandwidth from the peers. Hence, the problem of cheating would still exist because several players could group up and cheat together by modifying inputs of non-cheating player to be invalid in
the messages delivered to the arbiter. For these reasons, the author of this thesis concludes that PP-CA architecture is not a good choice for the development of commercial MOGs.

From the viewpoint of resource limitations, according to Pellegrino and Dovrolis (2003), PP-CA architecture has similar network bandwidth and computational power requirements than P2P. However, the overall bandwidth usage of the player is also in a relationship with the number of inconsistencies in the game. Compared to CS architecture, PP-CA requires less network bandwidth from the server whereas the computational power usage is close to the same. (Pellegrino & Dovrolis, 2003.) However, as already discussed, there is a possibility that PP-CA model is vulnerable to cheating, because the arbiter cannot validate the inputs transferred directly between the peers. The problem could be avoided by making the arbiter to send game states in regular intervals – as in CS architecture, but then developers would be better off using the CS architecture from the beginning.

### 2.3.4 Client-Multi-Server

CMS is a CS architecture with improved scalability. The architecture improves the scalability by using multiple servers for processing the events of the game (see Fig. 4d). In other words, a single game server is no longer responsible for more than a subset of the whole player population. (Yang & Sutinrerk, 2007.) By doing this, CMS architecture also circulates the problem of single point of failure because the crash of an individual server does not affect all players of the game (Bharambe et al., 2002). At the time of writing, at least two variations of the architecture exist.

According to Yahyavi and Kemme (2013), the first option with CMS model is to employ multiple servers which run independently from each other by having their own copy of a game world. The solution works similarly as CS architecture with only difference being the number of servers available to the players. That being said, the game needs to provide a mechanism for selecting a game server. The selection could be handled manually as in Day of Defeat: Source or automatically based on the geographical location of the player. All in all, the independent worlds approach is trivial to implement because server-to-server communication is not required. But that also means that the players residing at different servers cannot interact in the game. (Yahyavi & Kemme, 2013.)

The second option with CMS model is to split a game world into multiple zones which are distributed to different servers. (Yahyavi & Kemme, 2013; Yang & Sutinrerk, 2007). The solution allows players to interact in a same game world but transferring player from one zone to another is more complex to implement. The reason is that two or more servers needs to negotiate the change which should be invisible to the players. (Yahyavi & Kemme, 2013.) In some games – like MMORPGs – hiding the player movement between zones is easy because in-game portals can be used for connecting the zones (Chen et al., 2004). In others – like continuous world games – portals cannot be used as they break the flow of the game; a player residing close to a zone border should be able to interact with more than one zone at the same time (Yahyavi & Kemme, 2013). Making such system is difficult, because the implementation might require more sophisticated techniques, such as acquiring region and object locks (Assiotis & Tzanov, 2006).

From the viewpoint of resource limitations, CMS increases scalability of CS architecture by using multiple servers. Namely, the main problem with CS architecture is the fact that a single server cannot support more than 2000 – 6000 concurrent players.
(Knutsson et al., 2004). In reality, the actual performance can be worse because it varies between the game types (Yahyavi & Kemme, 2013). In any case, CMS architecture should only be used when the game under development requires high level of scalability. The reason is that employing servers is rather expensive. According to Kabus et al. (2005), costs of supporting 30,000 simultaneous players is over 800,000$. In conclusion, CMS is a good architecture and it is often used in MMORPGs, such as Ultima Online, EverQuest (Fiedler, Wallner & Weber, 2002) and World of Warcraft (World of Warcraft, 2004).

2.4 Application-level techniques for multiplayer online games

The findings of previous chapters indicate that MOG architecture sets rules on how a game taking place between remote players needs to operate. Hence, these rules must be reviewed on a game by game basis because different games require different properties from the underlying architecture. Also, there is the aspect of money which needs to be considered when choosing an architecture for a particular game. The reason is that some architectures are more expensive to maintain than the others. For example, CMS architecture requires that someone pays the costs of employing multiple servers whereas P2P relies solely on the processing capacity of players’ computers.

Nevertheless, according to Smed et al. (2002) there is a chance that choosing a MOG architecture is not enough to meet the requirements of a given game design. The reason is that an architecture helps to reduce resource requirements of a game but cannot get rid of them completely. For this reason, additional application-level techniques are needed for mitigating resource usage further, and increasing responsiveness and fairness of the game. (Smed et al., 2002.) That being said, the game is considered fair when all players have similar chance of winning, and responsive when the time between pressing a button and seeing the result does not frustrate the player (Brun et al., 2006). Naturally, complying these properties in MOGs is difficult because messages affecting the game state needs to be transferred over the internet (Chen & Maheswaran, 2004).

In this chapter application-level techniques relevant for MOG developers are being reviewed. These techniques can be roughly split between those improving playability and those reducing resource usage. Hence, within the context of this thesis the interest is towards the latter because they are more closely related to a design of MMORPG backend. In other words, techniques intended for improving playability is often employed at client-side rather than at server-side. For this reason, application-level techniques, such as local lag (Liang & Boustead, 2006; Malik Khan, Chabridon & Beugnard, 2008; Mauve, 2000; Stuckel & Gutwin, 2008; Yahyavi & Kemme, 2013), dead-reckoning (Brun et al., 2006; Malik Khan et al., 2008; Mauve, 2000; Sandeep, 1996; Smed et al., 2002; Zhang, Chen, L. & Chen, G., 2006), entity interpolation and lag compensation (Bernier, 2001; Source Multiplayer Networking, 2005) were recognized, but studying them in more detail is left as an exercise for the reader.

2.4.1 Delta coding

According to Glinka, Ploß, Müller and Gorlatch (2007), a delta coding is an application-level technique for mitigating network bandwidth usage by implementing a game server which sends value differences rather than whole states of in-game entities to interested receivers. In more detail, the solution works under an assumption that once a player has received initial state of an entity, such as Non-Player Character (NPC), the subsequent updates from the server does not need to carry the whole state information. The reason is twofold. First, the game client can be implemented as such that whole state of entities
are kept in-memory and their properties updated individually. Second, the behavior of entities is tied to rules of the game, and therefore the whole state of an entity rarely changes between current and next update frame. (Glinka et al., 2007.)

For this to work, Bharambe, Padmanabhan and Seshan (2004) continues that the server needs to keep a track of previously delivered game states to the clients. The reason is that once the server calculates a new game state, the state of entities can be compared against the values known by the clients. The result of this comparison is a so-called *delta update* which contains only the value differences of entities between the current and last update frame. Hence, the server needs to deliver this information to the clients, so that the game simulation running on players’ computers can be updated to reflect the current state of the server. (Bharambe et al., 2004.) Figure 6 demonstrates using of delta coding by modelling a behavior of NPC.

**Figure 6.** Example of delta coding. Here, an NPC is created to a position \( x = 5, y = 0 \) and then moved to a position \( x = 7, y = 0 \) during the course of the game.

All things considered, using delta coding in a FPS game, such as *Quake III*, mitigates network bandwidth usage significantly. The results show that using the technique in a game with 5 bots drops bandwidth usage from 3 Mbps close to 40 Kbps. (Bharambe et al., 2004.) Interestingly, however, applying delta coding to other type of games has not been studied a lot. For instance, Glinka et al. (2007) mentions that using delta coding is useful in the context of massively multiplayer online games (MMOGs) as well, but concrete results of applying the technique were not provided.

### 2.4.2 Interest management

According to Yahyavi and Kemme (2013), an *interest management* is an application-level technique for mitigating network bandwidth usage by sending players only information relevant to their location in the game world. In more detail, the technique works under an assumption that the only information relevant for the players is the one they can see or interact with. For example, current position of an NPC moving back and forth in the game is interesting only for the players which are close enough to see the NPC. Similarly, current state of an in-game object, such as treasure chest, is interesting only for the players which are close enough to open the chest. For this reason, MOGs containing a large game world – such as MMORPGs – benefits from the technique as
only fraction of the game state needs to be delivered to the players. Consequently, interest management also improves the security of the game because the players are given only the information they need to know. (Yahyavi & Kemme, 2013.)

Boulanger, Kienzle and Verbrugge (2006) explains that the interest management can be implemented using a so-called aura-nimbus model. In this solution, each player has an aura and nimbus which typically take the form of a fixed size sphere around or close to the location of the player. The aura means the area the player is exposing to others, and nimbus the Area of Interest (AoI) the player is receiving updates from. In its simplest model, aura and nimbus are equal and placed around the player. Thus, delivering a game state update to a player is simply a process of checking which auras of objects resides within the AoI of the player, and selecting the properties of those objects accordingly (see Fig. 7a). (Boulanger et al., 2006.)

![Diagram](image)

**Figure 7.** Interest management mitigates network bandwidth usage by sending players only the state of objects and entities within their area of interest.

By comparison, Boulanger et al. (2006) continues that another option for interest management is to use so-called zone-based model. In this solution, game world is split into multiple zones and players are given a fixed size sphere similar to nimbus for expressing the interest to certain area. But unlike before, the final AoI of the player is calculated by checking which zones the sphere is currently overlapping and subscribing to those accordingly. (Boulanger et al., 2006.) In its simplest model, players cannot listen more than one zone at a time because the zones are separated from each other and moving between them requires using of in-game portals (Knutsson et al., 2004; Yang & Sutinrerk, 2007). However, in some games – like continuous world games – portals cannot be used as they break the flow of the game; a player residing close to a zone border should be able to interact with more than one zone at the same time (Yahyavi & Kemme, 2013). In these cases, the zones typically take the form of a square or hexagon because of the low computational overhead and number of zones the player needs to listen at once (Boulanger et al., 2006; Yahyavi & Kemme, 2013). Hence, in order to limit network bandwidth usage, the zones should not be much larger than the view of the player (see Fig. 7b) (Fiedler et al., 2002).

In any case, the author of this concludes that interest management is a broad topic and more deeper explanations of all the available techniques falls outside the scope of this thesis. For instance, according to Boulanger et al. (2006), interest management is further complicated by visibility-based algorithms, such as ray visibility, in which an accurate AoI of the player is determined by drawing lines between the player and objects nearby.
By doing this, the solution achieves more fine-grained interest management as the state of an object residing nearby player, but behind a wall, does not need to be transferred. Consequently, calculating an accurate AoI results into using more computational power. (Boulanger et al., 2006.) That being said, selecting an interest management technique for a game is always a trade-off between computational power and network bandwidth.

In conclusion, interest management is an important technique for MMORPGs, but regardless of the used solution the problem of so-called hotspots still exists. Yahyavi and Kemme (2013) defines that hotspots are places – such as towns – in which large group of players likes to gather at the same time. Consequently, players may end up receiving more information from the server than their network connection is able to handle. For this reason, the development team needs to make sure that important resources of the game are placed to a map in a decentralized manner. Otherwise, players continuously grouping up to certain areas of the game mitigates the usefulness of interest management. (Yahyavi & Kemme, 2013.) On the other hand, more sophisticated interest management techniques also exist, such as locality aware load balancing (Chen et al., 2005) and dynamic region splitting (Yahyavi & Kemme, 2013).

2.4.3 Server discovery

A server discovery is an application-level technique for mitigating network latency by finding an optimal game server to a player or group of players participating to the game. Therefore, the technique can be used only with architectures employing multiple servers. (Gargolinski, St. Pierre & Claypool, 2005.) Hence, in order for this to work properly, game providers need to make sure that the servers are placed geographically apart from each other. Otherwise, finding a game server with low RTT to players coming from different parts of the world would not be possible. (Feng, Wu-cha. & Feng, Wu-chi, 2003.) That being said, developing a system which transfers players automatically to closest servers is trivial to implement. In reality, however, the game needs to give some control to the players because they prefer to select a game server based on multiple criteria, such as RTT, server map type and number of players (Gargolinski et al., 2005; Zander, Kennedy & Armitage, 2005).

As of today, server discovery is typically implemented using a so-called master servers approach (Gargolinski et al., 2005; Henderson, 2002; Zander et al., 2005). In this solution, a player selects a game server manually using a client-side program, such as server browser, which shows details of all the active game servers (Gargolinski et al., 2005). For this to work, according to Henderson (2002), the solution requires that the game provider starts hosting separate servers, called as master servers, which purpose is to maintain an up-to-date list of game server addresses. These addresses are received from the actual game servers during a so-called start-up procedure, because the master servers cannot locate the servers from the internet. In addition, every game server provides an application programming interface (API) for returning details about the ongoing game, such as server map type and number of players. By doing this, the role of a server browser is to receive game server addresses from master servers and query them individually to present necessary server details to the players (see Fig. 8). (Henderson, 2002.)
Figure 8. A screenshot of a server browser which is used in several MOGs, such as Day of Defeat, Half-Life and Left 4 Dead.

In any case, the author of this thesis concludes that server discovery is a broad topic and more deeper explanations of all the available techniques falls outside the scope of this thesis. The reason is twofold. First, master servers as described above already explains the purpose of server discovery for the reader, and the technique is proved to work in many MOGs, such as Half-Life, Quake III and Unreal Tournament (Henderson, 2002). Second, server discovery is further complicated by algorithms for selecting a game server for multiple players in a form of matchmaking (Agarwal & Lorch, 2009). Hence, matchmaking in particular is not a common feature to see in MMORPGs.

2.5 Microservice architecture vs monolithic architecture

The microservice architecture pattern is a new approach for building modern web applications, such as web services and APIs. As opposed to monolithic architecture, in which the application is build using a single codebase and deployed as a whole, the application based on the microservice architecture is composed by a set of services, each of them designed to perform separate and well-defined task. The architecture caught the eye of the software industry during 2014 when Netflix and Amazon adopted the new style with good results. (Newman, 2015.) Since then, the interest towards modelling systems using microservices has been increasing in a rapid fashion (see Fig. 1).

According to Fowler and Lewis (2014), the basic idea of the microservice architecture is to split a large application to a set of services which can be developed, deployed and invoked independently from the rest of the application. Therefore, unlike in monolithic architecture in which the application is scaled by running multiple instances of the whole application, the microservice architecture benefits from the logical split and allows to scale different services individually (see Fig. 9). Hence, the important difference between the two is the fact that microservices function together as a distributed system by making API calls – typically over HTTP/REST, whereas the monolithic application uses so-called in-memory function calls. Consequently, moving from in-memory function calls to a distributed programming model introduces its own set of problems. (Fowler & Lewis, 2014.)
The microservice architecture pattern is seen as a good alternative for the development of web applications because the architecture introduces several benefits over a traditional monolithic architecture. For instance, different parts of the system can be implemented using state of the art frameworks and programming languages (Newman, 2015; Taibi et al., 2017). Second, scaling the system becomes cheaper as the development team is able to provision hardware at service-level rather than paying the costs of running yet another instance of monolithic application (Newman, 2015; Villamizar et al., 2016). And finally, the architecture allows companies to form distributed teams which are responsible for developing individual microservices (Fowler & Lewis, 2014; Newman, 2015; Taibi et al., 2017).

On the contrary, the microservice architecture introduces several challenges as well. For example, when a large system is modelled as separate microservices, logging and monitoring the production environment becomes more difficult. Second, testing different parts of the system requires more work as high number of services are communication over the network. But most importantly, designing a system in a form of microservices is not an easy task for developers used to building monolithic applications. (Newman, 2015; Taibi et al., 2017.)

Speaking of challenges, because the architecture is still relatively new research topic, the overall benefits and disadvantages of using the solution is still unknown. However, the study conducted by Alshuqayran, Ali and Evans (2016) reveal that the microservice architecture might not be the silver bullet software companies have been looking for. Instead, the microservice architecture – like any other architecture – continues to have its own set of pros and cons. The results show that microservices face challenges also within the areas of communication/integration, service discovery, performance, fault tolerance and security. (Alshuqayran et al., 2016.)

As of today, the use of microservice architecture has not been studied in the context of MMORPGs in particular, despite the fact that MMORPGs are known for their scalability requirements (Chen & Muntz, 2006; Ferrara et al., 2010). However, the study
conducted by Cardin (2016) focuses on designing a horizontally scalable backend for mobile games using the microservice architecture. In this study, the requirements of the system are identified by analyzing most popular mobile games in the market. These requirements, such as chat, multiplayer and matchmaking are then turned into nine different microservices which make up the bulk of the system. Hence, the important part about the solution is the fact that services are not directly exposed to the internet. Instead, the solution uses multiple proxies and a so-called API gateway in front of the microservices. The API gateway is a piece of software which purpose is to provide a single point of entry for the game clients accessing the system by pre-authenticating, load balancing and routing the incoming requests. For this to work, the microservices need to register themselves to the gateway as well. (Cardin, 2016.)

Cardin (2016) continues that first of the microservices is a so-called Social Service. The purpose of the service is to provide necessary APIs for integrating with social networks, such as Facebook and Google Play. As an example, the service offers APIs for fetching user profiles, friends and achievements. However, the service does not store the returned data to its own database. Instead, the service acts only as an intermediate which is able to communicate with external services. For this reason, the social service is consumed by a so-called Authentication Service during registration and authentication requests. (Cardin, 2016.)

Speaking of Authentication Service, Cardin (2016) explains that the purpose of the service is to provide necessary APIs for registering and authenticating a player. In more detail, the registration and authentication procedure can be completed using a help of Social Service by creating an account which is linked to one of the 3rd-party providers via OAuth2. The OAuth2 is an authorization framework that allows a player to securely access a system $A$ using an account of system $B$. Alternatively, the authentication service supports creation of custom player accounts. However, the accounts created with the latter scheme cannot request information from the Social Service. Finally, the authentication service stores all accounts to its own database with or without the link to an external service. (Cardin, 2016.)

Cardin (2016) continues that third of the microservices is a so-called Shop Service. The purpose of the service is to provide necessary APIs for doing in-app purchases. The service improves the security of the system because the communication with 3rd-parties is taking place at the server rather than at the clients. Therefore, the service also eases the development of the game clients. The basic responsibilities of the shop service involve keeping an up-to-date list of products and offering an API for purchases. Hence, the shop service stores the products and purchases to its own database. (Cardin, 2016.)

According to Cardin (2016), the fourth of the microservices is a so-called Datasore Service. The purpose of the service is to offer APIs for storing and retrieving data which relates to the current state of the game and its business logic. Consequently, the service has the biggest database schema as it persists information regarding players, items, achievements, upgrades and more. The actual schema is designed as such that the player id is used as a unique identifier for player-related data. Hence, the same approach appears to be true for other services as well, such as Authentication Service, Leaderboard Service and Chat Service. (Cardin, 2016.)

Cardin (2016) continues that fifth of the microservices is a so-called Leaderboard Service. The purpose of the service is to offer APIs for keeping an up-to-date list of player rankings. For this to work, the service relies on the usage of multiple boards in which the players are ranked based on different factors, such as score and skill. The author emphasizes that the service should rely on the usage of in-memory database due to state synchronization and performance reasons. However, the actual performance of
the in-memory nor relational database is not measured in the given context. (Cardin, 2016.)

Cardin (2016) states that the sixth of the microservices is a so-called *Chat Service*. The purpose of the service is to offer APIs for sending and receiving chat messages. In more detail, the service implements *publish-subscribe pattern* which allows players to exchange messages by subscribing to a same abstract room at the server. The service also supports asynchronous communication – similar to email – by storing messages to its own database. However, rather than assigning email addresses to the players, a message from a sender to a receiver is identified using *outbox_id* and *inbox_id* respectively. (Cardin, 2016.)

Cardin (2016) continues that the seventh of the microservices is a so-called *Matchmaking Service*. The purpose of the service is to offer APIs for matching teams to compete against each other in a way that the game is fair. In order to do this, the service ranks players based on their skill level and assigns them to groups which are equal in strength. The skill level of a player is determined with a help *Leaderboard Service* or using an alternative criteria. The service works together with a *Multiplayer Service* to transfer the players to available game instances. (Cardin, 2016.)

Speaking of *Multiplayer Service*, Cardin (2016) explains that the purpose of the service is to offer APIs for transferring players to existing or new game instances. A game instance is a server-side program that accepts inputs from the players and calculates the current state of the game. In addition, the service supports managing of separate room instances which re-uses the previously mentioned *publish-subscribe pattern* for transferring messages between the players. The service allocates all instances on demand so the IP addresses and port are not known until a reservation occurs. (Cardin, 2016.)

Finally, the Cardin (2016) describes that a *Logger Service* is needed for tracking events occurring in the whole system. In more detail, the purpose of the service is to provide APIs for tracking events in multiple categories, such as faults, ads and in-app purchases. Consequently, almost every service in the system communicates with the logger service. Hence, the service offers separate web interface for browsing the logs. (Cardin, 2016.)

Table 1. A set of design principles for microservice-based applications (Fowler & Lewis, 2014).

<table>
<thead>
<tr>
<th>Design principle ID</th>
<th>Design principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP-1</td>
<td>Componentization via services</td>
<td>The functionality of a system should be decomposed using independent services rather than relying on the use of shared libraries.</td>
</tr>
<tr>
<td>DP-2</td>
<td>Organized around business capabilities</td>
<td>The services should be split around business capabilities so that microservices are developed by cross-functional teams, including programmers and designers.</td>
</tr>
<tr>
<td>DP-3</td>
<td>Products not projects</td>
<td>The development team building a certain microservice needs to take full responsibility of running it as well.</td>
</tr>
<tr>
<td>DP-4</td>
<td>Smart endpoints and dumb pipes</td>
<td>The microservices needs to be decoupled from each other and invoking a functional call needs to go through an API layer or message bus.</td>
</tr>
<tr>
<td>DP-5</td>
<td>Decentralized governance</td>
<td>Different parts of the system should be possible to build using different technology stacks.</td>
</tr>
<tr>
<td>DP-6</td>
<td>Decentralized data management</td>
<td>Each services should manage its own data rather than</td>
</tr>
</tbody>
</table>
In the final analysis, the author of this thesis conclude that coming up with a set of design principles for the development of microservice-based applications is essential. The reason is that the author of this thesis is not convinced that the approach taken by Cardin (2016) is the right way to employ microservice architecture pattern. The biggest problem with the solution is the fact that adding high number of services for multiple—and sometimes overlapping—purposes is going to increase the development time, complexity and amount of tests to be written to the overall system. The author of this thesis continues that adding of new services should be avoided as long as the functionality can happily coexist inside another service. Fortunately, Fowler and Lewis (2014) defines a set of design principles for helping the development microservice-based applications. These principles are listed in Table 1.
3. Research problem and methodology

The objective of this master’s thesis is to study how MMORPG backend can be designed using the microservice architecture? In this context, MMORPG backend means a server-side game system which needs to be in place for thousands of people to connect and role-play in a virtual world (Chen & Muntz, 2006; Ferrara et al., 2010). On the contrary, an application based on the microservice architecture means a distributed system which is composed by a set of services, each of them designed to perform separate and well-defined task (Newman, 2015).

3.1 Research method

The research method of this thesis is a design science methodology, and the research is conducted following the design science guidelines of Hevner et al. (2004). Hevner et al. (2004) defines that the purpose of design science research is to address the research problem by building and evaluating IT artifacts. These artifacts can be models, methods, frameworks or instantiations. An artifact, however, is considered viable only when it is developed to solve identified business needs in a new or more efficient manner. Meaning, the artifact cannot be a result of routine design in which a new product for certain company is being developed using the best practises. Instead, the quality and novelty of the artifact needs to be demonstrated using proper methods of evaluation and a form of communication which is understandable. For this reason, the purpose of an artifact is by no means to be a complete implementation of an information system because the usefulness of the artifact can be measured using a prototype as well. Hence, the search for an optimal artifact is an iterative process in which the solution is continuously evaluated and modified according to requirements of the system. (Hevner et al., 2004.)

In this thesis, the design science guidelines of Hevner et al. (2004) are adapted by building a proof of concept implementation of an MMORPG backend which is developed according to principles of a microservice architecture pattern. These principles along the requirements of the system are derived from the existing literature because the research is not conducted in a company setting. The knowledge base of this research is the existing work towards designing MMORPG architectures, and the studies describing microservice architecture pattern in more detail. The game backend is rigorously evaluated by comparing the design against the principles, and the functionality against the requirements using three different test strategies: unit testing, service testing and end-to-end testing. From the results of the evaluation, and the process of developing the system, the research problem can be addressed.

Hence, there are at least three different business for studying microservices in the context of MMORPGs. First, the microservice architecture makes it possible to implement a system using state of the art frameworks and programming languages (Newman, 2015; Taibi et al., 2017). Second, scaling the system becomes cheaper as the development team is able to provision hardware at service-level rather than paying the costs of running yet another instance of monolithic application (Newman, 2015; Villamizar et al., 2016). And finally, the architecture allows companies to form distributed teams which are responsible for developing individual microservices (Fowler & Lewis, 2014; Newman, 2015; Taibi et al., 2017).
3.2 Developing the artifact

The search for an optimal artifact – as required by Hevner et al. (2004) – is accomplished by conducting an iterative software development process. In this process, there are multiple steps in which the requirements of the system will be reviewed and the artifact incrementally improved. Hence, the artifact is considered complete when the requirements of the system are met, and iterating the artifact further does not provide meaningful additions to the knowledge base. The process ends to a final evaluation in which the artifact is compared against the principles of microservice architecture pattern and the requirements it was designed to work with. The main steps of the development process and their objectives are described below.

Requirements engineering

The objective is to develop a list of requirements for a proof of concept implementation of an MMORPG backend. In the beginning, the requirements are selected from the existing literature by identifying a class of problems that the system needs to solve. These problems are searched from the studies describing solutions for MMORPGs, including the studies that describe typical features of MMORPGs. As the study progresses, the requirements are reviewed and modified according to discovered needs of the system.

Design

The objective is to design a structure of an MMORPG backend which is able to meet the requirements of the system. The work begins by splitting the requirements to different microservices using the idea of loose coupling and high cohesion (Newman, 2015). According to Newman (2015), microservice-based system is loose coupled when changing the behavior of one service does not require changing the behavior of other services. On the contrary, a system contains high cohesion when related functionality of the system exist in the same place. (Newman, 2015.) After splitting the requirements, the development of microservices can begin. However, it is very unlikely that the initial design is the most optimal one. The reason is that implementing microservices in practice is expected to produce new knowledge, and therefore makes altering the design necessary. Hence, only the final structure of the system will be reported, because it is considered most valuable concerning the current and future research.

Implementation

The objective is to ensure that the theories made during a design phase can be implemented in practice. The result of this step is an MMORPG backend which is composed by a set of microservices. These services will be developed using the programming languages and frameworks which are fastest to work with. The reason is that the purpose of the implementation phase is to focus on prototyping rather than implementing most performant versions of the used algorithms. That being said, existing solutions are used for those parts of the system which does not have to be made from scratch.

Testing

The objective is to verify that the system is working as expected and that all functionalities have been implemented. For this reason, the author of this thesis performs a rigorous evaluation by employing three different testing strategies. First, unit tests will be used for verifying that method calls inside microservices works properly. Second, services tests will be used for verifying that calling microservices produces
appropriate results. Third, end-to-end tests will be used for verifying that microservices are able to work together as an integrated system.

**Final evaluation**

The process of designing, developing and testing the system ends to a final evaluation in which the system is compared against the principles of microservice architecture pattern and the requirements it was designed to work with. The purpose of this comparison and the process of developing the system is to produce the required knowledge for solving the research problem of this thesis.

### 3.3 Literature review

The objective of the literature review was to support the decision making in the system’s design, and inform the researcher about the difficulties of implementing online games. In the beginning, literature was searched from the digital libraries of ACM and IEEE. The search started by using broad terms, such as MMORPG architecture and massively multiplayer online role-playing game architecture. The objective was to find studies which describes MMORPG architectures in more detail. These searches ended up generating thousands of hits, but with very little relevance. In other words, only couple of the studies presented solutions for MMORPGs in particular. For this reason, the author of this thesis altered the terms to include massively multiplayer online game architecture and multiplayer online game architecture as well. After performing the search, it became obvious that majority of the existing research within the field focuses on designing architectures for MMOGs and MOGs, but only a few targets MMORPGs explicitly.

The author of this thesis continued the literature review by selecting a group of articles for light reading so that more knowledge about the development of MOGs could be gained. The results of this analysis revealed that developing MOGs is hard, and that there is a need for gathering material from multiple themes before attempting to design an MMORPG backend. Also, due to novelty of the research problem, the required material for conducting the research were split between academic research and industry know-how. For this reason, the author expanded the search to include the findings of online resources and books related to the subject. However, digital libraries of ACM and IEEE were still used as primary sources. In the end, the author decided to search material from five different themes which were expected to support the design of MMORPG backend.

First of all, the author of this thesis concluded that there is a need to explain the effect of the internet on multiplayer game development. The majority of the existing research within the field focuses on designing MOG architectures, but only a few explains resource limitations: network bandwidth, network latency and computational power in which MOGs must operate. For this reason, first of the themes was to review the aforementioned resource limitations briefly.

Second, the author of this thesis concluded that there is a need to define MMORPG genre. The reason was that in order to design an MMORPG backend, the requirements of the system needs to be derived from the literature as the research was not conducted in company setting. Therefore, second of the themes was to review MMORPGs so that mutual understanding about the common features of the games could be formed.

Third, the author of this thesis concluded that there is a need to describe most popular MOG architectures: P2P, CS, PP-CA and CMS in more detail. The reason was that
majority of the existing research within the field focuses on designing architectures for MMOGs and MOGs, but only a few targets MMORPGs explicitly. The architectures were reviewed from the viewpoint of MMORPGs so that findings of previous researches would support the process of designing an MMORPG backend.

Fourth, the author of this thesis concluded that there is a need to review so-called application-level techniques which are used in MOGs today. The reason is that Smed et al. (2002) states that there is a chance that choosing a MOG architecture is not enough to meet the requirements of a given game design. An architecture helps to reduce resource requirements of a game but cannot get rid of them completely. Very often, additional application-level techniques are needed for mitigating resource usage further, and increasing responsiveness and fairness of the game. (Smed et al., 2002.) For this reason, the author of this thesis identified a set of application-level techniques which could be useful in the design of an MMORPG backend.

Finally, the author of this thesis concluded that there is a need to review studies describing microservice architecture pattern and its context of use in more detail. After all, the research is about applying microservice architecture pattern to a design of an MMORPG backend. However, because microservice architecture is a relatively new approach for building modern web applications, it became obvious that studies applying microservice architecture pattern to a design of MMORPGs were lacking. Even so, the author of this thesis attempted to find microservice-related material which could be beneficial in the process of designing an MMORPG backend.
4. Requirements engineering

The empirical phase of this study began with the requirements engineering process. The purpose of this process was to develop a list of requirements for a proof of concept implementation of an MMORPG backend. In the beginning, requirements were selected from the existing literature by identifying a class of problems that the system needs to solve. Here, studies describing typical features of MMORPGs or solution proposals – such as game architectures and application-level techniques – were used as a primary source of information. As the study progressed, the requirements were reviewed and modified according to discovered needs of the system.

4.1.1 Selecting high-level requirements from the literature

The challenge of selecting requirements from the literature was the fact that every MMORPG is unique, and therefore is having their own set of requirements. For instance, *Eve Online* is a space MMORPG which allows a player to build her own ship and explore a galaxy consisting of more than 7000 star systems (*Eve Online*, 2003). On the contrary, *World of Warcraft* is a fantasy MMORPG which allows a player to portray herself as a customized character and explore a landscape containing interesting quests, dangerous monsters and friendly NPCs (*World of Warcraft*, 2004).

Consequently, the author of this thesis could not select requirements which were really game-specific as there could have been hundreds of them. Instead, the author decided to focus his attention on defining a set of high-level requirements which could provide a good generalization across wide range of MMORPGs. By doing so, coming up with a set of requirements for the proof of concept implementation of the system was easier because the findings of existing literature were treated more like a guideline rather than a single source of truth. Fortunately, this approach also gave the author an opportunity to use his industry know-how. In any case, the chosen high-level literature requirements and their purpose are explained below.

**Progression**

In all MMORPGs, the development of in-game character appears to be the primary goal of the game. For this reason, MMORPGs typically include some sort of progression system in which players earn *experience points* and *levels* for their actions in the game world. Naturally, these actions change on a game by game basis, but doing quests and fighting against monsters appears to be typical ways to progress in the game. (*Suznjevic et al.*, 2009; *World of Warcraft*, 2004.)

**Character customization**

In all MMORPGs, character customization plays an important role because it allows players to personalize the game experience in a way that it feels more meaningful to them. In its simplest form, character customization means that the players are allowed to create in-game characters which properties, such as name, race and skills can be altered. (*Achterbosch et al.*, 2008.)
Player interaction

In all MMORPGs, player-to-player interaction is an important element because actions of other players – such as quests, fights and purchases from shops – affects the shared game state of the server. Very often, player interaction is even required in order to progress in the game. Example of such cases are so-called raids in which large group of players needs to form an alliance to take down monsters which are too strong to defeat alone (Suznjevic et al., 2009). For this reason, MMORPGs typically provides different tools for facilitating communication between the players, such as chat so that using of external software is not required (Achterbosch et al., 2008).

Persistent world

In all MMORPGs, the novel feature of the game is the fact that it continues to run and evolve whether the player is online or offline. Hence, because a big part of these games is about progression through quests and combat, the game is implemented as such that the player is able to use the same in-game character between different gaming sessions. (Achterbosch et al., 2008.) For this reason, it is not surprising that many MMORPGs – such as Diablo 3 – provides permanent in-game stash for storing of goods and valuable items. Thus, it can be argued that having an in-game character there waiting for the future gaming sessions lowers the barrier of returning to the game.

Responsiveness

In all MMORPGs, the game needs to be responsive so that the time between pressing a button and seeing the result on the screen does not frustrate the player (Brun et al., 2006). Hence, according to studies of Claypool, M. and Claypool, K. (2006; 2010), MMORPGs needs to respond within 500ms or the delay starts to affect the user experience of the game. Naturally, complying these properties in MOGs is difficult because messages affecting the game state needs to be transferred over the internet (Chen & Mahesan, 2004).

Scalability

In all MMORPGs, the game backend needs to be designed as such that it is able to handle the increasing workload caused by more and more people joining to the game. As a rule of thumb, these games are expected to support simultaneous play for thousands of people which should reside at a same or separate instances of a virtual world (Chen & Muntz, 2006; Ferrara et al., 2010).

Security

In all MMORPGs, the game needs to be secure as players are not going to a MOG which is crowded with cheaters. Unfortunately, however, because the internet allows anyone, from anywhere, to connect and play the same game, attempts of cheating (Li et al., 2004) and game bots (Gianvecchio et al., 2009) are common to almost every MOG out there.

4.1.2 Developing the detailed requirements

The high-level requirements selected from the literature fits the description of wide range of MMORPGs, but they could not be used for implementation purposes as is. Consequently, the next step in the requirements engineering process was to develop these requirements further to more detailed requirements of a proof of concept.
implementation of the system. These requirements were developed by wearing the hat of a game developer and concerning all the things such system would need in order to support simultaneous play for thousands of people. In the end, having this mind-set proofed to be useful way of thinking the system’s internals, and a set of more detailed requirements could be formed. As the study progressed, these requirements were further reviewed and modified according to discovered needs of the system.

Table 2. A set of detailed requirements derived from the literature requirements. Here, the requirements which are marked with an asterisk are the ones which were added after iterating the system’s design.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-1</td>
<td>Create account</td>
<td>As a Player, I want to create an account so that I can access the game.</td>
</tr>
<tr>
<td>R-2</td>
<td>View account</td>
<td>As a Player, I want to view my account information.</td>
</tr>
<tr>
<td>R-3</td>
<td>Edit account</td>
<td>As a Player, I want to edit my account information.</td>
</tr>
<tr>
<td>R-4</td>
<td>Delete account</td>
<td>As a Player, I want to delete my account so that I can no longer play the game.</td>
</tr>
<tr>
<td>R-5</td>
<td>Log in</td>
<td>As a Player, I want to log in so that I can play the game.</td>
</tr>
<tr>
<td>R-6</td>
<td>Log out</td>
<td>As a Player, I want to log out so that I can leave the game.</td>
</tr>
<tr>
<td>R-7</td>
<td>List game servers</td>
<td>As a Player, I want to see a list of game servers so that I can select a server to play at.</td>
</tr>
<tr>
<td>R-8</td>
<td>Select a game server</td>
<td>As a Player, I want to select a game server so that I can play on a server with lowest RTT.</td>
</tr>
<tr>
<td>R-9</td>
<td>Register a game server</td>
<td>As a System, I want to register a game server so that there are more game servers available for the players.</td>
</tr>
<tr>
<td>R-10</td>
<td>Create character</td>
<td>As a Player, I want to create a character so that I can personalize the game experience.</td>
</tr>
<tr>
<td>R-11</td>
<td>Load character</td>
<td>As a Player, I want to load the progress of my character.</td>
</tr>
<tr>
<td>R-12</td>
<td>Connect to game world</td>
<td>As a Player, I want to connect to the game world so that I can interact with the players.</td>
</tr>
<tr>
<td>R-13</td>
<td>Save character progress</td>
<td>As a Player, I want to save the progress of my character so that I don’t have to start from the beginning.</td>
</tr>
<tr>
<td>R-14</td>
<td>Calculate game state</td>
<td>As a System, I want to calculate the game state so that inputs of the players can progress the game simulation.</td>
</tr>
<tr>
<td>R-15</td>
<td>Distribute game state</td>
<td>As a System, I want to distribute the game state so that players can see the current state of the game.</td>
</tr>
<tr>
<td>R-16</td>
<td>Basic movement</td>
<td>As a Player, I want to move around in the game so that I can find dangerous monsters.</td>
</tr>
<tr>
<td>R-17</td>
<td>Player vs enemy combat</td>
<td>As a Player, I want to combat against enemies so that I protect myself, gain experience and items.</td>
</tr>
<tr>
<td>R-18</td>
<td>Chat</td>
<td>As a Player, I want to be able to chat so that I can communicate with other players.</td>
</tr>
<tr>
<td>R-19</td>
<td>Log events</td>
<td>As a System, I want to log events so that they could be browsed later on.</td>
</tr>
<tr>
<td>R-20</td>
<td>View logs</td>
<td>As a Developer, I want to view logs so that I can trace errors and detect suspicious behavior.</td>
</tr>
<tr>
<td>R-21</td>
<td>Cross-platform support*</td>
<td>As a Developer, I want the system to work for cross-platform game clients so that I can maximize the sales.</td>
</tr>
<tr>
<td>R-22</td>
<td>Authorization*</td>
<td>As a System, I want to authorize a user so that I can detect his rights to complete an action.</td>
</tr>
<tr>
<td>R-23</td>
<td>Unified API access*</td>
<td>As a Developer, I want to have unified API access so that calling APIs is easier from the game client.</td>
</tr>
<tr>
<td>R-24</td>
<td>Activate account*</td>
<td>As a Player, I want to activate my account so that no one else cannot register with my email.</td>
</tr>
<tr>
<td>R-25</td>
<td>Suspend account*</td>
<td>As a Developer, I want to suspend an account so that malicious player cannot log in to the game.</td>
</tr>
<tr>
<td>R-26</td>
<td>List accounts*</td>
<td>As a Developer, I want to see a list of accounts so that I can find out who is playing the game.</td>
</tr>
<tr>
<td>R-27</td>
<td>Register service*</td>
<td>As a System, I want to register a service so that it can be located from the network.</td>
</tr>
<tr>
<td>R-28</td>
<td>Deregister service*</td>
<td>As a System, I want to deregister a service because it is no longer available.</td>
</tr>
<tr>
<td>R-29</td>
<td>Get service address*</td>
<td>As a System, I want to get service address so that I can communicate with it.</td>
</tr>
<tr>
<td>R-30</td>
<td>Marketing page*</td>
<td>As a Developer, I want to have marketing page for the game.</td>
</tr>
<tr>
<td>R-31</td>
<td>Monitoring*</td>
<td>As a System, I want to monitor instances so that I can detect when they are no longer available.</td>
</tr>
<tr>
<td>R-32</td>
<td>Load game items*</td>
<td>As a System, I want to load game items so that I can use them inside the game world.</td>
</tr>
<tr>
<td>R-33</td>
<td>Load game enemies*</td>
<td>As a System, I want to load game enemies so that I can use them inside the game world.</td>
</tr>
<tr>
<td>R-34</td>
<td>View inventory*</td>
<td>As a player, I want to view my inventory so that I can see my game items.</td>
</tr>
<tr>
<td>R-35</td>
<td>Collect a game item*</td>
<td>As a Player, I want to collect game items so that I can sell them and gain wealth.</td>
</tr>
<tr>
<td>R-36</td>
<td>Drop a game item*</td>
<td>As a Player, I want to drop low level game items.</td>
</tr>
</tbody>
</table>

Table 2 lists the detailed requirements which were derived from the literature requirements. For the sake of simplicity, the author chose only two different roles – Player and Developer – which are allowed to access the system.

### 4.1.3 Splitting the detailed requirements to different microservices

As a final step in the requirements engineering process, the author of this thesis decided to follow the advice from Newman (2015), and attempted to split the requirements of the system to different microservices using the idea of loose coupling and high cohesion. According to Newman (2015), microservice-based system is loose coupled when changing a behavior of one service does not require changing the behavior of other services. On the contrary, a system contains high cohesion when related functionality of the system exist in the same place. (Newman, 2015.)

However, the author of thesis quickly learned that coming up with a set of services meeting the criteria appeared to be difficult. The reason was threefold. First, the author had no prior experience on modelling a microservice-based system so each service ended up being responsible of either too much or too little of the overall functionalities.
Second, the author had no prior experience on developing an MMORPG backend having the exactly same set of requirements, so previous knowledge of organizing application code to different services could not be used as a reference. Third, the author did not understand the requirements well enough to be able to map them to appropriate microservices; Of course, conducting more rigorous requirements engineering process could have alleviated the problem, but the author argues that requirements are never fully understood until first lines of code gets written.

For these reasons, the author concluded that only option to overcome the difficulty of splitting requirements to different microservices was to learn more about the system by building something in practice. Consequently, the design and development of the system began with a set of assumptions listed in Table 3.

Table 3. A set of detailed requirements mapped to microservices before starting the development of the system.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-19, R-7, R-8, R-9</td>
<td>Discovery service</td>
</tr>
<tr>
<td>R-19, R-1, R-2, R-3, R-4, R-5, R-6</td>
<td>Account service</td>
</tr>
<tr>
<td>R-19, R-12, R-14, R-15, R-16, R-17</td>
<td>Game service</td>
</tr>
<tr>
<td>R-19, R-10, R-11, R-13</td>
<td>Character service</td>
</tr>
<tr>
<td>R-19, R-18</td>
<td>Chat service</td>
</tr>
<tr>
<td>R-19, R-20</td>
<td>Logger service</td>
</tr>
</tbody>
</table>

After designing and developing the system for a few months, the author gained the required knowledge for mapping the detailed requirements to different microservices. The author concludes that final structure of the system deviates from the initial assumptions; Altering the design was necessary because some of the important requirements were missing and working with the system produced new design knowledge.

Table 4. A set of detailed requirements mapped to microservices after finishing the development of the system.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-19, R-21, R-23</td>
<td>API gateway</td>
</tr>
<tr>
<td>R-19, R-27, R-28, R-29, R-31</td>
<td>Discovery service</td>
</tr>
<tr>
<td>R-19, R-1, R-24, R-30</td>
<td>Game website</td>
</tr>
<tr>
<td>R-19, R-22, R-1, R-2, R-3, R-4, R-5, R-24, R-25, R-26</td>
<td>Account service</td>
</tr>
<tr>
<td>R-19, R-22, R-32, R-33, R-34, R-35, R-36, R-10, R-11, R-13</td>
<td>Realm service</td>
</tr>
<tr>
<td>R-19, R-22, R-32, R-33, R-34, R-35, R-36, R-12, R-14, R-15, R-16, R-17, R-18, R-21, R-5</td>
<td>Zone service</td>
</tr>
<tr>
<td>R-19, R-20</td>
<td>Logger service</td>
</tr>
</tbody>
</table>

Table 4 shows the microservices after finishing the development of the system. The process of how and why the requirements were mapped to different microservices are discussed in Chapter 5.
5. Design

This chapter is reserved for describing the structure of an MMORPG backend which was designed according to principles of a microservice architecture pattern. These principles along the requirements of the system were derived from the existing literature because the research was not conducted in a company setting. Hence, only the final structure of the system is reported, because it was considered most valuable concerning the current and future research. However, each chapter includes a rational behind design decisions.

5.1 Architecture overview

The MMORPG architecture was designed to support the development of cross-platform game clients so that a company or an individual implementing the system could distribute the game on multiple marketplaces. In the beginning, however, the system was designed as such that implementing cross-platform games would have been impossible. The problem was that the author of this thesis overlooked the fact that web browsers in particular sets limitations for data transfer. However, further incrementing the design revealed that achieving a cross-platform support – today – would be trivial to implement. The reason is that microservices are primarily accessed using HTTP/REST whereas the real-time communication taking place between a game client and the game backend can be implemented using a so-called WebSocket protocol. The WebSocket protocol is a result of recent advancements in technology and it allows to implement a system which can be accessed using cross-platform game clients, such as mobile, desktop and web applications (The WebSocket Protocol, 2011). Consequently, using the protocol in the design of MMORPG backend was an obvious choice. The actual uses of the protocol and how microservices function together as an integrated system are explained in the following chapters. See Figure 10 for illustration.

![Diagram of MMORPG architecture](image)

**Figure 10.** The game backend can be accessed using cross-platform game clients.

The designed architecture consists of five different microservices, an API gateway and a game website. *Game website* is used for marketing the game and creating user accounts so that the registration flow is centralized to the web rather than implementing the same
functionality to multiple game clients. *API gateway* is used for providing a single point of entry to the system by proxying and routing requests to underlying microservices. *Discovery service* is used for locating microservices from the network by registering IP addresses and performing health checks. *Account service* is used for managing user accounts and returning short-lived access tokens in response to successful authentication attempts. *Realm service* is used for storing the business logic of the game and the progress of the players. *Zone service* is used for implementing real-time functionalities of the game by calculating and distributing the game states. Finally, *Logger service* is used for collecting logs to a centralized place so that tracing errors and detecting suspicious behavior in the system is easier for the developers.

![Diagram](image_url)

**Figure 11.** The high-level architecture of the system.

The Figure 11 illustrates the high-level architecture of the system. The system should be deployed to a proper cloud computing environment – such as *Amazon Web Services* (AWS) – so that running new instances of microservices could be accomplished without too much hassle. The AWS also helps securing the system in a form of reverse proxies and setting up proper firewall rules. In addition, AWS allows ordering servers from multiple regions so achieving responsiveness is easier for the game developers. Nevertheless, the actual deployment and configuration of the live environment is considered to be outside the scope of this thesis. Thus, the final structure of the system most likely vary between the games as the amount of services and load balancers depends also on the popularity of a game.

### 5.2 API gateway

The microservice architecture pattern favors autonomous services over shared libraries (Fowler & Lewis, 2014), and therefore the amount of running services in the system is expected to grow over time. However, regardless of the used architecture, calling microservices from a viewpoint of a game client should feel as if the system is
developed using a monolithic architecture. Meaning, the complexities of the chosen server-side architecture should not be a concern for the teams developing the game clients. The microservice architecture makes achieving such structure more difficult because a single point of entry to a system does not exist; The microservice-based application contains multiple services, each of them designed to perform separate and well-defined task (Newman, 2015).

The problem was solved by placing a smart reverse-proxy known as API gateway between the game clients and the game backend. The idea of implementing an API gateway was based on the studies of Cardin (2016) and Richardson (2017a). In this context, the purpose of the API gateway was to provide a single of entry to the system by proxying and routing requests to underlying microservices. For this to work, the API gateway needed to be aware of the locations of the requested services. Consequently, the designed system employed a separate Discovery service for registering and deregistering microservice addresses. As a result of doing this, the API gateway was able to route requests coming from external network to the internal microservices. The Figure 12 illustrates the communication between an API gateway and discovery service.

![Figure 12. The API gateway routes requests to appropriate microservices with a help of Discovery Service.](image)

The design of the API gateway was kept simple for a reason. The API gateway was the main entry point to the system so the gateway was expected to be scaled horizontally sooner or later by placing additional load balancer in front of the gateway. Hence, horizontally scaling a small service is going to be cheaper than scaling a monolithic application (Newman, 2015; Villamizar et al., 2016). Moreover, developing an API gateway with multiple purposes would also be against the principles of microservice architecture pattern. That being said, the author recognizes that the discovery service could be a part of the API gateway as well. But the service was kept separated for previously mentioned purposes but also because the author wanted that internal microservices do not contact the API gateway. The API gateway was used only for proxying and routing requests coming from the external network. Separate microservices need to call discovery service explicitly in order to locate each other in the network.
The author concluded that the future improvements of the API gateway could be storing of already queried locations of the microservices in-memory or in its own database so that chattiness between the gateway and discovery service is reduced. The chattiness could be also reduced if a separate message bus is used for transferring events of the discovery service. In the end, long polling addresses from the discovery could also work to a some extend.

Finally, in order for the API gateway to operate properly, the static IP address of the discovery service was hard-coded to a configuration file. Hence, the API gateway was implemented using Node.js because it was capable of forwarding HTTP and WebSocket connections.

5.3 Discovery service

The microservice architecture consists of multiple services which are distributed to the network. These services are often started and tear down on demand based on the current load of the system. Hence, because microservice-based applications are often run in cloud computing environments, using of auto-scaling groups and dynamically changing IP address is pretty common. However, because microservices are meant to be communicating together as an integrated system, they need to be able to locate each other from the network. For this reason, a separate Discovery service was needed for registering IP addresses of the microservices to a centralized place. The idea of implementing the discovery service was based on the studies of Gargolinski et al. (2005), Henderson (2002), Richardson (2017b) and Zander et al. (2005).

The registration of a new microservice worked by implementing an API which is able to store the given service name and IP address. The API was called by every microservice in the system during a so-called startup procedure. As a result of doing this, the discovery service stored the IP address and name of the service to the database and started checking the status of the registered instance. The status was checked using a long polling event in which the discovery service called /health-check URL of the registered service every 30 seconds. Hence, as long as the service was returning 200 OK, the discovery service did no further actions. However, once the service failed to reply back with proper status code, the discovery service removed the instance from the database. The Figure 13 illustrates the data model used by the discovery service.

<table>
<thead>
<tr>
<th>services</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Figure 13. The data model of the Discovery service.

In addition to deregistering a service based on the failed health check-request, the service provided an API for doing the same. The API was called by the microservices whenever they were shutting down so that the discovery service was able to update the list of active service addresses. Finally, the registration and deregistration requests coming from the API gateway to the discovery service should be blocked in a real deployment environment. The reason is that a malicious user could shutdown services on purpose and register their own. On the other hand, when service registration needs to
be exposed to the API gateway a separate API key could be used for protecting access to the system.

The author concluded that the discovery service could be improved in the future to include more detailed monitoring information because the health checking makes it possible to ask additional information. As a result of doing this, the service could open a separate API for returning a list of active services in the system and a snapshot of their performance metrics. Consequently, this information could be also used for sending automated email alerts to the developers of the system. Nevertheless, once the monitoring becomes truly an issue a separate microservice should be implemented for that purpose as well.

**Table 5.** A list of APIs provided by the *Discovery service*.

<table>
<thead>
<tr>
<th>Method</th>
<th>URI</th>
<th>Parameters</th>
<th>Requirement ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/services</td>
<td>name</td>
<td>R-29</td>
</tr>
<tr>
<td>POST</td>
<td>/services</td>
<td>name, ip</td>
<td>R-27</td>
</tr>
<tr>
<td>DELETE</td>
<td>/services</td>
<td>name</td>
<td>R-28</td>
</tr>
</tbody>
</table>

Table 5 lists the APIs which were provided by the discovery service.

### 5.4 Game website

Today, every game has a website of some sort whether the game is made to a mobile, desktop or web-platform. The website is primary used for marketing the game but it can be used for so much more. In this thesis, the game website was used for creating user accounts because centralizing the registration flow to the web means that cross-platform game clients do not have to implement the same functionality.

The registration flow was implemented with a help of *Account Service*. The account service is a separate microservice which was responsible for managing user accounts and providing API for authentication purposes. Hence, the author concludes that these functionalities could have been a part of the game website itself, but the approach would have not scaled well. The reason is that adding a customized registration flow later on to the game clients requires that the game website opens a separate API for account management. However, because websites in particular are implemented using techniques such as *Wordpress*, opening an API is less desirable.

The game website implements the registration process by offering a web form to a player. The information from the form is posted to an account service with the help of discovery service. The successful request ends up creating an account entry to the database of the account service and sending an activation link to the provided email address. The account is considered created at the time the email is sent but signing in to the system is prohibited as long as the player has not clicked the activation link. Once the link is clicked, the player is redirected to a game website which in turn calls the account service for activating the account. Finally, the website renders a view saying that the account is now activated. The full list of APIs used by the game website is listed in Table 6.
In final analysis, the game website could also be used for distributing the game. Hence, when the game is implemented as a browser-based application the website could have a separate view for playing the game right in the browser.

5.5 Account service

MMORPGs and games in general which has a continuous world or a need for storing a progress of a player for later gaming sessions needs to have a means for managing user accounts and providing API for authentication. In the developed system, this was the responsibility of the Account service. The account service was a separate system not only because of the microservice architecture pattern but also because the game backend was designed as such that the player could play in multiple Realm services using a single account. The solution works because the game authentication is implemented using json web tokens (JWTs). The Figure 14 illustrates the data model used by the account service.

![Figure 14. The data model of the Account service.](image)

The authentication and authorization of the system works by creating so-called JSON web tokens. In this solution, the player requested /auth using his username/password as in a normal form based authentication. The account service validated the credentials and then created base64 encoded access_token for the user with a JSON body containing account_id, first_name, last_name, role and username. By doing this, the token could be used to access separate microservices without contacting the account service. Hence, placing a role inside the JSON body allowed to implement an API-level authorization by checking the role – player or developer – in different microservices.

The author of this thesis concludes that the authorization could also be centralized to the account service, but that would require constant modifications to the scopes and unnecessary communication between the teams developing the systems. Mostly because...
adding or removing rights in a separate service require modification of scopes at the account service. Persisting only the user roles to the database is a better approach because a list of roles is not going to change as often as the APIs of certain microservices.

The another good thing in using an JWT is that the other services does not need to contact account service unnecessary because the access token contains relevant data about the account holder. Of course, the solution required that each microservice knows the shared secret of the account service so that the system is able to verify that the token has not been tampered in the transit. Second, the interesting thing about the access token is that because it has an expiration time, the token is basically as long as valid as the time is not expired or the shared secret is not modified. The API does not have separate API for extending the token time, because the access_token is meant to be used as a one-time token.

Table 7. A list of APIs provided by the Account service.

<table>
<thead>
<tr>
<th>Method</th>
<th>URI</th>
<th>Parameters</th>
<th>Requirement ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/accounts</td>
<td>page, per_page</td>
<td>R-26</td>
</tr>
<tr>
<td>POST</td>
<td>/accounts</td>
<td>first_name, last_name, email, username, password, password_confirmation</td>
<td>R-1</td>
</tr>
<tr>
<td>PATCH</td>
<td>/accounts/:id/activation</td>
<td>email, uuid</td>
<td>R-24</td>
</tr>
<tr>
<td>GET</td>
<td>/accounts/:id</td>
<td>-</td>
<td>R-2</td>
</tr>
<tr>
<td>PATCH</td>
<td>/accounts/:id</td>
<td>first_name, last_name, email, username</td>
<td>R-3</td>
</tr>
<tr>
<td>DELETE</td>
<td>/accounts/:id</td>
<td>-</td>
<td>R-4</td>
</tr>
<tr>
<td>POST</td>
<td>/accounts/:id/auth</td>
<td>username, password</td>
<td>R-5</td>
</tr>
<tr>
<td>POST</td>
<td>/accounts/:id/penalties</td>
<td>reason, expires_at</td>
<td>R-25</td>
</tr>
<tr>
<td>GET</td>
<td>/health</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The list of APIs used in the development of a account service are listed in Table 7.

5.6 Realm service

In MMORPGs, the development of in-game character is primary goal of the game. For this reason, the developed system contained a separate Realm service for storing the progress of the players and relevant business logic. The realm service included functionalities for persisting the progress of the players, list of allowed items and enemy types in the game. Hence, the central to the data model of the service was the usage of account_id. Meaning, creating a new character required that account_id was extracted from the provided JWT. The account_id was used as a unique identifier connecting the character to an account residing at the Account service. For this reason, using a single account on multiple realms was possible. See data model in Figure 15.
The realm service was launched in conjunction with zone service because every realm needs separate zone for the actual real-time functionalities. The realm service was designed only as an API-backend for storing the progress of the players. The service was separated from the zone because the author concluded that implementing real-time functionalities of the game using Node.js was easier. On the contrary, because RoR is better for working with relational databases, two different services were implemented. Separating realm service from the zone opened also more opportunities for scaling the game. Namely, the author could have implemented a system in which multiple zones per realm exists and moving between them requires using of in-game portals (Knutsson et al., 2004; Yang & Sutinrerk, 2007). However, implementing such system is not a trivial task because depending on the game design, using of in-game portals could break the flow of the game (Yahyavi & Kemme, 2013).

The important limitation of the realm service was the fact that a character created to a one realm could not be used in any other realm. In this regard, the solution works similarly to the architecture of World of Warcraft (World of Warcraft, 2004). The reason was that each realm was intended to support gameplay for a single game world. As a result of doing this, scaling the system was simply a process of starting new realm services each of them having their own database. Hence, implementing a game in which multiple universes exist is always arguable easier to scale than implementing a game – such as Eve Online – in which infinite number of players interact in a single universe. For this reason, the game backend used multi-universe approach.

**Table 8.** A list of APIs provided by the Realm service.

<table>
<thead>
<tr>
<th>Method</th>
<th>URI</th>
<th>Parameters</th>
<th>Requirement ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/characters/:id</td>
<td>-</td>
<td>R-11</td>
</tr>
<tr>
<td>POST</td>
<td>/characters/</td>
<td>name, character_type</td>
<td>R-10</td>
</tr>
<tr>
<td>PATCH</td>
<td>/characters/:id</td>
<td>experience, gold, damage, attack_speed, x, y</td>
<td>R-13</td>
</tr>
<tr>
<td>GET</td>
<td>/characters/:character_id/items</td>
<td>-</td>
<td>R-34</td>
</tr>
<tr>
<td>POST</td>
<td>/characters/:character_id/items</td>
<td>item_type, inventory_slot</td>
<td>R-35</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------</td>
<td>---------------------------</td>
<td>------</td>
</tr>
<tr>
<td>DELETE</td>
<td>/characters/:character_id/items/:id</td>
<td>-</td>
<td>R-36</td>
</tr>
<tr>
<td>GET</td>
<td>/items</td>
<td>-</td>
<td>R-32</td>
</tr>
<tr>
<td>GET</td>
<td>/enemies</td>
<td>-</td>
<td>R-33</td>
</tr>
<tr>
<td>GET</td>
<td>/health</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The list of APIs used in the development of a realm service are listed in Table 8.

5.7 Zone service

Unlike traditional APIs or web pages in which the purpose is to embrace the stateless nature of the system, in MOGs everything is about the game state which should be shared between the players interacting in the same virtual world so that everyone can see the progress of the game by rendering it on the screen.

In practice, before the players could interact with each other the players need to be connected to the same game server. The Zone service this goal as the actual real-time functionalities of the game happens in the zone. The zone service is a WebSocket-based game server and central to it is a so-called game loop in which the game state is calculated. Therefore, once the player connects to a zone it is automatically added to be part of the game loop so that he can send inputs and receive distributed game states.

The basic input flow of the zone service works in a form of remove procedure calls, so once the player makes some action at the game, such as picking up an item, or sending a chat message, these messages are sent to the zone service. The zone service validates the given inputs and distributes the updated game state for nearby players.

Initially, implementing a separate chat service was considered, but by further incrementing the system revealed that it should be part of the zone because the events occurring at the zone are very depended on the location of the player in the game world; Sending a chat message to players near him requires knowledge about the current location of the players. Similarly, asking a shopkeeper for a list of items on sale needs to validate the distance between a player and the shopkeeper.

Complicating the system and splitting functionalities of the game world to multiple services would have overly complicated the process of validating the actions of the players. Hence, the author argues that in order for team to focus truly on developing the server-side game system the service should not be split.

5.8 Logger service

The microservice-based system consists of multiple services which receives requests from a single or multiple clients and different microservices. These events are traditionally logged at the server to a separate log files so that the game developer is able to sign in to a certain production node and inspect the events. In monolithic architecture, inspecting logs is relatively straight forward because there might be only a few instances of the application running at the same time. Hence, tracing the actions of the application is simple because the use of in-memory function calls.
In microservice-based architecture there is even in minimal setup a distributed system in which microservices execute calls between each other. So inspecting logs of a single microservice, such as Account service, we cannot be sure about the complete flow the request has went through in the system. For this reason, a separate Logger service was needed so that logs occurring in different microservices could be aggregated to a same placed and viewed by the game developer.

For this to work, the system was implemented as such that every microservice started a socket-based connection to the logger service on startup. After that, when something needed to be logged the line was stored to a file but also sent to a separate logger service using socket connection. The service was responsible for accepting the logs sent over the wire and saving them to its own database.

The important part regarding logging is the fact that teams developing the microservices are responsible for logging properly by using the account_id stored in the access_token whenever such information is available. The proper placement of account_id made it possible to see how a certain player has called the different microservices.
6. Evaluation

This chapter is reserved for describing the process of testing and evaluating the proof of concept implementation of an MMORPG backend. The primary goal in this process was to verify that the system is working as expected and is able to meet the requirements it was designed to work with. For this reason, the author of this thesis performed a rigorous evaluation by employing three different test strategies: *unit testing*, *service testing* and *end-to-end testing*. The game backend was also compared against the principles of a microservice architecture pattern to find out possible similarities and deviations.

6.1 Test strategies and known limitations

The test strategies used in this master’s thesis were selected by following the microservice architecture testing guidelines of Clemson (2014). However, in place of *component testing*, the author of this thesis uses a term *service testing* because it better reflects the intend of testing a microservice through its API. Consequently, the test pyramid used within the study looks like the one illustrated in Figure 16.

![Test Pyramid Diagram]

*Figure 16. The test pyramid is a concept which means that the number of tests for a system should increase when walking down the pyramid (adopted from Clemson, 2014).*

In theory, the author of this thesis could have performed even more rigorous evaluation by adding a separate performance tests to the equation. But in practice, the amount of time and money required for hosting a live environment of the system were not available. Hence, executing performance tests against an un-optimized system would have produced false results. After all, the emphasis of the study was to find out whether microservice architecture pattern is applicable to a design of an MMORPG backend rather than implementing most performant versions of the used algorithms. For the same reason, existing solutions were used for those parts of the system which does not have to be made from scratch. The author concludes that employing three different testing strategies appeared to be enough for supporting the work of solving the research problem of the thesis.
## 6.2 Results of testing the system

The actual implementation and testing of the system was carried out using the power of *Ruby on Rails* (RoR) extensively. RoR is a popular web-application framework which is known for its ability to allow fast prototyping of applications but also running of large-scale production systems. In addition to RoR, the author decided to leverage the possibility of implementing different services using a different technology stack (Newman, 2015). However, rather than doing anything crazy, the author concluded that selecting one additional technology stack would be enough to validate the promise of a microservice-based architecture. In the end, the author ended up using *Node.js* because he has extensive knowledge on working with *JavaScript*.

### Table 9. A list of tools which were used in the development and testing phase of the system.

<table>
<thead>
<tr>
<th>Product</th>
<th>Ruby on Rails (v.5.1.4)</th>
<th>Node.js (v.7.2.1)</th>
<th>Testing framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account service</td>
<td>x</td>
<td></td>
<td>RSpec (v.3.6)</td>
</tr>
<tr>
<td>Discovery service</td>
<td>x</td>
<td></td>
<td>RSpec (v.3.6)</td>
</tr>
<tr>
<td>Game website</td>
<td>x</td>
<td></td>
<td>RSpec (v.3.6)</td>
</tr>
<tr>
<td>Realm service</td>
<td>x</td>
<td></td>
<td>RSpec (v.3.6)</td>
</tr>
<tr>
<td>API gateway</td>
<td>x</td>
<td></td>
<td>Mocha (v.3.5.3)</td>
</tr>
<tr>
<td>Zone service</td>
<td>x</td>
<td></td>
<td>Mocha (v.3.5.3)</td>
</tr>
<tr>
<td>Logger service</td>
<td>x</td>
<td></td>
<td>Mocha (v.3.5.3)</td>
</tr>
</tbody>
</table>

Table 4 summarizes the main technologies which were used in the implementation and testing phase of the system.

### 6.2.1 Comparison against the requirements

The MMORPG backend was evaluated by analyzing how it corresponded to the detailed requirements that were engineered before and during the implementation phase of the system. Each requirement was marked with a status of having been achieved, have not been achieved or have not been needed (see Table 10). The system’s ability to meet the detailed requirements are explained in more detail below.

### Table 10. Status of the evaluated requirements. (Achieved (✓), Not achieved (✗), Not needed (☐)).

<table>
<thead>
<tr>
<th>R-1: Create account</th>
<th>R-2:</th>
<th>R-3:</th>
<th>R-4:</th>
<th>R-5:</th>
<th>R-6:</th>
<th>R-7:</th>
<th>R-8:</th>
<th>R-9:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-10:</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-20:</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-29:</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-30:</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-31:</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-32:</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-33:</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-34:</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-35:</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-36:</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**R-1**: Create account  
**Status**: Achieved  
**Evaluation**: A player was capable of creating an account. The account creation was evaluated by writing unit/service -tests for the *Account service* and *Game website*. A successful API call to the *Account Service* returned the account as a JSON response, and
triggered sending an email with an activation link pointing to the address of the *Game website*. A separate end-to-end test was also performed from the *Game website* when the API gateway was directing the requests.

**R-2: View account**  
**Status:** Achieved  
**Evaluation:** A player was capable of viewing account information. Viewing an account was evaluated by writing unit/service -tests for the *Account service*. A successful API call returned the requested account as a JSON response.

**R-3: Edit account**  
**Status:** Achieved  
**Evaluation:** A player was capable of editing account information. Editing an account was evaluated by writing unit/service -tests for the *Account service*. A successful API call returned the modified account as a JSON response. Hence, trying to edit the details of someone else than the account owner returned *403 Forbidden*.

**R-4: Delete account**  
**Status:** Achieved  
**Evaluation:** A player was capable of removing account. Deleting an account was evaluated by writing unit/service -tests for the *Account service*. A successful API call changed *deleted=true* in the accounts –table and blocked further login requests. Hence, trying to remove an account of someone else than the account owner returned *403 Forbidden*.

**R-5: Log in**  
**Status:** Achieved  
**Evaluation:** A player was capable of logging in. Log in was evaluated by writing unit/service -tests for the *Account service*. A separate end-to-end test was also performed by first making an authentication request to *Account service* to receive *access_token* and then using the same token to log in to *Zone service*.

**R-6: Log out**  
**Status:** Not needed  
**Evaluation:** The system uses JWT-based *access_token* for authentication and therefore separate log out API does not have to be implemented. The logging out from the system is performed at client-side by disposing the token.

**R-7 / R-8 / R-9: Listing, selecting and registering game servers**  
**Status:** Not needed  
**Evaluation:** The system used *Discovery Service* for registering and deregistering all microservices. For this reason, the requirements for R-7, R-8 and R-9 was considered to be duplicate requirements for R-27, R-28 and R-29.

**R-10: Create character**  
**Status:** Achieved  
**Evaluation:** A player was capable of creating a character. The character creation was evaluated by writing unit/service -tests for the *Realm service*. A successful API call returned the character as a JSON response.

**R-11: Load character**  
**Status:** Achieved  
**Evaluation:** A player was capable of loading a character. The character loading was evaluated by writing unit/service -tests for the *Realm service*. A successful API call returned the requested character as a JSON response.
R-12: Connect to game world
Status: **Achieved**
**Evaluation:** A player was capable of connecting to game world. The connecting to game world was evaluated by writing a service test for the *Zone service*. A successful connection request returned the whole game world snapshot which contained initial details about the players currently residing at the server.

R-13: Save character progress
Status: **Achieved**
**Evaluation:** A player was capable of saving character progress. The character creation was evaluated by writing unit/service -tests for the *Realm service*. A successful API call returned the updated character as a JSON response.

R-14: Calculate game state
Status: **Achieved**
**Evaluation:** The system was capable of calculating game state. The calculation of the game state was evaluated by writing unit test for the *Zone service*. A successful calculation of the game state updated in-memory state of the stubbed players.

R-15: Distribute game state
Status: **Achieved**
**Evaluation:** The system was capable of distributing the game state. The game state distribution was evaluated by writing unit tests for the *Zone service*. A successful distribution of game state send an outgoing packet containing current state of the players and enemies.

R-16: Basic movement
Status: **Achieved**
**Evaluation:** A player was capable of moving in the game. The player movement was evaluated by writing unit tests for the *Zone Service*. A successful player movement managed to turn given x/y –coordinates to a new player location. The zone service performed slight validation for the coordinates so that moving too far was not allowed.

R-17: Player vs enemy combat
Status: **Not achieved**
**Evaluation:** The evaluation of the player vs enemy combat was postponed too long to be evaluated on time. However, the author of this thesis argues that the implementing a basic combat APIs to the *Zone service* would have been trivial.

R-18: Chat
Status: **Achieved**
**Evaluation:** A player was capable of chatting in the game. The chat was evaluated by writing a unit test for sending and receiving messages at the *Zone Service*. The filtering of chat messages based on player locations was considered but not tested.

R-19: Log events
Status: **Achieved**
**Evaluation:** The system was capable of logging events. The event logging was evaluated by writing a unit/service -tests for the *Logger service*. An event was considered successfully logged when it was received over a socket connecting and populated to the database. The author of this thesis concludes that more rigorous evaluation for event logging is needed in a proper environment.
R-20: View logs
Status: Achieved
Evaluation: A developer was capable of viewing logs. The event viewing was evaluated by writing a unit/service-tests for the Logger service. For the sake of simplicity, the viewing of logs was implemented by returning only 100 latest logs from the database against an API call.

R-21: Cross-platform support
Status: Achieved
Evaluation: The system was capable of supporting cross-platform clients. The cross-platform support was evaluated by using a web browser for connecting to the Zone service. Connecting to the socket-based system using a native application was not tested, but documentation claimed that to be possible.

R-22: Authorization
Status: Achieved
Evaluation: The system was capable of authorizing users. The user authorization was evaluated by writing a service test for the Account Service in which a user with a role=player tried to suspend another player. The service returned 403 Forbidden.

R-23: Unified API access
Status: Achieved
Evaluation: The system was capable of providing unified API access. The unified API access was evaluated by writing unit/service-tests for the API gateway. A successful API call managed to return a response from the requested microservice.

R-24: Activate account
Status: Achieved
Evaluation: A player was capable of activating an account. The account activation was evaluated by writing unit/service-tests for the Account service and Game website. A successful API call executed from the Game website rendered a message to the user for stating that the account is now activated.

R-25: Suspend account
Status: Achieved
Evaluation: A player was capable of suspending an account. The account suspension was evaluated by writing unit/service-tests for the Account service. A successful API call managed to add a penalty to the given account as a parameter. The further login requests for that account were prohibited for the duration of the suspension.

R-26: List accounts
Status: Achieved
Evaluation: A developer was capable of listing accounts. The account listing was evaluated by writing unit/service-tests for the Account service. A successful API call returned the list of accounts as a JSON response.

R-27: Register service
Status: Achieved
Evaluation: The system was capable of registering a service. The service registration was evaluated by writing unit/service-tests for the Discovery service. A successful API call stored the service address to the database and returned 201 Created.

R-28: Deregister service
Status: Achieved
**Evaluation:** The system was capable of deregistering a service. The service deregistration was evaluated by writing unit/service -tests for the *Discovery service*. A successful API call removed the service from the database and returned *200 OK*.

**R-29:** Get service address  
**Status:** Achieved  
**Evaluation:** The system was capable of returning a service address. The returning of service address was evaluated by writing unit/service -tests for the *Discovery service*. A successful API call returned the service address as JSON response.

**R-30:** Marketing page  
**Status:** Achieved  
**Evaluation:** A user was capable of viewing the game marketing page. The marketing page was evaluated by simply rendering empty view from the *Game website*.

**R-31:** Monitoring  
**Status:** Achieved  
**Evaluation:** The system was capable of monitoring microservices. The monitoring was evaluated by running *Discovery service* and *Account service* simultaneously so that *Discovery service* was able to ping the health check URL of the *Account service*. The *Account service* returned *200 OK* responses.

**R-32:** Load game items  
**Status:** Achieved  
**Evaluation:** The system was capable of loading game items. The loading of game items was evaluated by writing unit/service -tests for the *Realm service*. A successful API call returned the list of items as a JSON response.

**R-33:** Load game enemies  
**Status:** Achieved  
**Evaluation:** The system was capable of loading game enemies. The loading of game enemies was evaluated by writing unit/service -tests for the *Realm service*. A successful API call returned the list of enemy types as a JSON response.

**R-34:** View inventory  
**Status:** Achieved  
**Evaluation:** A player was capable of viewing inventory. The viewing of inventory was evaluated by writing unit/service -tests for the *Realm service*. A successful API call returned the list of inventory items as a JSON response.

**R-35:** Collect a game item  
**Status:** Achieved  
**Evaluation:** A player was capable of collecting a game item. The collecting of game item was evaluated by writing unit/service -tests for the *Realm service*. A successful API call returned the inventory item as a JSON response.

**R-35:** Drop a game item  
**Status:** Achieved  
**Evaluation:** A player was capable of dropping a game item. The dropping of game item was evaluated by writing unit/service -tests for the *Realm service*. A successful API call removed the inventory item from the player and returned *200 OK*. 
In conclusion, the author of this thesis argue that the designed system managed to meet the requirements it was designed to work with. However, the author also recognizes that performing more rigorous evaluation in a proper cloud-based environment could have produced new insight about the behaviour of the system.

6.2.2 Comparison against the design principles

The MMORPG backend was also evaluated by analyzing how it corresponded to the principles of the microservice architecture pattern that were selected from Fowler and Lewis (2014). The purpose of this comparison was to verify that the author of this thesis designed the system using the microservice architecture pattern as it is currently described by the industry experts. Consequently, the purpose of the comparison was by no means to be as rigorous as possible because architectural design by nature leaves room for own interpretation. Nevertheless, the system’s ability to meet the design principles are explained below.

**DP-1: Componentization via services**

The designed system favor componentization via services rather than using shared libraries. Hence, the shared libraries in this context was understood as domain-related libraries which are developed by the team itself. Meaning, the system does rely on the usage of JWT at different microservices for authorization purposes. The reason is that the well-established library helps securing the system and should not be implemented from scratch.

**DP-2: Organized around business capabilities**

The system was designed as such that it could be organized around business capabilities. For this reason, the author of this thesis made his best efforts for splitting the requirements to different microservices using the idea of loose coupling and high cohesion (Newman, 2015). In reality, however, because every business is different, following the principle accurately depends on the company structure as well.

**DP-3: Products not projects**

The designed system could not be compared in terms of products not projects - mentality. The reason is that following the principle requires that there are multiple teams working with the system – in a production environment – for a longer period of time.

**DP-4: Smart endpoints and dumb pipes**

The designed system was well decoupled by making each microservice to expose APIs for others to consume. These APIs were exposed over HTTP/REST or using WebSocket-based connections.

**DP-5: Decentralized governance**

The designed system was implemented using two different technology stacks: RoR and Node.js. Therefore, the author of this thesis managed to evaluate that different parts of the system can be implemented using tools which are best for the job.
DP-6: Decentralized data management

The different microservices in the system were responsible for managing their own database schema. This means that the other microservices in the system are not allowed to alter the data in any other way than using the provided APIs.

DP-7: Infrastructure automation

The infrastructure automation and related functionalities were considered to be outside the scope of this thesis. The design and development of the proof of concept implementation of the system appeared to be a huge job for a one man to handle. In any case, the author argues based on his previous experience that automating the system should begin by selecting a proper cloud-based provider and preparing different projects for container-based deploys. Good tools for helping the infrastructure automation would be the use of Docker and Jenkins.

DP-8: Design for failure

The designed system was partially designed for failure because the requirements of the system was split in a way that crash of a single microservice would not produce a bad user experience nor require every game client to be dropped out from the game. For instance, crash of an Account service could be easily handled at the client-side by stating that the server is down at the moment. Moreover, the Discovery service was designed as such that it could be extended for providing email alerts from the crashes by inspecting the results ongoing health checks.

In the final analysis, the author concludes that the system was designed following the microservice architecture pattern but studying the use of the architecture in a real company could have produce better results. However, because such opportunity was not available the research was conducted as a solo project.
7. Discussion

The objective of this master’s thesis was to study how MMORPG backend can be designed using the microservice architecture? In this context, MMORPG backend means a server-side game system which needs to be in place for thousands of people to connect and role-play in a virtual world (Chen & Muntz, 2006; Ferrara et al., 2010). On the contrary, an application based on the microservice architecture means a distributed system which is composed by a set of services, each of them designed to perform separate and well-defined task (Newman, 2015). The research problem was studied by approaching it with a design science research methodology. The methodology was applied by building a proof of concept implementation of an MMORPG backend which was developed according to principles of a microservice architecture pattern. These principles along the requirements of the system were derived from the existing literature because the research was not conducted in a company setting. The completed system was rigorously evaluated by comparing the design against the principles, and the functionality against the requirements using three different test strategies: unit testing, service testing and end-to-end testing. The results of the evaluation showed that the system was able to meet the requirements and respect the principles of the microservice architecture pattern. With them, the research problem of this thesis could be addressed by reflecting the results of evaluation, the process of developing, and the structure of the system to existing literature on designing MMORPG architectures and using the microservice architecture pattern in similar applications.

The research question of this thesis was: how MMORPG backend can be designed using the microservice architecture? Above all, designing an MMORPG backend using the microservice architecture pattern boils down to properly conducted requirements engineering process. A one size fits all solution is not going to exist because each game have different requirements. For instance, Eve Online is a space MMORPG which allows a player to build her own ship and explore a galaxy consisting of more than 7000 star systems (Eve Online, 2003). On the contrary, World of Warcraft is a fantasy MMORPG which allows a player to portray herself as a customized character and explore a landscape containing interesting quests, dangerous monsters and friendly NPCs (World of Warcraft, 2004). For the same reason, the author of this thesis needed to develop a set of detailed requirements for the proof of concept implementation of the system.

However, the mistake the author did was that those requirements should have been defined more rigorously using a real game design. Namely, reasoning the requirements of the game backend without designing a real game on top of it caused a lot of difficulties. Even so, similar approach was used by Cardin (2016) while designing an horizontally scalable backend for mobile games using the microservice architecture. In this study, the requirements of the system are identified by analyzing most popular mobile games in the market (Cardin, 2016). Consequently, the end result is that both of the backend architectures might be too bloated and hard to integrate with using a real game client. For this reason, implementing a game client should be preferred over using automated tests – as in this thesis.

The another mistake the author did was that he assumed that developing microservice-based system rather than monolithic application at first could work. Consequently, the author wasted a lot of time in trying to split the requirements to different microservices.
using the idea of **loose coupling** and **high cohesion** (Newman, 2015). The problem was threefold. First, the author had no prior experience on modelling a microservice-based system so each service ended up being responsible of either too much or too little of the overall functionalities. Second, the author had no prior experience on developing an MMORPG backend having the exactly same set of requirements, so previous knowledge of organizing application code to different services could not be used as a reference. Third, the author did not understand the requirements well enough to be able to map them to appropriate microservices; Of course, conducting more rigorous requirements engineering process could have alleviated the problem, but the author argues that requirements are never fully understood until first lines of code gets written. Interestingly, however, further investigating the issue revealed that Newman (2015) also suggest starting the development of the system using a monolithic architecture.

In terms of implementing a game client, the challenge in this thesis was the scope of the research. The previous research on designing MOG architectures focuses only on a fraction of the things which are required by a MMORPG backend. Meaning, the existing research focuses primarily on real-time communication taking place inside a game world in a form of P2P, CS, CMS and PP-CA architectures (Pellegrino & Dovrolis, 2003; Smed et al., 2002; Yang & Sutinrerk, 2007) or application-level techniques (Gargolinski et al., 2005; Glinka et al., 2007; Yahyavi & Kemme, 2013). However, the MMORPG backend designed by the author of this thesis is more comprehensive because the solution included features such as logging, account management, authentication, persisting game state and more. Hence, the real-time communication referred by the previous researches is a job of a single microservice, known as the **Zone service**. Obviously, for the same reason the author did not find significant ways to improve the behavior of zone service in terms of what is already known through previous researches. The reason was that the purpose of the implementation phase was to focus on prototyping rather than implementing most performant versions of the used algorithms.

In the end, the designed MMORPG backend consists of five different microservices, an API gateway and a game website. **Game website** was used for marketing the game and creating user accounts so that the registration flow was centralized to the web rather than implementing the same functionality to multiple game clients. **API gateway** was used for providing a single point of entry to the system by proxying and routing requests to underlying microservices. **Discovery service** was used for locating microservices from the network by registering IP addresses and performing health checks. **Account service** was used for managing user accounts and returning short-lived access tokens in response to successful authentication attempts. **Realm service** was used for storing the business logic of the game and the progress of the players. **Zone service** was used for implementing real-time functionalities of the game by calculating and distributing the game states. Finally, **Logger service** was used for collecting logs to a centralized place so that tracing errors and detecting suspicious behavior in the system was easier for the developers.

The amount of microservices used in the system was less than in Cardin (2016). Hence, despite these solutions are not directly comparable, the author of this thesis learned in the process of developing the system that adding more microservices should be avoided as long as possible. The reason was that implementing the communication taking place between the microservices started after a while to be a pain in terms of testing and development effort. Therefore, the focus quickly shifted from creating new services for trying to keep the service count to a minimal. The author of this thesis concludes that it is very easy to plan high number of services on paper, but once the implementation phase begins, the problem is realized. Hence, Cardin (2016) in particular did not implement the system in practice.
Finally, one interesting finding during the development system was the fact that using microservice architecture in the context of MMORPGs challenges the way services can be split. The reason is that in MOGs the communication taking place between a game client and the server relies on the calculation of a game state which should be shared between the players (Chen & Maheswaran, 2004). This made splitting microservices more difficult. In the context of this thesis, Chat service was merged to be a part of Zone service because otherwise sending chat messages to a players nearby would not have been possible. The problem could be partially solved by sharing the game state at the server-side between services, but managing such system could cause more bugs than answers. The author of this suggests that only a single service should be responsible for calculating and distributing the game state.

In conclusion, there are multiple steps in the process of designing an MMORPG backend using the microservice architecture pattern, but the requirements engineering is the most important. Having a detailed understanding about the game requirements makes it possible to construct a distributed system in which multiple microservices works together to serve player goals. Hence, the architectural design developed in this research helps game developers to construct their very own MMORPG backend.
Conclusions

The objective of this master’s thesis was to study how MMORPG backend can be designed using the microservice architecture? In this context, MMORPG backend means a server-side game system which needs to be in place for thousands of people to connect and role-play in a virtual world. On the contrary, an application based on the microservice architecture means a distributed system which is composed by a set of services, each of them designed to perform separate and well-defined task. The research problem was studied by approaching it with a design science research methodology. The methodology was applied by building a proof of concept implementation of an MMORPG backend which was developed according to principles of a microservice architecture pattern. The completed system was rigorously evaluated by comparing the design against the principles, and the functionality against the requirements using three different test strategies: unit testing, service testing and end-to-end testing. The results of the evaluation showed that the system was able to meet the requirements and respect the principles of the microservice architecture pattern. With them, the research problem of this thesis could be addressed by reflecting the results of evaluation, the process of developing, and the structure of the system to existing literature on designing MMORPG architectures and using the microservice architecture pattern in similar applications.

The results showed that there are multiple steps in the process of designing an MMORPG backend using the microservice architecture pattern, but the requirements engineering is the most important. Having a detailed understanding about the game requirements makes it possible to construct a distributed system in which multiple microservices works together to serve player goals. Hence, the architectural design developed in this research helps game developers to construct their very own MMORPG backend.

The future work is still needed for conducting more detailed performance tests of the overall system in a production environment, such as Amazon Web Services or using equivalent cloud-provider. Also, implementing a proper game on top of the system could born new design knowledge and help to fix problems.
References


