Design of a horizontally scalable backend application for online games

Christian Cardin

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 10.10.2016

Thesis supervisor:

Prof. Antti Ylä-Jääski

Thesis advisors:

M.Sc. Teemu Kämäräinen

M.Sc. Mikko Virkkunen

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Christian Cardin

Title: Design of a horizontally scalable backend application for online games

Date: 10.10.2016 Language: English Number of pages: 6+83

Department of Computer Science

Professorship: Mobile Computing - Services and Security

Supervisor: Prof. Antti Ylä-Jääski

Advisors: M.Sc. Teemu Kämäräinen, M.Sc. Mikko Virkkunen

Mobile game market is increasing in popularity year after year, attracting a wide audience of independent developers who must endure the competition of other more resourceful game companies. Players expect high quality games and experiences, while developers strive to monetize. Researches have shown a correlation between some features of a game and its likelihood to succeed and be a potential candidate to enter the top grossing lists.

This thesis focuses on identifying the trending features found on the current most successful games, and proposes the design of a scalable, flexible and modular backend application which integrates all the services needed for fulfilling the common needs of a mobile online game.

A microservice oriented architecture have been used as a basis for the system design, leading to a modular decomposition of features into small, independent, reusable services. The system and microservices design comply with the Reactive Manifesto, allowing the application to reach responsivness, elasticity, resiliency and asynchronicity. For its properties, the application is suitable to serve on a cloud environment covering the requirements for small games and popular games with high load of traffic and many concurrent players.

The thesis, in addition to the application and microservices design, includes a discussion on the technology stack for a possible implementation and recommended setup for three use case scenarios.

Keywords: scalability; online games; microservices; architectural design

Acknowledgements

I would like to thank Aalto University for the invaluable learning opportunities and life experiences I had while accomplishing my master studies.

Thanks also to my advisors Mikko Virkkunen and Teemu Kämäräinen for their feedbacks and guidance on this thesis work.

I am grateful to all my colleagues at Tunnel Ground Oy who gave me the inspiration for this project and their assistance in everything.

I wish to give a heartily thank to my family and all my dearest friends for the constant encouragement and support, and for being with me in this awesome adventure.

Otaniemi, 10.10.2016

Christian Cardin

Contents

Symbols and abbreviations

Abbreviations and Acronyms

API	Application Program Interface
SDK	Software Development Kit
P2P	Pay To Play
F2T	Free To Play
V2P	View To Play
IAP	In App Purchases
Ads	Advertisements
MMO	Massively Online Multiplayer
MMORPG	Massively Online Multiplayer Role Playing Game
FPS	First Person Shooter
RPG	Role Playing Game
RTS	Real Time Strategy
RDT	Round-trip Delay Time
AI	Artificial Intelligence
CRISP-DM	Cross Industry Standard Process for Data Mining
SOA	Service Oriented Architecture
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
MQTT	Message Queue Telemetry Transport
AMQP	Advanced Message Queuing Protocol
RMI	Remote Method Invocation
IPC	Inter Process Communication
TCP	Transmission Control Protocol
ACID	Atomicity, Consistency, Isolation, Durability
BASE	Basic Availability, Soft-state, Eventual consistency
CAP	Consistency, Availability, Partition tolerance
XP	Experience Points
MMR	Matchmaking Rating
QoS	Quality of Service
TTL	Time To Live
RBI tree	Red-Black Interval tree

1 Introduction

1.1 Digital Game Market

Games are currently playing a major role in the mobile market ecosystem. With the advent of new technologies like smartphones, the evolution of the Internet and social networks, games have seen a drastic change in almost every aspect from implementation to delivery. One change was the decline of retail shops in favor of digital downloads from application stores or online games played through a browser [8]. The contribution of Facebook was particularly significant for the diffusion of online games since 2007, when it opened its platform to third party developers and released social APIs to let the player interact with their friends, for example by sending gifts or inviting other people to play. The "friend" concept begun to appear directly in the game mechanics, making it part of the core loop and, therefore, encouraging people to involve as many friends as possible for a better experience and advantages[13]. Thanks to this particular design, some games became viral and earned a consistent capital in a very short time. A case example is Farmville, by Zynga, who reached 10 millions daily active users in only six weeks after the launch [31]. After Facebook, another great contribution to the ecosystem came from Apple and Android when they introduced their app stores in 2008. From that moment onward, games quickly climbed the rankings to the most downloaded apps.

A completely new ecosystem stimulated new ideas and business models for enhancing the monetization of games. Nowadays there are different models [37] which suit different market sectors, but the most popular one is the Free to Play, also referred with the acronym F2P. According to the F2P model, a user can acquire and play a game free of charge and optionally purchase virtual goods to enhance the experience. The advantage from the user's point of view is immediate because he has access to the product without having to pay beforehand, but for the developer there's a bigger risk since the profitability is less certain. Based on the theory of engagement, the longer a user plays, the more chances he has to buy virtual items, so developers started to put emphasis on experience as a way to increase their revenue.

1.2 Typical requirements for a modern game

To have a deeper connection with players, the game should emphasize emotional commitment through narrative techniques, introduce customizable elements, deliver a high quality gameplay, induce players to come back regularly with assiduity rewards and new updates. The engagement should be assessed by continuously collecting analytic data and tuned iteratively. [13] For this reason, modern games have started to integrate a series of services aimed to enhance the player's game experience. Research shows [26, 40, 15] that the most important feature for the majority of players is the ability to interact with friends, to compete or collaborate with them towards a goal. A typical way to introduce competition is by ranking players on a public leaderboard, and compare them with their closest friends or with the other players in a local geographic area. Instead of competition, there can be cooperation

by implementing a mechanic which allow one player to send gifts to his friends, also contributing at the virality of the game itself. The possibility to create clans and alliances with other players is a great improvement for the social aspect [38], players will feel more involved and prone to a long term commitment with the game. For "collector" players, a common option is to provide achievements and special prizes when they manage to accomplish a specific goal in the game. This is particularly useful for increasing the satisfaction and retention. Moreover, people tend to play games in many devices tied to a specific account, so the progress should be portable from a device to another, with an option to resolve an eventual conflict between game states. Online personal inventory is an option that brings a considerable advantage for both players and the company. From the player side, players can manage their virtual goods from a browser, purchase directly from an online store, trade and sell items to others. From the company side, the company gather useful statistics on how the game is performing and respond quickly to discrepancies in the game design.

At last, there is the multiplayer support. The definition of multiplayer games can be summarized as a network of social interaction in which players (human actors) interact both with each other and with the system (nonhuman actors). At the time of write, summer 2016, multiplayer games ranks on top of the grossing charts of both Android and iOS, and so they did in 2015 [5]. Games can offer diverse multiplayer experiences. The most common one is realtime multiplayer, where two or more players play in the same virtual environment and the action of one player has an immediate effect on the others. This is the case of massively multiplayer online games (MMOG) like the famous World of Warcraft where millions of players impersonate a hero and proceed through quests to gain experience and stronger items. The players move in a vast virtual world and the actions of one player are instantaneously synchronized with the other players. There are also turn-based multiplayer games, like the lucky card game HearthStone. In this type of games, the gameplay proceeds by discrete actions executed one after another, in turn, by the players. The final state of the game is synchronized at the end of the turn or after every performed action. Some of these games need a matchmaker in order to gather the necessary players together, choosing from a pool of players waiting to join an instance, or "match" of the game. Lastly, there are other types of games which have real-time and turn-based elements mixed together.

1.3 Recurrent needs, not a unified way to deal with them

Currently, there are some providers offering similar services to users who integrate their SDK into the application. The most known are Google Play Games, Apple GameCenter and Steam. But they are not the only one, there are other less known providers offering similar content including Amazon GameCircle, Microsoft and Samsung. All of them expose accessible APIs or distribute an SDK, but with different interfaces to the services and data formats. Often the feature set changes from a provider to another, or they only expose a partial set of services. Implementing all the providers' technologies into one game is often a desirable business requirement, but very time consuming for the developers. Integrating many third party development

kits in the same project requires an additional layer of abstraction to achieve a seamless access from the core logic, but it increases the overall complexity. Moreover, developers have to create special test cases and environments to ensure the same behaviour in different platforms. Another limiting factor is that most of these services are not meant to be cross platform. As an example, Apple restricts the use of its services only to Apple devices. In general, developers who rely on third party services don't have full control over the data they collect from the users, not even retain the ownership. Game companies must adapt to the provider's terms, conditions and technical limitations, and accept the compromise between imposed restrictions and customization freedom.

There are many game Backends-As-A-Service solutions available on the Internet [41], offering a complete suite of tools to fulfill all the features a modern online game may require. Although these game backends solve most of the common problems relative to the backend implementation, they are commercial products and none of them allows the developer to really customize the system in all its parts. Game BaaS may be a worthwhile choice for companies without special requirements on backend, who don't care about vendor lock-in and ready to invest a consistent capital in the long run. There is a clear lack of options for those companies who need a tailored game backend solution, who want full control over the system and the data, who want to fine-tune performance and scale operations at a microservice level. There are frameworks which partially address the problem but far from being complete, the project is discontinued or documentation is not fully available in English, or not available at all. Example of game frameworks are: Pomelo[1], NugServer[2] and Scut[3].

1.4 Architectural design challenges

There are some technical requirements to take into consideration for implementing a system capable of providing all the common features desirable in an online game. First of all, there is the problem of abstraction. Games are so different that it is not possible to make any assumption on the data model or mechanics. Every aspect of the system must be designed to be data agnostic, from storage to algorithms, so the developers can use it regardless of their specific data model. This imposes a restriction on how the database is accessed, queried, and the kind of operations that can be done with the game's specific business data. However, some kind of data format must be agreed in order to have a clear protocol for interfacing different services.

It is clear how a distributed and concurrent application is more resilient, reliable and scalable than a monolithic counterpart. But these advantages come at the price of simplicity: the architecture becomes more complex as different parts of the systems could be running on separate machines connected in a network. As a consequence, the access to resources can't be controlled and synchronized similarly to a normal application. Additionally, network communication is expensive in execution time

[1] `http://pomelo.netease.com/`
[2] `http://www.nugserver.io/`
[3] `http://scutgame.com/`

and can possibly fail due to many reasons, machines can crash and disappear from the network at any time, the state is not guaranteed to be always consistent. For mission-critical applications, uptime and recovery from failure is a non negotiable option which require extensive effort on operational tasks, especially for monitoring the application status and maintenance of the whole system. These issues become even more challenging when they are taken into the context of online games, mostly because of the high number of concurrent users, fast variations in user activity and near-realtime communication requirements.

1.5 Thesis Goals

The objective of the thesis is the design of a flexible, lightweight horizontally scalable game backend application that provides developers with many of the most common services, that they can deploy on commodity hardware or existing IaaS providers. The goals can be summarized in the following points:

1. Study the current mobile game environment to recognize and classify the most used services and their use cases.
2. Design a scalable backend application which exposes the aforementioned services. The design will focus on these five core concepts:
   (a) **Modularity:** the application must be composable and it must be possible to turn the services on and off on demand.
   (b) **Flexibility:** parts of the application can be customized, and it can also be extended with additional extra functionalities.
   (c) **Abstraction:** Storage, communication protocols and algorithms must work with any possible data model, imposing the least restrictions as possible.
   (d) **Scalability:** parts of the system must be able to scale up and down on demand in terms of resource utilization.
   (e) **Concurrency:** the system must be able to work concurrently with other components and able to manage the available resources.
3. Proposal of a setup for a minimal usage, a medium usage and intense usage
4. Technology evaluation for the actual implementation: platform, programming languages, databases, communication protocols, data formats.

2 Background, literature and theoretical focus

This section will introduce the reader to the topic covered in the thesis and explain the motivation behind the work: why games are important? What are the problems that games (and game developers) have in common? Why games need a scalable architecture and who wants it? What does it mean "horizontally scalable architecture" and why is it better from the others?

2.1 The Gaming industry

This chapter describes, in general, the vision and goals of the modern game industry. It will focus on the impact that games have in our society and how we benefit from them, for example for entertainment, learning, scientific and technological advancement. The chapter will also cover the role of the industry in the economic market and why there has been an increasing interest in games in the recent years, the current trends and the possible evolution in the next future given the latest data available from the recent statistics. Special mention to the current market distribution and fragmentation, focusing especially in the mobile ecosystem.

2.1.1 Beyond Entertainment

At the time of writing the game industry is riding the wave of a phenomenon that can be referred as "democratization of game development". For the first time ever, the dream of many independent developers, or wannabe game developers, is a reality: everyone can build a game in his garage. If compared with the situation during the 1990s and 2000s, an age dominated by console and PC games, the idea of creating a game as a hobby was mostly impracticable. Making a game was a very big deal, only few studios had the experience, knowledge and production structures required for the task, and acquiring this knowledge required a considerable investment in terms of time and money.

However, things have changed dramatically, especially in the last five years when the first game engines became available for the masses. A special mention goes to Unity Technologies, which greatly lowered the entry level barriers for the non specialized developers and game aficionados: they created an easy to use graphical editor and tutorials for moving the first steps into game programming. Since then, more and more tools became available throughout the years, more powerful but more accessible at the same time, that encouraged the flourishing of a vibrant community currently counting tens of millions of people. Thanks to the new tools and the increasingly interest around the indie game culture, a profound change has come bringing new and elaborate concepts, designs, business models and career opportunities. The economy is moving with the fingertips of game developers.

The influence of videogames on society it's not only limited at the entertainment and its relative market sector, but spans over other inherent disciplines as games improve. For example, games are the main reason for pushing forward the technology of graphic processors and the science of computer graphics, as new games tend to

look more photorealistic than the predecessor. In addition, handheld devices need more capabilities in order to fulfill the increasing hardware requirements of the newer games. For this, new algorithms must be researched and optimized to take full advantge of the limited resources available, also new materials are being tested for the next generation batteries to guarantee more autonomy. We are living in a connected world, and also games have evolved to take advantage of the thriving internet services available nowadays. Fast, reliable and lag-free connection is now an indispensable requirement for many online games, especially for realtime multiplayer games. For maintaining this condition, server infrastructures and technologies must evolve in order to keep the pace with bandwidth usage and ensure the maximum availability even with millions of daily active users. As a side note, high performance networks and infrastructures are required for a relatively new way of distributing high-end games into low-capability devices: cloud gaming.

Another important drive for technological advancement is design. Game designers are experimenting with innovative gameplay and interactions, which leads to completely new experiences for the player. Often, for making these interaction possible, it's necessary to invent new devices. One remarkable example is Kinect, a motion sensing input device by Microsoft for Xbox game consoles and PCs. Thanks to devices like Kinect and other affordable sensors, a research group of Aalto University is trying to use different body movements for the human-computer interaction, like jumping, kicking and dancing [22]. Another recent and interest project aims to develop an augmented climbing wall for bouldering [29] for improving climber's skills using also game-like elements; the research included studies on image recognition and path optimization, but also physiology and psychology. Enhancing performance in sports, like in climbing, need intense training to practice motion skills, building strength, endurance, knowledge and so on; it usually takes many repetitions before the desired level of competence is reached. Fortunately, technology united with gamification can help to make training less strenuous. Simulation games have come so closed to reality that games are even used to train professional pilots and drivers.

It's clear that computer games provide a compelling context for children's learning. Research [19] has advanced the idea that games can be effectively applied in a serious educational context, motivating students to learn in a more effective way that the traditional frontal lecture. Initially, educational games were not different from the typical academic exercises (spelling words, solve arithmetic problems), although somehow more appealing because of a higher degree of interaction, instant feedback and cool animated payoffs. But the real advantage comes in when the educational content is truly part of the gameplay, so children can exercise the targeted skills and knowledge in a natural and seamless way. In addition, this approach encourages children to see the educational content as useful and fun not because the content is presented alongside other appealing elements, but because they are having fun and achieving goals by using the educational content itself.

2.1.2 Trends and current evolution of the gaming ecosystem

Understanding the evolution and structure of the game industry has an enormous strategic and economic implication for all the parties involved in the gaming market sector. The game business market is highly dynamic thanks to the continuous technological advancement, which opened the road for a multitude of new participants, while other had to re-position themselves or adapt for not losing their place.

Compared to the entire entertainment market, video games occupy a relatively small portion of it, most directly competing against the movie and music industry. Many trends are promising in the current game industry, especially after the incredible proliferation of mobile games: in 2006 the industry was generating $10 billion [13] which grew to $91.8 billion in 2015 and $99.6 billion expected for 2016 (Statista, 2016). Impressive numbers, but still behind the astonishing $286.17 billion revenue for filmed entertainment. However, game market is expected to grow at a faster rate (+6.6% yearly) for the period 2015-2019 than film market ("only" +4.5%). While the film and music industry monetize from many different sources (direct sales, licences to broadcasters, rentals etc.), video games producers make money by selling directly to consumers, from subscription fees for online games like MMORPG, or by showing advertisements. In this context, videogame industry growth is relatively stable.

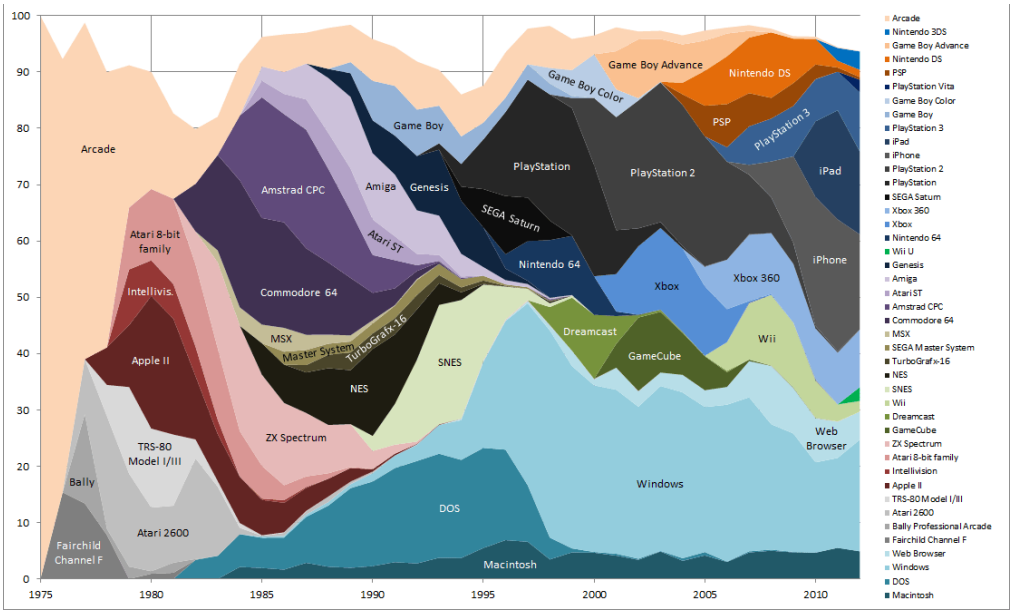


Figure 1: Distribution of game platforms over the years

Trends starting from 1990 onwards shown a strong constant growth of the gaming industry, despite the financial crisis of 2007. Previously, when the scene was mainly dominated by home console, the market worked in cycles of about 5 years: once a new generation of machine was released, it brought a burst in revenue due to the hardware sales and new games designed for them. Eventually, the most popular consoles would engage in "console battles" to gain the biggest piece of the market

share. An interesting point is that the bigger console manufacturers (Sony, Nintendo, Microsoft, Sega, and Atari) aren't generating much revenue from selling the hardware itself, since they're sold on a 5-year cycle. Indeed, the greatest part of the profit is coming from sales and distribution [36]. The development costs for a (good) console game rose from $17 million to about 20$ million in the last decade [37], sum that not many companies can afford. As a consequence, the publishers finance the development of video games for the platform and take the most of the income, and then pay back licensing fees to the manufacturer. This is the reason why fighting (and winning) for the market share is so vital for manufacturers: a large consumer base means large game sales, which then encourages developers to make more game for a specific platform.

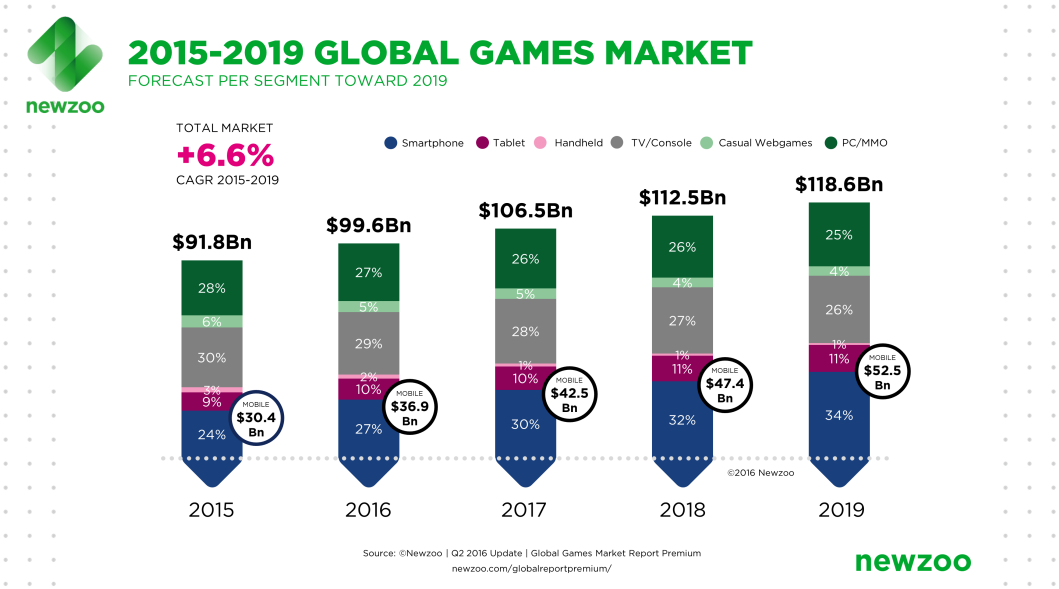


Figure 2: Global game market forecasts

Until the seventh-generation console cycle (2005), Sony led the market with their PlayStation 1 and 2, although challenged by the Xbox from Microsoft. Still, it was Nintendo's Wii that, after the failure of GameCube, took the lead with most analysts' surprise. Instead, Xbox 360 and PlayStation 3 fought hard for the close second place. In the latest console generation it seems Sony has come off to the best start with the PlayStation 4, whereas both Xbox One and Wii U are struggling to gain momentum. PlayStation has historically appealed more to the hardcore gamers, and many non-hardcore gamers are abandoning the consoles in favor of the vast mobile alternatives [16].

Indeed, mobile platform has seen a continuous, steady rise from 2006, quickly gaining an important piece of the gaming industry previously dominated by consoles. Everything changed in the year 2007 when Apple introduced the iPhone, the first smartphone to become a global hit and bringing also a profound change in the culture and society. But the merit behind the achievement should not be attributed only at

the iPhone itself, but to the App Store: the tightly platform-integrated system that enabled simple application downloads, management, and payments. The success of the App Store led to a massive entry of small companies and independent developers, in addition to encourage other mobile platform manufacturer to develop their own platform ecosystem. The latter was the case of Android who launched the Play Store a year later, in 2008. From that moment onward, mobile network operators lost their influence and control on the mobile ecosystem in favor of mobile platform manufactures and app developers. A new sector was born. Before, a developer who wanted to publish an application had to contact a mobile network operator to have their app visible on the operator's portal, often the only channel available for mobile users. But thanks to the introduction of an integrated app store, individual developers could deliver their products directly to the consumer, cutting all the costs involved in a longer product chain. App stores became the primary gateway for end users to mobile applications and content.

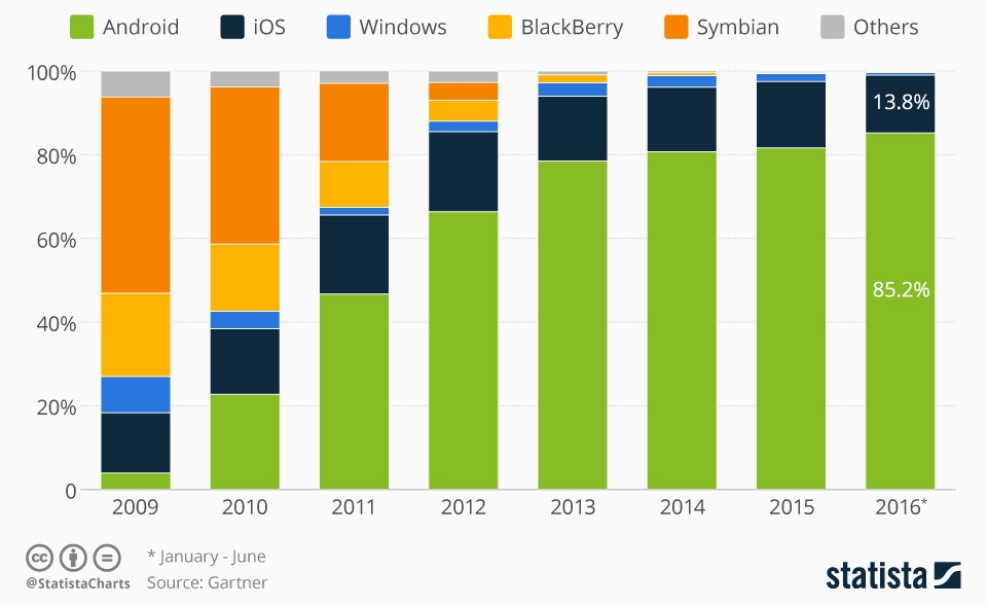


Figure 3: Worldwide smartphone operating system market share, based on sales (Statista, 2016).

After few years, the innovative ecosystem built around Android and iOS platforms crushed the competitors who didn't react in time to follow the trend, growing into the today's gigantic strongholds of the mobile market. The former bigger player was Nokia with its Symbian operating system, who ended a flourishing mobile phone business in April 2014 after ten years of success.

2.1.3 The business side of gaming

The growth of the videogames market favored the flow of an enormous amount of money making the fortune of many companies in the sector. In this very moment

millions of people are playing videogames, most of which are considered casual players. Casual player is the term used to define the profile of a person that plays games without a long-term commitment. They are not usually bound to a particular genre, and tend to prefer games with simple rules and short game loop. A game targeted to this kind of audience is called casual game. It's interesting noting that casual online games appeal are appealing for all audience segments, including hardcore PC gamers and mid-core players (Fig. 4). The prosperity of the mobile ecosystem, the popularity of app stores and the shift of interest towards mobile games favored the birth of new types of business models, pricing and contents. The most common are: Pay-to-Play (P2P), Free-to-Play (F2P), subscription, early access, hybrid, player to player trading. The graph shows most popular types of video games according to worldwide consumers in 2014. Based on Limelight Networks research, approximately 50 percent of respondents chose casual games such as Candy Crush Saga as their favorite type.

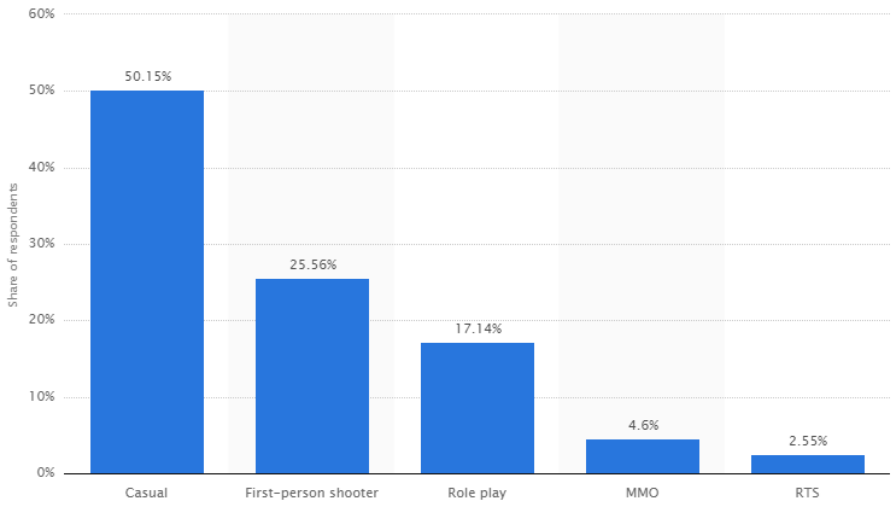


Figure 4: Most popular types of video games according to worldwide consumers in 2014, (Statista, 2016).

The P2P model consists, from the customer's point of view, in three distinct phases: first, the player buys the game (monetization), then he plays the game and discovers the gameplay (acquisition), finally he eventually engage in the game and keep playing (retention). In this case, the content is sold on a one-time premium purchase which lets the buyer to enjoy the full experience as a unique and encompassing service. The retention is only at the end of the process, but it's important because it can motivate the player to wait for the sequel or discover other products from the game's same franchise. With this model, the producer's intention is first to create demand for the product, and then understand the consumer behaviour watching at the consumption data. Players with willingness-to-pay higher or equal to the P2P price will buy the game, while others will not.

In contrast, F2P has a different scheme, also described with the ARM funnel [34]. The ARM funnel is a framework for analyzing the performance of a social network

game created by the company Kontagent. The player interaction is categorized in three main stages: Acquisition, Retention and Monetization, from here the acronym ARM.

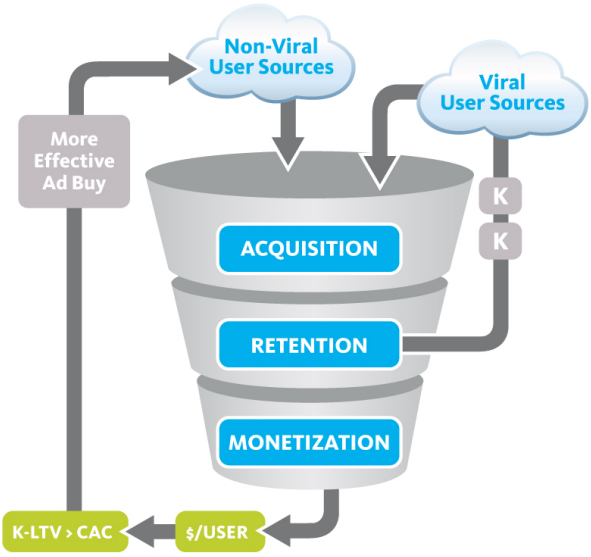


Figure 5: Acquisition, Retention, Monetization funnel

The first step, acquisition, is the process of generating new players for the game through two channels: viral and non-viral. The non-viral channel comprehends the category of users acquired from advertising campaigns, cross promotion, offer walls and other services where usually the company invests money in. On the other hand, the viral channel conveys all the users attracted by other users already in the game using word of mouth, sharing the game on social networks or by using a built-in referral system; usually there are no economical costs for the company to withstand. A good indicator for monitoring the game's ability on generating users through viral sources is the K-factor, which describes how many new users are gained through virality for each user. The second step, retention, measures the game's ability to maintain its existing users. Since most of F2P games get most of their revenue from active players (IAP or Ads), the profitability of the game can be related to how long a player spend time in the game. The parameters used for the analysis are sessions per user, average session length, average lifetime, retention rate (how often the player returns to the game). The final step, monetization, indicates how much revenue is generated from the players. There are many indicators of monetization, the most common are the average revenue per user (ARPU) and average revenue per paying user (ARPPU). Since, in F2P games, many players uninstall the game after the first time they play it, the number of downloads is not considerate a good indicator for success. Instead, developer prefer to look at the number of daily/monthly active users (DAU/MAU). The latter can be then combined with the aforementioned statistic to get ARPDAU/ARPMAU (average revenue per daily/monthly active users).

The main objective of the F2P games is to emphasize the experience before monetization. Thanks to the app stores, distribution costs for developers is almost zero, which allows player to get the game at no cost. Being the game free, the acquisition step is facilitated by the huge amount of potential users which can try the game at any time with no cost. If the player likes the game he will continue to play and, depending by his level of engagement, will eventually involve other friends to play with him. Actually, the first games to adopt the F2P model were online games directly integrated in social networks just to exploit the virality potential of a social platform. Based on the theory of engagement, the more a user plays the more likely is to buy some virtual item; so game designers aim to create more engaging core loops to lock in the user. This is why a F2P game requires a lot more effort in design and strategy than a normal P2P game Zynga found a gold mine when they launched the free-to-play game Farmville in 2009, reaching 10 million active users in only six months and, at its peak, 32 million people were playing the game every day [31]. The Farmville's success is attributed to its strong social component and they beneficial effect that friends have in progressing with the game. Thanks to this feature, the game underwent to a viral diffusion among Facebook users. For example, friends can speed up a player's progress up by allowing them to plant and yield crops instantaneously, or use them as helpers to automate some tasks. To cite Wright Bagwell, FarmVille's director of design: "This isn't just a virtual, traditional game board, It's a breathing world and everything is really interconnected. I grow my crops to feed my animals, my animals fertilize my crops, I need water to grow my crops and my trees, and it tells a nice story that you're trying to bring this farm back to life." When the player is engaged in the game, it may come the monetization step (M). Indeed, a research conducted by GameAnalytics [12] shows that only about 2% of the user totality will ever turn into paying users. The results are summarized in the table 1

	Population	**Amount spent**	**IAP revenue**
Non-monetizers	98%	$0	0%
Minnows	1%	$0 - $5	1%
Dolphins	0.8%	$5 - $50	12%
Whales	0.2%	More than $50	87%

Table 1: Monetization by user category. eMarketer, Q2 2016

The users are divided in different tiers based on how much they spend in a game. Minnows are low-core spenders, dolphins are mid-core and whales hard-core spenders. The big amount of non-monetizers can be explained by a typical behaviour of the average casual gamer: installing an app, trying it once, and uninstalling it. While the non-monetizers are prone to play several different games, dolphins and whales are more committed to a small number of games, in which they decide to invest time and money in it. A common strategy is try to elevate a non-paying user to the small group of paying users (becoming a minnow), and then inducing him to invest more to become a dolphin or a whale. The economic theory suggests that players are more

likely to purchase a virtual good if it's in line with their preference and the way they're experiencing the game. But games are complex environments in which the player's perception evolves with the game's progression in a very personal way, so game developers must must offer a wide variety of differentiated items to capture the maximum possible value from the users. Even if a player is converted to a paying user, there are no guarantees that it will continue, it greatly depends by his level of involvement and how deep the monetization mechanism is built in the game's core loop. Still, a large number of users is necessary for monetizing a F2P game. Looking at data coming from the games in mobile stores [17], 90% of them are free to play and they generate the majority of gross revenue. There are different monetization channels, more or less effective. Here are some of them:

Direct monetization: In-App Purchases (IAP)
Also referred as Microtransactions, IAP consist in direct sales through purchase of virtual goods inside the game. The list of most successful games using IAP as their primary revenue is usually indicated in the app store's top charts under the "top grossing" section. At the time of write, both iOS and Android top grossing charts are dominated by Pokemon GO (Niantic) generating a revenue of $2M per day, followed by RTS games like Game of War (Machine Zone), Clash of Clans (Supercell) and Candy Crush Saga (King).

Indirect monetization: Advertising
The game has some form of advertisement embedded into it. The more the number of players that plays the game, the more times the ads are impressed and clicked, thus the revenue from advertising is directly proportional to the user base's size. A good way to identify games who are monetizing with ads is looking at the top download charts. This monetization method is getting a considerable attention lately, so much that companies started to adopt a new business model focused on the monetization of video advertisements, called View-to-Play (V2P). The peculiarity about this model is that ads are not shown intrusively anymore, like static banners or interstitials, but player choose to watch a 30 seconds long video in exchange of a in-game reward or advantage. So far, the user reception was overwhelmingly positive with more than 50% [3] of players stating that it's their favourite way for "paying" the game. Once, players used to pay to remove ads, now they value games for having a built-in V2P mechanism. Revenues coming from this channel exceeded already the revenue coming from IAP in 2015, because video advertisements are targeted to the 97% of non paying users. (Futureplay, 2016).

Complementary monetization: Merchandise
As a side note, there are other ways in which a game can monetize. Some of them, for example, allow the players to create content and sell to other players through an in-game store in exchange of virtual currency or real money; game creators detain a percentage from any transaction. This is particularly advantageous for companies because they don't have to invest too much in creating the content, since the community will do it. Some companies with a strong brand can monetize with complementary

goods such as merchandise, books, movies or the brand itself. This is the case of Rovio, who earned the 45% of its $71 million profits from consumer products in 2012. Obviously, for both of these business model to work, the game must be reasonably popular.

Turns out that, unfortunately, even a free game is not enough to reach the visibility needed to ignite the monetization machine, especially in the last years. The app stores are so saturated with free to play games of all sort that one can not simply publish a game and wait for players to come. Since the mobile game environment is competitive, developers invest more time to create quality titles in terms of design, graphics, usability and gameplay. Furthermore, players who are accustomed to this rise in average game quality tend to be more critical when playing games, eventually underestimating a good game, relegating it as "cheap", because it does not have an AAA graphic. Small studios and individual developers struggle to compete against the more resourceful companies, and 90% of the startups in the gaming industry close after 2 years. There are some exception, though. A way for gaining visibility and acquiring users is through advertisement campaigns. But planning and running an advertisement campaign is costly and requires specialized knowledge but it's a necessary investment for facing the huge competition affecting the mobile market. Nowadays there are many services and advertisement providers which offer a full set of features to control and customize a campaign, for example by letting the developers to choose the target audience who may be more interested in the game.

2.2 Services in games

2.2.1 What users want

Understanding what's desirable for players gives a direction to game designer for creating a more enjoyable game that, hopefully, will have a better chance to engage and retain users. Directions turn into ideas, ideas shape a design, and design translates into requirements. At the last stage, after that the artistic process is complete, it's easier to analyze recurring patterns and find what are the services that games need in order to give players what they want.

Once upon a time, a japanese game designer with the passion of collecting insects decided to make a game that he would have liked to play. There was no business sense included, only his love involved in the creation. Somehow, what he created for himself managed to appassionate other people in his country, and then people in other countries as well. The game was not build to sell, nor to become viral. It was made only for the love of something, and to make something that others can love. That designer is Satoshi Tajiri, the creator of Pokémon. It's not possible to tell empirically what is enjoyable about a game and what is not. Creating experiences and emotions is an innate process, a form of art, that designers chase using their intuition and inspiration. For this reason, it's wrong to study "what players want" purely from a marketing point of view; a good, original game has always some inner spark in it, something unique and special that makes it different from anything else. In general, players don't want clones of other games, even if the game is "new"

in how it looks but the gameplay is not original; however it's possible to look at older and successful games as references to better understand what players consider compelling. Once a correlation between games and players' preference is found, it's easier to divide players in groups and then outline statistics and infer what are the most influential features in a successful game.

So why people play games, and why some people prefer it at a book or movie? For Richard Rouse [40] games are way more appealing because they provide some sort of challenge. Reading a book or watching a movie are passive activities: the user is the spectator, who lives the experience in third person. A game engages the player in an entirely different way: in most of the cases the player IS the main character and he have to actively think for progressing in the story, solving riddles, making choices which may alter the experience completely. In games, the player lives and creates the the story in first person. In this context, when a person overcomes a challenge, that person also learn something which can eventually be applied also in the real life, even if they don't realize it (this is the core concept behind educational games).

It's true that many games are played alone, in single player mode, but there are many multiplayer games which makes way more revenue. People seem to enjoy a social game experience more than a single player one, especially in genres like first person shooter (FPS), role playing games (RPG) and real time strategy (RTS). Typically many console and PC titles of this genre offer both a single player campaign and then a multiplayer option, which is an adaptation of the single mode, both with the same rules and mechanics. Years ago, common multiplayer tournament sessions were organized by groups of friends sitting in the same room, with their PCs physically connected to each other in a LAN network. It was not only an occasion to play, but also to socialize and doing fun activity together. Eventually, larger groups would organize big competitive LAN parties, giving birth to what now is the phenomenon of eSports. Another category of multiplayer games is the "massively multiplayer online" (MMO), in which a great amount of players meet in a persistent universe called also "virtual world", the same for all the player involved, and move and interact in real time with each other or with characters controlled by the game. These kind of games are played over the internet, so the only way to socialize is with the support of in-game features, for example an in-game message system. The socialization aspect is integrated directly into the game, which encourage players to meet and organize teams to have a better chances to proceed in the adventure. This is also possible because, usually, MMO games are more slow-paced than FPS, and players have the time to chat while playing. Games which take place in virtual worlds have the potential to attract a large portion of users who are interested in a multiplayer experience.

One immediate advantage of multiplayer games is that a human opponent is much more challenging in comparison to a character controlled by an artificial intelligence (AI). Even if the AI can be well-designed to mimic human behaviour, they turn to be predictable and easy to beat once the player understood how the bot's logic work. But real players are unpredictable, and able to create strategies and react much better in many different situations. Yet, not all participants in an open world

are there for socializing, for example for those players who enjoy a more explorative experience, or the satisfaction to achieve certain results all by themselves [15]. Still, the collaborative nature of most activities in the game is the main difference for many players who engage in multiplayer and, most importantly, for the advantages, rewards and reputation that this entails. People have a fundamental need of belonging. With the rise of MMORPG (Massive multiplayer online role-playing games) a number of institutions have emerged to fulfill this need; guilds are the primary institution to encourage cooperative gameplay and sharing of success. In World of Warcraft, for example, 66% of players belongs to a guild, and the rate increases to 90% for advanced players [30]. All together, these shared experiences can increase the longevity of the game, more attractive and dynamic. For example, user-created stories and quests can sometimes be much more appreciated than the normal quests created by designers. Especially in multiplayer games, players want to win respect from others. People able to reach important achievement, or able of high performance, will be surely prized by other in the game community and, sometimes, also in real life. This sensation of contentment is often correlated with an increase of self esteem, they feel they can accomplish something that others can't, and an occasion to show pride for a result they manage to accomplish.

2.2.2 Case study: what features make a successful game?

The analysis of a game's features can be done from many points of view, it all depends by what meaning is given to the word "feature". There have been studies in the literature which tried to identify the relationship between in-game features and successful games [18, 4]. Despite the gap of five years between the cited studies, outcomes are similar. The study analyzes the trends in the free to play mobile game market, exploring the features that have the greatest impact for the success of a game. A list of 37 features was collected [7] using the ARM funnel as a reference model and then examinated with an approach inspired by CRISP-DM method. The Cross Industry Standard Process for Data Mining (CRISP-DM) is a method for developing data mining and knowledge discovery projects subdivided logical sequences of construction steps such as business and data understanding, data preparation, model building and evaluation. The first phase consisted in collecting business intelligence data from the top ranking apps in the stores. Filho et al., 2014 considered 30.000 entries sampled from the top downloads and top grossing lists in Google Play in the period 11.04.2013 - 12.05.2013; Alomari et al., 2016 instead considered 46 entries from Apple App store top grossing list. Both studies then followed a three steps procedure for analyzing the data.

First step: modeling of a decision tree
A decision tree is a structure that includes a root node, branches, and leaf nodes, automatically generated by an algorithm starting from a given dataset. The resulting top node corresponds to the best predictor from which the tree will continue. Each internal node denotes a test on an attribute, each branch denotes the outcome of a test, and each leaf node holds a classification.

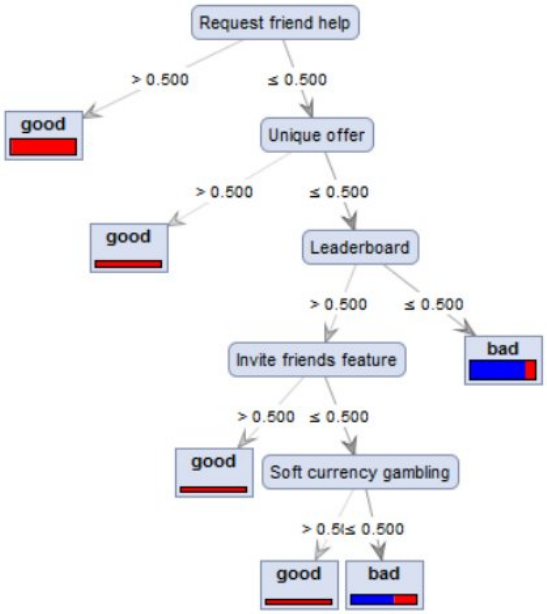


Figure 6: Decision tree, 2014 research

Attribute	*Coefficient*
(intercept)	0,131
IAP Potential	-0,345
Hard Currency	-0,295
Time Skips	0,324
Hard Currency Gambling	0,482
Soft Currency	0,272
Leaderboard	0,195
Request Friend Help	0,274
Consumable	0,172
Facebook	0,177
Levels	0,159

Figure 7: Feature influence, 2014 research

Figure 8: Decision tree, 2016 research

Nodes	Importance
Invite friends feature	0.4356
Skill tree	0.1708
Leaderboard	0.0869
Unlock content	0.0341
Soft currency	0.0341
Facebook	0.0341
Customizable	0.0341
Event offer	0.0341
Request friend help	0.0341
Time skips	0.0341

Figure 9: Feature influence, 2016 research

In fig 6 there's the decision tree generated from [18], which shows the feature "friend help request" as the strongest indicator for game performance in revenue. The second indicator, in order of importance, is the "unique offer" feature, meaning that players are more likely to invest in good deals if there is an incentivation for it. The results indicate that games with a leaderboard are usually ranked higher in the

top grossing charts. Fig 8 depicts the produced decision tree from [4] .

Second step: linear regression and evaluation
It may not be as intuitive at a first glance, to get a more meaningful result the authors performed a linear regression analysis in order to extract additional knowledge about relationships among predictive and class attributes. Linear regression is a technique used to infer a relationship of cause-effect between a dependent variable (performance of the game, in this example) and one or more explanatory variable (game features). The results of regression are numbers called regression coefficients, which indicates the significance of each explanatory variable over the output model. In this way, it is possible to identify most of the representative variables in the problem. Finally, a performance evaluation is made using the ROC Curve, or "Receiver Operating Characteristic". The ROC Curve is a graphical plot illustrating the performance of a binary classifier system. Accuracy is measured by the area under the ROC curve: a value equal to 1.0 represents a perfect test; an area of 0.5 represents an unreliable test. Study [18] has an accounted accuracy of 79% and precision of 86.86%, thus confirming a reliable outcome for the experiment. Study [4] also presents statistically significant results, with a level of confidence similar to the first study.

Third step: analysis of results
The result of the two studies are depicted in table 7 and 9. The first table shows, under the "+" sign, the features who have a positive effect on performance and under the "-" sign the ones that have a negative impact. Surprisingly, IAP seems to be a negative feature on games, but it may be an outlier because the study does not take into account soft and hard currency consumption and consumable goods such as lives and power-ups. Since freemium gamers spend most of their games on consumables, the bad performance of this variable makes complete sense. From the second table it's evident how "Invite friends" stands at the first place as most significant feature for a successful game. Skill tree is in second place, and leaderboard stays in third position. It's interesting noting that, in the two studies, there have been similar features among the most important, but in different positions. This difference in result may be caused by the difference in the initial sample data and because the mobile game market has evolved during the five years that intercoured between the two researches.

2.2.3 Occurrence of features in games

The previous section showed what are the features that can make a game potentially successful. However, for having a better understanding of the mobile game environment, it's important to know what are the most popular features in terms of frequency. The goal of this thesis is to study the current mobile game environment to recognize and classify the most used services and their use cases. An analysis of game services is presented in chapter 3.1. Figure 10 shows the occurrence of features for 48 top grossing mobile games, and a short description for each of them in table 2.

Ranking	Feature Name	Description
1.0	Hard currency	In-game virtual currency bought by paying with real money, or redeemed in special occasions
2.0	Leaderboard	Ladder where players are uniquely ranked, with the purpose of comparing them.
3.0	Facebook	Facebook's functionalities are present in the game, such as login, friend list and other services.
4.0	Soft Currency	In-game virtual currency earned while playing, to be reinvested for progressing
5.0	Invite Friends	Social action feature. Often used as play accelerator, users may invite their friends to play in exchange for a reward
6.0	Levels	Visual representation of the player's progress
7.0	Unlock Content	The player can unlock some content after some action/milestone in the game
8.0	Single Player mode	Typical for premium games. Player can engage in single-player campaigns, story mode or play against bots
9.0	Request friend help	Feature that allows a player to ask the help of a friend for completing a difficult task or progressing faster
10.0	Unique offer	The game offers a bundle of in-game items in exchange of real money, usually at a discounted price, for a limited time, only once.
11.0	Energy session restriction	The game loop is restricted by an energy consumption mechanic: energy is consumed by playing and regenerates over time.
12.0	Random elements	The game is affected somehow by random choices.
13.0	Event Offer	In-game bundles purchasable by real money or hard currency, available for special events (e.g. Christmas, Summer..)
14.0	Achievement	Bonus features, unlocked when the player reaches certain goals
15.0	Competitive play	Games with some sort of ranking system which encourage competition and skill development.
16.0	Power-up upgrade	Power-up items can be upgraded in the game
17.0	Cumulable rewards	The game includes rewards which increase in value over time, usually every day.
18.0	Consumable	Items that can be used and have an immediate, temporary effect. Usually they're cumulable in an inventory.

19.0	Daily offer	A small discount that help to convert users to paying users.
20.0	Item Upgrade	Items can be upgraded in the game
21.0	Status Upgrade	Character's statistics can be upgraded in the came
22.0	non-cumulatives	The user receives the same reward for coming back every day
23.0	Customizable	Games with customization options, usually in an avatar.
24.0	Gambling	The game includes gambling mechanisms
25.0	Chat	The game includes a chat functionality, which can be in-game or while meta-game
26.0	Timed boost	For a limited amount of time, the player gets a boost on rewards for playing (double coins or exp)
27.0	Time skip	The game uses timers, that the player can decide to skip by paying currency
28.0	Skill tree	Common in RPG, the character evolves following a tree-like chart with branches; the player decides the evolution path
29.0	Versus	Games with competitive mode which creates one-to-one competitions.

Table 2: Feature ranking and description

2.3 Game production process

The creation of a game is both a form of art and an engineering challenge, for this reason the whole process demands a large variety of competences, depending by the game. But the typical expertise in a gaming company is split into five major areas: design, art, programming, management and testing. Every project has its own development time, also in games: from AAA games, which take years to develop due to their high quality level in every detail, to simple mobile games that can be developed in few weeks. But all of them follow a similar production line: concept definition, design, development and testing.

Concept definition Phase The original idea is condensed into concepts. The team participate in brainstorming sessions to add details to the starting idea until the main mechanics and goals are decided, in the limits of the team's resources, time and capabilities. At this point, the team analyzes the game feasibility, the producer writes a game proposal to attract funding bodies and a plan for the project development and budget projections.

Design Phase The design phase transforms the concept into a detailed description of the game's features, mechanics, user experience, user interaction and narration. There are many iterations until a final design is agreed. Also, the graphics style

Figure 10: Feature occurrences on top grossing games, 2016

for the game world and the user interface is also decided in this phase. A design document is written to collect all the directives and salient points from the design, and it may be different depending the size of the company. Typically, small companies where most of the team members work closely together don't need a verbose report, but big gaming companies with tens of employees need to deliver all the information to all the interested people at the same time, and the better way is by reporting all the details in a shared document, so every person involved in the process have the same reference for continuing their work. Nowadays, one of the best way to proceed with the design of a game is through prototyping: developers implements a quick and essential mockup of the feature following the design document specifications, so that designers can actually see in action the mechanic beforehands, having a better understanding whether the feature may work or not. At the same time, the technical team must decide which technology to adopt for the development. This usually involves the choice of a game engine, a software that provides most of the tools for creating a game, for example a physics engine, rigidbody and softbody collision detection, rendering and software compilation. The most used game engines at the time of writing are Unity3D and Unreal Engine 4. The first has attracted millions of indie developers with its free licence and an easy to use visual editor. The website gathers many useful tutorials from the beginner level to more advanced users, and the engine can be expanded with plugins, purchasable from a well supplied store or made in-house. It is the most popular because it is a truly multi-purpose engine for both 3D and 2D games, and targets a wide range of platforms from consoles to smartphones. The latter was the engine used by Unreal Technology for its Unreal games, but then opened to the public as a licensed product with a free option; it focuses on AAA level quality in graphics and it is highly optimized for 3D games, for consoles and PC. Finally, after that the team concluded the design and tool selection, the development phase can start.

Development phase The beginning of the development usually includes the creation of a working prototype. A prototype is a fragment of the game with the intention to demonstrate its main features and a basic playability example. The purpose is to give developers, marketers, investors, and others a feel for the game, a good measure of the "fun" factor and a proof that the design and features work together as expected. Prototyping is the preferred way to correct the design on the go, before spending resources to refine the game. A good prototype is also used by developers to secure further funding from publishers. Once the prototype has reached a satisfactory progress, the team start with the actual production process. This task include the creation of the game's code, graphics and sound, which can take several weeks to several years depending by its the size and complexity.

Test and release phase After all the different pieces of the game are made, they are connected together to form the first version of the game, commonly referred as alpha version. It contains all the elements for being playable, but it lacks of final polishing and balancing. The alpha version is used internally for further testing and finely tune the balance and playability. Playtesting is crucial during this phase to check that the game is actually fun to play, intuitive, challenging and balanced enough so the player won't incur in frustration, the worst case for a healthy monetization. The alpha test ends once all the problems found during the gameplay test have been corrected, opening the beta testing phase. A Beta version is a refined, almost ready to publish version of the game that is subjected to further gameplay test conducted by strangers belonging to the game's target audience. Testers are invited to play, while developers observe their interaction, reactions, and collects the final comments. After the game have been refined even more, incorporating the results from the beta testing, the game is ready to be published into a limited number of countries, to study the monetization incomes and gather additional analytics on a larger userbase. Finally, the game can be correct, if necessary, and released to the public. For many companies this is not the end of the development process: through the player's feedback and the analytics it's possible to detect problems which need to be fixed. For this reason, a part of the team may eventually follow the evolution of the game by creating patches for bugs, additional content for improving the monetization and many more subsequent changes for reaching a satisfactory level of user engagement, retention and monetization.

2.4 Backend as a Service

2.4.1 Building blocks of cloud computing

The term "Cloud Computing" can have diverse meanings depending by the context, but essentially it is an abstraction on top of the physical computing infrastructure for pooling physical resources and serving them as virtual resources. Virtual resources are intended as computational power and storage space that can be dynamically allocated, configured, reconfigured and deallocated as needed. It is a recent shift of paradigm

for delivering resources on demand, for staging applications, and for accessing services independently from the underlying platform, bringing a revolutionary change in the way enterprise computing is created, deployed and managed. Applications can be developed and distributed with minimum costs for the companies who utilize the cloud computing, enabling them the flexibility of scaling the business at will. Cloud computing must not be confused with grid computing, which is an infrastructure for dividing large tasks in smaller ones and computing them in parallel on multiple servers. A standard definition of cloud computing comes from the United States National Institute of Standard and Technology (NIST) [32] Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

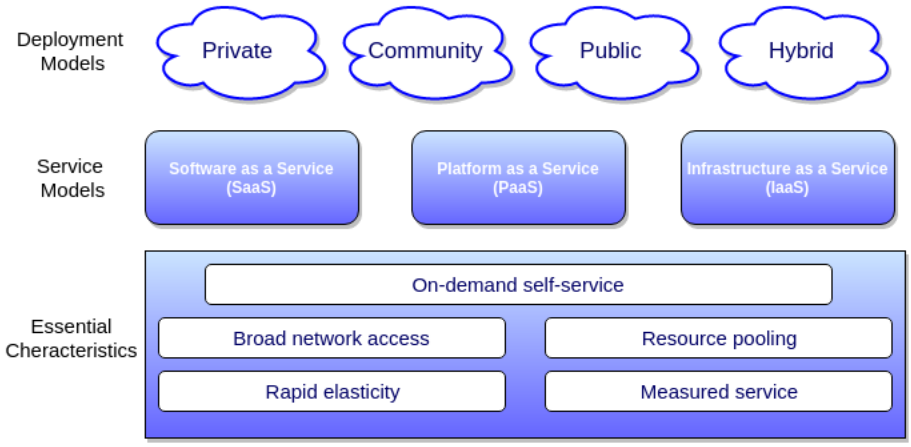


Figure 11: Cloud computing model, NIST 2012

This cloud model is composed of five essential characteristics, three service models, and four deployment models, shown in picture 11. In addition, further in the specification the NIST requires the support of multi-tenancy and network virtualization.

Essential characteristics

1. **On-demand self-service:** A consumer in need of computing resources (such as computation time, network storage, bandwidth) shall acquire them automatically without the need of human intervention.

2. **Broad network access:** Computing resources shall be delivered over a network, e.g. Internet, and accessed through standard mechanisms to promote heterogeneous client platforms (such as laptops, smartphones and other smart devices).

3. **Resource pooling:** The provider's physical and virtual resources shall be pooled and dynamically assigned and reassigned to multiple consumers who

demanded them, in compliance with a multi-tenant model. Since the cloud abstracts the physical resources, the final consumer has no control or knowledge over the real location of the resource. Nevertheless, he may be able to specify the location at a higher level, such as continent, country or datacenter level.

4. **Rapid elasticity:** Consumer's allocated resources can be rapidly scaled upwards or inwards according with demand. From end user's perspective, resources appear to be infinite.

5. **Measured Service:** Resources usage per tenant is monitored, controlled and reported through a metering system (usually pay-per-use or maximum available quota) in order to guarantee transparency for both the service provider and the consumer.

Service Models Although nowadays there are many types of popular cloud services, NIST categorizes them in three big families.

1. **Infrastructure as a Service (IaaS):** The cloud service provider has the responsibility of managing and maintaining the hardware appliances, such as processing power, storage, network and other computing resources. The consumer is allowed to install and execute arbitrary software and operating systems on top of the cloud infrastructure, and has restricted control over specific networking components (commonly firewalls). In order to efficiently manage physical resources, IaaS provider make extensive use of virtualization. Virtualization softwares can deploy virtual machines (VMs) isolated from each other and independent from the hardware, so that they can be eventually copied/transferred to different physical locations and adjust the resources allocated to single VMs to meet the consumer's demand. Amazon EC2 and Microsoft Azure are examples of IaaS cloud service.

2. **Platform as a Service (Paas):** The cloud service provider offers a hosting and development platform allowing the consumer to deploy consumer-created applications written in a supported language. In addition, the platform provisions tools, frameworks and middleware to facilitate the development phase. The user has control on environment settings and the deployed applications. An example of PaaS is Google AppEngine and Heroku.

3. **Software as a Service (SaaS):** The consumer has access to an application running in the provider's cloud infrastructure through a thin client such a web browser running on an internet-enabled device. The consumer has no control on the underlying platform, and the only customization options are the one available through the application itself. SaaS often employs a multi-tenancy architecture for minimizing resource consumption. The are countess of SaaS example currently active, some of them are: Google Docs, Gmail, Adobe Creative Cloud.

Deployment Models A deployment model defines the purpose of the cloud and the nature of how the cloud is located.

1. **Public cloud:** Currently the dominant form of cloud computing. The cloud is available for public use or targeted for large industry groups. The provider has full control and ownership over the platform, as well as policies and pricing.

2. **Private cloud:** The private cloud infrastructure is operated exclusively by a single organization, managed by the organization itself or by a third party entity. The actual location of the infrastructure could be both in- or off- premise. The benefits of a private cloud are custom optimization and maximizing the utilization of internal resources. Secondly, privacy and security concerns may induce companies to favour a private cloud. Lastly, because organizations necessitate full control and supervision over mission critical activities.

3. **Hybrid cloud:** Platform composed of multiple interconnected clouds, of any type, which share data and information together using proprietary or standard technologies. Every individual cloud service maintains its identity.

4. **Community cloud:** It's a cloud infrastructure used, built and administered by a community, or group of communities, which share a common vision.

2.4.2 Cloud computing stack

Fig 12 shows in detail the Cloud Reference Model architecture [42], a representation of the service models in terms of hardware and software stack. The bottom level comprehends the hardware infrastructure, which has also the lowest level of integration. Every layer inherits the capabilities of the service model on which relies on; the higher the position in the stack, the higher is also the level of integrated functionalities.

Cloud computing is built upon the same technologies developed for large distributed network applications accessible over the Internet. The advantage brought by cloud computing is the system virtualization in which resources can be pooled and partitioned on demand; software running in these virtual environment can also be coupled in multiple different locations. Abstraction through virtualization and resource metering is the core distinction between cloud computing and a general n-tier service architecture, where resources can be intended as static.

For the elastic nature of the cloud, an application running on a cloud platform is usually built from a collection of component, a property called composability. In a composable system, standard components can be assembled into services targeted to a specific purpose. For reducing complexity and enabling maximum flexibility, a component must be modular and stateless. A modular component is a self-contained and independent unit that can be reused. A stateless component does not keep data about the external environment; all the information must be provided by the requester in order to fulfill the task. A component with no dependencies on other modules brings powerful benefits for the application, as it can be easily upgraded or replaced with another module, provided that the interface remains the same. Despite that, there's no obligation for a component to have stateless transactions, but it surely increase coupling and accounts for a more complex system. Some cloud computing solutions manage state using brokers, service buses and transaction monitor; full

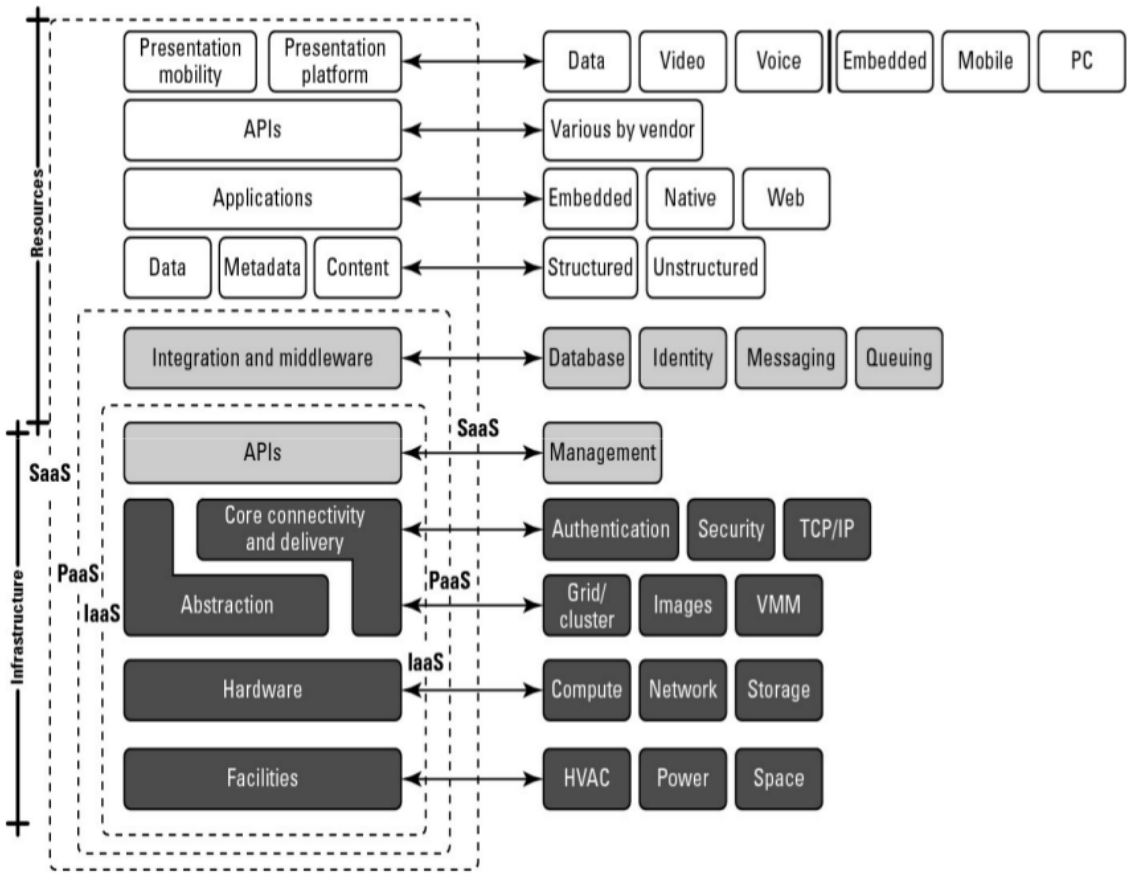


Figure 12: Cloud computing stack according to the Cloud Reference Model

transactional and stateful system are possible but hard to scale in a distributed environment. Service Oriented Architecture (SOA) is a design pattern for cloud applications promoting the use of composable services, which expose a well-defined interface and communicate together using standard protocols.

Infrastructure IaaS providers, in order to deliver servers that can run applications, build their cloud systems on top of virtual machines. The virtualization software can emulate a virtual server and assign to it a defined number of processors, processor cycles, storage space, ram and network bandwidth. Such virtual servers are commonly referred as virtual images, or instances. In fig 12 the IaaS stack is shown in dark grey, and represents the part of the cloud who is closely associated with "server". Server farms (i.e. big collection of computer servers) need equipped facilities to guarantee the proper conduct of operations. One of these requirements is an energy source able to supply enough power for the computers, combined with adequate protections against faults, overloads and short circuits. Electrical devices produce heat, and a server farm yields immense amount of heat which, if not properly controlled, can cause severe damage. Hence, the facility must be equipped with cooling systems and security measures against fire accidents. Some companies deploy

computer farms on the bottom of the sea or in cold regions, like Greenland, with the purpose of saving on the costs of cooling. Such appliances are often controversial, because of the environmental issues and pollution risks related to them. As a consequence, governments and several organizations around the world push towards an environmentally sustainable IT industry, known as Green Computing. Green Computing goals are primarily to reduce the use of hazardous materials, maximize energy efficiency during the product's lifetime, and promote the recyclability or biodegradability of defunct products and factory waste. The IT industry is interested in investing and researching in the Green Computing mostly for reducing the energy consumption of their appliances, aiming to cut costs in electricity and cooling systems.

Right upon the hardware, a low-level software called Hypervisor (or Virtual Machine Monitor, VMM) creates virtual instances, runs them in an isolated memory space, assigns resources to them and directs their I/O. The infrastructure layer is also responsible for providing connectivity, security and authentication services. Sometimes, the cloud provider can decide to expose APIs to interact with the lower layers.

Platform Virtual machines can run any software and operating system desired by the user. In PaaS clouds, the vendor installs custom services aimed to facilitate application development, within the range of the platform's capabilities. For giving an illustration, services can include: application lifecycle development, simultaneous collaboration, data management and databases, testing and analytics tools, plain storage space, transaction management to guarantee data integrity. It's convenient for operating system producer to move their development environments on a cloud platform, promoting their proprietary technologies. Microsoft Azure, for example, powers its platform on Hyper-V VM, integrates a SQL Server and ASP.NET frameworks so that developers can code using Visual Studio as client application. Platforms include all the necessary middleware to create, run, maintain and monitor an application. However, the consumer can not access the platform's features directly, it is limited to use the provided APIs.

Virtual Appliances The top layer of the cloud stack are virtual appliances: preconfigured virtual machines images, runnable on a hypervisor. One substantial difference between a platform and virtual appliance is that the former is built from many components and accessed through an API, the latter encapsulate one application and the minimum required dependencies, operating system included. Specifically, virtual appliances can be platform. Virtual appliances can be pre-configured and optimized for their purpose, so it can be copied and deployed several time without the need of configuring each virtual machine one by one. They can be used to assemble more complex systems by interconnecting more virtual appliances. Major cloud vendor like Amazon has a wide offer of virtual appliances that developers can deploy, usually they include an operating system, both proprietary and open source, a series of enterprise software and complete sets of development stack (LAMP: Linux, Apache, MySQL, PHP is one example). However, virtual appliances are generally large in size (from 500MB) because they bundle a full environment, including the

operating system, making them harder to distribute. The next section will explain in more detail how virtualization works and compare it with another popular method for distributing applications: containers.

2.4.3 Load balancing and Resource Virtualization

In order to split resources among several users, the cloud needs a low-level software called Hypervisor. Hypervisors can be of two types. In Type 1 Hypervisors, the software is installed directly on the hardware, creating a fully virtualized system. This is the first type of VVM made, and currently the most performant. Type 2 Hypervisors are applications run on the host's operating system. Type 2 hypervisors sometimes does not virtualize a machine completely. For efficiency and simplicity, some tasks can be processed by the host OS outside the virtual environment through some APIs exposed by the VVM, but the guest OS must be ported to support these APIs. Instead, an emulated virtualization simulates all the hardware, removing any dependency with the host system.

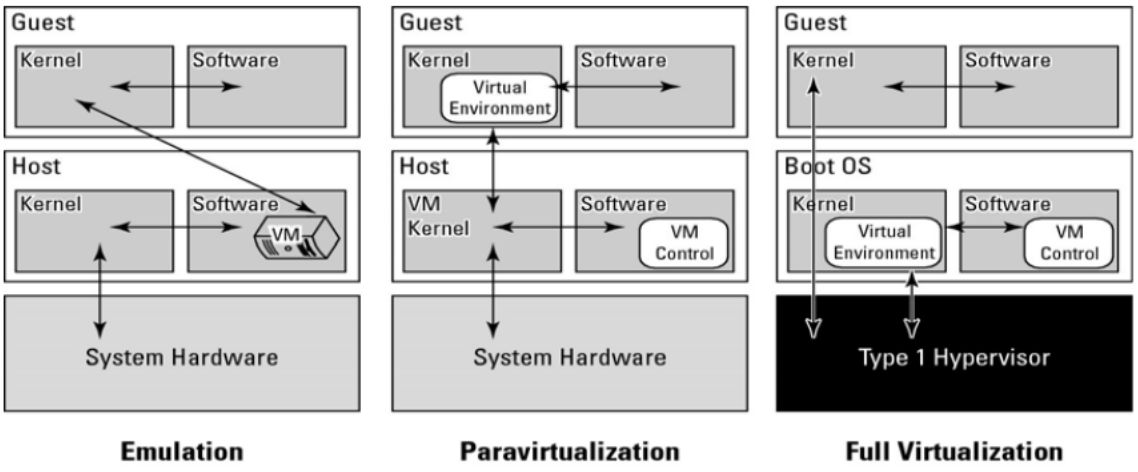
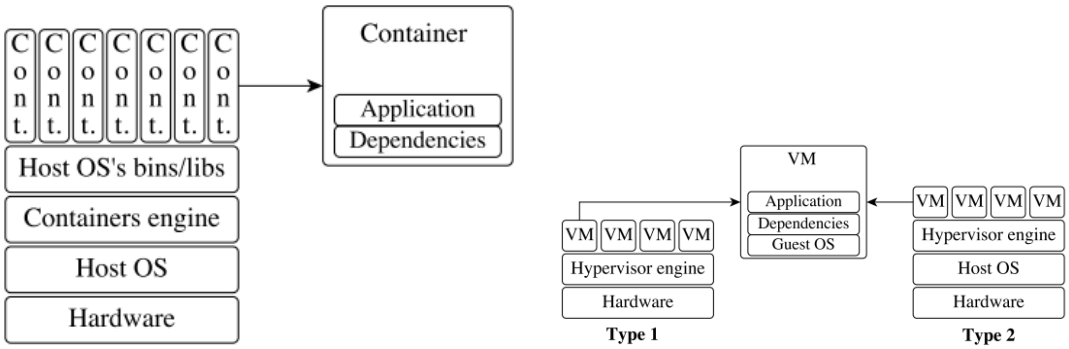


Figure 13: Different techniques of virtualization.

A summarization of virtualization method is shown in fig 13 Emulation is the technique with the most overhead, in which applications run slower. Generally, every form of virtualization incorporates some unavoidable overhead, but the advantages it brings justify the drawbacks. However, a relatively young technology is gaining popularity, and is quickly being adopted by an increasing number of developers: container-based virtualization.

Container-based virtualization approaches the problem of abstraction and isolation in a different way than hypervisors. While hypervisors abstract/emulate hardware, containers isolate processes at the host's operating system level, avoiding the overhead of virtualizing a full guest OS. That said, many containers can be executed on top of the same host's operating system kernel, sharing all the OS libraries and kernel access. A container can run one or more processes, in complete isolation from

other container's processes. Figure 14 shows the main differences in virtualization architecture [11].



(a) Container based architecture (b) Hypervisor based architecture

Figure 14: Difference between container and virtualization architectures

Due to the less overhead in containerization, it's possible to reach a higher density of virtual processes. In addition, without the need of a guest OS, the image size is smaller and deployment time is much faster compared to a normal hypervisor-based virtual machine. Researches have shown that containers offer better performance than hypervisors [33], also confirming how containerization causes no overhead on the host system. They do come with some payoff and security concerns: at the moment containers are not portable among different operating systems, and resources are not completely isolated because the kernel is directly exposed. Nevertheless, containers is the current buzzword and the technology is evolving in a promising direction.

Load balancing Virtualization is the technology at the cloud core, allowing it to dynamically assign and dismiss resources on demand while guaranteeing security and separation between virtual instances. The services in a cloud can be accessed from a network, and the component responsible for the optimal distribution of resources to requesters is the load balancer.

Load balancing is an optimization technique used for accomplishing different goals: decreasing latency time, increasing throughput, distributing load uniformly, reduce response time. Load balancers can be implemented in software or hardware, they are an essential part of the cloud infrastructure due to their ability of redirecting requests in case of failure. Load balancers help to create fault tolerant systems when coupled with redundancy and failover mechanisms. In its simplest operation mode, a load balancer uses a scheduling algorithm to match the resource with the requesting client. Depending on the state (static or dynamic) of the system and who initiated the process (sender, receiver or symmetric), different algorithms can be applied [2]. A dynamic algorithm always considers the current system status, it never accounts for past states. In distributed environments, the algorithm is executed in all nodes and the load is shared among them in a cooperative strategy, trying to increase the overall performance. However, nodes can also adopt a non-cooperative strategy and

aim to optimize the performance on a local scale. In non-distributed environments, one node can be responsible for balancing all the traffic (centralized algorithm) or, alternatively, nodes can be distributed in clusters and every cluster elects a master node to perform the balancing within the cluster.

After the algorithm matched the requester with the resource, a session token is created to redirect all the subsequent requests from the same source to the same resource. The token can either be stored by the cloud side and replicated across multiple load balancers, or it can be sent to the client as a cookie.

2.5 Achieving horizontal scalability

Scalability is one of the most important quality attributes of today's software systems. Yet, despite its importance, scalability in applications is poorly understood and there is no generally accepted definition. This chapter explains what that are the common techniques to achieve a scalable system, the advantages and the payoffs.

2.5.1 Overview

The concept of scalability has many interpretations depending by the field of application. D.Hill [25] unsuccessfully tried to give a definition of scalability in the year 1990, and by far no formal definition exists. The most widespread meaning of scalability shared by experts is the non-functional property of a process, system or network to handle an increasing workload and remain effective in terms of cost and performance when resources are added. This definition is usually associated to a distributed system, but it can be applied to a wider context. As an example, an organization who rely on a scalable business model has the potential to sustain an economic growth due to a larger volume of sales or services provisioned. Scalability is a desired feature of a system, but it does not automatically translate in performance, which is the ability of a system to complete a task in a certain amount of time. If well designed, a system can reach better performance under high load through scalability, however not without sacrificing other aspects of the system. There are two ways in which a system can be scaled: vertically and horizontally. In the case of vertical scalability, the existing computation node is upgraded with more or better physical resources to enhance its effectiveness.

Resources can be more CPUs, more memory, more storage, or technology upgrade (i.e. replace the hard disk drive with a solid state one, buy newer generation RAM or faster processors). Having more available resources affects directly the number of virtual machines that the system is capable to run, or the maximum amount of resources that every machine can get. On the other hand, horizontal scalability can be accomplished by adding more nodes to the system, in order to distribute the computation to many similar peers instead of having only one big, powerful central node of computation. Often, the nodes in this kind of system employ low-cost commodity hardware that can be conveniently dismissed if any failure occurs. Hardware costs for infrastructures may seem significant at a first glance, but the cost of software development for large applications becomes more expensive over

time. As a consequence, applications are built in a way that does not require any additional work when scaling is a necessity. But scaling vertically or horizontally have a great impact on the cost that a company must face. As seen in picture 15 the cost of scaling vertically does not scale linearly with the amount of processing power needed. At a certain point, the cost for upgrading a single node would become unsustainable. Even with a large budget available to face the exponential growth in costs, there will be a limit imposed by the technology. Anyhow, the major advantage of scaling vertically is that the application's architecture is really easy to create, and does not need any special attention when upgrading the system.

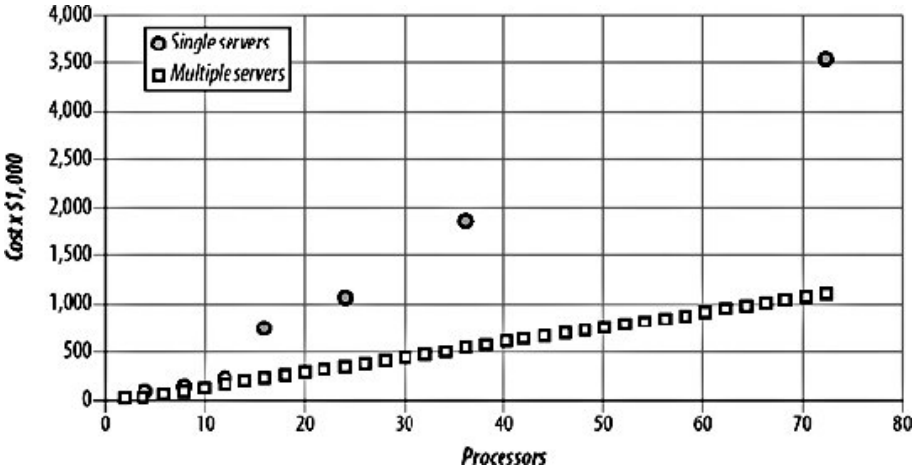


Figure 15: Cost of scaling vertically vs scaling horizontally [23]

On the long run, scaling horizontally using commodity hardware is the only viable solution. As drawback, many more machines require space, more costs on maintenance and an unavoidable more complex application structure. Moreover, as horizontally scalable system are distributed, there are other properties that the system must satisfy in order to be effectively scaled without running into issues, for example high availability, reliability and performance. A recurrent feature in distributed systems that may help an application to scale is concurrency. Concurrency is the property of a system to be decomposed into units capable of working independently by the order of execution. This means that the outcome of a certain task remains the same even if the components are executed in different or overlapping order. It should not be confused with parallelism, which is the ability of processes to be executed simultaneously. Concurrent tasks may be executed in parallel, thus increasing system performance and efficiently scaling out when more resources are added (nodes and/or CPUs).

Architectural design plays a fundamental role in achieving scalability, the ability of the system to scale depends on the types of data structures, algorithms and protocols used by components to communicate and execute business logic. As data and functions consume memory and execution time, this should be also taken into account in measuring the scalability of a system. A. Bondi describes four general types of scalability [10]

- **Load Scalability:** A system capable of remaining functional without unwanted delays or unnecessary waste of resources during heavy loads. The factors undermining load scalability can be an inefficient scheduling of shared resources, inadequate exploitation of parallelism with frequents deadlocks, the scheduling of a class of resources in a manner that increases its own usage (self-expansion).

- **Space scalability:** A system in which memory requirements grow at a sublinear rate when scaled, at most. This concept applies mainly to data structures.

- **Space-time scalability:** A system is time-space scalable when it implements algorithms having a reasonable asymptotic complexity, allowing it to function gracefully as the number of objects it encompasses increases by orders of magnitude. Space scalability is a necessary requirement to achieve space-time scalability.

- **Structural scalability:** A system that is able to increase the number of components it encompasses independently by the chosen technology, protocols or implementation. This definition is relative, because a system can be limited by the current technology. For example, a packet header has a limited size for the receiver address, which limits the number of possible addressable nodes in the system.

There are many ways to approach how to think about scalability of a platform, one useful example is the representation of a three dimensional cube addressing three approaches to scale that we call the AKF Scale Cube [1].

The AFK Scale Cube represents three characteristics of scalability on the three spatial axes X, Y and Z. The origin represents the worst case scenario, a monolithic service completely unable to scale. In a monolithic application all functionally distinguishable aspects (I/O, data processing, error handling, and the user interface) are all interlaced together, rather than contained on architecturally separate components, and is limited by the finite resources provided by the single machine. On the other side, the top right corner represent an highly optimized ideal system that can achieve infinite scalability.

The X axis represent the degree of replicability of the system, useful for spreading computational load across several nodes. This approach is effective for increasing load scalability, and it is the solution that most companies implement first. The solution consists in employing N machines, each of them running the service application, and place a load balancer in between the client and the multiple servers so that the load sustained by each node is $\frac{1}{N_t h}$ of the total. The implementation of node replicability does not involve significant costs or re-architecturing the application, unless it makes use of a state preserved in one specific server. Eventually, the system can be easily refactored to use a centralized state server, to support statelessness or use sticky sessions in the load balancer. While scaling along the X axis is sufficient for improving load scaling, it exposes potential bottlenecks like memory constraints when caching data, since all nodes must access all the data. In addition, it does not tackle the problem of increasing development and application complexity over time.

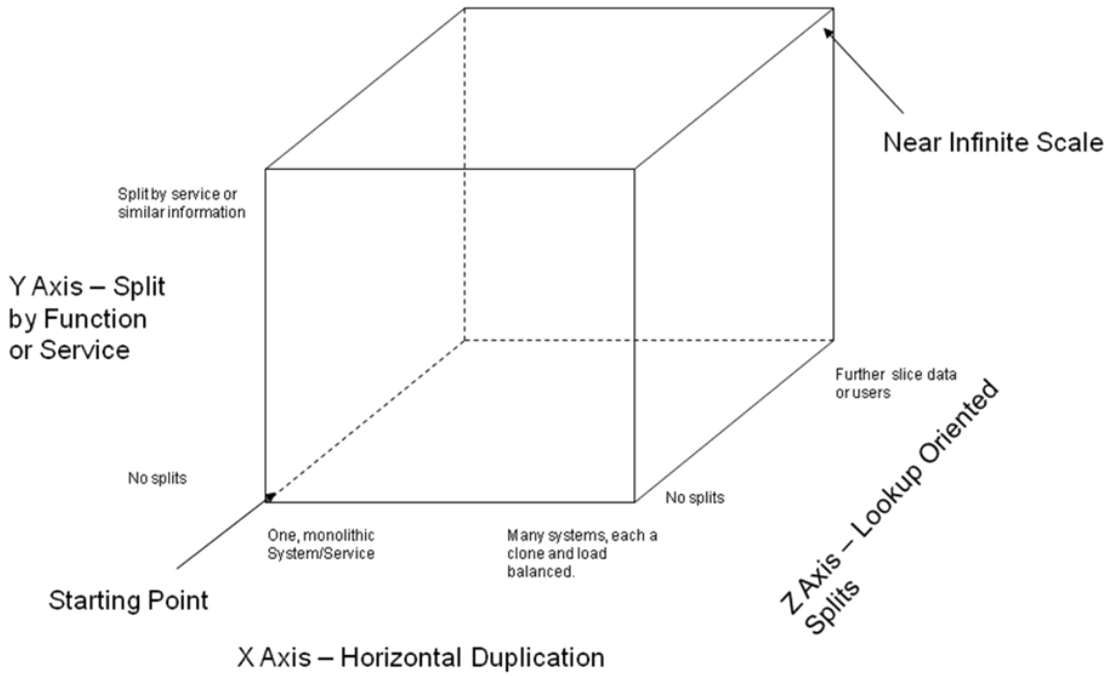


Figure 16: AKF Scale Cube

The Y axis shows how much the system is split in modular components, each of them encapsulating a certain function, service or resource. Each module can implement a set of use-cases, for example for managing a user login and processing requests for user account information. Rising on the Y axis promotes the distribution of each task to its specific module, thus distributing the amount of memory required across the system. More importantly, it localizes the memory on a single module, optimizing the use of cache. The benefits come with a higher price for architectural design, which needs to be re-engineered to carefully split the application into modules of the appropriate size and distribute roles and responsibility accordingly. In large organizations this approach is preferred because it allows developers to focus on specific, independent areas and benefit from a more gentle learning curve and better quality.

Scaling on **the Z axis** is similar of scaling on the X axis: in both cases there are cloned nodes executing the same application, with the difference that a node is responsible for only one portion of the data. Requests are equally routed to nodes by computing a hash of the request, or by inspecting its type. As a result, the system becomes fault-isolated since a failure will only affect a limited portion of the data; the memory would also benefit from a better cache utilization and reduced I/O traffic per node. Z-scaling is especially useful for sharding (or partitioning) databases across multiple servers. That said, it can be coupled with X-scaling to replicate each partition by deploying servers on a master/slave configuration, allowing simultaneous reads and writes without loss of performance. Amazon has based its DynamoDB scalability model on data partitioning [14] using consistent hashing, treating nodes

as part of a fixed circular space, or "ring".

2.5.2 Microservice oriented applications

Although there are many solutions for achieving scalability depending by the use case, but the most acclaimed approach is to use a microservice oriented application: units of modular software with a well-specified interface and well-specified set of roles and responsibilities, which follow a share-nothing principle. Microservices architecture is part of a service-based architecture family, the same as SOA (Service-Oriented-Architecture), distinguishable from other architectural designs for their emphasis on modular services as primary building blocks for an application. One commonality of service-based architectures is that they are designed for distributed systems, meaning that the communication with the single services happen through the use of a remote-access communication protocol such REST (Representational State Transfer), SOAP Simple Object Access Protocol), MQTT (Message Queue Telemetry Transport), RMI (Remote Method Invocation) and many more. A service module (or component) is a self-contained application that can be individually designed, developed, tested and deployed with minimum dependency on other services. The difference between Microservices and SOA is in granularity. While services in SOA can be of any size and complexity, Microservices are small, single-purpose and highly optimized. While splitting the architecture to create Microservices, it's important to consider the overhead caused by an excessive subdivision. Every message sent from a service to another implies a small delay due to network communication. If two microservices exchange too many message, there is the risk of a bottleneck and it's a clear signal that maybe the services can be incorporated into one. Microservice architecture divides the services in two categories, functional services and infrastructure services, as shown in figure 17.

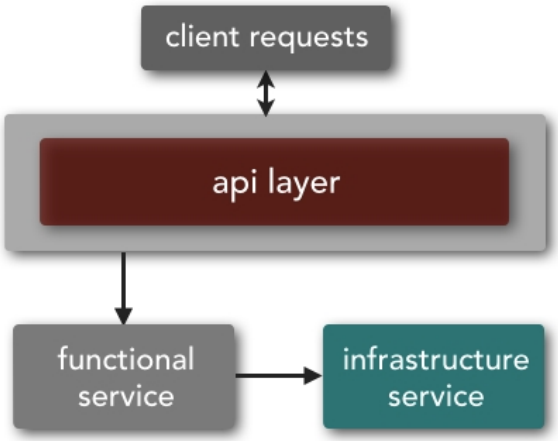
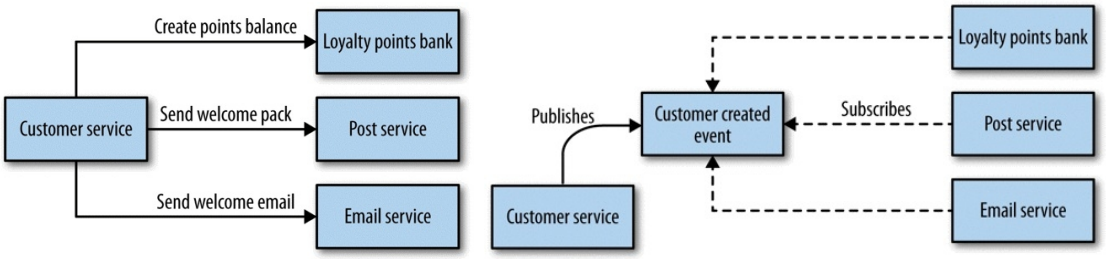


Figure 17: Microservice architecture model, in a very generic chart [39]

Functional services implement different part of an application business logic, they are accessed externally by the means of some API and generally are not shared any

other services. Infrastructure services are shared services providing non-functional tasks like authentication, logging and monitoring. They are private and not accessible from outside the application, typically shared with other services. Since microservices only provide one very limited functionality, business logic must be implemented by calling the appropriate services in the right way. There are two main strategies for managing logic flow in the application: orchestration and choreography. The differences are shown in fig 18, using the example of a customer service software that registers the creation of a new customer



(a) Service orchestration (b) Service choreography

Figure 18: Different control flows in microservice architecture [35]

In an orchestrated application there is a central node which drive the execution of a process, calling the services required through a series of request/response calls. The central node can keep track of the progress and report any fault during the execution. Although convenient, this model has drawbacks. The central node can become large as the application grows, turning into a potential single point of failure. With choreography, the central node only triggers an asynchronous event. The services subscribe to the event source and act only when they are informed that a certain thing happened. A disadvantage of choreography is in the difficulty to backtrack errors and monitor the system correct behaviour. To overcome the problem, it's a good practice to build testing and monitoring tools that explicitly match the application's logic, to reproduce the event flow and analyze how the services react. However, choreographed applications are generally more loosely coupled, tend to be more flexible and responsive to changes. Microservice architecture favors choreography for its distributed nature, but too much choreography between functional services can lead to coupling and inefficiency. For this reason it's important to divide the system in services at a right level of modularity, especially if services are organized in a chain: the longer a chain of service is, the longer it takes for satisfying a request due to the accumulation of network delay; and also higher chances of failure due to unexpected errors. A too fine-grained subdivision of tasks can undermine the benefits brought by a microservice architecture. If two functional services are sharing data with each other, they could be integrated into a unique one. Eventually, in the case in which a service is shared with many others, it's preferable to violate the DRY principle and incorporate the functionality in the services who require it. This expedient increases overall performance by reducing the number of remote calls and makes the system

more robust because there is less probability of faulty services. Finally, less services implies less API to maintain, simplifying the development and service maintenance.

A clear advantage of Microservices over monolithic applications is the flexibility to changes, a component can be redesigned or rewritten in another language without affecting the whole application; this concept applies even more to microservices, in which services are split in minimalistic modules. Service-based architectures promote technology heterogeneity. Every service can be designed and implemented with the technology that better suits the purpose, also helping the company to adopt new technologies quickly and incrementally. Microservices help to increase resilience, if a service fails then only a little portion of the application is damaged. Other services could then intervene to re-establish the faulty component or mitigate the functionality gracefully. However, accomplishing a full resilience is a non trivial task. Compared to a monolithic application, understanding how to react on failure points is much more difficult, especially when multiple services are involved for the execution of a higher level task.

Services are usually accessed remotely, so it is important to guarantee a certain level of security, typically authentication and authorization. In microservices there is not a middleware which handle security, so every service must handle security by their own. One approach is having two distinct services for authenticating and authorizing users, but a better solution is letting every service to handle the authorization by itself, creating a stronger context within the service and removing a dependency from another service.

Transactions in service-based architectures is challenging because, in distributed environments, the ACID (Atomicity, Consistency, Isolation, Durability) consistency model can not be satisfied. Instead, the predominant model in this context is BASE (Basic Availability, Soft-state, Eventual consistency). The former guarantees that every transaction is successfully executed or rolled back in case of failure, every requests does not overlap, the system remains on a consistent state and data is persisted even when faults occurs. On the contrary, the latter does not have this high degree of certainty, but ensures availability and consistency over a period of time.

3 Feature and requirement survey

Firstly, this chapter presents a survey on the common features and use cases for online games, followed by a list of non-functional requirements that the system must meet in order to satisfy the thesis goals.

3.1 Feature analysis of recent successful games

In order to have a better understanding of the very current needs of mobile gaming, the author will examine a selection of the top grossing games available in App Store and Play store and report the services used by each game. The selection has been made on date September 1st 2016. The chosen games for the analysis are: Clash Royale and Clash of Clans from Supercell, Pokemon GO from Niantic Inc, Candy Crush Saga from King and Hearthstone from Blizzard Entertainment.

Clash Royale Clash Royale is a fast paced multiplayer battle arena, which combine elements of tower defence with collectible cards. It was released on March 2nd 2016, and still ranks on the top 10 grossing mobile games since then. Players engage with other real players over the internet in one versus one matches, trying to defeat the opponent's towers using their minions, summoned by using cards randomly extracted from a deck. In case of victory, the player is rewarded with Trophies, Chests, Crowns and Coins, otherwise he loses Trophies on defeat. Player can also obtain new cards and coins by opening Chests; it's possible to open a chest only after waiting a certain amount of time depending by the Chest's type. Skipping the timer by paying with gems is also possible. Player gets Experience Points (XP) by upgrading minion cards, some features are unlocked only when the player reaches a certain levels. The major game elements and features are the following:

- **Player Profile:** Login with Facebook, Google Play or Apple Game Center. Player's progress is tight to the profile, and can be migrated from a device to another.
- **Virtual currencies:** Coins (soft currency) are gained by winning matches, and are used solely for purchasing and upgrading cards. Gems (hard currency) can be acquired with IAP, and they can be converted into gold or used to buy Chests. Experience Points can be acquired by upgrading cards. Crowns are awarded in battle and are spent to get free chests.
- **Inventory and upgradeable items:** the player can collect Minion Cards and arrange them to form three decks. Every Minion Card has its own statistics, type and peculiarity. Cards can be upgraded with coins and with enough Card Points. Chests are also part of the inventory.
- **Real-time multiplayer:** players engage in a fast-paced battle where multiple elements are synchronized in real time. Acceptable Round-trip delay time (RDT) is in the order of tens of milliseconds.

- **Matchmaker:** players can battle any other player in the world with similar level and rank.
- **Achievements:** Rewards are given to players who complete certain tasks. Achievements are incremental, and can grant a little amount of hard currency.
- **Leaderboard:** Players are ranked by the number of Trophies accumulated in battle. The game shows several leaderboards: global, regional, clan and Facebook friends.
- **Alliances:** Player is incentivated in joining a Clan, where members in a clan can engage in friendly battles.
- **Chat:** Players can chat with all members in the clan in a shared space.
- **Gifting and help requests:** players can gift or ask for cards among the alliance.
- **Tournaments:** The game is suitable for eSports thanks to an embedded tournament system, allowing the creation of public and private matches.
- **Push Notifications:** Device receive a notification on specific events

Pokemon GO
Niantic's Pokemon GO was launched globally in July 6 2016, breaking five world records after the release for grossing and number of downloads. It is the only game ever to have grossed $100 million in 20 days, and the most downloaded game ever in one month. It's a mobile location-based augmented reality game, it promotes physical activity through the gameplay. The goal is to find, capture and collect all the 151 Pokemon by physically walking in the real world. The game exploits the device's GPS to track the player's position, when he approaches a location with a Pokemon, the Pokemon is revealed and the player can try to catch him. The Pokemon is shown on a overlay on top of the camera view, giving the impression that the Pokemon is really there (although all serious players disable this feature because it makes the capture harder). Players can collect items by going to certain points of interests (POI) and battle gyms with their Pokemon, claiming its property for the team they belong.

- **Player Profile:** Login with Google Play (Android) or Game Center (iOS). Account is portable between devices.
- **Virtual Currencies:** Coins (hard currency) can be bought with IAP or obtained by conquering gyms. Stardust and Candies (soft currency) to power up Pokemons can be acquired during the gameplay, along with Experience Points

- **Inventory and upgradeable items:** Pokemon GO is all about collecting. Pokemons are the most important collectibles, and they can be powered up with Stardust and Candies, obtainable by capturing Pokemons. Some items are unlocked after the player have reached a certain level.
- **"Almost" real-time single player:** player fights other player's pokemon, controlled by an AI, for taking control over gyms. After the battle, the new state is globally visible. Client devices also need constant communication with the system for receiving Pokemon's location and for validating the capture. Acceptable RDT is in terms of one second.
- **"Almost" real-time multiplayer:** Multiple players of the same team can join the fight in a gym at the same time. RDT requirements are the same.
- **GPS tracking:** Eggs can be hatched by walking, the game keeps track of the distance traveled by the player and its current position in real time, to show the Pokemon hidden at that location.
- **POI and Location Data:** Pokestops and Gyms are placed in POIs all over the world according to their popularity. Data was originally extracted from popular geo-tagged photos on Google. Pokemons have affinity with the real environment, the system must be able to determine the physical properties of a location's terrain in order to spawn the appropriate type of Pokemon.
- **Achievements:** Players are awarded with medals after that certain goals are satisfied. No useful reward is granted to the player.

Candy Crush Saga
Candy Crash, developed by King and released on April 12th 2012, is available in all the major mobile platforms and as a browser game. It was nominated "Game of the year" in 2013 after its positive receptions and gross income, it's been in the top grossing mobile game charts for years. Candy Crush is a casual match-three puzzle game where players have to switch candies on a grid to form a line of three or more candies of the same type. The game is organized in predefined levels of increasing difficulties, and highly leverages on Facebook's social features to involve player's friends as much as possible.

- **Player Profile:** Login with Facebook. Account is portable between devices.
- **Virtual Currencies:** Gold Bars (hard currency) can be bought exclusively with IAP. Lives can be considered a renewable soft currency, the game poses a negative feedback after losing a level by subtracting a life. Lives are replenished over time.
- **Inventory:** Player can accumulate boosters of different type, usable in levels for having an help or advantage.

- **Daily reward:** Player is rewarded with boosters of increasing value every day in a stack.
- **Timed offers:** discounted bundles available to purchase for a limited amount of time. Special offers are proposed at special occasions.
- **Leaderboard:** Every level shows a leaderboard to compare the player's best score with the friends' one. In addition, player's friends are ranked by number of level cleared, their icons are shown in the main level map.
- **Gifting and help requests:** The game strongly encourages gifting to maximize the chances of virality. Player can gift lives or boosters, and request lifes from his friends through Facebook messages.
- **Social Network Features:** The game uses Facebook specific API to interact with player's friends and share high scores.

Clash of Clans
Clash of Clans is the most famous game of Supercell, the first of the "Clash" line. Although it shares some similarities with Clash Royale, the two games are deeply different. It is a multiplayer strategy game with tower defense elements in which players build, upgrade and defend a village. In addition, player can train troops to raid other villages to get resources and Trophies, ranking the global leaderboards.

- **Player Profile:** Like Clash Royale, players can login with Facebook, Google Play or Apple Game Center and play freely in any device.
- **Virtual currencies:** Elixir and Gold are the main soft currencies, generated by structures and obtainable from raids. Gems (hard currency) can be purchased with IAP or in other little amounts by playing and completing achievements. Experience is gained for building and upgrading structures.
- **Inventory and upgradeable items:** The player's village can be seen as an inventory, structures and troops as items that can be upgraded. Also, structures have an arrangement.
- **Indirect Multiplayer:** Players perform raids on other people's villages, trying to break through their defences. Although the server controls the defensive structures, only the attacking player is directly involved in the fight. The defender player will only get the notification of the battle outcome. Since no real-time multiplayer is involved, the game can easily employ lag compensation techniques to make the game run smooth also with high RDT.
- **Matchmaker:** players are matched according to the number of Trophies they have.
- **Achievements:** Incremental achievements that grants rewards, including hard currency.

- **Leaderboard:** Players are ranked by the number of Trophies accumulated in battle. There are several leaderboards: global players, regional players, global clans, regional clans and player's friends.
- **Alliances:** The game is highly focused on clan battles. Players are encouraged to create or join clans and collaborate together to rank leaderboards by defeating other clans.
- **Chat:** Players can chat with all members in the clan in a shared space, or send inbox messages to specific people.
- **Gifting:** There is the option to donate troops to other clan members.
- **Push Notifications:** Device receive a notification on specific events

Hearthstone
Originally released on PC and successively for mobile, it is a turn-based collectible card game developed by Blizzard Entertainment. The game borrows the theme of World of Warcraft with a gameplay typical of a physical card game: player collects cards by buying and opening virtual card packs. There are nine deck categories, each of them feature a Hero with different effect in game and can accept maximum 30 cards, both deck-specific or generic. Cards are distinguished by rarity and can be classified in Minion, Spell and Weapons, each with different usage and effect. Players contend in head to head random matches. Despite its simplicity, Hearthstone has a deep gameplay which require strategic thinking and careful deck builds to succeed.

- **Player Profile:** Login from any device with Battle.net account or Facebook.
- **Virtual currency:** Gold Coins is the soft currency used to purchase card packs, arena keys or unlock adventures. Coins can be obtained by completing daily quests. Arcane Dust can be obtained from duplicate cards, and used to purchase other cards.
- **Shop:** The game has no hard currency, instead IAP are used to directly purchase card packs and unlock adventures.
- **Inventory:** Cards are the central and most important collectible items in the game. Decks are arrangements of cards that the player can create and save.
- **Turn-based multiplayer:** actions are performed turn by turn, orchestrated by the game. A turn has a maximum duration of 75 seconds with no interaction with the other player, so the acceptable RDT is in the order of several seconds.
- **Matchmaker:** Players are matched based on their matchmaking rating (MMR) score similarity. Also, players can create friendly matches.

- **Leaderboard:** There are different game modes. Ranked mode favours competitive play by granting MMR points to players after each victory, and removing point after every loss. There are 25 ranks, depending by the player's score. Leaderboards are resetted after every month.
- **Chat:** a simple emote system is available in game with some pre-defined words. Chatting with friends is possible outside matches.
- **Friends:** Rewarded referral system. Friendship through Battle.net account.
- **Quests:** Daily quests are available to complete in exchange of gold.
- **Downloadable content:** New content is added seamlessly with in-game updates which does not affect the game client. Updates are downloaded from the server when needed. Adventures and expansion sets are two examples.

Despite the diversity in genre, some features occurs in all five analyzed games. It can be seen how user profile is an important element for mobile games. Players can authenticate in few seconds using 3rd party providers, usually Facebook, Google Play or Apple Game Center. Through authentication, player's progress can be persisted online and retrieved in a second moment on another device, allowing true portability of the user's information, more security and privacy. Leaderboards and achievements are cornerstones in modern games, as they increase the player's engagement and interest in the game, contributing to a better retention (and monetization). Inventory and virtual items are very common elements. Virtual Currencies, shops and IAP are also omnipresent, being one of the major monetization channel for games, together with rewarded video ads. Depending by the genre, player's friends can be more or less important. Some games are built around interaction with friends, incentivizing players to involve as many friends as possible, like Candy Crush. Others have a broader concept of friend which is not strictly limited to a social network connection, and give the opportunity to find other players directly in the game to form alliances. Anyway, a form of real time chat or inbox is a desirable requirement for allowing people to communicate. Although not all games involve matches and tournaments, the opportunity to create multiplayer experiences is appealing to many gaming companies, but also being a big limitation due to the high complexity of multiplayer interaction. The next sections describe in detail the requirements that will be used to compose the final scalable backend application.

3.2 High level requirements

This chapter lists the high level requirement that will be taken into consideration for the application design. The design will take inspiration from the directives recommended by the Reactive Manifesto[4], a collection of best practices and architecture traits from the industry aimed towards the production of scalable, resilient and highly

[4] `http://www.reactivemanifesto.org/`

available software applications. The characteristics of a Reactive system are shown in picture 19.

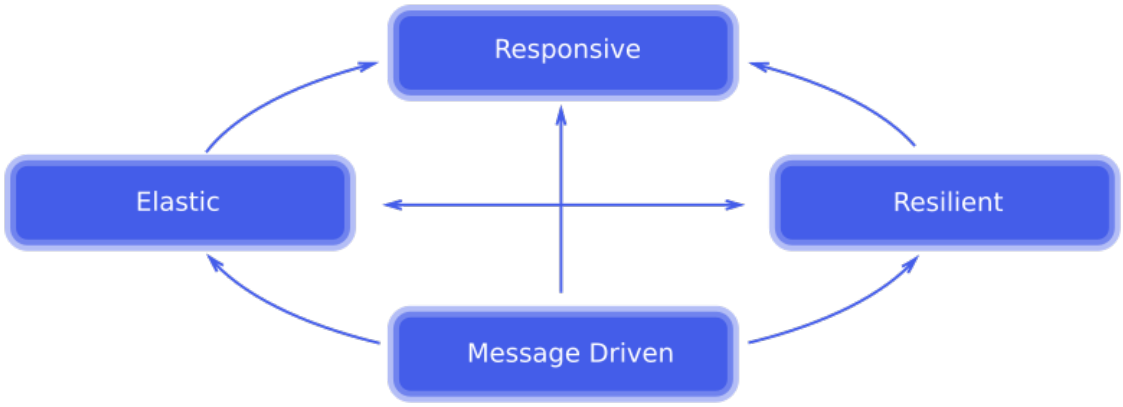


Figure 19: System properties according to the Reactive Manifesto

Responsive:
Responsive system are focused in providing high quality of service (QoS) with consistent response time, acceptable delay and well-defined upper bounds in resource consumption. Responsiveness empowers fidelity on the end user by guaranteeing usability and service utility.

Elastic: Elasticity ensures the responsiveness of a system under varying workload, both at high loads and low loads, and to efficiently allocate or dismiss resources in order to reduce costs for the service provider. Elastic systems are naturally distributed and avoid central bottlenecks or single point of failures, and consent to equally partition the workload among replicated or sharded components. Algorithms and data structures shall meet the elasticity requirement too, in a cost-effective manner on commodity hardware and software platforms.

Resilient: The system remains responsive even if failure occurs. Ideally, the system shall become unavailable only after a complete failure of all the components the system encompasses. Key properties of a resilient system are replication, containment, isolation and delegation. Single components shall be eligible for replication in order to increase redundancy and high availability; faults shall be contained within a single component to isolate other parts of the system from cascading failures. Finally, recovery of faulty nodes is delegated to an external monitoring component where it applies.

Message Driven: System based on asynchronous message passing decrease coupling between components and reinforces isolation. Messages can also be intended as self-describing events observable by a subscriber component; publisher/subscriber design pattern is especially suitable for non-blocking communication. In addition, component's physical location is transparent to messages, since they can be sent

seamlessly to nodes in a network or threads in a process. Message queues are effective for load balancing and flow control, as well as indicators about the system health. In case of congestion (e.g. a message queue length exceeds a certain threshold), the component can gracefully respond to the load by informing the user about the issue.

In addition to the aforementioned properties, the system shall observe the following requirements for facilitating development, operational tasks and deployment.

Modularity: The system shall be designed as a collection of standalone self-contained services, uniquely defined by a clear API. All data required for the internal operations must be encapsulated and never exposed with other services, in contemplation of the share-nothing principle. It must be possible to create non-trivial high-level logic by composition of single services using the orchestration or choreography approach, whereas the situation requires. A modular system allows the single components to be treated individually during development phase, improving testability and lowering deployment time.

Customization: The system must abstract as much as possible from implementation details, leaving to developers the task to customize the low-level design according to their application-specific requirements. This will affect mostly API definition, communication protocols, data models and database schemas. A good level of configurability ensures flexibility of components, so that they can be used as blueprints for the most common use cases. Moreover, it shall be possible to easily extend the system functionalities with new custom ad-hoc components.

Configurability: The application shall work with no restriction regarding the virtual topology or physical location of single service instances. There shall be no difference, from the application point of view, to run all the services on a unique physical machine or a virtualized environment spanning several datacenters. The final user must be able to configure the virtual topology of the application, deploying and removing instances at will. Services must adapt to topology changes automatically. Options for automating the deployment and allocation/dismiss of resources are desirable.

Monitorability: Final user shall be able to access metrics regarding the system at the component level. Especially, resources consumption (e.g. CPU, memory, storage, bandwidth) shall be metered and reported for performance tuning. All faults must be promptly alerted through a notification system, service uptime and congestion levels shall be monitored continuously. Automatic failure recovery mechanisms are desirable.

Security: Every part of the system must guarantee protection from unauthorized access and from theft of sensitive data. Connection between system and clients must be encrypted to prevent basic threats.

3.3 General Service Requirements

Services are meant to be highly dynamic components which enclose specific system features in an isolated logical space, which expose only the bare minimum APIs to access business data and operations. The following is a list of required properties for a service

Reactive compliant
The service must observe all the requirements for a Reactive System, already mentioned in the previous section.

Standalone A service is a software component that can be developed, tested and deployed independently of other components in the system. It shall encapsulate all the data needed for accomplishing the task.

Replicable
Requirement that supports Elasticity: the service must be designed for concurrency, so that adding more instances of the service increase the performance of the system. Multiple instances of the same service shall not interfere negatively with each other, but they may cooperate by sharing resources whereas it is necessary or convenient in order to enhance performance.

4 Application design

The previous chapters have given the motivation and background information for building a scalable system for online games. The current chapter will show the architectural design of the system as a whole and the details of every single component. For each component, specific requirements will be illustrated together with the necessary UML diagrams. As described in chapter 2.5.2, a microservice-oriented architecture is especially suitable for achieving horizontal scalability, modularity and flexibility, thus making it the first choice as reference for the architecture. Abstraction and concurrency can be achieved with a careful software design.

4.1 Overview

The application is based on standalone microservices that can be replicated as needed, communicating by the means of asynchronous message exchange as the Event-Driven Architecture pattern suggests. In this context, messages can be intended as events. Services create, detect, consume and react to events. An overview is depicted in figure 20.

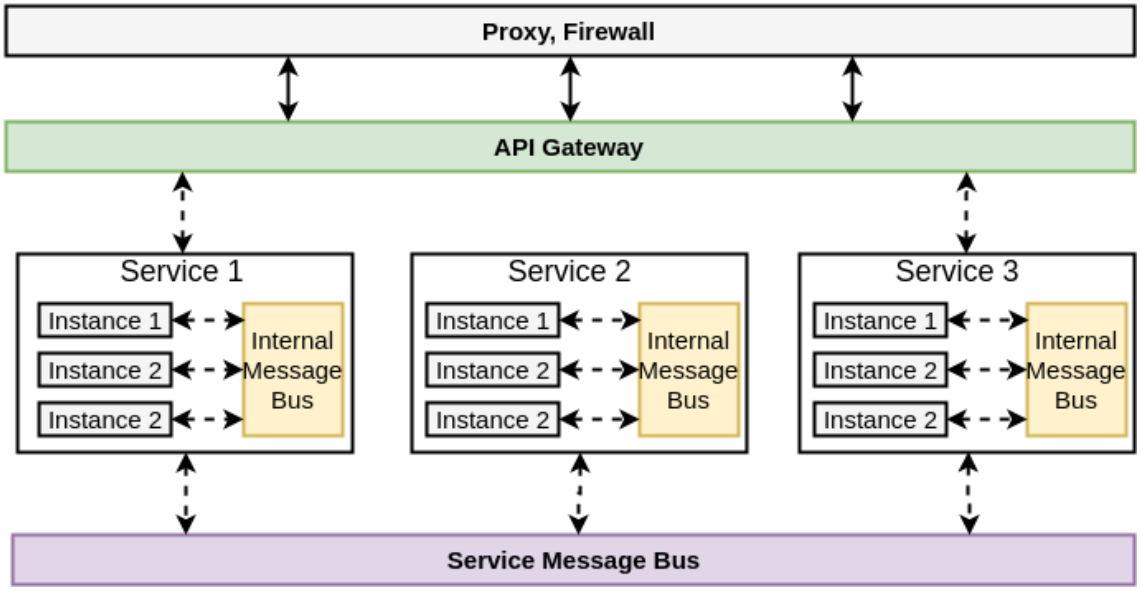


Figure 20: High level system architecture

Dashed lines represent asynchronous message exchange, while continuous lines indicate a persistent connection. Clients do not directly talk to the services, the application core is placed behind a layer of proxies that handles HTTP traffic and TCP connections. A reverse proxy has many advantages: it provides security by obfuscating the internal topology and real addresses of the application, handles the compression and encryption of requests and acts as a cache for certain type of contents. Thanks to proxies, the application is more secure because every service instance can block all traffic from external addresses, and whitelist only the trusted

sources. Also, the backend can be relieved from decrypting requests and dealing with certificates. Additional security can be granted with a firewall as protection from DDoS attacks and other threats. The Proxies route the incoming traffic requests to one of the API Gateway servers. **The API Gateway** is a server (or collection of servers) with the task to analyze incoming requests and route them to the appropriate service, at the same time guaranteeing protection against faults and adequate security. For dynamically routing the requests to the correct service and offering protection against faults the API Gateway uses a communication channel called Message Bus. A representation of the message bus is shown in figure 21.

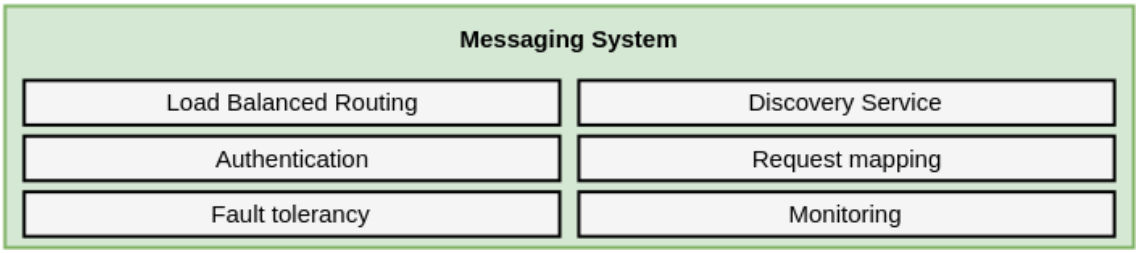


Figure 21: Schematic representation of the Message Bus

The Message Bus is a reusable system to route any type of request in the application, in the form of message. Message Bus can serve to send messages to every part of the application and stands as an independent service on its own. This way, the components are decoupled since they communicate indirectly to each other through the Message Bus, which can route requests dynamically according to the current load, exclude faulty nodes and invalidate stale requests. Messages can be intended as events, which are delivered asynchronously.

- **Discovery Service:** It's a service which gathers all the instances of all the other services running in the application, with their address and port. Every time that a service is started, it logs itself into the Discovery Service server. It will then check the health status of all services by sending heartbeat signals and immediately notify faulty nodes when occurring.

- **Load Balanced Routing:** Every request should be routed to a service that can satisfy it. The Router is a simple component that associated request types to services. For example, requests can arrive in form of HTTP, so they can be routed according to their URI. With TCP, the protocol should agree on message types. Once the request is identified, the list of available services is retrieved from the Discovery Service and one of them is chosen according to their load and health status.

- **Authentication:** If a request needs authentication, the player must include an authentication token with it. Before forwarding the request to the service, the API Gateway contacts the Authentication Service for checking the user's identity and fetch his permission. In case that the authentication fails, the

request is denied. For flexibility, authorization is delegated to the specific service.

- **Request mapping:** For a reliable system, it's important to inform the requester about the outcome of the operation. In a distributed system where messages are passed from service to service asynchronously, there must be a mechanism for notifying the sender if the request was executed correctly or generated an error. This can be accomplished by incorporating a *request_id* and *requester_id* in the message, so the reply message can be routed back to the origin. The requester is usually a service. In case of the API Gateway, which forwards requests on the player's behalf, it maps each request with the incoming connection so it will always able to return the message to the correct player.

- **Fault Tolerancy:** Nodes can fail and disappear, network communication can interrupt, messages can get delayed, processes can overload. If one service fails completely, there must be mechanisms to prevent cascading failures that compromise all the application and a graceful service degradation. Effective techniques for guaranteeing this property are known as **circuit breaker** and **timeout**.

- **Monitoring:** each time a failure occurs, the system reports the fault to a log service or supervisor service, who can activate to investigate the cause of the fault. This is useful for restarting a dead service, killing poisonous processes and notify the rest of the application that something went wrong, allowing it to adapt to the new condition. Also, the error log is useful for developers for debugging purposes.

The sequence for satisfying a request can be seen in the flow diagram in figure 22.

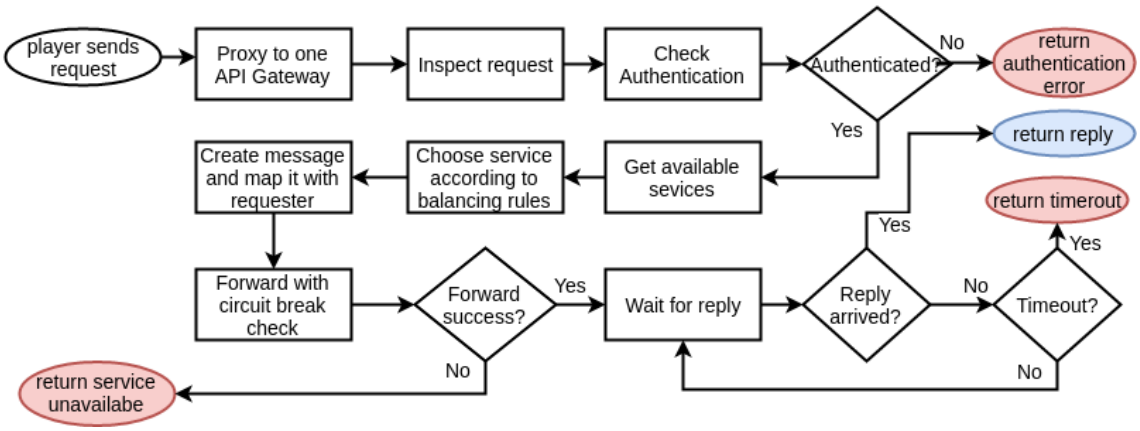


Figure 22: Flow diagram for sending a request to a service

The use of timeout guarantees that even if a service is not responding or a message is lost for any reason the caller will always be notified of the failure, so it can react

accordingly by the case. If the caller is the API Gateway, it will return an error message to the client. For messages sent by services in the Service Message Bus and Internal Message Bus the flow is the same, except for the authentication check (since all the internal services are protected) and the proxying to the gateway. Services can handle errors in the more appropriate way, eventually returning it on cascade until the initial caller is met. Actual message exchange is accomplished with **Message Queues**: every service has an inbox where messages from other services are collected waiting for processing. The service constantly checks the inbox for new incoming messages. Message queues are good for withstanding bursts of requests, they act similarly to a buffer, collecting tasks waiting that some service will execute them. If the queue size exceeds an upper limit, then the system can deploy more service instance to quicken the execution. Is worth noting that every service instance share the same Message Queue. Unfortunately, services can become overloaded and stop fetching messages from the queue, leading to an accumulation of requests until when the queue will be completely full. In this case, messages could be lost if some sort of fallback mechanism is not in place. A solution to this problem is using the circuit breaker pattern, shown in figure 23.

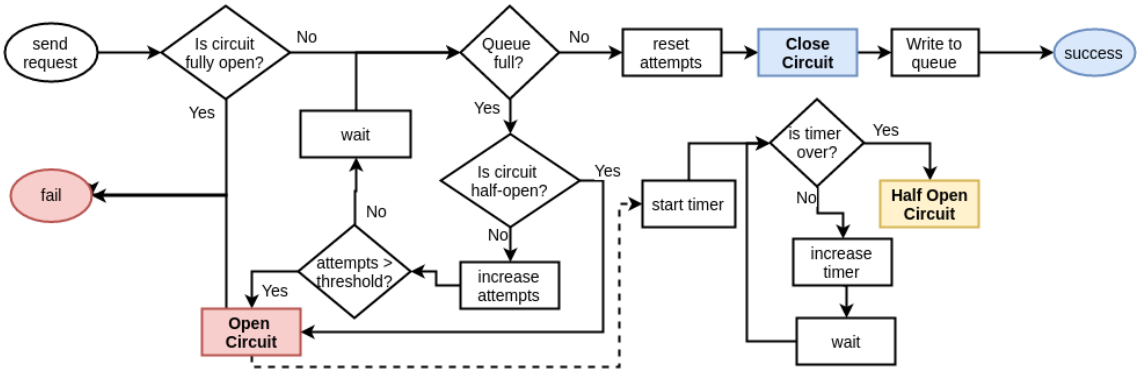


Figure 23: Circuit breaker pattern

Essentially, the circuit breaker uses timers and timeouts in case of full queue to repeatedly retry the delivery of a message in case of full queue. It can be in three state: close, open and half open. A closed circuit is healthy. At every request, the message queue is checked and if it's not full the message is pushed in the queue. Contrarily, if the queue is full it will try several attempts after waiting a certain amount of time. If the queue does not recover before the number of attempts exceed a threshold, the circuit is opened and all subsequent messages will be refused. After a defined time, the circuit will switch to the half open state, and start to accept messages again. At this point, if the queue has recovered the circuit will be completely closed recovering the normal behavior, otherwise it will return to the open state.

4.2 Social Service

As games are becoming more and more social, integrating it with the most widespread social platforms can be problematic to handle for developers. Typically, the integration between games and social networks happens through a provided SDK that needs to be included in the client application, so some part of the interfacing must be handled by the game itself. Nevertheless, many social networks have APIs accessible from servers. The Social Microservice serves this goal.

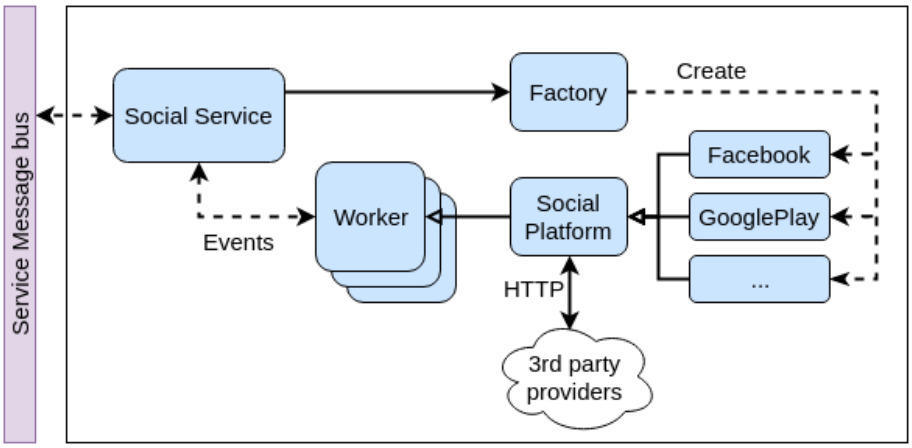


Figure 24: Social microservice architecture

The Social Microservice is an infrastructure service, not meant to expose any public API, but solely communicate through the application's messaging system. Figure 24 shows the architectural structure. **SocialService** is the microservice's entry point and responsible to accept incoming requests for tasks and emit results, a list of possible messages are shown in table 3. The service implements the Factory design pattern: a **SocialPlatform** (Worker) is an interface which declare the contracts of the functionalities provided by the service. SocialService does not need to depend on specific implementation of a platform, indeed it asks a **Factory** object to create them. The created objects implement platform-specific code to fulfill the all the operations. Since the worker is provided with all the data needed for completing the task, and all requests are independent by each others, they can run concurrently. Once the task is completed, the service entry point collects the answer and notifies about the operation result.

In order for the service to scale, each message must be complete with all the required information. The platform type and user access token, for instance, are always mandatory fields. The data carried by the message can vary depending by the target platform. Messages and answers declare their intent with a verb and a complement, and carry data. Answers also use semantic to distinguish their type and carry with them the data retrieved by the service. The service is suitable for infinite X-axis scalability and Z-axis when messages are routed according to the player's ID.

APIs (prefix: /social)		
GET /:platform/profile	GET /:platform/friends	GET /:platform/achievements
DO /:platform/login	DO /:platform/share	DO /:platform/invite

Table 3: Social microservice example APIs

4.3 Authentication Service

Games uniquely identify players in order to offer a customized experience and account portability among devices. When a user installs the game for the first time, the game client request the user to perform a login procedure or, in case he has not done it already, register to the service. User registration and authentication is often done through a third party identity provider, the most popular being Facebook, Google Play and Apple Game Center. These services implement custom protocols for authentication and require different sets of data to deal with. Moreover, in some cases other registration and authentication strategies are desirable, for example with a custom username and password database or an anonymous guest account. Functional requirements for the Identity service are:

1. Player shall authenticate to the service using a given **Authentication Strategy**. Required authentication data will be provided to the service which validates the request or returns an error to the client.

2. The service will retrieve the user ID from the identity provider and checks whether the user ID is already registered in the system. If yes, the system will return the data associated with player's identity, completing the login. If not, the service initiates a registration process.

3. A registration process involves the creation of a new **Player Identity**, identified by a unique ID, and an Access Token. A new **Player Account** representing the authentication strategy used for the registration is also created, and univocally linked to the Player Identity.

4. A player can have one or more Accounts connected to its Identity, as many as the supported authentication strategies.

5. Additional accounts can be linked to the Identity after the player has successfully authenticated to the service and authentication data for the new Account is provided. Linked accounts allow the service to determine the player's identity by using that account's login information. Unlinking of existing accounts can be possible if necessary, as long as there is always at least one connected to the Identity.

6. It must be possible to store player's related data along with its Identity, the system shall be able to update this information if required.

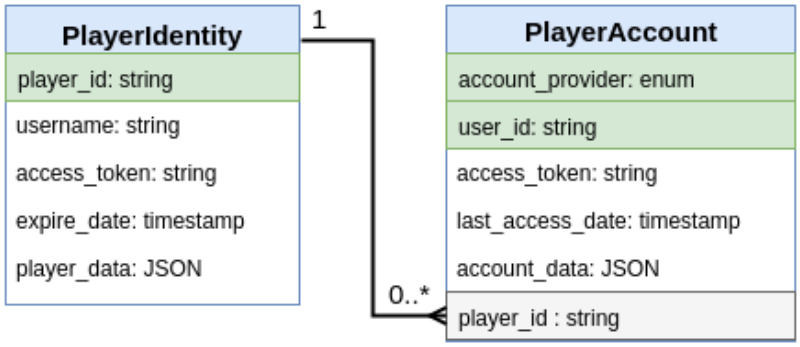


Figure 25: Data model for Authentication service

The proposed solution for the authentication microservice is a stateless functional service who exchange asynchronous messages with both the API Gateway and Message Bus for maximum scalability and flexibility. For conveniency, the service depends on the Social Service to handle the user's authentication with the 3rd party identity providers, but if necessary this functionality can be included under the Authentication service. If for example, the user choses to authenticate through Facebook, the Client application will initiate the OAuth2 handshake with Facebook's identity servers to obtain an authentication token. The Client will then use the Facebook token to login to the authorization microservice, who will confirm the identity and release a token usable for accessing the rest of the system. In the context of OAuth2, the Client sees the system as the Resource Server. Token-based authentication enhances security since the system will be able to access 3rd party services on behalf of the user without knowing its private credentials. From the functional requirement analysis, four main elements emerge: Authentication Strategy, Player Identity, Player Account and Player Data. Fig 25 shows the data model used by the microservice. **Player Identity** stores the system-specific user unique identifier and has a one to many relationship with **Player Account**, connected by the foreign key *PlayerAccount.PlayerId*. The latter model contains the information collected from the 3rd party service during the authentication phase. Both *playerData* and *accountData* fields contain custom application-specific details that developers may decide to include.

APIs (prefix: /auth)		
DO /:strategy/login	DO /:strategy/signin	DO /:strategy/link
DO /authenticate	DO /unlink	DO /logout
Events		
PUB auth:user_signin	PUB auth:user_login	PUB auth:user_logout

Table 4: Authentication service example APIs

The table 4 lists the functions exposed by the microservice. Login, Signin and Link are actions which depend by the type of the chosen authentication provider, so

the *:strategy* parameter must be provided along with all the custom details needed. The microservice publishes events at the occurrence of certain action, so the other microservices who subscribed in the topic can react automatically, for example by caching the user data from the database at login, creating records for new profiles or dismissing resources at logout.

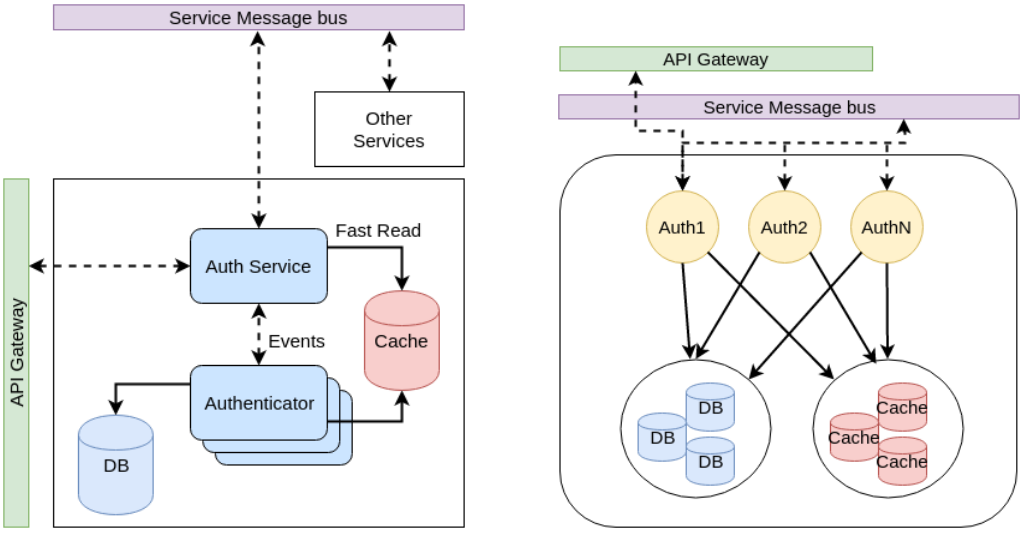


Figure 26: Authentication microservice architecture and scalability model

Fig 26a shows the logical structure of the service. The **Auth Service** is the microservice entry point, which listens for incoming messages from the API Layer and delegates other units for their execution. **Authenticator** can be implemented as workers who run concurrently, or as a thread pool, each of them satisfying one request. For requests who require a direct interface with the identity provider, the Authorization Service must rely on the Social Service, who has the capability to interact with the platform on behalf of the user. The result of the operation can be returned asynchronously to any Auth Service instance which eventually propagates it to the Authenticator for finalizing the operation. Authenticator has the task to create new Identities, Accounts and perform the link/unlink between them by assigning a value to the PlayerAccount foreign key. At the operation completion, the Auth Service can send a message back to the API Gateway informing it of the operation. The most common query the service will have to satisfy is authenticating the player given his access token. PlayerIdentity can be stored on a fast in-memory key-value database using the *accessToken* as key. Auth Service can check the cache before calling the Authenticator and return it directly in case of a cache hit. Otherwise, it will load PlayerInfo when the player logs in. As the service is stateless, every node or thread is the same. Fig 26b shows the service at maximum scale: the authentication logic is replicated in multiple nodes, each of them connected to a clustered database and cache. If the messages are partitioned among nodes using the player's ID as partition key, the microservice satisfy the XYZ scalability and each node could have

its own database and cache cluster, instead of sharing them with all the other nodes. In such scenario, the login functionalities shoud be moved from the Social service to the Authentication Service.

4.4 Shop Service

In App Purchases still play a major role in games monetization. Similarly to Social Networks, most games rely on external 3rd party SDK to implement a shop feature directly on the client device. As illicit ways to fool these SDKs with fake purchases has spread, it's important that developers protect their economy by validating the purchases in a secure environment. A Shop microservice is an indispensable tool for guaranteeing a healthy monetization and preventing cheating. The requirements for this services are:

- The service shall have a catalog of products available for purchasing. Catalogs may be different for different versions of the game, platform or provider, but in general every product will have a unique identifier given by the shopping platform (iTunes, Google Play, Amazon etc..).
- It shall be possible to validate purchases executed on the client device.
- It shall be possible to manually update the catalog if the shopping platform does not expose APIs for retrieving the catalog automatically.
- The service shall store all transactions to face eventual complaints from dissatisfied customers.

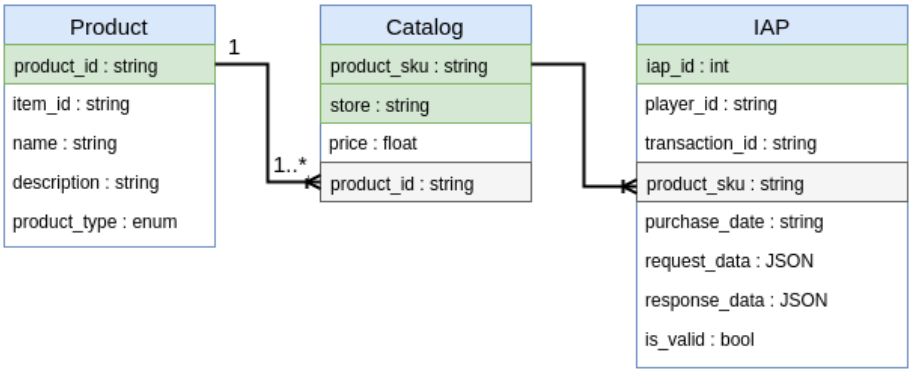
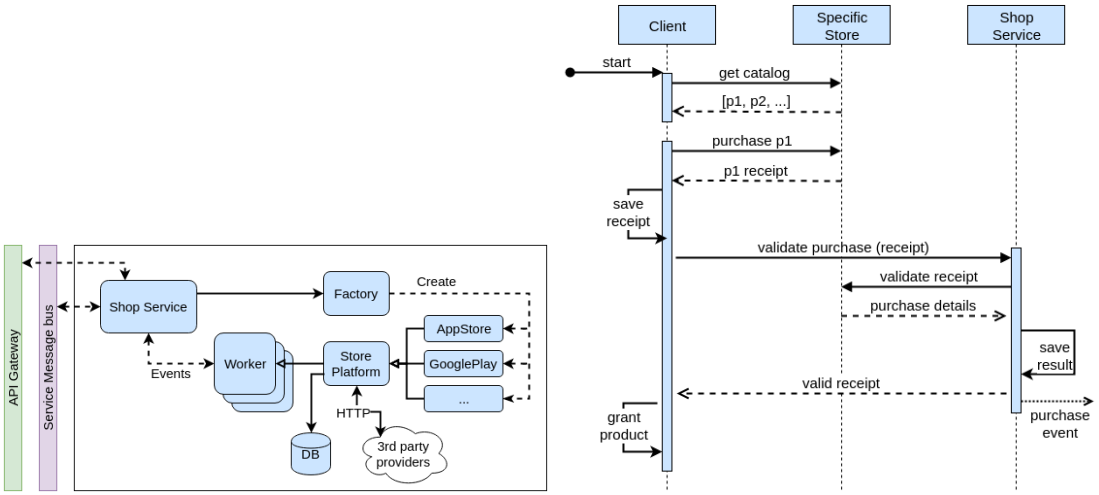


Figure 27: Data model for Shop service

A very simple catalog data model, every **Product** available for purchase is registered by the developers and **Catalog** is populated with the store specific information about the product. The most important field is *Catalog.product_sku* which must be unique in within a single store provider. For example, there can't be two products named "gold.100" in the same store, but there can be one product called "gold.100"

in Google Play Store and another “gold.100” in the App Store. Every product in the Catalog must have a corresponding entry in the Product table, in order to easily query statistics of sales. The **IAP** table reports all data representing a monetary transaction, it comprehends the *player_id*, the original receipt sent from the client and the result of the validation. Fig 28b shows the sequence diagram for a purchase operation.



(a) Shop microservice architecture (b) Sequence diagram for purchase operation

Figure 28: Shop Microservice

The architectural design is similar to the Social Service, with a Factory design pattern for interfacing with an external API provider. It is accessible from the API layer as a functional service, so it exposes APIs After the completion of a purchase request, the outcome is written in the database and a purchase event is emitted. Statelessness is guaranteed because all the information is given with every message, and the service only executes tasks on demand. Hence, X and Z scalability can be applied.

APIs (prefix: /shop)		
GET /:platform/catalog	DO /:platform/validate	DO /:platform/purchase
Events		
PUB shop:purchase		

Table 5: Shop service example APIs

4.5 Datastore Service

Saving player’s data online is fundamental for every game. It allows true account portability and definitive security for data integrity and prevention from client hacking.

Most of the state is persisted within the Datastore microservice, other services can rely on it to fetch, update or create content depending by the game logic. The Datastore shall:

1. Store user's data, including currencies and inventory. Data can be accessed and modified by the system at any time.
2. Store game business data, non necessarily related to player.
3. Store player's achievements.

Modeling data to suit every game would be not possible, since a predefined data model would greatly limit the flexibility. Instead, only a conceptual modeling will be given in Fig 29. As data needs to be flexible and easily editable, but at the same time organized in a complex nested structure, a document oriented database would be an appropriate choice for the underlying persistence technology. The service architecture is the same as the Authentication service (Fig 26a): a DataService entity listens for incoming messages and delegates the work to concurrent tasks, which will execute it and return the result in form of an event.

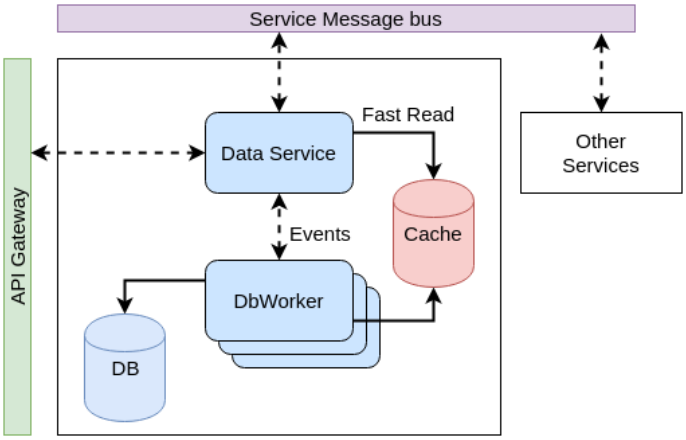


Figure 29: Datastore microservice architecture

Many modern NoSQL databases expose APIs to perform basic CRUD operation and also query data in efficient way across sharded nodes; the service's only duty is to wrap the specific database APIs to fulfill incoming requests. Examples of APIs are depicted in table 5. Datastore listens at events generated by Authorization service so it can allocate new space for the new user and initialize the data with default values.

Examples:

GET /data/users/xyz123/status: will answer with the Status object of the user xyz123.

SET /data/users/xyz123/wallet?gold=100&gems=50: will edit the wallet of the user xyz123 by setting 100 gold and 50 gems.

DO /data/query?select=item&category=weapon: will return all items in the "weapon" category

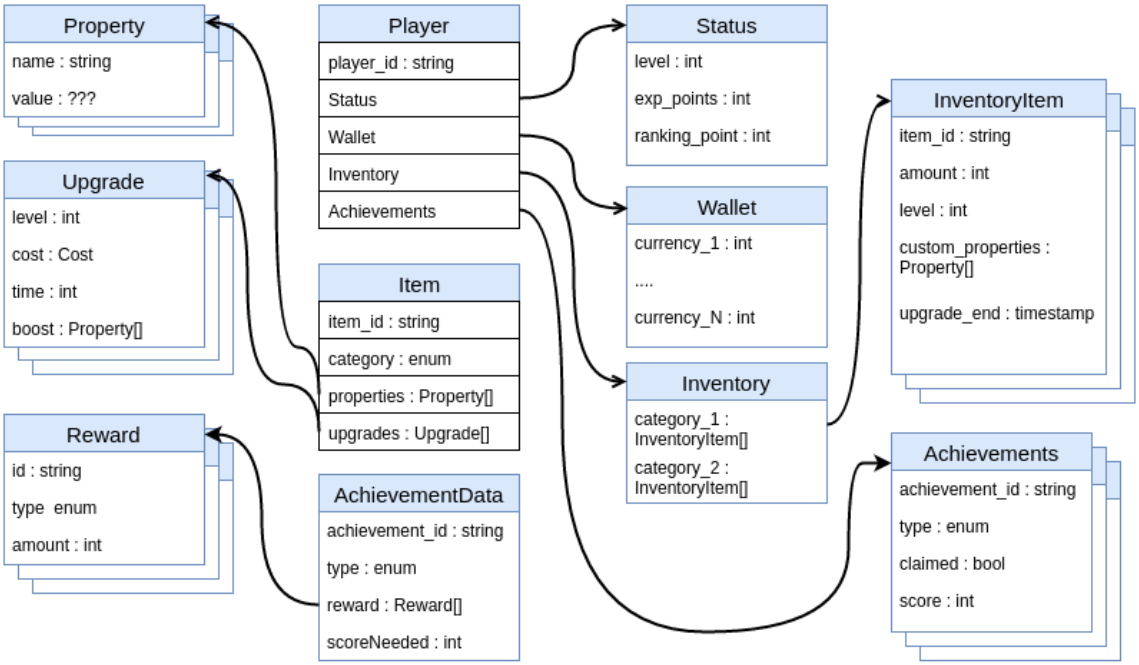


Figure 30: Data model for Datastore service

APIs (prefix: /data)		
GET /users/:id/...	SET /users/:id/...	DO /query
GET /achievements/	GET users/:id/achievements/	GET users/:id/achievements/
Events		
SUB auth:user_signin		

Table 6: Datastore service example APIs

4.6 Leaderboard Service

As shown in previous chapters, direct competition and skill improvement can be considered one of the most important ingredient for a successful game. As a consequence, a tool to better compare players is needed. This tool is the Leaderboard. A functional and flexible leaderboard should at least fulfill the following requirements:

1. Players are ranked by the means of a score, representing their skill level or progress in the game.
2. Multiple leaderboards shall be allowed, for different aspects of the game.
3. Players shall be able to compare them with the rest of the whole playerbase, with players in their same region, and their friends. Players can belong to only one region.

A leaderboard is stateful, it must preserve the players' information and their sorting order. It can be interpreted as a sorted hash set, in which every player is

uniquely identified by a key and sorted by a numeric value, representing the rank. In this setup, it's trivial for a set of nodes to concurrently process data from a shared dataset. The use of a clustered key-value in-memory database is the best choice for this scenario, as it keeps the complexity of state synchronization away from the service's logic and maintains very high performance for frequently changing data.

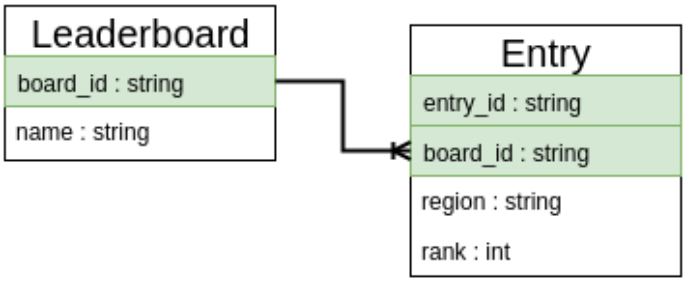


Figure 31: Data model for Leaderboard service

APIs (prefix: /leaderboard)	
GET /	Get a list of all available leaderboards
GET /:board/players?from=1&to=100	Get players from a global leaderboard. Parameters should allow to specify a range (e.g. from position 1 to 100)
GET /:board/players/:id	Get one specific player's rank
GET /:board/players?list=1,2,3	Get ranks of players given in a list
GET /:board/:region/players/	Get players belonging to a specific region
ADD /:board/players	Adds a new player to the leaderboard
SET /:board/players/:id?rank=123	Sets the rank of a player

Table 7: Leaderboard service example APIs

Leaderboard can be implemented as separate sorted hashsets, indexed by ID. Each leaderboard can contain a multitude of entities. There can be many configurations for organizing the data, each arrangement has influence on the scalability. One possible setup is using one Sorted HashSets to represent the whole set of leaderboards, and use lexicographical range queries to get different results. In this case, on a key-value store, an entry can be modeled as **<leaderboard_id:region:player_id, rank>**. For example, getting the list of all players in the leaderboard "boardX" can be done by querying for "boardX:*:*". For having the players in Finland the query will be "boardX:fi:*". For a specific player, the query will be "boardX:*:player_id", assuming that a player is unique in any leaderboard. This approach is good for optimizing space needed for the data structure, but is not convenient for maintaining performance. The cost of a lexicographical query is $O(\log N + M)$, where N is the size of the set and M the number of returned elements. As the number of players scale, having everything in one unique set can become a bottleneck. Moreover, not

every database supports lexicographical queries. A better solution would be to have separate indexed sets to query only when needed, as shown in picture 32

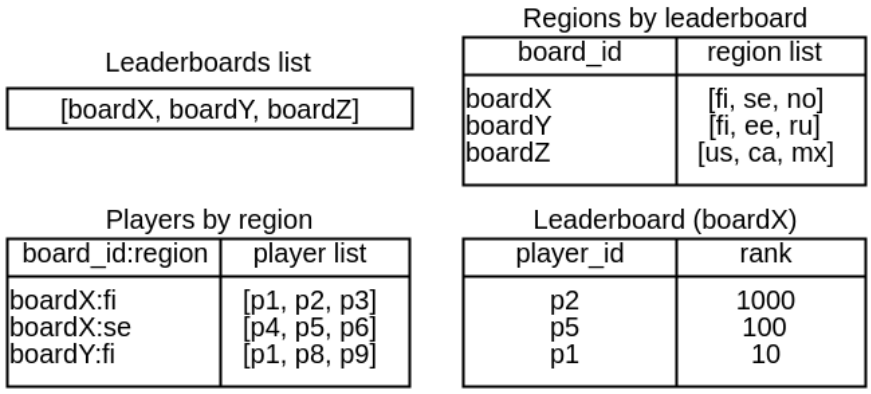


Figure 32: Data structures for Leaderboard Service

This simple data model allows the maximum flexibility and performance for queries. Having the indexes separated prevent the need for lexicographical queries, and all requests can be performed with the minimum time possible. Fetching the player's rank from a leaderboard can be done in time *O(1)*, getting the rank of M players costs *O(M)*. For retrieving all the players in a region, the cost is *O(1) + O(M)* where M is the number of the players in the region. The first cost is for querying the player list from the "players per region" set, the second cost for fetching player ranks from the leaderboard. The microservice will listen for messages coming from both the API Gateway and the Message Bus. Common requests, like "top 50 global players" can be cached in memory for a faster response time with a Time to Live (TTL) dependent by the number of updates (for frequent reads and updates, even 1 second TTL can greatly save performance). Since the state is preserved in a central clustered database, it's possible to achieve X-scalability by replicating the microservice among different nodes, all connected to the clustered database. Z-scalability can also be achieved if groups of nodes are partitioned to serve a subset of the leaderboards. Sharding a single leaderboard in different databases is not convenient, but even with a very large leaderboard with millions of entries (given that indexes are reasonably small) a commodity computer can suffice.

4.7 Chat Service

A chat service is the implementation of a message hub which players can connect to retrieve messages left there by other players, or for instant messaging. Messages in mailboxes are permanent, while instant messaging are usually non persistent. Concerns to an efficient chat system are the same as a general message system: it should be completely distributed and fault tolerant, which is not trivial to accomplish. For the messaging capabilities, the chat server will reuse a reserved Message Bus only for delivering messages n between the service nodes. The database is used

for persisting the messages sent by the players or the application. The in-memory database is used for volatile data such as connected user list and for supporting the implementation of the messaging protocol.

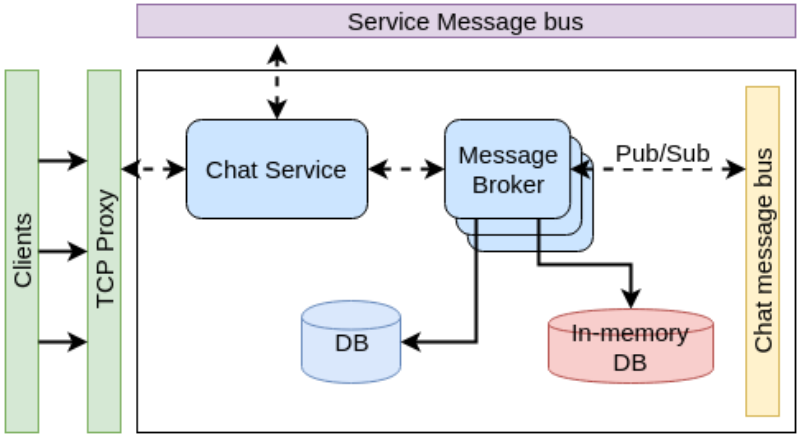


Figure 33: Chat microservice architecture

ChatService dispatches messages to message brokers, which are threads connected to an internal message bus implementing the PUB/SUB pattern. It accepts and redistributes messages representing the APIs in table 8. **MessageBroker** is a concurrent task that accepts messages, interprets them and executes the request. The main actions will be writing to one of the databases or to the internal message bus, and returning messages to the Entry Point to propagate to clients or the application.

APIs (prefix: /chat)	
CONNECT /players?id=xxx	Registers the user as connected to the service.
REGISTER /rooms?id=xxx	Creates a room, in which users will be able to broadcast messages. More options can be available, for example whether the messages should be persisted
JOIN /rooms/:room_id?player=id	Put a player into a room. It's equivalent to SUBSCRIBE, it will start to receive all messages published in the room
LEAVE /rooms/:room_id?player=id	Removes a player from the room, equivalent to UNSUBSCRIBE. Player won't receive messages anymore
SEND /rooms/:room_id	Equivalent to PUBLISH into a room. Parameters are used to specify the message and the sender
SEND /players/:player_id/:box	Sends a message to a player given its ID. Also a mailbox must be specified, in case there are more than one (it can be PM, whisper, friendlist etc...)

DISBAND /rooms/:room_id	Forces UNSUBSCRIBE for all the players registered and removes the room
DISCONNECT /players/:player_id	Player disconnected from service

Table 8: Chat service example APIs

The added value of the Chat service is that it can persist messages in an inbox, waiting for the player to fetch them when it will connect the next time. The data model for the database is shown in picture 34, it depicts a very simple relationship between a Mailbox and its messages. An entity (such a player, or a service) can own several mailboxes, each of them can be categorized with a type and uniquely identify by an ID. Every message generated must contain both the *inbox_id* and *outbox_id*, by which it's possible to identify the sender and receiver.

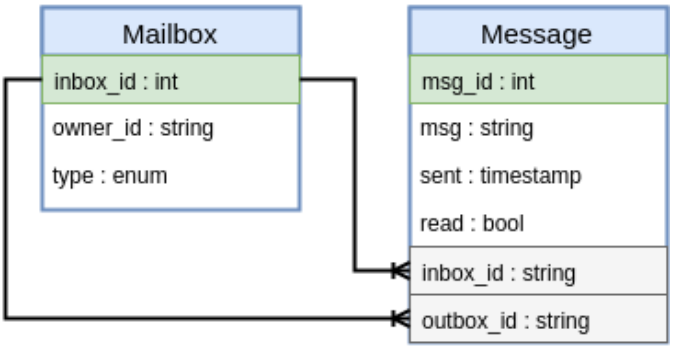


Figure 34: Data model for Chat service

4.8 Matchmaking service

Almost every multiplayer game has a matchmaking feature. Matchmaker is the software responsible of matching every player with the optimal team so that all players involved in the match have 50% probability of victory. First of all, a matchmaker relies on a value known as Matchmaking Rating (MMR), which determines the player's skill. It can be the same value used to rank the player in the leaderboard, or another calculated value in function of his statistics. There are several proven algorithms for calculating the MMR for a player, some of which derive from the ELO rating system used to rank chess players based on their skills. Currently, the Glicko2[20] and TrueSkill [24] algorithms are used in several popular games. Regardless the algorithm used to calculate the rating, a matchmaking algorithm only needs the MMR value. The service shall:

1. Create balanced teams so that the discrepancy between the players with maximum and minimum rating is minimized.

2. On each contending team, the player with the highest MMR should be as close as possible

3. Players can join the queue alone or in teams
4. Players shall be able to set preferences, and be matched with players who chose the same preferences.
5. Players shall be able to join existing games, if the case allows
6. Wait times should not be too long.
7. "Just let me play!" functionality to throw a player into a separate quick FIFO queue at the expense of balance.

A Matchmaking algorithm task is to sort players into separate groups under some conditions. The most simplistic condition is that every member in the group must have the minimum distance from the group's calculated average. This is a well known problem in data mining, and it's referred as **k-means clustering**: given a set of d-dimensional points, divide the set in k groups so that the sum of the distance functions of each point in the cluster to the center is minimal. Unfortunately, the problem is categorized as NP-Hard, computationally hard to solve. Many different variants of the algorithm have been proposed [27] to cover the most widespread application in data mining, most of them requiring that the initial set of data is known and static. This does not apply to a multiplayer matchmaker, where players join and leave waiting queue continuously. A streamed clustering algorithm could be applied to ensure that elements are partitioned already after one scan [21], but it would still require prior knowledge about the number k of total groups in which the set must be partitioned. Since the affluence of players cannot be predicted, k-means clustering does not represent an optimal solution for multiplayer matchmaking.

A more promising approach is the **K-Nearest Neighbour** (KNN) with aggregative algorithm. It starts from considering every single node a cluster by its own, and then iterate over all clusters by merging them together two by two, according to some distance metric, for example Euclidean Distance. This method produces very good approximations [9] but it also require that the initial set is known, and computation time is large if neighbours are computed for every new entry $O(n^2)$. Instead, another approach is explored using **Red-Black Interval Trees**, which will be referred as RBI tree from now on. An Interval tree is a data structure useful for storing numerical intervals, and efficiently tell if any given interval overlaps with others. It is similar to binary search trees and executing operations of search, insert and remove cost $O(\log N)$ where N is the size of the entries (in this case, the players queuing for a match). The red-black property prevents the worst-case scenario in which the nodes are disposed in long branches; ensuring that the tree is always balanced improves performance without considerable additional costs. It consists of attributing a color for each node coded in only one bit of additional information, and re-balancing the tree when the rule of Red-Black trees are violated.

Every player in the service is represented with its MMR, a weight and a timestamp. Actually, the service interprets the MMR as an interval of values $[MMR_{-}, MMR_{+}]$ that expands as the player ages. The interval growth rate can be tuned by developers. PlayerGroup is an extension of Player, useful for players who want to join a match

together. Weight is equal to the player's count, and MMR range can be calculated as the average range. A Group is just a container of players with a MMR interval equal to the average of all the players' intervals in the list. A static *Optimize()* function splits the given group in other optimal sub-Groups containing the best configuration of teams taken from the original groups. The actual implementation can be left to developers, since it really depends by the type of game.

Age is surely a parameter to take into consideration when creating matches, to guarantee that every player have the opportunity to play. If the Group is full or exceeds the maximum number of allowed players it always returns at least two sub-Groups, one of which will be eventually confirmed as a match and returned. The function can return the same group if it does not need optimization. The RBI tree is easy to implement: the base implementation is the Interval Augmented Tree variant, at which are added properties of a Red-Black tree in the Add and Remove functions. A class diagram is shown in figure 35 while in figure 36 there is a schematization of the microservice structure.

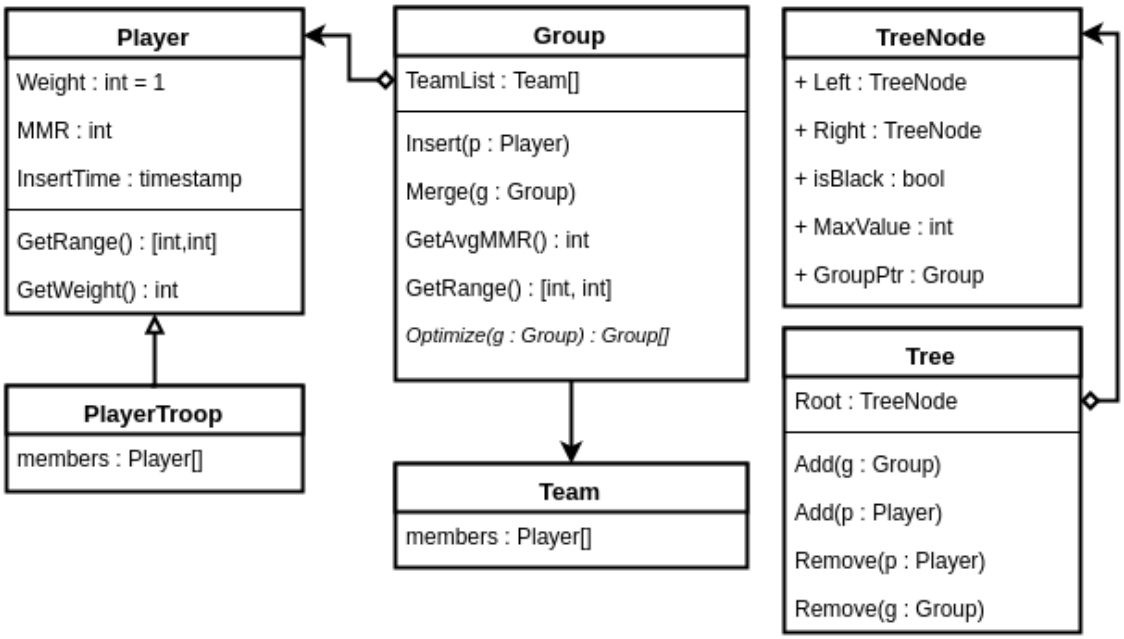


Figure 35: Class diagram for Matchmaking service

The entry point of the service, the **MatchmakingService** listens at requests incoming from the application message bus. In addition, it is connected to a high priority internal Message Bus used to exchange fast notifications only with the other nodes involved in the matchmaking. It may be useful for synchronization, or in case in which a player can't find a match and is aging too much, so the player is propagated to other queues for better chances to be matched for a game. For single-node deployments, the internal bus is not needed. It also collects the results of computation from the active tasks. After a message is received, it is delegated to one of the appropriate worker or back to the message bus. The service also tracks all

running games and players in them in a shared in-memory database, updating the state every time a game is created, disbanded or players leave.

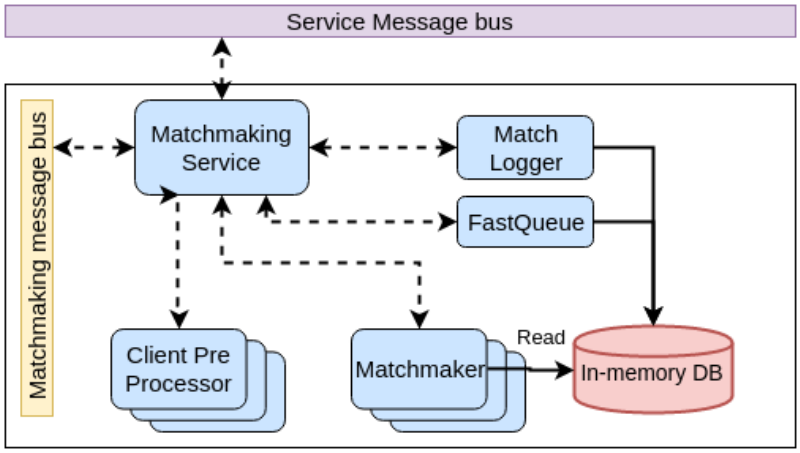


Figure 36: Matchmaking microservice architecture

PreProcessor is responsible for arranging players into preliminary groups using a RBI tree. This component helps to relieve the load from Matchmaker and mitigate bursts of requests, it can run in parallel without sharing memory with the main process because the task does not have external dependencies. If parallel computations are used, players are assigned to threads according to their MMR to increase chances of a match. After every defined time period, the PreProcessor gathers all the formed groups, executes the *Optimize()* function and returns them to the service entry point for further elaboration.

Matchmaker is responsible for executing the second step of the elaboration. It contains a RBI tree structure and a list with all the groups (referred as GroupList) currently waiting for a match. It can also match players with currently running games if some player is missing from them. The task consists in four main phases executed in a loop:

1. **Fetch:** accept the incoming list of pre-processed Groups from the Service and add them to GroupList. The sorting order is irrelevant. Also read the list of running games from the in-memory DB, filter out those outside the matchmaker range, create groups where existing teams are made as a unique PlayerTroop.
2. For every group G_i in the list:
   (a) **Search:** search in the internal RBI tree all the other groups overlapping with the G_i's MMR interval. If no group is found, then insert it as a new node and skip the Match phase.
   (b) **Match:** choose the group G_j with the most similar interval and merge it with G_i. Remove G_i from GroupList. Save G_j in a temporary list MergedGroups.

3. **Optimize:** Run *Optimize()* for every group in MergedGroups. Move the resulting full groups from GroupList to a temporary list MatchesFound. Add the newly created groups in GroupList.

4. **Report:** Return MatchesFound to the MatchmakingService.

It is worth noting that every time that the phase 1 starts, the previous groups will have a larger MMR interval because of their age increases with time.

FastQueue implements the "Just Let Me Play!" feature: when a "JLMP!" message arrives, the FastQueue adds the impatient player in a FIFO list stored in a in-memory database, shared among all the other matchmaking microservices. However, the player is not removed from the normal matchmaking process for not negatively impacting the waiting time for players who still want a fair play. If the list is full enough to accommodate a game, the FastQueue forms a Group containing the players in the list returns it to the Entry Point. The writes to memory should be synchronized to prevent multiple insertions at the same time while others are reading, leading to false Matches.

MatchLogger has the task to validate all formed Groups into Matches. This is a form of synchronization needed for preventing that a player is simultaneously matched both in the Matchmaker and in the FastQueue. To validate a Group, the MatchLogger must check from the shared memory that that every player is not already registered in a Match. If not, it removes those players in the "JLMP!" list and creates a new Match in the shared memory, marking them as "registered". If the validation fails, the misplaced players are removed from the Group and the Group is reintroduced in the MatchMaker.

4.9 Multiplayer Service

The Multiplayer service is a compound service used to automatically launch a game instance and let players connect to it. In addition it can create "Rooms", a virtual space in which clients connect and exchange real time messages.

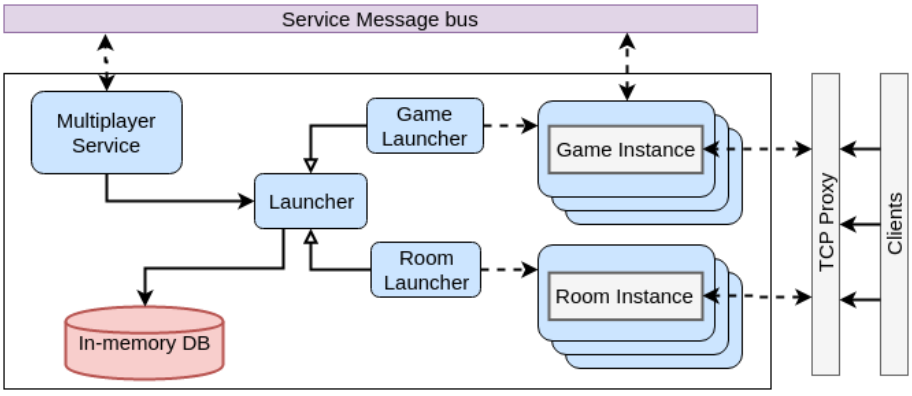


Figure 37: Multiplayer microservice architecture

The **Multiplayer Service** entry point listens for requests from the application, and uses a **Launcher** to create instances of games and rooms. A **Game** can be any software that accepts incoming connection from clients and executes the game logic. It is a trusted component so it can have access to the application's internal message bus for example to fetch the player's state before beginning the match. A **Room**, on the other hand, acts like a hub that only allows connected clients to exchange fast messages between them. In the latter case, the game logic can be run completely on the client side. Since delay time should be minimized, clients should connect directly with them without passing from the application message bus. Instances can be implemented as containers or virtual machines running in the same hardware, so to confine clients into a restricted space to enhance security. The Launcher has the task to create Rooms and Games when required, and register them on a in-memory database to keep track of all running instances. An instance usually has a lifetime. This can be decided case by case, but typically an instance is closed when the game ends or all the players disconnects. **Instances** could eventually be pooled to reduce response time. The matchmaker service can create and disband instances, give information about the status and the list of players in an instance. It listens from messages from the Matchmaking service to allocate an instance whenever a match is found, and prepare it according to the desired parameters for the match. After that an instance is created, a free port is assigned to it. The address and port are dynamically mapped in the proxy with the instance id, so clients only need to know the instance id to open a connection.

4.10 Logger Service

Logs and analytics are extremely important for game developers. It helps during the production phase to collect information on how the players use the game, records events of all sort and, most importantly, reports all errors and faults of the application, so it's possible to understand the causes of faults and promptly find a solution. Such service is easy to create and implement: it consists on a write-intensive distributed database with APIs to write different types of events. Table 9 shows possible APIs for the service, accessible also from the API Gateway.

As usual, the Entry Point is just a message broker and launcher for the real tasks. **Data Compactor** receives logs to write, but before writing them there could be some optimization to do like formatting, validation or additional elaboration. After the data is processed and ready to store, it can execute the query to the database. **Data Extractor** does the opposite: it executes queries from the database and elaborates the result in meaningful ways, for example to be plotted on a chart for visualization. The service also serves web pages to authorized users, such as the development team. An internal file server can be used if multiple microservices are run at the same time.

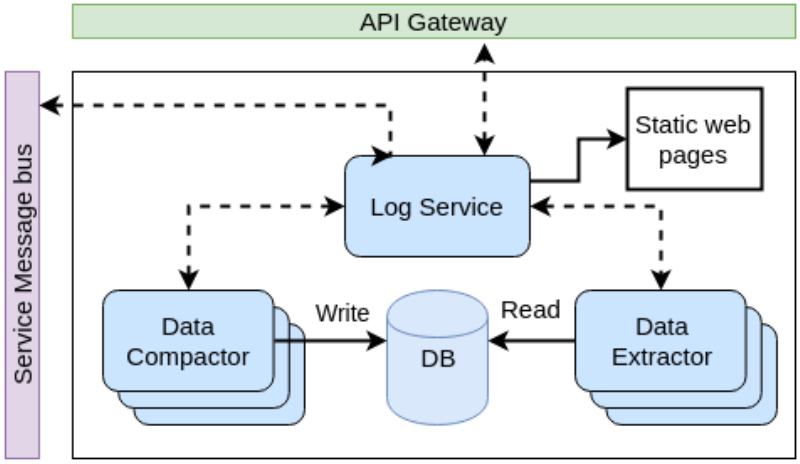


Figure 38: Logger microservice architecture

APIs (prefix: /log)	
GET /web/...	Serves static web pages if requester has the authorization
POST /event/:category	General event coming from the application. They can specify a category for the event and additional data related to it with parameters.
POST /heatmap/:category	Game-related event with spatial coordinates. It's useful to analyze particular events happening in the game, for example in which point players dies, camp and so.
POST /economy	Related to IAP and monetary transaction in general. Data of interest is from which platform the income is coming, what is the most popular item and from which country.
POST /ads	Advertisement are also a very important source of income that needs to be kept under control. For example, the game can send a message every time an AD is shown. This information can have incredible benefit.
POST /error/:level	Errors can be divided in critical, warnings, info and many other depending by the needs. A string message and source of error are also saved.

Table 9: Logger service example APIs

5 Use Cases and technology stack

5.1 Use Cases

This section shows some possible app configuration by proposing three different use cases, representing respectively a possible scenario for a small startup, a medium-size company and a big corporation.

5.1.1 Use case 1: Minimum setup

The first example explains a convenient configuration for a startup with low amount of expected concurrent players and a limited budget.

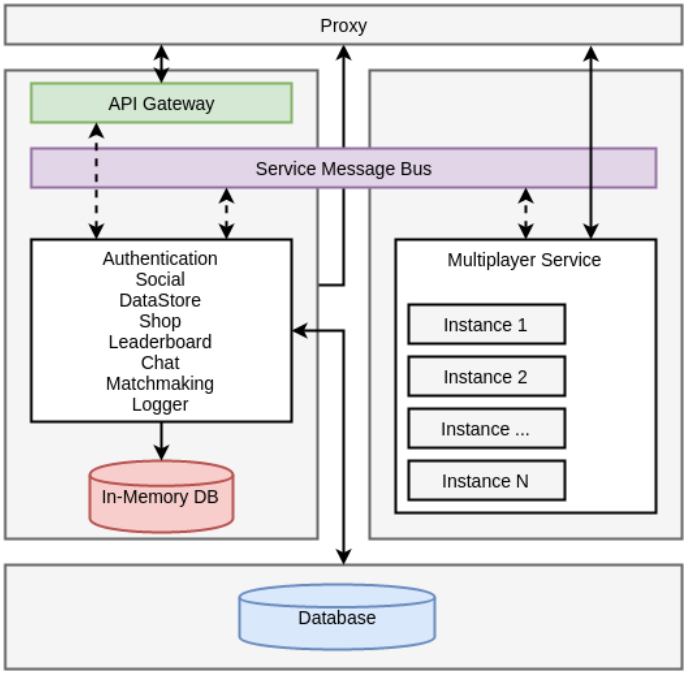


Figure 39: Minimal setup for a possible deployment use case

The four machines consist in a software reverse proxy server, a Service server, a Game server and a Database server. The Multiplayer service has a dedicated machine because multiplayer capabilities are the most critical and delay sensitive, so it should be separated from other services for not having negative impact on performance. The Service server comprehends all the services together, and all of them share the same in-memory database. The database for storage is kept on a separate machine with less computing power but more storage capabilities. Eventually, the architecture can be simplified even more by incorporating the database in the Service server. If the game does not need the multiplayer feature and the company is not concerned about external threats, all the system can be reduced to a single machine with the clients connected directly to the API Gateway. This configuration is still scalable on multicore machines since the services can run in different cores, and single services

could create threads to parallelize some tasks. However, if the hardware fails the system will be completely unavailable unless manual intervention from an operator.

5.1.2 Use Case 2: Triple redundancy

A good practice to prevent single points of failure is duplicate critical part of the system. In this case, the most critical parts are the message buses and the databases. Triple redundancy help to achieve fault resilience and high availability, if a node fails there are at least another two to sustain the incoming requests while a system utility will restart the faulty process.

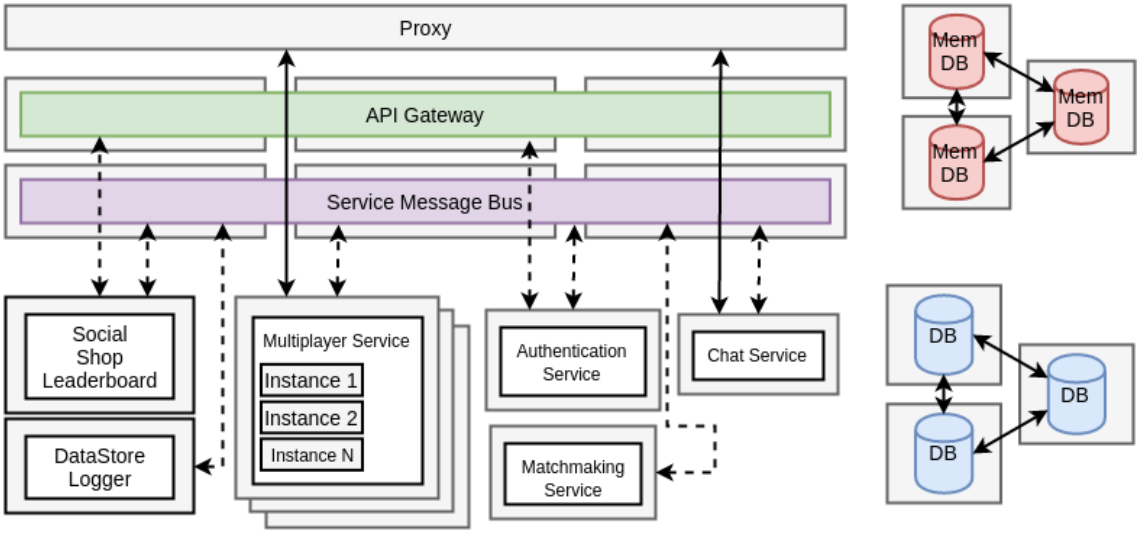


Figure 40: Setup for a discretely successful game

Here, both API Gateway and Service Message Bus span over three machines. This not only will make the messaging layer resilient to errors, but also performances increase because the load can be splitted into more computational resources. Database and in-memory database are clustered in a three node replication and shared with all the services. Some databases can be configured in master/slave mode, so that the application could write in the master instance and perform fast reads on slave replicas. Also, if the game is popular, more power is needed for running game instances and services can be assigned to more machines to split the load and keep the application performant.

5.1.3 Use case 3: Multiple Datacenters

For very popular games, world coverage is a necessary step for supplying a high quality of service and the best possible player experience. In datacenter configuration, every service is clustered and have a reserved database or clusters of databases, message buses have dedicated hardware to speed up delivery, and several proxies accepts incoming traffic from millions of users. Data is replicated among different

datacenters for a better protection against disasters. Similarly to service message bus, a Datacenter Message Bus can be implemented to communicate between datacenter in different Availability Zones for maximum redundancy and failover in case of severe faults without affecting the system availability.

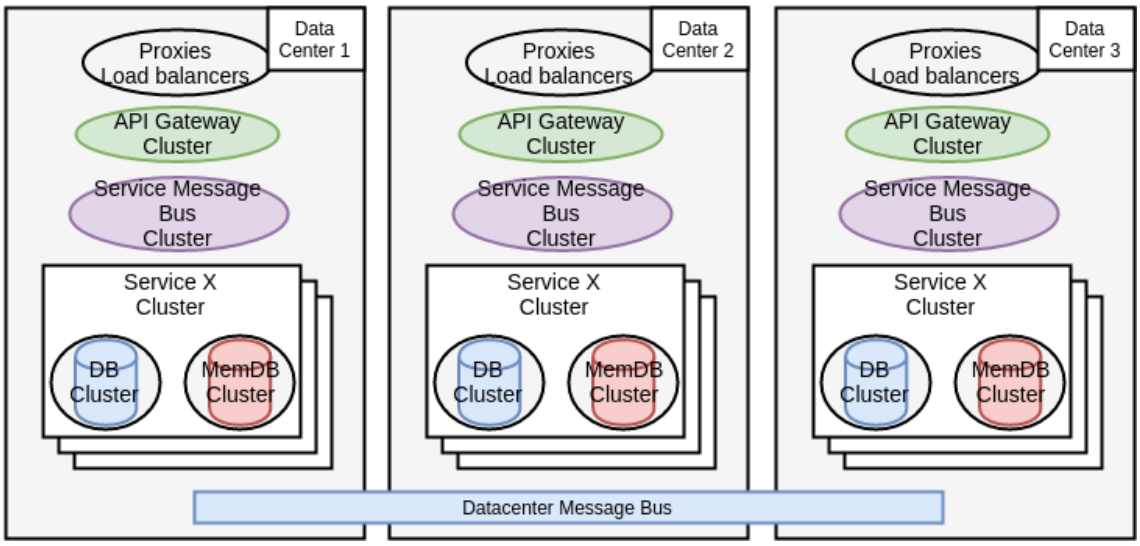


Figure 41: Multiple datacenter setup for a very popular game

5.2 Technology Stack

This section briefly provides an overview of the set of tools, protocols and technologies usable for the application implementation, and a comparison between them whereas multiple viable solutions are present.

5.2.1 Platform and infrastructure

The application is completely unaware of its physical location, it can run seamlessly on any physical or virtualized environment, on a single machine or in a cloud. This flexibility brings the advantage of letting the final customer to choose the platform to run the application. For their heterogeneous nature, microservices benefit from containerization for a better and more efficient management. Moreover, many platforms support containers and provide specific tools to ease the deployment and operations of containerized applications. **Docker**[5] is the current favourite technology for application containerization, and it's also the recommended choice for the amount of available tools for facilitating the development and deployment of the application, in addition to the thriving community around the project. The first and most obvious possibility for an entry-level company is deploying the application on virtual servers from a IaaS provider. Among the most popular IaaS providers there is **Amazon**

[5] `https://www.docker.com/`

AWS, **Microsoft Azure** and **Google Compute Engine**. Azure is convenient when working with Microsoft service and products, but limited in flexibility. Google's platform is a simple yet powerful IaaS that features automatic scaling in and out and full control over the network. Amazon AWS is the leader in the field of cloud infrastructure, it features a full set of services and tools for automating many types of operations, all composable together in a smooth and easy way. The resource scaling is automatic, and adding more instances is only a matter of few clicks. For companies willing to manage their resources on a private cloud, viable solutions are many. **OpenStack**[6] is an open source virtualization platform for cloud computing that can be installed in datacenter machines to have a unified control on the available resources. Resources are managed independently by several components, making OpenStack a great and solution for a flexible and customized private cloud. Unfortunately, this tool requires an expert operation team to install and maintain. More user friendly alternatives exist, at the cost of flexibility. **CoreOS**[7] is a minimal Linux based operating system for clustering containerized applications on distributed computing resources. It relieves the burden of operational maintenance from the development team, it promises automation, ease of use, security and scalability. Moreover, it comes with a built-in mechanism for service discovery and configuration sharing. If coupled with a container cluster manager like **Kubernetes**[8] or **Docker Swarm**[9] the application can be configured to be fully scalable and highly available.

The best candidate product for the application's platform is **DC/OS**[10], a distributed operating system for development, deployment and scaling of containers in production environment. It's based on the open source Apache Mesos project, a high scalable and available distributed kernel. The advantage of DC/OS is that it can be easily deployed on anything, from bare metal to virtual machines to clouds, and exposes the distributed resources as they were only one. Instances can be created and deleted with ease from a GUI or command line, the platform comes with monitoring tools already implemented or offers the possibility to integrate third party open source tools as plugins. Also, it has built-in load balancing and service discovery features, which can be used as base to build the application's Message Bus.

5.2.2 Proxies and load balancers

Reverse proxies are useful in many ways: for their capability of separating the application's interiors from the outer Internet, caching frequently requested content, dealing with request compression and encryption and balance traffic among different parts of the application for keeping it responsive. They can be implemented in software or hardware, but since hardware solutions are expensive, only software proxies will be considered. Several software proxies and load balancer exist, two of them are selected for comparison as the best viable possibility for this application:

[6] `http://www.openstack.org/`
[7] `https://coreos.com/`
[8] `http://kubernetes.io/`
[9] `https://docs.docker.com/swarm/`
[10] `https://dcos.io/`

Nginx[11] and **HAProxy**[12]. HAProxy is a fast, open source software that focus on high availability, load balancing, HTTP and TCP proxying. It is suited to application that must deal with high traffic loads and deployed on clouds, and fits very well to the application's needs. It's lightway since it does not do anything more than that. HAProxy runs on a single thread using a non blocking event loop, all I/O operations are blazing fast and scheduled with an priority-based scheduler. The architecture is optimized for moving data as fast as possible, and it's possible to configure it to overcome port exhaustion when more than 64k connection are open at the same time, thus making it a perfect candidate as part of the applications stack. Nginx is a fully flavoured, battle tested web server with proxy and load balancing capabilities. It is probably the most popular tool for developing highly scalable systems and web applications. Its popularity brings the community to build plugins and several integrations with third party tools which make Nginx an attractive option. It also comes with health check features to automatically detect faulty nodes in the backend. Both Nginx and HAProxy can be integrated with **Consul**[13], a highly available and distributed service discovery which can span multiple datacenters, making them a powerful tool not only for sustaining a large number of concurrent client connections, but also for building the application's routing and discovery service for inter-process communication.

5.2.3 Message Queue

As stated in the previous sections, an effective message routing system can be built exploiting load balancers in conjunction with a service discovery tool like Consul. However, there are also other possibilities worth exploring. A famous framework that serves at this purpose is **Apache Kafka**[14], a distributed streaming platform built by LinkedIn for internal use. Like message queues, Kafka is used to send asynchronous messages across the application in real time with very high throughput at a very small resources cost. It supports both publisher/subscriber pattern and the more classical queue, with an at-least-one message delivery guarantee. Kafka relies on Zookeeper, a discovery service needed to locate nodes in a distributed cloud, which has been reported [28] to have issues in recovery from faults in HA setups. On the other end, **RabbitMQ**[15] is a more mature broker-based solution which guarantees reliability and flexible routing of messages. It supports many protocols, including HTTP, Mqtt, STOMP and AMQP and it is very good to deal with messages of any type. While Kafka is a distributed system, RabbitMQ is centralized, and this can be a concern in some setups where availability is essential. Other solution include **ZeroMQ**[16] and its successor, **Nanomsg**[17]. These are very lightweight distributed asynchronous messaging frameworks, with embedded message routing, targeted for

[11] `https://www.nginx.com/`
[12] `http://www.haproxy.org/`
[13] `https://www.consul.io/`
[14] `http://kafka.apache.org/`
[15] `https://www.rabbitmq.com/`
[16] `http://zeromq.org/`
[17] `http://nanomsg.org/`

high throughput and low latency scenarios. Nanomsg is a more compact version of ZeroMQ, but removes most of its predecessor's unnecessary complexity to really focus on performance and features. It exposes several patterns: publisher/subscriber, request/reply, pair, pipeline, bus, survey. ZeroMQ has more. Due to their flexibility they don't impose any specific restriction, they just provide the building blocks for asynchronous, concurrent application. Nanomsg and ZeroMQ are recommended for those developers who want to build the Message Bus from scratch, otherwise both Kafka and RabbitMQ can be equally viable solutions.

5.2.4 Database

Throughout all the application, service make extensive use of databases and in-memory databases. The first are used for data storage, for preserving the service state, the second is used as a fast cache and a shared memory space visible by all the distributed nodes which compose the microservice. Since microservice have different requirements, the choice for the database vary according to the use case. What is certain is that all database must be partition tolerant. According to the CAP theorem, any distributed system can't be, at the same time, Consistent, Available and Partition tolerant. And given that a scalable application must be partition tolerant, the compromise is between consistency and availability. Similarly for the choice of specific database, every microservice can be implemented as consistent or available depending by the use cases and the specific game requirements. NoSQL databases (Not only SQL) approach the issue from different perspectives. There is indeed an increasing interest in NoSQL databases over the traditional relational databases, essentially because the traditional models don't scale well and are not partition tolerant at all, exception for some special cases. For services who are mostly writing data, have to deal with large datasets and prefer availability over consistency, like the Logger service, would benefit from a database like Apache Cassandra. **Cassandra**[18] is an open source column-based, fully distributed NoSQL database especially suited for large datasets. It's based on the Amazon Dynamo paper [14] consisting in a series of nodes disposed in a ring, communicating with each other and receiving notifications about changes in the topology thanks to a gossip communication protocol. In masterless architectures every node is the same, and it scales linearly by adding more nodes to the cluster. Data is partitioned in nodes according to a calculated function of the data primary key and a node's partition key; data is also automatically replicated in different nodes for maximum fault tolerancy. What makes Cassandra special, and thus suitable for many purposes, is that it can also satisfy the consistency property if explicitly required (in spite of availability). This is particularly useful for services like Shop or Datastore, where consistency is a must. Alternatively, document-based databases like **MongoDB**[19] shines especially for services who have to deal with complex data and need querying capabilities, something not easily practicable in Cassandra. Consistency over availability is a plus in some cases, where data can be cached or the service is not read-intensive.

[18] `http://cassandra.apache.org/`

[19] `https://www.mongodb.com/`

Similar to MongoDB is **RethinkDB**[20], a JSON distributed database with a rich query support and real-time update notifications. About in-memory database in short: **Redis**[21]. Redis is the number one choice for a fast, key-value database meant to keep all data in memory. It comes with built in data structures like hashtables and sets for fast lookups and caching. Combined with **DynomiteDB**[22], a dynamo like distributed and high available database system with pluggable backend, the system can benefit from a scalable, globally distributed cache. Like Cassandra, also DynomiteDB is masterless, avoiding the problems coming from a master/slave architecture (for example, coordination overhead) and preferring high availability over consistency, indispensable requirement for performance and responsiveness.

5.2.5 Programming Language

As databases, the choice for the right programming language depends on many factors. Microservice oriented systems are very flexible because they can be developed separately, with different set of tools and languages. There is a fundamental consideration to do when dealing with concurrent systems, the programming language should be able to somehow reflect the concurrency model suggested by the architecture. Concurrent and functional programming languages suitable for scalability are gaining more and more attention lately, and being widely adopted by both corporates and mid-sized companies. Functional languages treat computation as evaluation of mathematical functions, and enforces the use of immutable data and fixed state. The return value of every function depends only on the input parameters, avoiding involuntary side effects elsewhere in the program. Some popular languages that embrace these characteristics are Scala, Erlang and Go. **Scala** is a language which promote concurrency and distribution through the use of Futures and Promises, placeholder objects that represent the result of an asynchronous operation that will eventually be completed. It incorporates both the object oriented as well as the functional programming paradigm, it runs on the JVM and it's fully interoperable with Java. Moreover, functions are first class objects, meaning that a function can be treated like a variable and passed around with great flexibility of application. **Erlang** is another functional language used principally to build highly scalable, available and real time applications with features directly supporting distribution and concurrency. Among its features, there's the possibility to update the application's code without any interruption or restart. Erlang processes are very lightweight and OS independent, making it a cross platform language. A collection of libraries is also available called OTP, collecting the patterns and best practices from years of real-world expertise. **Go** is an open source programming language, with a concise syntax and built in concurrency primitives: Channels and Goroutines. Goroutines are similar to threads, with the difference that they are more lightweight and works efficiently also in single core machines. Channels are used as message passing mechanism between goroutines, synchronization and buffers. All the aforementioned languages can fit the application design since the

[20] `https://www.rethinkdb.com/`
[21] `http://redis.io/`
[22] `http://www.dynomitedb.com/`

services make extensive use of concurrent tasks that can be parallelized. As a special side note, also the popular **Node.js** is a good choice. A node application runs on a single process, executing requests coming from outside in a loop called Event Loop. Time consuming I/O operations and network calls are executed asynchronously in background, and a callback function is registered for execution when the process returns. When the asynchronous operation completes, it sends an event to the main process which triggers the callback function. Node.js can be run in cluster mode: a copy of the application is run for each CPU core, each on one separate process sharing the same port, and incoming request are distributed between child processes. In cluster mode, node applications partially avoid the complexity and disadvantages of multithreading.

6 Final remarks

This chapter presents a brief evaluation of the proposed application design, assessing whether the application could achieve a horizontal scalability according to the thesis goals. For concluding, the final section will suggest guidelines for a possible future work and improvements

6.1 Discussion

From the feature list analyzed in chapter 3.1, nine major recurring services were recognized: multi account user authentication; shopping service with receipt validation; social service integrated with Facebook, Google Play and Game Center at least; Datastore service for saving player related data and progress; Achievements, which was incorporated into the Datastore service because of similar functionalities; a leaderboard service supporting multiple regions; a chat service for both instant and permanent messaging, supporting player-to-player communication or broadcast to all participants in a room; a Matchmaking service to match players according to their ranking similarity; a Multiplayer service to manage custom game applications and allow realtime communication between players; a logger service to collect useful statistics from the clients and application itself.

A microservice oriented architecture was chosen as main design reference due to the proven capability of microservices to enable highly scalability for their decoupled nature. The application was designed following the Reactive Manifesto best practices, which promote responsive, resilient, elastic, message driven systems. Each service was designed as a separate, standalone application according to the share-nothing principle. Each of them can be accessed by the means of an API, they collaborate together exchanging messages in a choreographed manner using the inter-service message bus. These properties ensure a high level of decoupling, with no direct dependencies between services. Messaging is realized by a separate component, the Message Bus, which was designed as a distributed message queue with load balancing and service discovery capabilities in order to react when nodes are added or removed, and promptly redirect the traffic to healthy nodes. In addition, fault tolerancy and resilience are accomplished by implementing a circuit breaker pattern in the message bus, preventing system overloads and allowing a graceful service degradation and error handling exploiting message backtracking. All errors can be reported to the logger service, giving developers insights about the possible bottlenecks and possible causes of faults.

Services are secured behind a layer of proxies, representing a SSL termination point that take care of request encryption and decryption, so that the traffic inside the application can be unencrypted for better performance. Inner services are never directly exposed to clients, so they can block all traffic coming from nodes outside the private network. Moreover, each request needs to be authenticated before being routed. Every service must authorize the player for every request (or before accepting an incoming connection) by inspecting the player's permission from an access control list.

Conforming with the AKF Scale Cube, the application can scale in all the X, Y and Z axis. At a high level, the application complies with Y scalability by splitting the complexity in modular, separated and encapsulated components. Then, at the service level, each service can be scaled along the X axis by just replicating it on new nodes and, similarly, along the Z axis by partitioning the data to elaborate and to the same group of nodes. The concurrency model chosen for the application adopts stateless workers with asynchronous message passing, exception made for the matchmaking service which implements an assembly line [6]. Due to the complexity of creating a fully automatic elastic system, third party tools will be used for supporting automatic deployment or disposal of resources according to load. In the same way, fault recovery will be performed by separate tools.

Threads were chosen to guarantee true scalability also among CPU cores, to exploit all available resources, and because most programming languages have native support of threads, or have at least some simple mechanism for parallelizing code execution. Unfortunately, in some cases threads are not the best way to achieve high performances due to frequent message-passing between processors and possible bottlenecks due to possible cache invalidation. Threads are perfect for workers that have to undergo a huge amount of repetitive work with no frequent message passing, for example the matchmaking service. In other cases, an unique event loop per service would be more beneficial for services who only do mostly I/O or network operations. Environments like Node.js allows a process to run in cluster mode where a process is run in multiple cores, and a master process load balances the requests to the child processes via IPC. Since the performances are very implementation dependent, an interesting experiment would be to implement the same service in three different ways (with threads, with one event loop and with cluster mode) and observing its performance under increasing amount of load.

Three possible configuration scenarios were given, respectively for a minimal deployment, a medium company with a limited number of players and a big company maintaining a very popular game. Lastly, possible stack technologies are investigated including the platform, load balancers and proxies, message queue implementation, databases and programming languages. Preferences about the platform in which the services will run are also explicated (Docker containers on top of DC/OS).

6.2 Future work

As the thesis only describes the architectural design, the obvious next step for the future is continuing the project with the implementation phase. Firstly, a simple prototype should be made using as much available software as possible. The first tests should be deployed on a configuration resembling the minimum setup shown in chapter 5.1, aiming to evaluate the actual performance and behaviour of the system under increasing load. Tests should be developed to measure each service both independently and together with other services. Tests should report response time, number of operations performed per second, number of successful requests and failed requests, time for the system to recover from faults and time needed to adapt to varying load conditions. The second step will be to fine tune the system

and adjust the design to overcome the bottlenecks found during the testing phase, eventually rewriting services or replacing technologies in the stack, and iterate until a satisfying level of performance and scalability is met. Successively, new service can be added to cover even more use cases and necessities which were not considered in this thesis but are important for the company. Some additional services can be advanced analytics for advertisement campaigns, a unified channel attribution service to track application installs, a push notification service, a file storage service for serving static files (e.g. patch updates, downloadable content), and a dedicated solution for MMO games.

6.3 Conclusion

Mobile game market is increasing in popularity year after year, attracting a wide audience of independent developers who must endure the competition of other more resourceful game companies. Players expect high quality games and experiences, while developers strive to monetize. Researches have shown a correlation between some features of a game and its likelihood to succeed and be a potential candidate to enter the top grossing lists. Some of the currently most popular mobile games were analyzed in order to recognize the most common services, confirming the results occurring in precedent researches. There is the need for a comprehensive application which provides all the services to support the creation of online games, without the restrictions and costs that many existing service providers impose, so that developers can concentrate in creating top quality game experiences rather than spending time and resources in building a backend application for their game. This thesis describes the architectural design of a horizontally scalable backend application for online games that provides essential services for mobile games. The design followed the Reactive Manifesto guidelines to achieve both inwards and outwards horizontal scalability, modularity and flexibility thanks to a microservice oriented architecture and an event-driven communication between application components. Services are designed as stateless, share-nothing parallel tasks to facilitate the implementation with the current programming languages. As the application is designed to cover the most common game requirements, is far from being an all-inclusive solution that can be blindly applied to every mobile game. However, the proposed design and technology stack can be used as a starting point and adapted in a way that best suits the specific feature that the game needs. The author encourages to share and refine the present work, with the hope that it will contribute to the creation of the next awesome game.

7 References

[1] Martin L Abbott and Michael T Fisher. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Pearson Education, 2009.

[2] Ali M Alakeel. A guide to dynamic load balancing in distributed computer systems. *International Journal of Computer Science and Information Security*, 10(6):153–160, 2010.

[3] Gamasutra Alex Wawro. Survey: Video ads are the no.1 way players prefer to 'pay' for mobile games. `http://www.gamasutra.com/view/news/269819/Survey_Video_ads_are_the_1_way_players_prefer_to_pay_for_mobile_games.php`. Accessed: 2016-08-22.

[4] Khaled Mohammad Alomari, Tariq Rahim Soomro, and Khaled Shaalan. Mobile gaming trends and revenue models. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 671–683. Springer, 2016.

[5] App Annie. App annie 2015 retrospective. `https://www.appannie.com/insights/market-data/app-annie-2015-retrospective`. Accessed: 2016-10-02.

[6] Jenkov Aps. Concurrency models tutorial. `http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#parallel-workers`. Accessed: 2016-09-22.

[7] Peter Askelöf. Monetization of social network games in japan and the west. 2013.

[8] Rahul C Basole and Jürgen Karla. On the evolution of mobile platform ecosystem structure and strategy. *Business & Information Systems Engineering*, 3(5):313–322, 2011.

[9] Nitin Bhatia et al. Survey of nearest neighbor techniques. *arXiv preprint arXiv:1007.0085*, 2010.

[10] André B Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203. ACM, 2000.

[11] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[12] GA DataScience. How to identify whales in your game. `http://blog.gameanalytics.com/blog/how-to-identify-whales-in-your-game.html`. Accessed: 2016-08-17.

[13] Myriam Davidovici-Nora. Paid and free digital business models innovations in the video game industry. *Digiworld Economic Journal*, (94):83, 2014.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[15] Nicolas Ducheneaut and Robert J Moore. The social side of gaming: a study of interaction patterns in a massively multiplayer online game. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 360–369. ACM, 2004.

[16] Simon Egenfeldt-Nielsen, Jonas Heide Smith, and Susana Pajares Tosca. *Understanding video games: The essential introduction*. Routledge, 2016.

[17] Statista Felix Richter. Freemium is the no.1 pricing strategy in most app categories. `https://www.statista.com/chart/1733/app-monetization-strategies/`. Accessed: 2016-08-10.

[18] V. V. Filho, Á V. M. Moreira, and G. L. Ramalho. Deepening the understanding of mobile game. In *2014 Brazilian Symposium on Computer Games and Digital Entertainment*, pages 183–192, Nov 2014.

[19] Shalom M. Fisch. Making educational computer games "educational". In *Proceedings of the 2005 Conference on Interaction Design and Children*, IDC '05, pages 56–61, New York, NY, USA, 2005. ACM.

[20] Mark E Glickman. Example of the glicko-2 system. *Boston University*, 2012.

[21] Sudipto Guha, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams. In *Foundations of computer science, 2000. proceedings. 41st annual symposium on*, pages 359–366. IEEE, 2000.

[22] Perttu Hämäläinen, Joe Marshall, Raine Kajastila, Richard Byrne, and Florian Floyd Mueller. Utilizing gravity in movement-based games and play. In *Proceedings of the 2015 Annual Symposium on Computer-Human Interaction in Play*, pages 67–77. ACM, 2015.

[23] Cal Henderson. *Building scalable web sites*. " O'Reilly Media, Inc.", 2006.

[24] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill^{TM}: A bayesian skill rating system. In *Advances in neural information processing systems*, pages 569–576, 2006.

[25] Mark D Hill. What is scalability? *ACM SIGARCH Computer Architecture News*, 18(4):18–21, 1990.

[26] Shang Hwa Hsu, Ming-Hui Wen, and Muh-Cherng Wu. Exploring user experiences as predictors of mmorpg addiction. *Computers & Education*, 53(3):990–999, 2009.

[27] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[28] Tomasz Janczuk. From kafka to zeromq for real-time log aggregation. https://tomasz.janczuk.org/2015/09/from-kafka-to-zeromq-for-log-aggregation.html. Accessed: 2016-09-12.

[29] Raine Kajastila and Perttu Hämäläinen. Augmented climbing: interacting with projected graphics on a climbing wall. In *Proceedings of the extended abstracts of the 32nd annual ACM conference on Human factors in computing systems*, pages 1279–1284. ACM, 2014.

[30] Daniel King, Paul Delfabbro, and Mark Griffiths. Video game structural characteristics: A new psychological taxonomy. *International Journal of Mental Health and Addiction*, 8(1):90–106, 2010.

[31] Tracy Lien. Farmville 2 represents the next generation of social games, says zynga. http://www.polygon.com/gaming/2012/9/5/3290747/farmville-2. Accessed: 2016-08-22.

[32] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[33] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: a performance comparison. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 386–393. IEEE, 2015.

[34] SocialTimes Neil Vidyarthi. How lessons learned in social will give you a head start in mobile. http://www.adweek.com/socialtimes/how-lessons-learned-in-social-will-give-you-a-head-start-in-mobile/88692. Accessed: 2016-08-16.

[35] Sam Newman. *Building Microservices*. " O'Reilly Media, Inc.", 2015.

[36] Investopedia Ravi Srikant. The economics of gaming consoles. http://www.investopedia.com/articles/investing/080515/economics-gaming-consoles.asp. Accessed: 2016-08-11.

[37] Thierry Rayna and Ludmila Striukova. 'few to many': Change of business model paradigm in the video game industry. *Digiworld Economic Journal*, (94):61, 2014.

[38] Matt Ricchetti. Gamasutra: What makes social games social? http://www.gamasutra.com/view/feature/6735/what_makes_social_games_social.php. Accessed: 2016-08-05.

[39] Mark Richards. *Microservices vs. Service-Oriented Architecture*. " O'Reilly Media, Inc.", 2016.

[40] Richard Rouse III. *Game design: Theory and practice*. Jones & Bartlett Learning, 2010.

[41] Soomla Blog Sid James. Top 10 parse alternatives for your game backend. `http://blog.soom.la/2016/02/top-10-parse-alternatives-game-backend.html`. Accessed: 2016-07-28.

[42] Barrie Sosinsky. *Cloud computing bible*, volume 762. John Wiley & Sons, 2010.