Proximity-as-a-Service

Letting ISPs Sell Fog Computing as a Cloud Service

Kim Smedsrud

Thesis submitted for the degree of
Master in Network and System Administration
30 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

Proximity-as-a-Service

Letting ISPs Sell Fog Computing as a Cloud Service

Kim Smedsrud

Proximity-as-a-Service

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

We live in a society that is becoming increasingly digitized. Cloud computing is more popular than ever, and digital extensions appear everywhere.

Over the past decade IoT and edge computing have emerged with smartwatches, cars, and all sorts of devices and applications requiring faster response time.

CDN and PoPs are still able to meet latency requirements for most applications, but for real-time interaction, milliseconds are of essence. Many applications only need the computing power, and not data storing, like gaming for example.

In this thesis we assume the perspective of the ISP in order to improve customer QoS as well as opening new venues for revenue. We investigate how we can use recent technologies to enable edge computing at hubs and nodes to increase user experience and proximity.

We have developed an independent model and a prototype which achieves to simplify the ISP infrastructure. This model has been used to create three distribution strategies that easily can deploy the desired number of containers. In order to create a sellable product the ISP can offer to content providers and other parts of the digital entertainment industry, we apply the ”pay as you go” price model.

Essential results form this project is that the scale of a complex deployment has successfully been abstracted away.

Contents

List of Figures

List of Tables

Preface

This report is the master's thesis for Kim Smedsrud and concludes my study in Network and System Administration (NSA) at the University of Oslo (UiO). The thesis is written spring of 2020.

Acknowledgements

I would like to extend a big, and sincere thank you to my supervisor Kyrre Begnum. He has been a super in every way possible, and a great support these past months.

Thanks to my fellow students, especially Ali Arfan. We have had some interesting discussions and supported each other this semester.

Thank you Jonathan Lu for some general tips and for taking the time to read parts of my work and giving me feedback.

Thank you OsloMet for providing the ALTO Cloud, which I have used to host my virtual machine.

I would also like to thank my neighbours, close friends, and family for support and uplifting me during difficult times these months.

Chapter 1

Introduction

We are moving towards a world where all parts of society eventually will have a digital extension of themselves. Some decades ago, computers were only seen in the workplace. Today our cellphones are more powerful, and more affordable than ever. Technology has evolved at a speed no one had anticipated, and the latter introduces more robust tools which allow us to collaborate and communicate in more effective ways. In addition, many companies invest a lot of time, resources and business in technology, as the latter may strengthen and increase the companies revenue and reputation at scale. At the same time, all kinds of information are available with just a few keystrokes. Critics are concerned about this breakneck speed and are uncomfortable with this accelerated digital development and transformation, since the privacy and security are two of the many factors which are not being taken seriously.

A lot of social functions and elements are depending on technology and digital availability. Not many years ago, manually setup and configuring servers in offices were common. Today more or less everything still runs on servers, but their location has changed from office buildings, to huge data centers around the world. They make up what we now call the cloud. However, the lift and shift to the cloud has been gradual. In the end, everything runs in the cloud, which is nothing more than hundreds and thousands of machines working together in one or more data centers. The cloud has become the one choice for many companies since servers now are operated and managed by cloud providers.

The cloud creates opportunities, but a problem that comes with the cloud is that it is extremely complex. Still, many nontechnical people understand the concept, but they do not care about the architectural complexities below. The word cloud masks its complexity, which can be considered as positive, since the underlying technical concepts are hidden. Neither should we forget that the cloud enables distribution and scale, and shapes the future for many companies across the world.

The cloud is complex, but necessary. In some cases, the architecture cannot be ignored. Applications and services have different requirements,

and today most of them run in the cloud. Some do not need all the power the cloud provides, but they still run there because it is smart, simple and cost-effective, while many others depend and require the power the cloud provides. This is an opportunity to scale their distribution. Netflix, Facebook, Amazon and Google are good examples as they have billions of users all over the world who are utilizing their services. Another example is online gaming which stands out in the matter of latency amongst others.

For most service providers, the cloud covers all, or almost all, requirements. However, when the data centers in the cloud are not enough, the architecture becomes the problem. There are rarely any problems with the technology itself, but for some, geography becomes a problem. In essence, the closer the server is to the end-user geographically, the better the service the user most likely will experience. Companies like Facebook and Netflix have for several years already challenged and worked with the cloud vendors, experimented and made use of their economic advantage to navigate and develop solutions that could help themselves as well as others to get closer to the end-user, at the network edge. Far from every service or content provider is that fortunate.

The third, often overlooked, but important party, namely Internet Service Providers (ISPs). ISPs have a distinctive advantage when it comes to geographical proximity to the user. Their business is to deliver services to the end user at the network edge. Service providers like Netflix already utilize the concept Point of Presence [1], which is located at many ISPs, and is often just a server rack that helps them put geographically relevant data near the users. For example, they can cache the next episode in popular series. This is not edge computing, but maybe one step closer to another paradigm where service providers get the opportunity to use existing infrastructure locations to deliver improved and new services?

As we just mentioned, some digital service providers have a rack geographically placed at ISPs, but for most service providers, this is not an option, and there is no mass product offered by anyone to make it possible.

Lack of products similar to what we have described here does not directly imply that those products would not have been competitive in today's market. However, lack of knowledge and practice are possible challenges that keep these products from emerging over the past decade. There has been brought more and more attention to edge computing, which is a good thing, because it can provide knowledge in the domain of providing computing power at the network edge. But at this point, it is unfortunately mainly practiced in the academia.

However, new technological developments, such as containers, may satisfy the need and could potentially give more value to the matter. Containers are lightweight, easily managed, and do not require a lot of hardware resources. The network edge can be in someones home, different sensors out in the streets in lampposts, or cameras. Many edge locations are not very spacious, which make small computers a good choice in hand,

and small computers are rarely very powerful, which satisfies the need.

The industry is already experimenting with Internet of Things (IoT) in some ways, but if it is profitable, containers would allow us to explore the academic ideas in more realistic ways. This would help the development of future products available to a broader customer base.

1.1 Motivation

To be able to use a technology where intended manual work is the only possible way to change the configuration, and still keep required resources to a minimum, could enable the possibility to place lightweight computers closer to the end-user, and thus provide digital services in a whole new way.

If we could use containers as a suitable platform for an edge-computing service, it would enable the widespread use of products for those whom it was unattainable for before. It could let the ISP rent out a form of runtime capacity within its own network infrastructure which would let content providers push their content to the almost nearest possible physical location outside of their users home. As such it would provide the concept of proximity as a service. Meaning that content providers can use a product that will enable them to optimize the proximity of their own service towards their users.

Problem statement: *How can recent technologies like containers contribute to facilitate edge-computing towards ISPs delivering Proximity-as-a-Service (PraaS)?*

Containers is a widely used virtualization technology paradigm. We could also have used other lightweight type of systems, like unikernels. What they all have in common is that they are lightweight and platform independent, which is the most important attribute for this project.

Edge-computing is the concept of providing computation-capabilities into a network infrastructure. It is not the same as establishing data centers in neighborhoods at multiple locations, but rather adding computing power at edge locations of a network infrastructure.

This project does not see itself as a end station, but rather a starting point which would stake out a path for future projects to follow. We want to contribute to *facilitate towards* future work with ISPs.

The vision this project strides towards, is that the idea of *proximity* could become a type of service that would be comparable and recognizable as the other types of cloud services that are available today.

Chapter 2

Background

This chapter will provide background and research, introduce technologies and concepts, and provide discussions when it fits the purpose. Several paradigms and technologies will be presented from different perspectives, to get an understanding of the complexity, but still, how connected some fields within computer- and information technology are.

IoT, cloud computing, and virtualization are well known research areas. The relationship between ISPs, fog- and edge-computing, however, have limited focus. Hence, the amount of available literature is less too.

The author of this thesis has worked in an ISP, for almost five years, not to mention worked with computers in different settings for many years. During these years, the author has experienced how technology has been used to create both services and devices.

The chapter tries to identify similarities and differences between the industry and academia within several topics and fields.

Academic research will be at the center, but, some topics have been more popular in academia than others, therefore knowledge combined with experience from the industry and the author will be presented as part of this chapter.

2.1 The Ancestry of Edge Computing

In the past few decades technology has developed rapidly. Computers and the internet, virtualization and the cloud, are all comprehensive topics that there have been written several books about. To better understand the term edge computing, we have to take a small journey.

2.1.1 The User's Journey

Edge computing aims to provide computing power close to the end-user. Just a little bit more than a decade ago, around 2005, smartphones were

rare. Phones were used to call each other and for text messages. 4G was unheard of, and stationary computers and laptops were the only clients with a keyboard connected to the internet for everyday people. Few years later, around 2010, smartphones were used to check emails and browse the internet, and the number of clients connected to the internet was exploding. By 2015, smartphones became as powerful as old or new cheaper computers. Sharing of pictures, videos, and other information were becoming normal. From around 2015 up until today, technology has become so small, yet powerful. Small computer chips can be inserted into pretty much everything and connected to the internet via 4G or WiFi. Back in 2015, Internet of Things (IoT) was becoming a hot research topic, because everyone knew that number of connected devices to the internet was going to multiply many times in the future.

2.1.2 Internet of Things - Everything Connected

In a short period of time, IoT became a buzzword and famous in academia, as well as slowly, but steadily something people knew what was, but not quite caught the scope of. Today we have all sorts of devices around us. Smartwatches, washing machines, sensors and cameras, not to mention cars and TV's. All of this, connected to the internet and communication with something over the internet, is a part of IoT. There are challenges with security and privacy, but this thesis will not cover these topics comprehensively.

IoT in Research

A lot of research has been done on the topic of IoT since 2015, like in [2], where lightweight virtualization and functionality is central in requirements to build and create edge and IoT solutions, and in [3], where security, cloud-centric architecture, pitfalls and durability are discussed. Even a new field has even developed from it, IIoT (Industrial IoT) [4]. IoT is just a part of edge computing's background, but the research points towards a society where everything is connected to the internet, Internet of Everything (IoE). A consequence of that is that there will be a lot of different software and hardware connected to each other. All of these pieces must, in many cases, be able to communicate seamlessly. Heterogeneity which is is discussed in [2], will be one of the challenges that rises from this consequence. Washing machines and all sorts of sensors have already been mentioned as different devices. But there will be many, many more, for example in vehicles [5] [2] and surveillance equipment. More or less all of them must be able to communicate in some way.

2.1.3 The Cloud Is No Longer Enough, We Demand High Quality Services All the Time

Since 2010 the cloud has made it possible for people to chat instantly, upload and download files to store safely online, share in a whole new way and do their job with innovative tools. The cloud has provided us with great computing power, but services and content are demanding more bandwidth and power. In addition we have computers around us everywhere. How are we to continuously bring better services to more devices? Because, that is what is happening. Streaming services, cars, all sorts of sensors, and much more, generate enormous amounts of different data. The data are stored and analyzed for future use, or watched, or played. The cloud is still very powerful, but the urge for more power at work, at home, or simply everywhere, is starting to put real pressure on the response time (latency) the cloud is able to provide. We need some cloud power where all the users are.

2.1.4 IoT, an Ancestor of Edge Computing

As the number of connected devices to the internet has expanded rapidly, users, services, and content has become more demanding of quality, resources, and response. The cloud on its own can no longer give us what we require. The videos we stream on Netflix have such high quality and real-time ads and other content from Instagram and Facebook happens all the time. All sorts of applications, with gaming requiring latency as close to 0 milliseconds as possible. Exceeding 50 milliseconds can ”destroy” the user experience. All these demands by services and devices are pushing the cloud towards the users, on the network’s edge.

IoT along with demanding users and services have pulled the cloud from data centers to the network edge. It is not in our homes yet, but at our ISPs and other CDNs.

2.1.5 The Relationship Between Academia and the Industry Within Edge Computing

We do not need to go back more than a decade to see the emergence of cloud providers. Virtualization was a hot topic then, but the cloud could provide computing power on a higher level. From around the year 2010 until today the industry has tried to keep up with more and more demanding users, while academia has tried to keep up with the industry, as academic research often is funded by what is considered by politicians to be important for the industry in the future.

Normally, only large companies with several hundred or even thousands of employees do their own research, as research is an expensive affair. Universities and other research institutions are granted research funds

if their fields seem to be important, relevant, not to mention applicable to the industry later on.

In academia there has been done extensive research in wireless sensor networks, and we are starting to see an adoption into the industry, but only on a small scale. This problem domain tends to be very theoretical, which increases the complexity and time consumption. But, sensor networks are at the network edge, very close to the end-user, where IoT has made a big name of itself.

Edge computing has been mainly in research and academia as a lot of other domain fields like IoT, nanotechnology, and 5G, but as the cloud continues to grow, services and users are becoming more and more demanding, computing power on the network edge has taken a small step from simulations and research in academia and research communities, to something the cloud and service providers tend to be needing more and more in the future, as the society requires more and more real-time computing.

It is safe to say that IoT has been a buzzword for the past few years. But we also experience the urge from the industry to comply with what seems to become a trend. Today, washing machines can be connected to your home network, and smartwatches are a consumer product, just like smartphones. This was not the case 10 years ago, but both the industry and academia predicted a rapid development back then.

In other words, there seems to be a pattern that the industry picks up on technology when they see they can make a profit on it in the long run. Academia does the main research, also in collaboration with the industry, but the time from a subject or problem domain enters academia to becoming interesting for the industry, takes a while.

Artificial intelligence (AI) and machine learning is an example. Facebook has since the beginning used AI and algorithms to create better services from data collected. Netflix uses it to provide user-specific content suggestions. Today we call it data science, or big data, and it is a vital part of many businesses. Within the field of IoT, which is becoming more and more consumer-friendly, it is very important for data analysis. It is the base for general improvement. [3] talks about preservation of the data, as well as the challenge regarding heterogeneous systems in IoT along with security. The industry has been forced into AI, security, and IoT (to a certain level), to meet consumer demands, much more than just a decade ago.

2.1.6 Edge Computing From a Conference Point of View

Before 2017, edge computing was rarely mentioned at any conferences around the world. The Usenix Large Installation System Administration (LISA) conference did not have any talks on it at all [6–9]. It had not been mentioned, as a paper or a topic of its own.

Edge computing is often identified as a subtopic of cloud computing,

and has been mentioned at the HotCloud, USENIX Workshop on Hot Topics in Cloud Computing in from 2017-2019 [10][11][12]. In 2017 HotEdge, USENIX Workshop on Hot Topics in Edge Computing was finally introduced, but there are only workshop schedules available from 2018 and 2019 [13][14].

2.1.7 EU Funding

In Europe we have something called the European Commission. One of their tasks is to hand out funding to different parts of society. It enables research for several years, to dig deep into exciting topics. In our case, funding for research and work that strives to develop and research less common, new and up and coming new technologies. Next we present patterns in funding from 2014 up until today [15] [16].

The list below represents some patterns from 2014 to 2020:

- Smart system of different kinds for the future internet.
- Big data, machine learning, and artificial intelligence (AI), which we today know as data science.
- IoT, which is one part of the future internet.
- Cloud an related subjects, like big data and IoT.
- Health, food, and nanotechnology.
- Climate and energy.

An interesting observation is that smart future internet in general (Cloud, IoT, Big Data) and clean energy along with health and climate has got a lot of attention. Below all these, we find edge computing, because the cloud won't be able to provide the capacity in the future. Latency and bandwidth will also become a problem. Fortunately, we at least see that edge computing has been emerging into a central field in the research community in the past years.

2.1.8 Edge Computing in Different Settings

Latency is becoming more important as services and users are demanding more. The cloud can only provide that much. For now it is enough, but the internet is evolving. Mobile devices in all formats are going to enter the world in the next decades. Many have already done it, and smartphones were just the start.

One example is Tesla, which has used technology for several years now, to build cars. Vehicular edge computing [2] has just begun.

4G has been a part of the industry for several years, and 5G is about to surface and be available to consumers. Mobile networks, were data analysis and distributed processing is done at the network edge are emerging [17].

Video in general deserves attention as well. Streaming and interactive media in general, real-time interaction and gaming [18]. Latency, preferably as close to zero milliseconds as possible. It all exists today, but will most likely be cutting edges in the early future.

2.2 From Nothing to PoPs to CDN's

To better understand how cloud and service providers can distribute their services and content like they do, we have to take a short dive back several decades, because this is how long caching has been used. Caching lets you store data to have more convenient and faster access to it. When you cache something, you save it for later use or to have more convenient and faster access to it [19]. This is how caching in the form of PoPs and CDN's work as well, but on a much larger scale than before.

Before PoPs and CDN we had to go all the way, even cross continents to get the data we wanted, because there were no PoPs at ISPs or somewhere else, or CDN. Figure 2.2 gives an illustration. To reach a service, requests were sent to the ISP, as it does to today as well, but from the ISP, the journey where much slower, due to old and bad infrastructure, and the data were often stored in one place only.

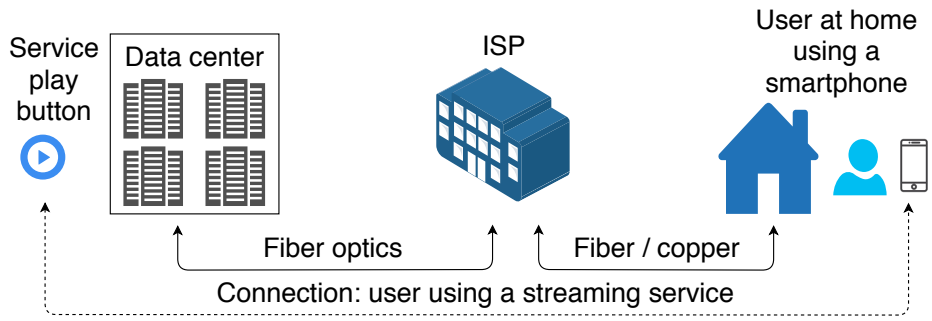

Figure 2.1: Illustration of streaming without PoPs or CDN. The video is stored at a data center, but not necessary in the same country or on the same continent. To play a video we have to get the data from the original data center, most likely on another continent

Point-of-Presence (PoP) [20] is one or more server racks placed at a single geographical location. These racks consist of computers with large amounts of Random Access Memory (RAM) and Hard drive (HDD) space and powerful Central Processing Units (CPU), often with several cores.

Along with these resources, software to configure the computer as a giant cache server is installed. PoPs are often placed at Internet Service Providers (ISPs) by Content Service Providers to improve the performance of their own service delivery. Figure 2.2 gives a simple illustration. A user connects to a streaming service. Fortunately, there is a PoP at the ISP, which results in a quick response back to the user.

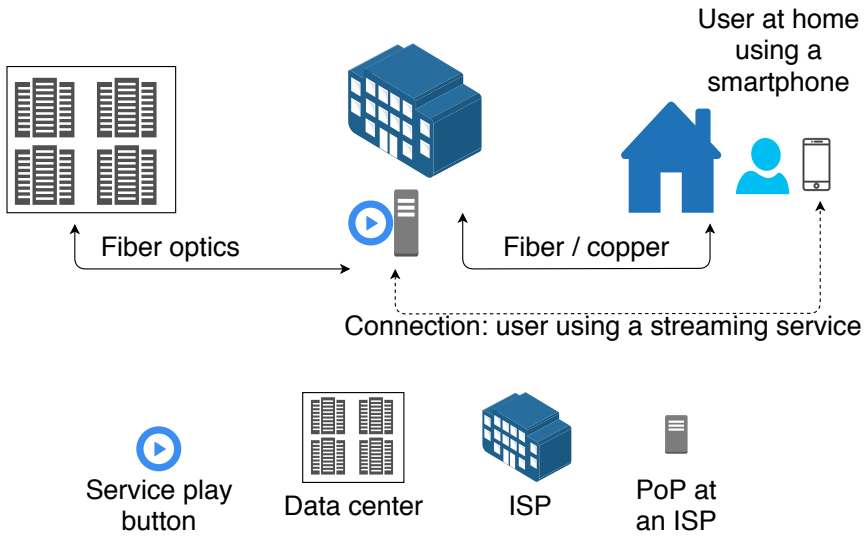

Figure 2.2: Illustration of caching with a PoP. A PoP at the ISP results in a quick response, because we have the video stored nearby, not on another continent

Content Delivery Network (CDN) [20][21] is a network of PoPs, and was actually born in the 1990s [22]. A CDN consists of several PoPs placed at different geographical locations around the world. The use of CDN has grown fast in the past years, due to requirements from users, and restriction in capacity and performance in the cloud. PoPs are no longer enough, because the constant need for low latency is so demanding. The fact is that a lot of traffic on the internet is going to and from CDNs. For example, in 2014, only 10% of traffic to and from Google and Facebook, was served by CDNs. In 2017, that number had become 80% for Facebook and Instagram. For YouTube, it already was 80% back in 2014. These findings were confirmed by the ISP staff [22]. Without CDNs, the internet would simply not have been like it is today. Many applications can deal with response time of around 100 ms, but that is not for all, especially when dealing with video and streaming of content. Also simple websites benefit from CDN. CloudFront [23] is Amazon's CDN. When using AWS you choose a region somewhere in the world which represents a data center. But AWS has PoPs around the world to reduce latency. Using CloudFront means to use these PoPs. Simple Storage Service (Amazon S3) [24] works as network storage. Here static files are stored to be used, for example with CloudFront. Figure 2.2 illustrates how CDN works. PoPs at ISPs, and the ISP connects users and data centers. All these data centers and PoPs make

up CDN.

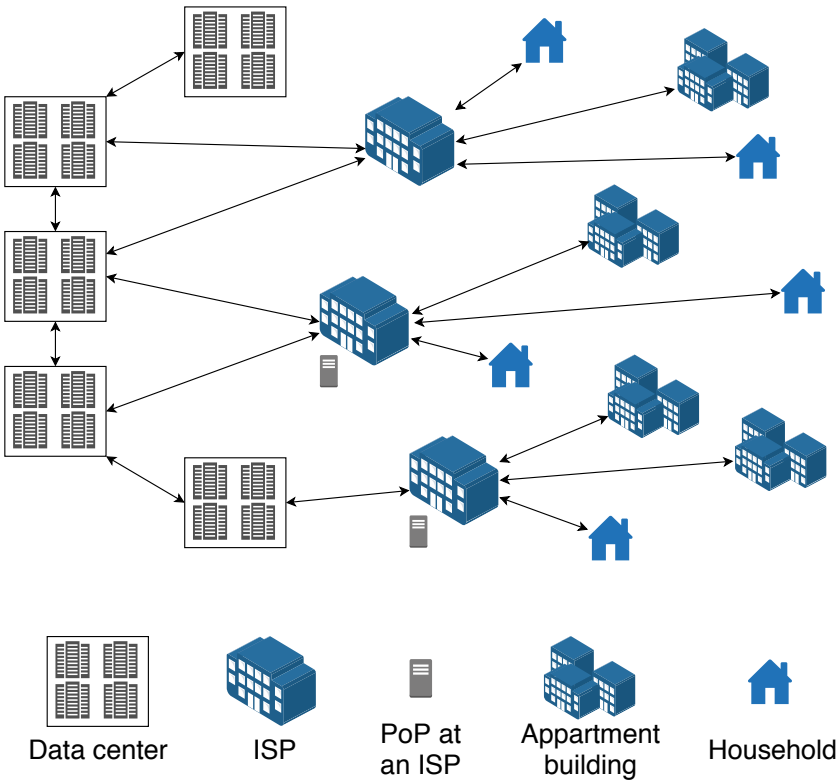

Figure 2.3: Illustration of CDN. Four data centers connected to three different ISPs, two of them with PoPs, and users are connected to the ISPs

Amazon Web Services (AWS) was founded in 2006 [25], and has become one of the largest cloud service providers in the world. It started simple, with a few services, like the S3 and Elastic Compute (EC2). Shortly after CloudFront was released [26], which is AWS's response to Content Delivery Network (CDN). In addition to data centers, AWS is able to provide low latency worldwide as their CDN contains over 200 PoPs, located around the world [27].

There are several CDNs today. Akamai [28] is a large CDN provider, who had 170 000 servers in over 100 countries which delivered 15-30% of all web traffic a few years ago [29]. CDNs are actively used by content service providers, to make sure the end-user experience has as low latency as possible. CDNs also offload major internet cables around the world. Service providers like Netflix[30], Facebook [31] and Google have their own CDNs. Many of them use Akamai [29]. They have an origin server where content is stored, but in data centers all over the world, they got PoPs that make up their CDN, which also creates redundancy. These PoPs consist of content used by users. This way, when a user requests content or data, it

is received from a PoP in the CDN, which reduces response time several times compared to what it would have been. For example a user in Oslo, Norway is playing a video game. The origin server is located in the USA, but has a CDN that reaches Europe and possibly Norway. That reduces the latency, and improves user experience.

Cloud providers also use CDNs, and rent server capacity from each other. Microsoft, Google and Amazon who are the largest cloud providers in the world, also have data centers on several continents [32][33][34]. But sometimes data centers on all continents are not enough. Certain applications, like gaming, or live TV for example are best experienced with latency below 50 milliseconds. When problems like this have to be faced, PoPs and CDN have to be used. Also streaming of popular TV shows and movies has been, and continues to occupy more and more of internet traffic. For example Game of Thrones, which has been a world wide extremely popular show for almost a decade.

It is easy to forget the ISPs as we often take the internet for granted, but without them there would be no internet as we know it today. The different ISP infrastructures connect us together. They enable us to communicate across countries and continents. This way we can connect to and use different services, such as gaming, streaming, and communication applications together.

But being an ISP today is not an easy task. Yes, they reach the end-users with physical infrastructure like no one else, but their expenses are just as high, and besides TV and internet, their abilities to provide something else to gain profit, is limited.

In the core of ISPs data centers, which usually are smaller than other data centers, like Digiplex [35], who have several data centers in Nordic countries where companies no matter the size can rent server-racks, we can find PoPs from other service and content providers which is a part of their CDNs [22]. This is a business avenue for the ISP. But with this simple operation for AWS, Netflix and other providers to get much closer to the user, while the ISP really has not gained anything specific. Service providers like Netflix and Facebook, among others, depend on cloud and CDN providers to distribute their content as flawless as possible. So, while so many build new business opportunities on top of the infrastructure managed by the ISPs, ISPs still do the heavy lifting. Therefore, it is a legitimate question whether this constitutes an opportunity missed and a potential source of income captured by other companies.

For Netflix and others to reach the end customer fast, is one thing, but the largest ISPs also own most of their infrastructure, all the way into the customer's apartment, or at least the apartment building or house. This gives big opportunities for edge computing. CDN and PoPs have taken it from the cloud to the ISPs, which can be considered their edge, because that is where their customers are connected to the internet. There are many facilitations for ISPs, where cables in most corners of the world represent

the core, but there is a lack of money, and proven ways to succeed, at the very edge. Only the largest companies like AT&T and Telenor have funds to do their own research and work in their preferred fields.

2.2.1 Streaming, Gaming and Real Time Interaction

We have been watching videos and playing video games for many, many years, but mostly on local devices, like TVs or gaming consoles. In recent years we have been waiting for cloud-based gaming. Nvidia has been testing its platform 'GeForce Now' for several years, and Google recently launched its 'Google Stadia'. This is real-time interaction with video and games, which is talked about in [18], and CDN is an important part of the user experience using these platforms.

2.3 Fog and Edge Computing, Two Words for the Same Thing?

There are mainly three different kinds of distributed computing. Cloud, edge and fog. The public cloud, like AWS or Microsoft Azure, provides access to powerful data centers. Fog and edge computing tends to be defined as the same and used interchangeably. That is not completely wrong, because both happen outside data centers and within a Local Area Network (LAN) and close to where data is generated. But there are a few small differences. Edge computing processes data close to the device, like a sensor, or a component like a phone or another specific client or device. This way the data does not even have to be transmitted across any network, just processed locally at the object, which reduces latency to a minimum [36].

Edge computing is fitting for simple calculations and maybe small analysis. In 2014, Cisco created the term fog computing as a way to bring cloud computing to the network edge [36]. Because, fog computing is also done at the network edge, but mainly at LAN level where local gateways and other physical larger devices are located. This way, extensive computing can be done quickly, without the need for the public cloud. This decreases bandwidth used between the device where data was generated and the cloud as well [37][38].

In essence, fog is the standard, and edge is the concept [36]. Fog enables the computing power of the cloud to be pushed to the network edge(LANs). For now, edge and fog are by many interpreted as the same, and in many ways it is. But if we must differentiate, fog is servers and racks, kind of like PoPs at ISPs or other places, like large LANs, and edge are devices used by users or very close to them, which Figure 2.4 illustrates.

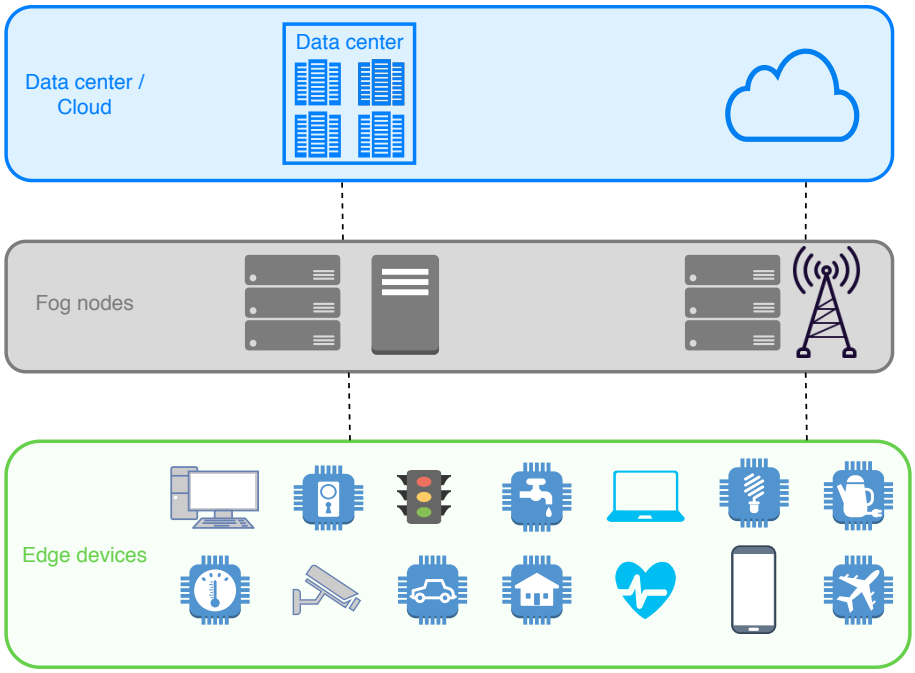

Figure 2.4: Model of cloud, fog, and edge computing. The cloud / data centers on top. Fog nodes like 4G cell towers, PoPs, and local servers in the middle, and edge devices on the bottom which can be a cellphone, a coffeemaker, a camera, or even door lock

2.4 Internet Service Providers, Very Important, but Almost Forgotten

We talk about the cloud, IoT, Netflix, Amazon, Google, and many other big players in the world of technology. But what we seem to have forgotten in the fog of all the technology, are the ISPs. We rely on ISPs to communicate with each other in the digital space. Most people want an internet connection and do not put much thought into it, or how it works, but the fact is that without ISPs, there would be no internet. In many cases we take the internet for granted.

There have been conferences within a lot of subjects like cloud and virtualization, for example the LISA15 conference [6], even economics and energy in different ways. For ISPs, it seems to be conferences within telecommunication that are the closest it gets, which makes sense, because ISPs have been part of the telecommunication community for a long time as many of them started out as telecommunication providers (TP) providing ISDN technology so people could talk with each other. But after the year 2000, they have become more and more ISPs and less TPs, as the telecommunication industry has evolved a lot, due to technological development.

On conferences for cloud, virtualization, edge, and other aspects of IT, ISPs are sometimes mentioned, but not in particular. Research done in this thesis on conferences regarding ISPs, indicates that there are no specific conferences only for ISPs, but on telecommunication in general, as mentioned. ISPs do not seem to be an area like cloud, but a business that has been, and still is, trying to keep their infrastructure up to date. They work with the rest of the industry to develop new services and products, to stay ahead of what the future will bring.

How can we tell if an ISP has succeeded? An indication is how big part of the total population the ISP has as customers. But some ISPs are local, while others are national, even international. Therefore, if the ISP has customers, a stable infrastructure, and their services work as they should, it is safe to say that the ISP has succeeded. But many people also just use their phone and 4G/LTE today. The same criteria can be applied to these providers. Telenor for example provides both.

It has always been a goal for businesses to get as close to their customers and users as possible, but in the last couple of years it has become even more important. ISPs are in a great position here, especially when it comes to the geographical aspects.

2.5 Security and Privacy

Security and privacy won't have a big role in this thesis, but both subjects deserve a little attention concerning the subjects of this thesis, especially IoT. In recent years, security and privacy have become very important. So many challenges have surfaced, due to social media and the technological development within streaming, and a large number of free services, in exchange for personal information. Many are concerned about where their information is stored, who has access to it, and how it is used.

2.6 Related Work

ISPs have not been in the spotlight of academic computer science research. However, ISPs utilize technologies that have been researched a lot. They are also partially mentioned in other work, but not like for example containers or edge computing. Therefore there are limited directly related work to ISPs and what thesis aims to provide, which is services as a product on the network edge using recent technologies. One recent technology is containers.

Now we see that recent lightweight technologies, edge-computing, bandwidth, and latency are used in other work and have similarities with our task. However, we want to help the ISP to utilize their infrastructure in a new way. They focus more on IoT in relation to edge computing, thus a different focus.

The work by Morabito Et Al. demonstrates a project where someone from the industry, in this case Ericsson and someone from the University of Delft collaborates within edge computing [2]. They explain how lightweight virtualization (LV) and functionality are central in requirements to build and create edge and IoT solutions. They have used containers and unikernels.

Applications and services requiring more and more bandwidth and latency reduction are emerging every day. [39] talks about the post-cloud era, where devices and activities that require even lower latency than the cloud can provide. Increasing demands in bandwidth because of the amount of data sent and received across millions of devices is also becoming a problem. Not necessary because the cloud can't handle it capacity wise, but because we only want to process the data as fast as possible. Pushing cloud resources to the edge can also reduce latency for devices with limited capacity.

The amounts of data consumed by users puts more pressure on the infrastructure. Trevistan Et Al. discovered that an ordinary broadband subscriber in 2017 downloaded more than twice as much as they used to do 5 years before [22]. They found that FTTH not necessarily has a big impact on the data consumption compared, but that the traffic in general from applications like Instagram is comparable to traffic from services such as Netflix and YouTube. This means that our work has clear relevance, because we want to reduce the load on the core backbone network by enabling edge computing on nodes at the edge.

2.7 Container Technology

Container and virtualization technologies have changed the way system administration, development and deployment are done in regards to software and hardware. Virtual machines and containers are central in these regards.

2.7.1 Virtualization and VMs

Virtualization is a very popular paradigm, which still has an important role in operations today since it provides the opportunity to administrate tasks and many machines at once, while improving scaling and overall hardware-resource utilization. Virtualization enables us to create several machines on the same hardware. Instead of several physical machines, we can use one machine to run several virtual machines(VMs). We give each machine some of the hardware resources (RAM, CPU, etc). These resources can be reduced and increased as desired. To do so requires a hypervisor. A hypervisor is the software that allows us to run VMs. There are two types. Type one runs directly on the hardware. Type two runs on the host operating system. VMs are in many ways the predecessor to Docker. Figure

2.5 illustrates the difference between type 1 and type 2 virtualization.

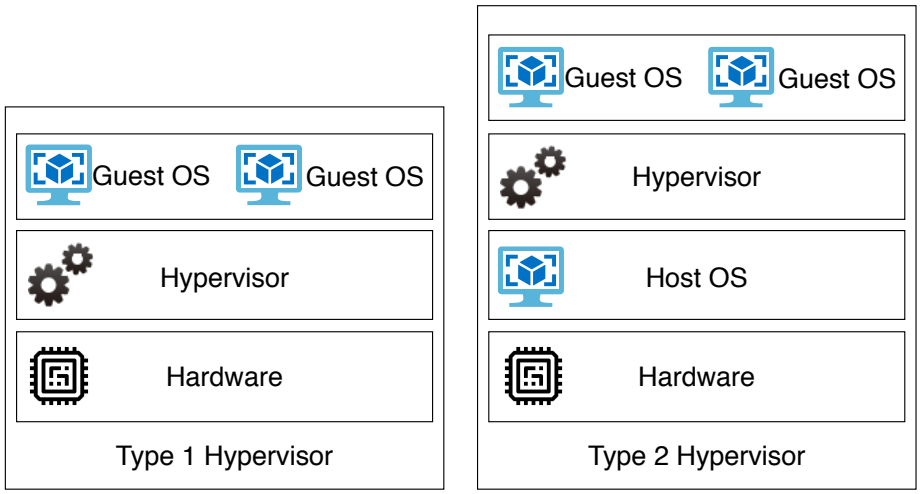

Figure 2.5: Virtualization, Type 1 and Type 2. Type 1: the hypervisor running on the hardware, and guest operating systems (OSs) running on the hypervisor. Type 2: A host OS running on the hardware. Hypervisor program (software) installed on top of the OS, and Guest OSs running on the hypervisor

2.7.2 Docker

Docker is a platform for building, running and shipping applications in containers. It is used by developers, system administrators and enterprises to isolate software from its surroundings and to easily move complex configurations. This way Docker helps to separate different software environments from each other and makes it possible to update or change software when necessary, without having to worry about the state, or versioning of the surroundings. This applies to development and production environments.

Docker enables virtualization in a simple way. It allows you to use the same software environment anywhere, not depending on the operating system in use. The environment being a container. In short terms, Docker is a computer program that performs operating-system-level virtualization, also known as ”containerization”. Containers are just as isolated from each other as VMs, but more lightweight.

Containers

Containers have become very popular because they are simple to create, use and deploy. A container can be viewed as a lightweight VM. Multiple

containers can be spawned in a short time span, often seconds, and just as easily be deleted and removed, because they are small at size. Usually between 10-50 megabytes, but if an operating system like Ubuntu is installed, a few hundred megabytes. A container only includes the requested software and packages to run the desired application. They do not need an operating system and other dependencies to function, which is why they are so lightweight and fast. To run a web server for example, all you need are Apache and necessary files to make a website work.

Docker Image

Containers are created from images. An image is a file-system image configured with the necessary software and its dependencies. For example an image with Apache installed along with some PHP libraries on top of Ubuntu to work as a web server. These images are created from a Dockerfile.

Dockerfile

A dockerfile defines the container's environment. A dockerfile is a configuration file. With the dockerfile we can specify certain actions to be performed on a file-system base. We can set permissions, create, copy and move folders and files. The dockerfile is the configuration of a container. Listing 1 shows a dockerfile that is based on Ubuntu 16.04, runs an update for the system, and installs Apache2 before everything that is located in the folder this dockerfile is, is added to the /var/www/html. Then index.html is added and "hello world!" is added to index.html. At the end port 80 is "exposed", this way we can access the content of index.html from the outside through port 80 on the container.

```
1 FROM ubuntu:16.04
2 MAINTAINER Name "email"
3 RUN apt-get update
4 RUN apt-get install -y apache2
5 ADD . /var/www/html/
6 RUN touch /var/www/html/index.html
7 RUN echo "hello world!" > /var/www/html/index.html
8 EXPOSE 80
9 CMD /usr/sbin/apache2ctl -D FOREGROUND
```

Listing 1: Dockerfile that installs Ubuntu 16.06 and creates a webserver with Apache2. General updates with "apt-get update" are installed first. /var/www/html/index.html is added, and port 80 is exposed to enable access of this container's web server on port 80.

2.7.3 Immutable Computing

Mutable systems can change state after they are created. For example, an operating system can be updated with a new patch, or rolled back to a previous state, which is the typical approach to managing the state of systems.

The opposite of this is immutable systems, or immutable computing, which has become a popular paradigm. Immutability means that a system cannot be changed after it is created. When we use immutable computing we stop operating with direct changes, but instead re-create the system with the change we want, delete "the old" system, and deploy the re-created one. We basically apply the idea of "disposable computing". This approach has become very popular because time and money can be used to create and develop software and systems rather than debugging something that does not work as we want.

Containers are a very good example of immutable computing. We program the dockerfile to install software and run our desired commands and operations. The result is an image. From this image we create a container. If something is missing out, the wrong version of a software was installed, or for some reason, something does not work as intended, the container is removed and deleted. To change the deployment to its intended state, the dockerfile or the application-files that is to be deployed into the container is changed and deployed again. The alternative is spending time debugging with rollbacks and software versions, which can take unnecessary long of time.

2.8 Microservices

Microservices is an architecture where many different services run separately. It is an alternative to traditional service architectures like the classical 3-layer application stack, database, application and presentation layer.

2.8.1 Service Discovery

Microservices depend on service discovery. In many ways, service discovery is a Domain Name System (DNS) for microservices. When we type in a URL in a browser, DNS is the system that translates the domain name to an IP that can be used to find the server where the content we want to see is located.

Service discovery is a method, used by applications to locate each other. This can be a challenge with microservices, because the environment is dynamic and in constant change [40]. DNS can be used, but is more fitting for external use, outside the application environment, while service discovery is for internal application cluster use [41].

Registry

A key component of service discovery is a service registry. This is a ”table”, that contains currently-available instances of each service and how to connect to them. The registry is maintained by a heartbeat mechanism to see if services are still running or not, have changed IP, or anything else, that might be essential for a connection. Anything that is not used, is removed to keep the registry up to date.

There are two ways to register with a service registry. A service instance registers itself with the service registry to make itself available, and unregisters on shutdown. This is called self-registration. The other way, called third-party registration, is when another third party system takes care of the registration. Consul [42] is a system like this. It is a service networking tool that allows you to discover services and secure network traffic.

Client Side vs Server Side Service Discovery

There are two main types of service discovery. Client-side and server-side. If we use client-side service discovery, the client contacts a service registry to obtain information about the relevant service. The client then contacts an available instance using a load balancing algorithm. A drawback using this method is that logic must be implemented for all programming languages, if more than one is used. Server-side service discovery on the other hand uses a server load balancer, not an algorithm, as opposed to client-side. The client contacts a load balancer, which consults the service registry to select an available instance to use, and routes the request to this instance [43][40]. An example of this is AWS Elastic Load Balancer (ELB) [44]. ELB can be configured for applications, networks, and across other EC2 instances to operate on more than one level. The ELB receives a request from the internet, finds an available instance, and routes the traffic there. The user only wants the content and do not care about the source.

2.8.2 Kubernetes

When we want containers to work together, at the same time, orchestrate a cluster of VMs, across multiple data centers where containers are created and removed, Kubernetes can help. Kubernetes is an open-source container-orchestration framework for deploying, scaling and managing clusters of containers and VMs, where we run our applications.

It has built-in service discovery, and is a platform well suited for large and complex systems. Docker has Docker Swarm, which is Docker’s way to cluster containers and administer a large number of containers. But, compared to Kubernetes, Docker Swarm is not as extensive [45]. They are two different technologies, but they work well together, and can very well be combined [46].

2.9 Adapting Paradigms

Now, we have a robust infrastructure with PoPs and CDNs. Technological development which tends to push limits all the time, and containers with Kubernetes and Docker have taken dynamics a step further.

Can these mindsets and architectural paradigms, from a birds-eye view, be adapted into the ISP infrastructure? From an architectural point of view, ISPs have large backbones with servers, routers, and services. Netflix for example has been using microservices for a long time. They also have a large backbone in data centers around the world running Netflix. Here, there are many similarities, like distribution.

2.10 Background Summary

In this chapter we have learned that edge computing in many ways comes from IoT and that the internet today would not be the same without PoPs and CDN. Content providers need it to distribute their services, but what we tend to forget in the end are the ISPs, who we need to even be connected.

Edge computing, or fog computing as Cisco calls it, has the past few years surfaced from just being in academia, to the industry, as IoT has grown. Sensors, smartwatches, cars and washing machines are just some parts of it, and we will see much more of this paradigm in the years to come.

The internet has grown to become a big part of our everyday life. Cloud and service providers like Facebook, Netflix, Google and Amazon have become big players we often take for granted. Due to enormous infrastructure around the world that consists of large fiber cables, data centers and PoPs that make CDNs, Facebook and others reach all consumers and work pretty well.

But in the fog of all technologies, cloud, content, and service providers, it is easy to forget the ISPs that connect us. There have been conferences where they have been present and they really are a huge part of the internet. The fact is that everyone relies on ISPs for internet connections, but they have mainly been left to themselves and forced to evolve as everyone else does.

Chapter 3

Approach

This chapter will describe and further outline activities that aim to provide an answer, or answers, to the problem statement: *How can recent technologies like containers contribute to facilitate edge-computing towards ISPs delivering Proximity-as-a-Service (PraaS)?*

We know that the cloud is very powerful, but we also know about the challenges regarding latency and bandwidth. In the introduction we said that the world and our society is becoming more and more digital. Users and different services are also becoming more and more demanding. Therefore, it would be useful to push more capacity to the network edge. This way we can make the user experience better in general. Our background told us that there has been done research on edge computing in regard to IoT and other topics. But the ISPs role has rarely been mentioned or accounted for in a considerable extent.

This thesis wants to make the ISP more central than it has been. If we can find a way that enables the ISP to have a bigger role, we can help academia and the industry with a piece of work that can make a slight difference, but still put more light on edge computing with ISP in a more central and deserved role.

There are mainly three different research methods; Exploratory, survey, and comparative.

Exploratory

Exploratory research explores problems that have not been clearly defined yet. In exploratory research, the outcome is not so clear, which opens for surprises as well as inspiration, but it is difficult to estimate time, hence unclear outcome [47].

Survey

Survey research uses previous research done within a field, statistical analysis, or literature to help move research within a field forward. Surveys have a direction, but there is not enough literature to carry out research as desired [47].

Comparative

Comparative research uses already well-defined statistics and data to draw conclusions. This type of research is used to help identify similarities and differences. [47].

This thesis is exploratory because we have an idea about the outcome, but can face surprises as well, and therefore be inspired along the way.

3.1 Objectives

The objective of this thesis is to investigate the possibilities of pushing cloud capacity to the network edge, with the ISP in a more central role. We want to see if recent technologies like containers can help us make it easier to get one step closer to enable capacity at the network edge, not just routing and other network functionalities. There will most likely be discovered new technologies in the future, but today we know containers as a great option, which have been used in a lot of work with significant success.

We still need the cloud as the "supercomputer" it is for heavy analysis, computing and storage. We know that PoPs and CDNs have decreased latency. Based on that, it is safe to assume that we will get some expected and desired results in our experiments, pointing out that containers can be used to reduce it even more at the edge, without high space requirements. But we want, and need, to find out the pros and cons of this with different architectures.

Many different technologies are used in different ways today, to create models and implement designs. Like virtualization, containers, trees and nodes in algorithms. JSON is a widely used method to structure data in general, for data analysis and AI, and data analysis and IoT are emerging as we speak. Can we adapt relevant technologies to our advantage? We have containers, which are programmable to fit different cases and scenarios, as well as lightweight, and easily managed, like we mentioned in the introduction of this thesis. Virtualization is a heavily used technology everywhere. If we can use VMs to simulate different machines, we have a base that can be expanded if needed.

The need for knowledge has been talked about and discussed briefly in the background. Knowledge is gained from research, failing, and learning

from our mistakes, experimenting, and testing. We know that an ISP infrastructure is large and has complex architectures on several levels, like operational and technological. We can also say something about how the results will be in our experiments, but that would only be touching the tip of the iceberg, and not be able to tell for sure, because lack of knowledge. What we want to do in this thesis, which is to use containers, distribute them, somehow, to improve the end-user experience. More knowledge on this problem domain would help.

3.2 Design

Further, in the design chapter, we will look at the ISP infrastructure as we know it today, before we look at new ways to use it. Can we push caching and computing power closer to the user? We will look at different ways of distribution, and reach at least one, preferably two or three models that we can dig deeper into. When we look at distribution, examples, diagrams, and drawings will be given to better explain. Slight talks about algorithms may also be mentioned. We want to end up with models that are dynamic, but also takes (hardware) resources into perspective. The pros and cons will be evaluated. In the end we want a model where we can use containers in the infrastructure, and gain better user experience with an eye to minimize resource consumption.

If we use and adapt technologies and models to an ISP infrastructure, it would help to create innovative designs that can fit real life distribution models to implement. Storing of data is required almost all implementations, no matter what. Non-sensitive data can safely and easily be stored in a JSON file, and virtualization can help to simulate an ISP infrastructure. If on top of this we use containers that can be distributed according to a number of models, we need to program scripts that make decisions based on our distribution models.

3.2.1 Use Cases

Use cases describe a series of events, steps, and actions on how the system reacts when someone interacts with a system. In order to associate designs to the real world, not just theory, use cases mimicked to realistic scenarios will be presented and then used to see how our designs apply themselves.

Use cases can help to see strengths and limitations, or weaknesses with a model.

A weakness in a model can be that we beforehand think that it is a good fit in a scenario. But when we use it in a use case, we find that it is a lot more complicated than we expected, or it is simple, but takes a lot of extra time and effort to get it done. Is it worth it then?

A strength is that a use case can show that the model can be used to

simplify something complex. We cannot expect everything to run smoothly all the time, but it works well enough or very good. Sometimes we come across edge cases where we need to take shortcuts we rather won't take, but to make it work, just good enough, we take these shortcuts. Use cases are a great tool to find these edges and limitations.

Use cases are not unique for this kind of thesis. Use cases are very common in general. They have been used in a lot of work, also work that can be compared to this thesis in regards to implementation and deployment of or in infrastructure. Like in [48][49], where use cases have been used to show and simplify the complexity of deploying, managing and scaling of VMs.

A weakness with use cases itself, is that we might use them to distinguish what is easy and what is not. These results are then used to pick only what is good, and not challenge it. The end result is that it now looks like everything went according to plan. The reality is that it did not, we were just very selective. We should challenge models, even the good ones. We should also work with what may seem hard, but can have great end result if time is spent on it.

More than one use case is preferred and should be present. Still, different use cases, that can challenge the same thing from different perspectives, is important. This combined with an open mind should cover and show that extensive testing has been done.

In our use cases we will imagine a large ISP like Telenor or Get, in Oslo, Norway. The use cases will contain the ISP, the ISP customers, or users, and possible content or service providers like Netflix and Epic Games (creator of the worldwide popular game, Fortnite). These services are to be distributed into the infrastructure, based on models.

The evaluation will be from an operations perspective of complexity. An expert evaluation from a traditional computing and enterprise operations perspective.

3.2.2 Tools, Components and Technologies

ISPs already use several technologies today. For our design in this thesis, we will use the technologies listed below. We use these technologies because it is convenient for us. Other languages and kernels that can do the same job are applicable as well.

- Python. Python is a widely used scripting and programming language, that is highly applicable in general.

- Linux and bash. Linux (UNIX) is the core system kernel used by many operating systems, where Ubuntu is one of them. Bash is the language of Linux.

3.3 Implementation

The implementation chapter will create and implement a prototype based on models and design carried out in the design chapter. In the implementation we will create tools and a way to represent data, so it is simple and applicable. The purpose of the implementation is to verify that it is possible to apply our designs and models to an ISP infrastructure, using programming and known technologies.

The implementation will be based around Python-programming and simple data storage. A main element of the prototype will be simplicity as it enables us to better evaluate later how well this could be transferred to other technologies. Python is chosen because it is a straight forward language, that is easy to read and understand. For data storage we will use basic formats, such as JSON, in order to better read and understand the data as it changes. The prototype should be on a functional level such that we can use it to do data-manipulation that represents management operations in an envisioned ISP infrastructure. It should enable us to run several simulations and review the end result. It should do such in a manner that we can rerun experiments in order to see the effect of any subsequent changes.

The focus will be on the data model which mirrors a picture of the ISP infrastructure, and how it can be manipulated using our tools. It is important that a model provides clear results as to the proximity to users. We will also apply the prototype to our use cases.

One could ask is there not a tool we can integrate our tools with, like why is not Kubernetes used. That question can be addressed to almost everything. The plan is not to integrate a tool to something that already exists. However, the implementation will inform us on the feasibility which will be discussed in the analysis.

3.4 Analysis

The analysis will look at the design and the implementation from an expert's point of view. We are interested in how our model and prototype has reduced complexity. Is it feasible in a technological way regarding programming languages and data structures? What do the results from use cases tell us in regard to the pricing of the distribution models? Are they applicable to a broad audience, and could we have done more to bullet-proof the prototype?

Chapter 4

Design

In this chapter we look at the already established ISP infrastructures, and what we can do with it. We look at possible designs to be used when creating a prototype in the implementation. The implementation will be done in the implementation chapter. This chapter presents models and different architectures, describes them, and puts them up against each other, to reveal pros and cons.

4.1 ISP Architecture Overview, Today

ISPs provide users with an internet connection, whether it is as a SIM card in their phones, or via a router in their home, which usually has wireless functionality. Behind this router, or sim card, there are a lot of infrastructure.

An ISP infrastructure in general is very big. If we look at this type of infrastructure from a bird's eye view, it looks like some kind of a tree structure, illustrated in Figure 4.1. At the root of this infrastructure we find a data center that often is in the ISP building itself. This data center connects the ISP customers to the rest of the world. Between this data center (root), and the user at the network edge, there are a lot of cables and devices, like fiber-optic (cables), servers and routers. All of these servers and routers, make up hubs and nodes. Hubs have quite a lot of capacity and have many nodes connected to them. They usually cover a geographical area of one or a few postal codes. A node is the physical hop that connects an apartment building to the hub. Between the node and the users router, we often find a switch of some sort. The tree structure in 4.1 is simplified with just the ISP and nodes. Hubs are left out, to simplify, but the principle remains the same, a tree structure. This will be our base model when we work with this infrastructure.

Users use their PC, mobile or some other client, and hit buttons to interact with applications. For them, the internet is just there. Some of them know they get content and services from somewhere and have an

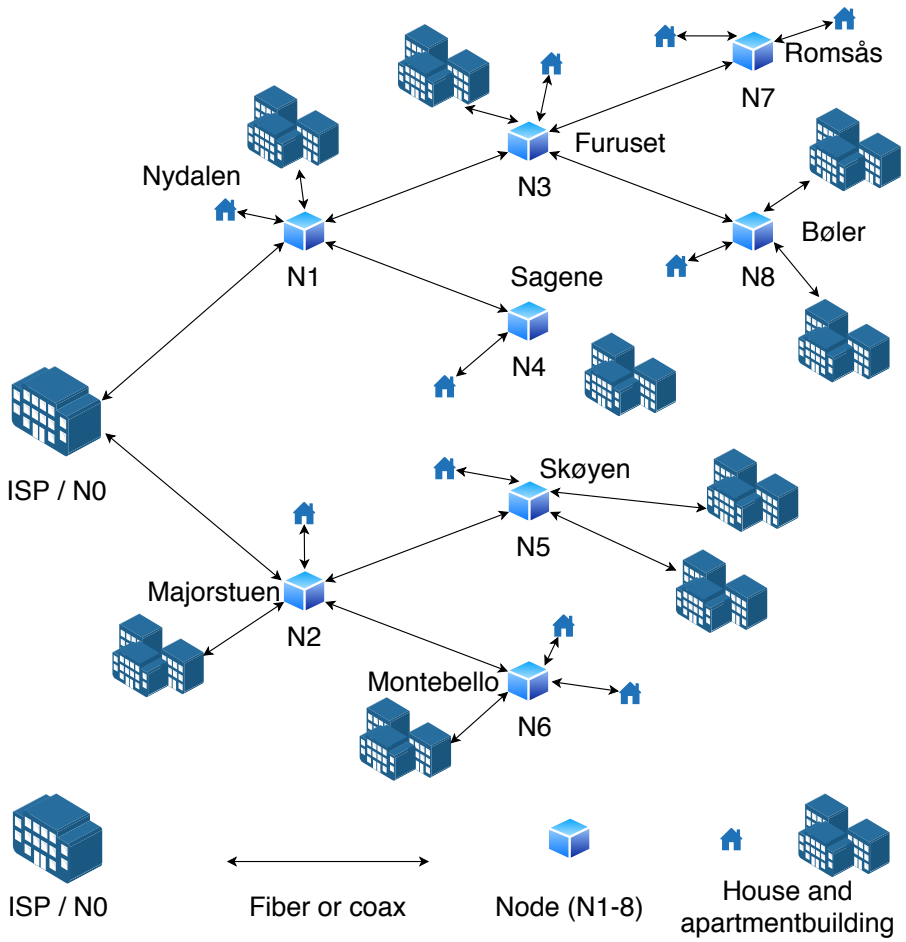

Figure 4.1: ISP infrastructure tree figure. ISP/Node0(N0) as root. N1-8 make up the rest of the tree. They are connected as children an parents. N0 children: N1 and N2, N1 children: N3 and N4, N2 children: N5 and N6, N3 children: N7 and N8. Some households are connected to N1-8 to illustrate.

internet connection through a company, in a sense like 2.2. But all they really need to know is the password to their wireless network, or a network cable, nothing more.

From the moment a user hits a button until the response appears on the screen, a lot has happened. Information and data are sent through an enormous infrastructure. This infrastructure consists of servers, routers, and other types of hardware configured by software to route traffic, or data packets in the right direction. The direction is towards the server where the service itself is, or the client requesting information. If we look at this infrastructure from a bird's eye view, it has some kind of a tree structure from an ISP point of view. Each router in this infrastructure becomes a hop which again increases latency. Most often, not by much, but some. Managing and maintaining this infrastructure is not an easy

task. It costs a lot of money and is complicated. In both the introduction and the background chapter we talked about the ISPs being the ”big loser”, because content and service providers like Netflix, Facebook and Amazon use their infrastructure without putting too much effort into it. This also falls into the category of managing and maintaining an ISP infrastructure. After all, we need to remember, the ISPs rely on a good user experience.

Now we know what it is like to be an ISP, the general challenges they face in their everyday life. This thesis maintains the focus on their possibilities, because it is not completely dark. Soon, or by around 2023 and the years after towards 2030 we will have 5G with all its perfections and imperfections, not to mention the already existing infrastructure.

An important thing to mention when we talk about infrastructure, is the difference between the real one between an ISP and the user, and what we see if we use tracert/traceroute. When we use tracert/traceroute, we just see the big hops, so to speak. We do not see all hubs and nodes the real infrastructure consists of, but instead big routers of IX (internet exchange) points and ISPs. Like in Figure 4.2, ”Tracert hops”. ”Real life infrastructure” of 4.2 illustrates that the real-life infrastructure has several more hops and is very complex.

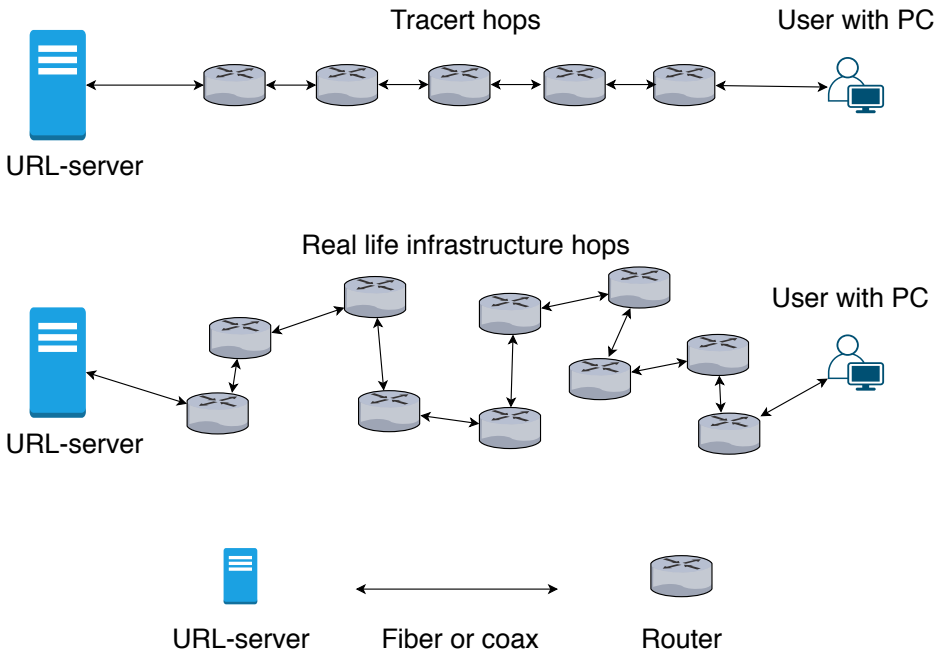

Figure 4.2: Tracert hops vs a more real-life infrastructure scenario. Real-life infrastructure has got several more hops (nodes) than what tracert/traceroute shows.

4.2 New Supplies to the Existing Infrastructure

The ISP infrastructure is big and can be used to accomplish a lot. Further in this chapter we want to find a new way to use this infrastructure. A new way to think. Our problem statement refers to containers. In this thesis we want to look at ways to use containers in an ISP infrastructure.

4.3 Containerization and Service Migration

In the approach chapter we talked about the everyday life of ISPs in the past, up until today. In this chapter we look at possibilities and discuss designs, look at models and more. We know that ISPs have a very distributed infrastructure, with hubs and nodes. The root is at the ISP data center, and the tree structure appears from that. It becomes a cluster of hubs and nodes. At this root, in the ISP data center, we want to run services in containers, and distribute them in containers throughout the infrastructure. But we need to keep in mind that capacity shrinks the further away you get from the root, and from hubs to nodes.

We have talked about hubs and nodes in the ISP infrastructure. Hubs have more capacity than nodes. That is the main difference between a hub and a node. A hub links multiple nodes together. Hubs and nodes are connected by fiber cables. In total, it constitutes the ISP infrastructure.

A node is a large computer, geographically placed close to one or more apartment buildings or houses. It covers maybe a few hundred customers. Hubs can handle several thousands. A node is configured to route traffic (data packets) between the users and the hub it is connected to. It is physically at the size of a small server rack approximately, which means it can host a computer of quite decent size. For a user, that would mean quite a lot of capacity, but that changes when we move the venue to a node. It becomes a computer that we need to be careful with in the matter of overloading, because not only one user will use it, but possibly several services and users. The capacity is increased, but must be used smart way.

For the end-user, or the user experience, a computer running services on a node just outside, or few meters down the street, means a very short distance to the desired service. The service will be running at the root (the ISP) as well, but that will be several hops and possibly several kilometers away, which increases latency, and we do not want that. Having a service only running in one place also increases the load heavily. It is wise to spread out, if possible, to reduce latency and general server load.

A service, or multiple services, because there are many different services requested by users all the time. These services are to run in containers. A service may run at many containers at the same time, but never more than one service running on a container at a time. We build and run a container, let us say with Netflix, on the root node, the ISP data

center. From here the same container is spawned on another node closer to end-users who want this content. How it spreads out depends on models and user demands.

Figure 4.3 illustrates how a container is placed at a node in the infrastructure. There are fewer nodes between the user and the node where the container is running, than the user and the URL-server itself. This is a fitting illustration of how we imagine services, as containers will be placed out. Closer to the user, shorter distance and time between the service and the user, and relieve the load on servers. On this figure there are router icons to demonstrate hops. Nodes are used on most of the other figures.

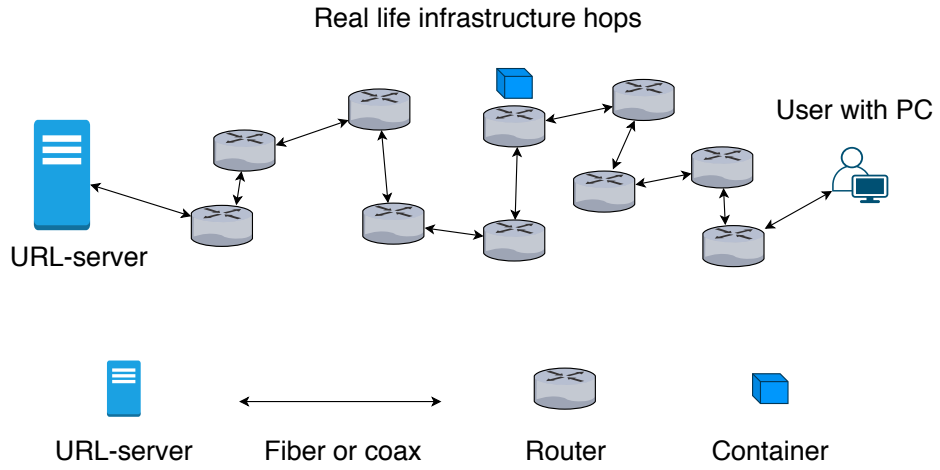

Figure 4.3: More realistic infrastructure with a container running at the middle node closer to the user.

Figure 4.4 illustrates roughly how a node is meant to work and look like. The physical node itself already have network hardware and functionality. We want to add hardware and software for computing and caching functionality, to be able to host containers.

4.4 Distribution Strategies

In the matter of distribution we look at what distribution capabilities we want. Do we want to be able to move a container, copy it, or maybe both, depending on the situation, resource requirement, existing load or something else. We are looking at scaling, because we want to run one service in more than one place. But, we need to keep in mind that hub and node capacity decreases as we move further away from our root.

Moving containers from the root is one way to do it. This way, we can free up space, for new services. Copy is another way. This way we can

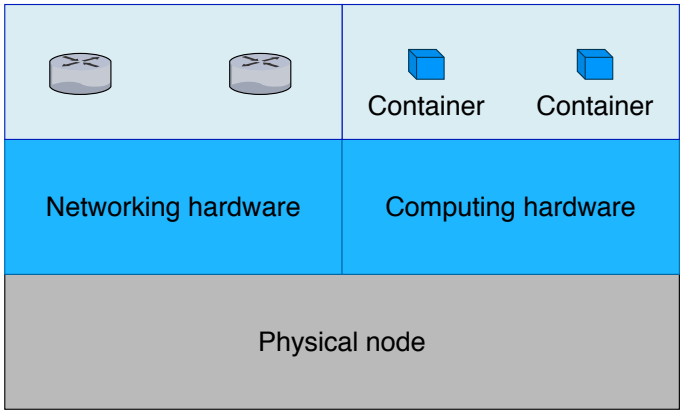

Figure 4.4: Physical node with networking hardware and computing hardware. The network hardware is running network and routing functionality, and the computing hardware is running containers.

duplicate the service to meet higher traffic loads, which creates a dynamic flow where a service runs at multiple nodes at the same time.

Both moving and copying means that we take up resources at a new node. If this node already is peaking at its resources, more load can to more damage than good. Therefore, there is a need to manage this in a good way.

When we talk about containers like we do here, it is important to point out that to copy or to move a container is not the same as doing it with a VM. A container is stateless. To alter its state, changes have to be done to the docker configuration file itself. A container instance has never been meant to be copied or moved. In both cases, we create a new instance from the existing dockerfile. If we want to copy it, the running container is preserved. If we want to move it, the running container must be stopped and deleted, and a new instance must be spawned.

In the end, there are users at multiple locations, all the time. These users rarely use the same service at the same time, all the time, which means that the ability to scale, is important. When a popular TV-show is launched, traffic related to this show takes up capacity. The same when a new video game is launched. The difference is that a video game has higher requirements regarding latency than a TV-show. In the end we want multiple services running at multiple locations at the same time, and meet high capacity demands.

When we want to copy or move, the ability to make decisions, is important. We need to make decisions on what to run where, and how. There are many different algorithms for this. Algorithms where the most recently used object is kept, or the object using the least resources is deleted,

to mention a couple.

To run services smoothly requires optimization, especially in large environments, with many unknown factors. Algorithms can be used to optimize services and applications the way we want. This way we can utilize resources both locally and in the network for the best. The result is an optimized user experience.

If we look back to our problem statement, *How can recent technologies like containers contribute to facilitate edge-computing towards ISPs delivering Proximity-as-a-Service (PraaS)?*, we want to achieve a way to make a sellable product.

4.5 Distribution Models

Now we have talked about how containers are meant to be managed and worked with. But how do we want to distribute services, as containers? We are going to talk about three models, the balloon model, the fog model, and the micromanagement model. The balloon model and the fog model have certain differences that separate them from each other. The micromanagement model on the other hand, is somehow more free. It has not got that many preset rules or ways to work. It aims to be as flexible and dynamic as possible.

It is important to be precise when we talk about these models in general. Because, we have three models, and we assume that these three models can be used at the same time. That means, one model can be used in one use case by a service, and another one can be used in a different use case, by the same service. How smart is it, is a dilemma that arises if multiple models are used at the same time. It complicates the job for the ISP, but it also enables several ways to use the infrastructure to distribute content simultaneously. Capacity on nodes are limited, but with today's technology, and in the future, it is not wrong to assume that capacity is a problem that must be managed, one way or the other.

4.5.1 The Balloon Model

The balloon model starts at the root and expands. The first step will be hop number 1, then number 2 and so on, as it expands. When the balloon expands, containers deployed at previous hops, are not removed. This is good and bad. A disadvantage is that it takes up resources that could have been used by other services in the meantime. An advantage is that the nodes now are prepared for traffic that may come later on, and traffic patterns do change. A service that is not needed at a time, may be popular in thirty minutes. This way the balloon model prepares the infrastructure for what may come in the near future. The balloon model is static in the matter of hops, but is not restricted to only one balloon. One balloon can

only contain one service, and there are many services. If one balloon has the newest episode of a poplar TV-series, and another has games like Pokemon GO or Youtube clips, there will be multiple balloons.

Figure 4.5 illustrates the balloon model. Each red line represents a balloon at each hop in the infrastructure. Normally, there is a container at a node which is covered by a balloon, which on this figure are all nodes, including the root ISP(N0). However, not in this figure, because it is just a demonstration. To demonstrate how the balloon can expand, thereby creating a ”before” and ”after” state, Figure 4.6 and Figure 4.7 illustrate a scenario of ”before” and ”after”. Figure 4.6 is the balloon model ”before”, with a container running at the N0 (depth 0), and at N1 and N2 at depth 1. Figure 4.7 is ”after”. The balloon has expanded, and the container is now running at N3-6, at depth 2, in addition to N0 and N1-2 at depth 1. Now the balloon covers all depths from 0 to 2.

4.5.2 The Fog Model

The fog model also starts at the root and cannot suddenly appear at a random node, but expands differently, not static, hop by hop, and is rather more dynamic. If there is a need for more capacity at a certain node, the fog will be increased to that specific node, but on the way, also take up resources on other nodes on the same tree path. These nodes may not need that service at that time, but that can change fast, so it is not a total waste. The fog can also expand to a node beside the original taken path. But, it will always take up resources where it is, no matter the demands at different geographical locations. So, it is more dynamic, but can also create other challenges that must be dealt with, or accepted.

Figure 4.8 illustrates the fog model. In this model some parts of the infrastructure are covered with a red fog. In this particular example the fog covers N0, N1, N2 and N4. The next step could be that the fog is increased out to cover N5 as well.

4.5.3 The Micromanagement Model

The micromanagement model is without a doubt the most flexible, in theory. It is very dynamic, but also extremely complex. The micromanagement model allows us to do adjustments to accommodate small and large changes in traffic and demands at one or several locations at the same, and at different times. No balloon or fog, but deciding exactly where to run what, when and where. In an ideal world we would like to manage our services and devices in detail like that. Very complex algorithms and Artificial Intelligence (AI) custom-made for this purpose alone would probably help, but as said, it is extremely complex. It is relevant, but unfortunately out of the scope of this thesis, and therefore only addressed purely in theory. Figure 4.9 illustrates the micromanagement model. Same base tree infrastructure as with both tree

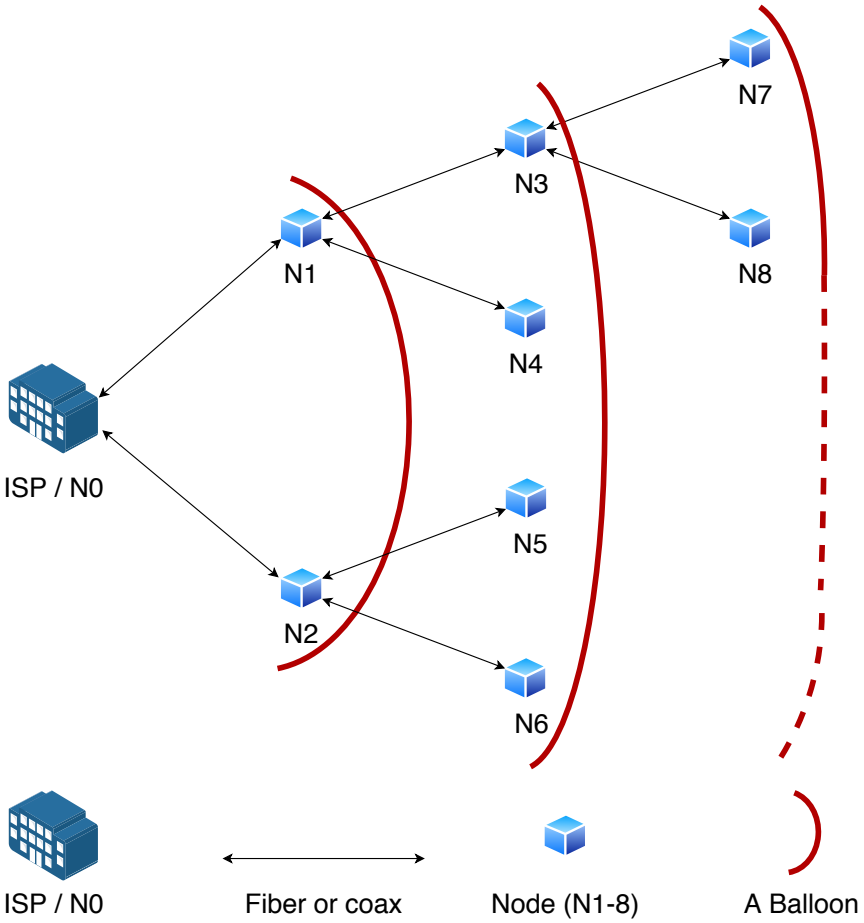

Figure 4.5: The balloon model. Here the balloon is on depth 1-3, but there are no containers running on any nodes. This is just a simple demonstration of the balloon model. The red lines signify which depths the balloon has expanded to.

and fog, but there is no pattern of where a container can spawn. Figure 4.9 has containers at N0, N1, N4, N5 and N8 in this scenario.

4.6 Profiting From Models

We now have three different models that different algorithms can be applied to. We will connect these to use cases after they are presented in the next section, use cases.

We want a way to utilize the ISP infrastructure for them in a new way to make money and preferably gain a profit. For this to be possible price models are valid to introduce and discuss. Cloud services are based

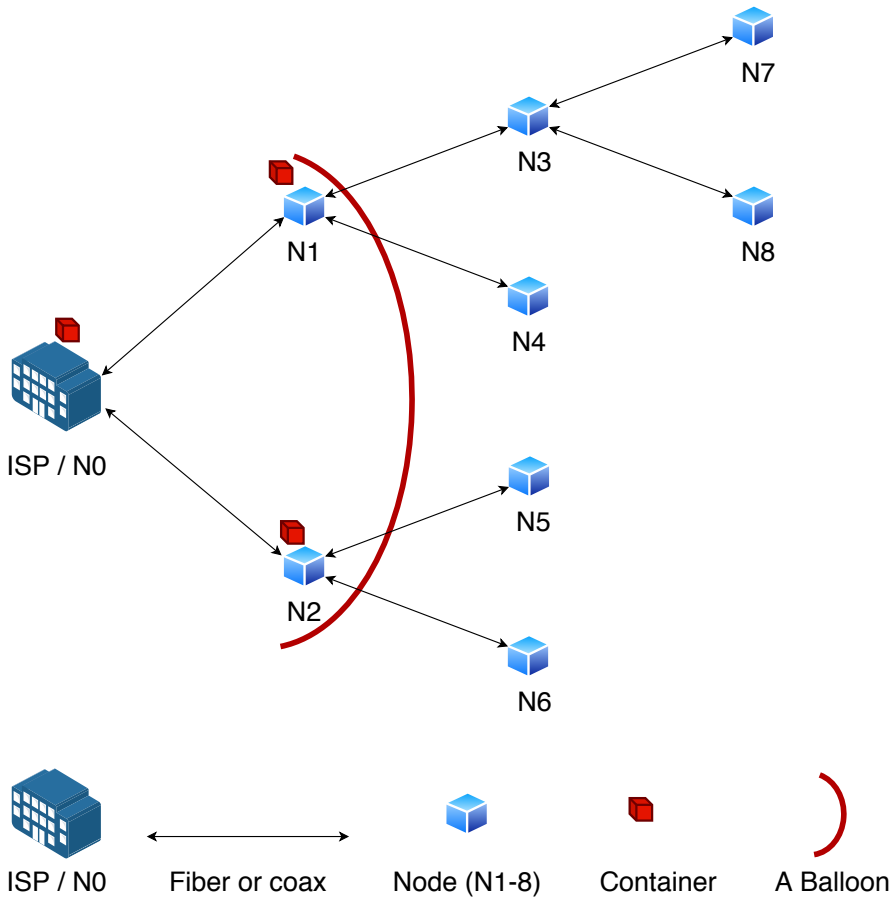

Figure 4.6: The balloon model, before. One container running at N0(depth 0). The same container running at N1 and N2 (depth 1), because the balloon has expanded out to depth 1. The red line signifies the current depth of the balloon.

on the ”pay as you go” model, and have proven to be very popular. That is because you only pay for what you use. Adapting this to our distribution models would enable the ISP to gain a profit. We can let the different models be a base for price. The micromanagement model is the most advanced, therefore the most expensive, the fog model comes second because it covers all nodes along the path, but still dynamic and requires administration to a certain point. Lastly, the balloon model. It is cheapest because it is simple and covers all nodes at each hop it takes, which is not very complicated. From here, it is natural that the price depends on the number of containers in use. That fits the pay-as-you-go price model.

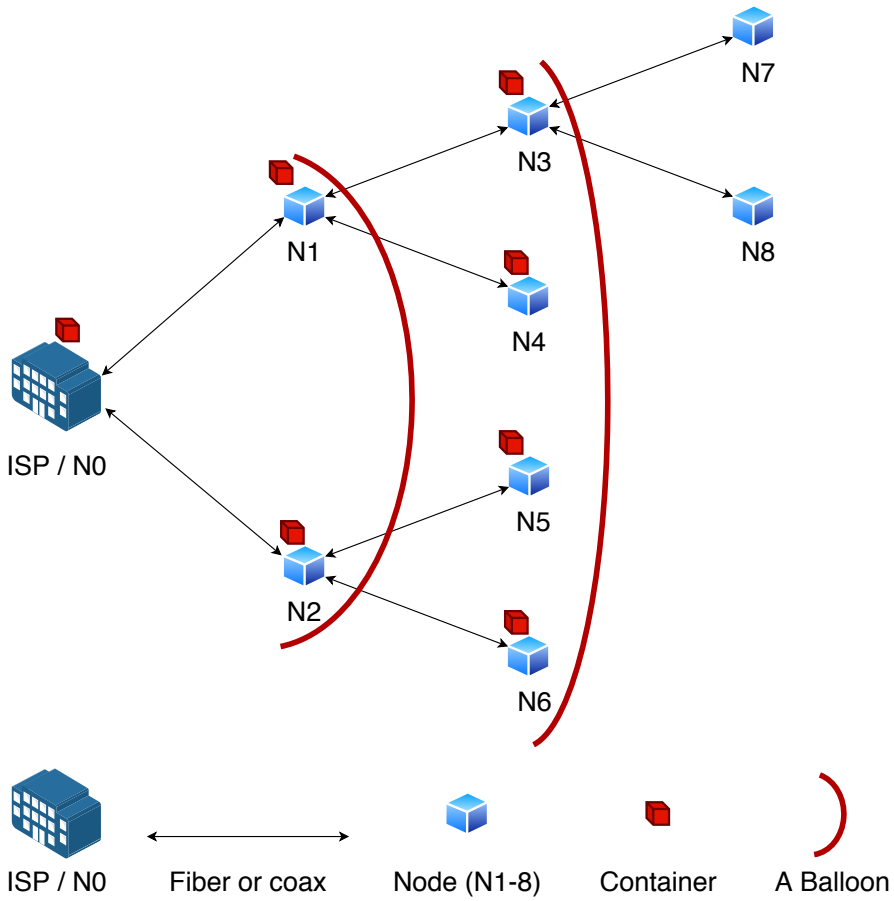

Figure 4.7: The balloon model, after. Same container as before, but the balloon has expanded to depth 2 which is signified by the red lines, and the container is now deployed at N3-6 in addition to N0-2.

4.7 Use Cases

This section provides use cases. We want realistic use cases, so we can find out if our models are applicable to more than just modeling. We have three use cases. The first one is about live video streaming and stream cache. The second one is about streaming of a TV-show episode. The third and last one is about gaming. These cases all have three actors, which are the end-user, the service owner, for example Netflix, and the ISP. The ISP wants to offer a distribution of services to a service owner, so the end-user experience of the provided service, like Netflix or video games, are as good as possible.

Previous figures of the infrastructure are simple models. A real ISP infrastructure, is significantly more complex. Figure 4.10 shows a more realistic picture, but this is only a model of an infrastructure that actually is even more complex. We have previously looked at nodes in these figures, a more accurate description would be to look at nodes as hubs. This extends

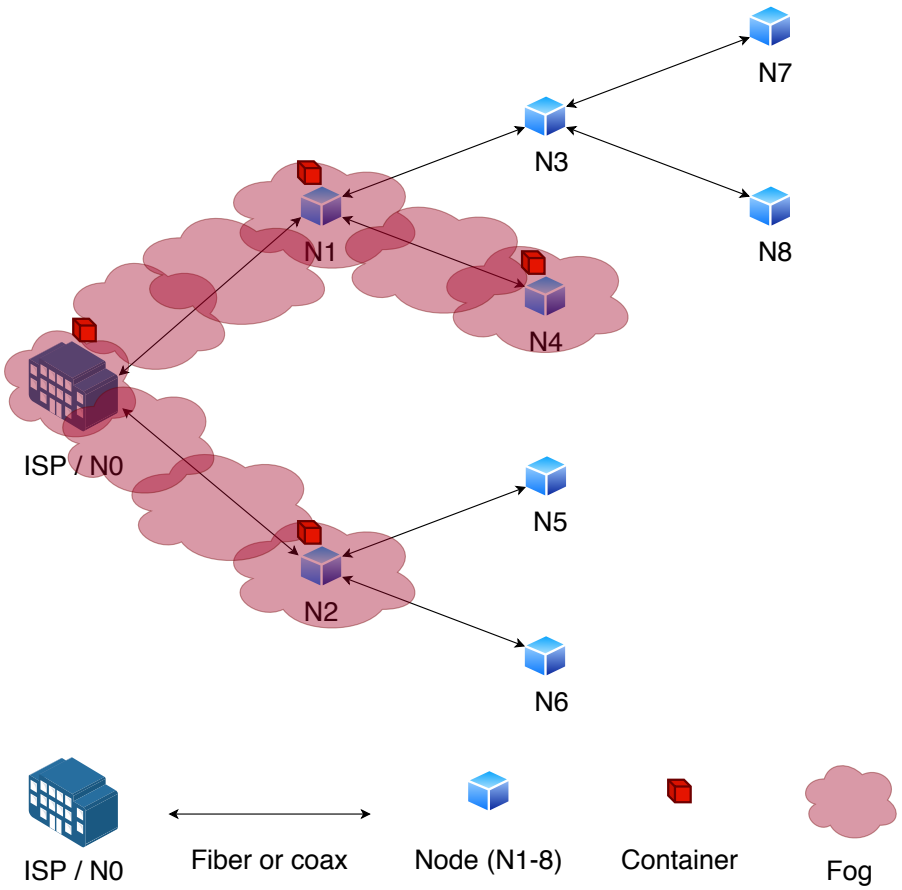

Figure 4.8: In this figure of the fog model, a container is running at N0, N1, N2, and N4. Therefore the red fog also covers only these nodes.

the complexity, because as already mentioned, several nodes are connected to one hub, and there are several hubs.

4.7.1 Live Streaming and Stream Cache

In this use case we imagine that the football team in Lillestrom named ”Lillestrom Sports Klubb” (LSK), in Norway, right outside of Oslo in Lillestrom, is playing a match, and LSK wants to distribute this match digitally.

The End User

For many people in Lillestrom, this is a big event. There are people at the stadium, of course, but far from all fans are here. They are at home and at pubs to watch the match. These, including those who could not watch the match live, would often want to see it on a later occasion.

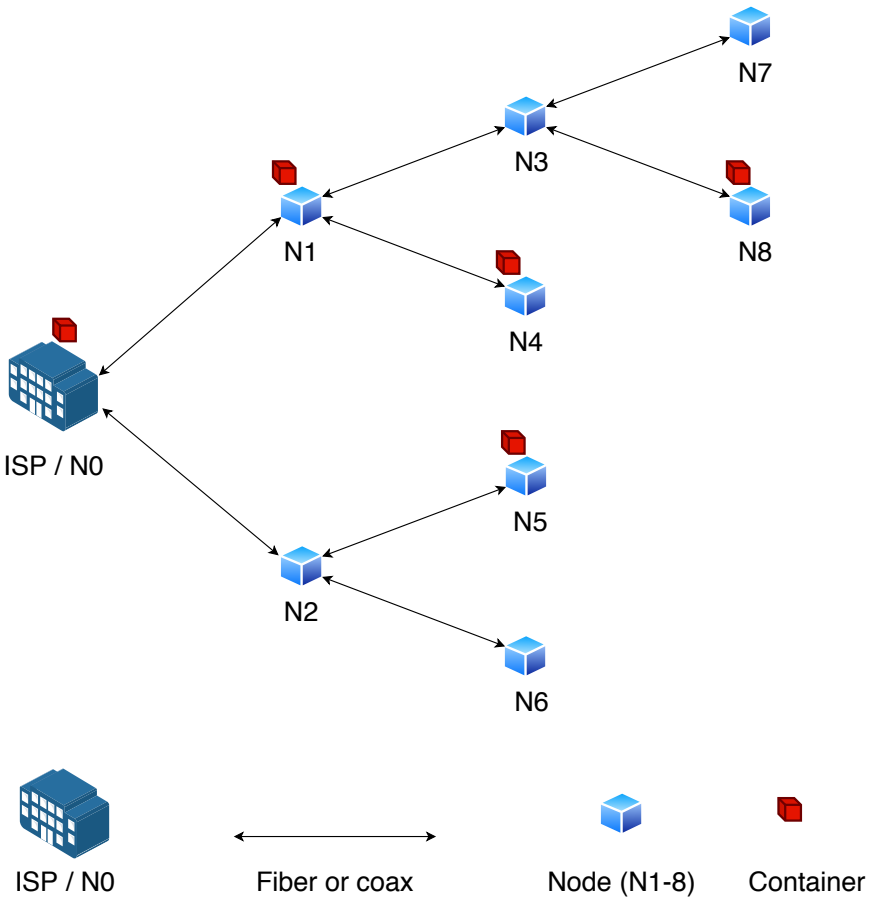

Figure 4.9: The micromanagement model. In this figure, there is a container at N0, N1, N4, N5, and N8. This model demonstrates that the micromanagement model is very dynamic, but also more administrative demanding.

The Service Owner, LSK

LSK has a digital strategy because they want to distribute football matches in a new way to reach a wider audience, and a way to get more money from sponsors. To make digital distribution possible, money is essential, because digital distribution is very expensive, which means, these two go hand in hand.

To distribute football matches digitally, cameras and a production team are necessary, and both are expensive. In other countries like the USA, football preseason is an arena for each club to distribute their content like LSK wants to. Each club has its own TV-channel in different scales. But, LSK is in a very different financial situation. One option is to hire an IT-company to handle servers and caching of the recorded match, but this will likely be very expensive in the long run. The end-user is also demanding to

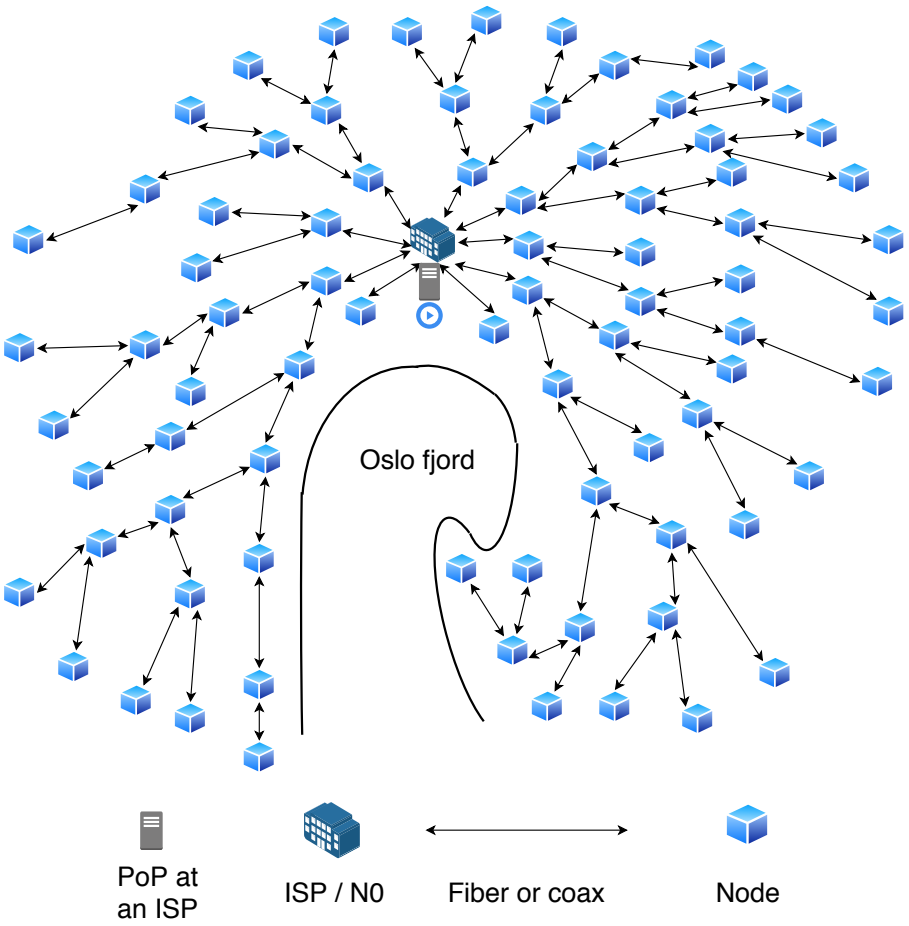

Figure 4.10: A more realistic infrastructure, but still only a model of something a lot more complex. This infrastructure figure has 80 nodes including the ISP/N0. The tree structure persists.

video quality. Many prefer 1080p or higher, which requires both network bandwidth and capacity in general. All factors point to the need for a service that can make LSK as independent as possible.

The ISP, Get

The ISP Get, has a service, called stream cache, which can communicate with the software that is used to film the match. This way the match can be distributed locally in Lillestrom after it is finished as well. Because the match is filmed and then distributed to be cached in the infrastructure, locally in Lillestrom, we avoid unnecessary load on links and nodes between Lillestrom and Get. Streaming of the cached match could be done with a service like Youtube, from the PoP at Get, but then we would put load on parts of the infrastructure that can be used for something else.

We have our three models for distribution. The balloon model

distributes to all nodes at every hop it ”iterates” through, which makes it a bad choice for this use case, because we only want the match to be distributed in Lillestrom, and maybe a few other places in and around Oslo. The fog model on the other hand enables more dynamic distribution, which makes it a good fit for this use case. The best model would be micromanagement, but it generates a lot of work for Get who possibly need to have someone monitoring at all times, because this model is very dynamic. That is not economical for Get, as they want to make a profit. This use case is small in the big picture, and the potential of making a profit is therefore small. That is why the fog model is the most applicable in this use case. Figure 4.11 illustrates this use case. The top right corner of the infrastructure in this figure is magnified. The ellipse above the infrastructure itself, illustrates even closer how a realistic infrastructure looks like. Over 40 nodes connected to 13 hubs within Lillestrom. A more accurate picture of how much infrastructure there are in a small geographical area.

4.7.2 Streaming, Game of Thrones

In this streaming use case, we assume that an episode of a popular TV show like Game of Thrones (GoT) is launched. This series is known to be extremely popular and has crashed servers, which have made the service unavailable to use. When a new episode is launched, a lot of people, in many different locations, connected to many different nodes, want to see the episode at the same time.

The End User

Users all over the world pay to see content from HBO, and GoT is a particularly popular TV show. Many pay just to see this show. They gather with friends and family for dinner and drinks or a late-night snack. Both video and audio quality is essential in this use case, as many have large TV’s with Ultra HD (4K) resolution support.

The Service Owner, HBO

HBO is the distributor of GoT. They know from experience that there are great risks associated with the premiere moment, and the first hour or two after launch, because the server load is enormous. A lot of money has been used on the production and marketing beforehand, so although HBO has quite a lot of money, there are limited amounts left to use on measures to ease the load on servers for everyone, and therefore also limited amounts left to use on other measures and services as well.

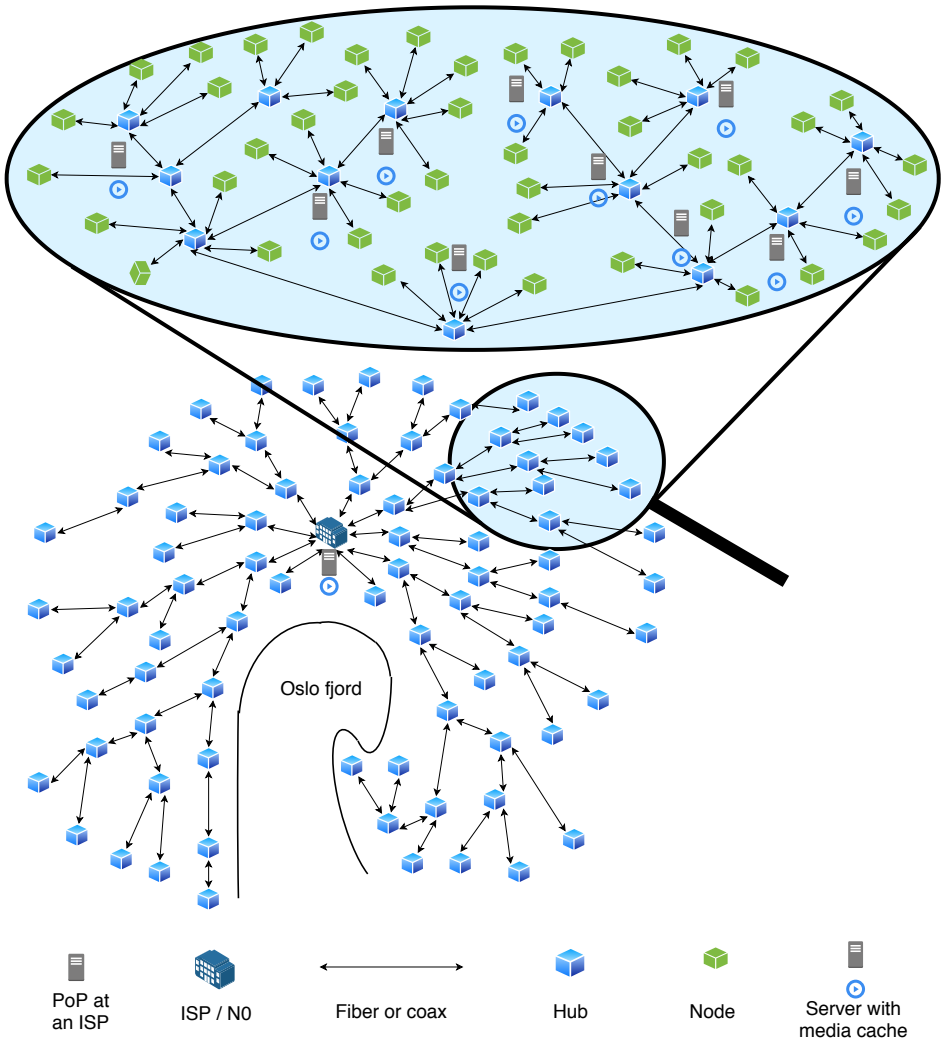

Figure 4.11: LSK stream cache use case. The top right part of the infrastructure is magnified and "zoomed" in on and showed in the ellipse right above. The ellipse shows 13 hubs and over 40 nodes connected in total.

The ISP, Get

In this use case, the ISP, Get has a service called "media cache". This enables the content provider that distributes GoT, which is HBO, to create a container for distribution within Get's infrastructure. GoT is popular, and therefore desired to distribute broadly. Broad distribution is not cheap, but end users demand stability. At the same time, Get wants this to run without too much administration, like with the Lillestrom use case. Users will use the service, almost no matter what, because they want to watch the show. However, HBO has limited financial resources left and can't afford to pay "the premium" price. It seems like Get is the winner in this situation. Fortunately the "big storm" of traffic occurs on launch, and stagnates later

on, so HBO does not have to worry a lot.

Get can offer distribution closer to the user, but HBO can't afford to be at the very edge. So, a model that pushes broadly and reaches infrastructure core, closer to end-users is a good fit. Micromanagement is the preferred, but for Get it is not. The fog model is easy to control, but in this case, we want to reach broadly. The balloon model is therefore a good choice. This also keeps the infrastructure stable and away from overload, because we spread the traffic out neatly. Figure 4.12 illustrates the balloon distribution in this use case.

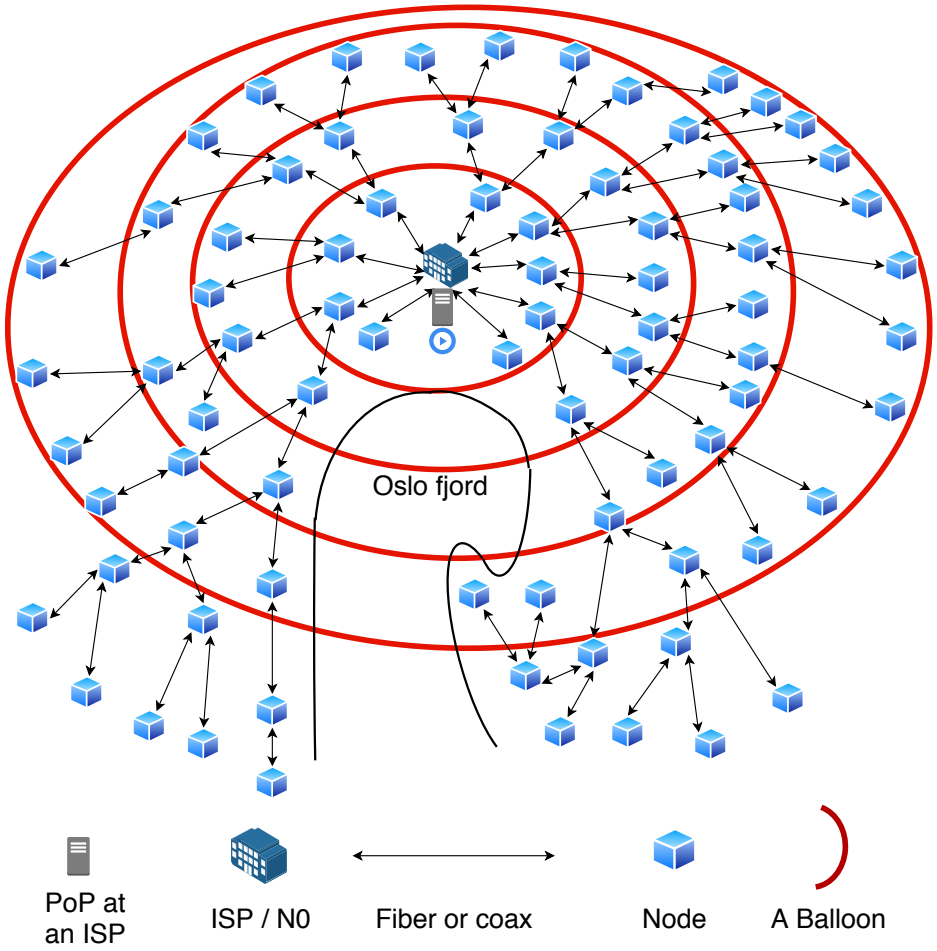

Figure 4.12: Game of Thrones media cache use case. In this use case, the balloon represented as red lines, has expanded to and covers all nodes within depth 1-4.

4.7.3 Gaming, Fortnite

In this gaming use case we use the worldwide popular video game, Fortnite, which has tens of millions of players, and is developed by Epic Games. Gaming is different to both stream cache (LSK use case) and media

cache (GoT use case), because gaming is real-time interaction. Therefore latency, or Round Trip Time (RTT) is of great matter. While stream cache and normal streaming can buff data, this is not the case with real-time interaction like gaming.

The End User

For a player playing a video game, latency (RTT) is of great of matter. RTT above 50 milliseconds can, at worst, ruin the gaming experience. They also want to play against players with the same RTT as themselves.

The Service Owner, Epic Games

Epic games want players to get online as fast as possible and play with low RTT and with people with the same RTT as themselves. Different RTT can result in unfair fights online, because one player may react slower than the other. But if the RTT is different, this reaction time may not matter anyway. Therefore Epic Games wants game servers in clusters in the infrastructure where there are people playing. That is not necessarily as broadly as with the GoT use case. What is important to remember here, is that players not are active all day and all night long. Therefore, Epic Games would want to pay for a service when the players are active, not all the time.

The ISP, Get

As mentioned already, gaming is different from streaming, because it is real-time interaction. In this case, Get can provide the possibility to host game servers spread out into the infrastructure. But, they need to be careful in their promises to the players and to Epic games regarding what to expect, because gaming is highly sensitive to latency and performance degradation. There are rarely large data packets related to gaming, but there are many and small packets, which require very fast processing. In addition to these requirements, chances are that not only Epic Games want this type of service. There are many other popular online video games.

Get needs to take into consideration how much equipment they can equip their infrastructure with. Hardware required for game servers is expensive. They cannot fill up every hub and node. There is not enough money or physical space to do that. To empower the largest backbone of hubs and some nodes on the other hand would be smart to spread out the total load. This might also help other services like HBO or Netflix as well, because the general capacity is increased.

The model that best fits this case, will be the micromanagement model or the fog model, depending on how many people who are playing, where they are located, and the load on different locations at different times, among others. At peak, gaming is a highly demanding service. But for

Get, micromanagement is not preferred at all. Epic Games might be able to afford the expenses, but a question to ask, is if it is worth it over time, which it most likely is not. Therefore the fog model is the most applicable to this use case. An illustration of how the game servers can be pushed out to the infrastructure, can be seen in Figure 4.13.

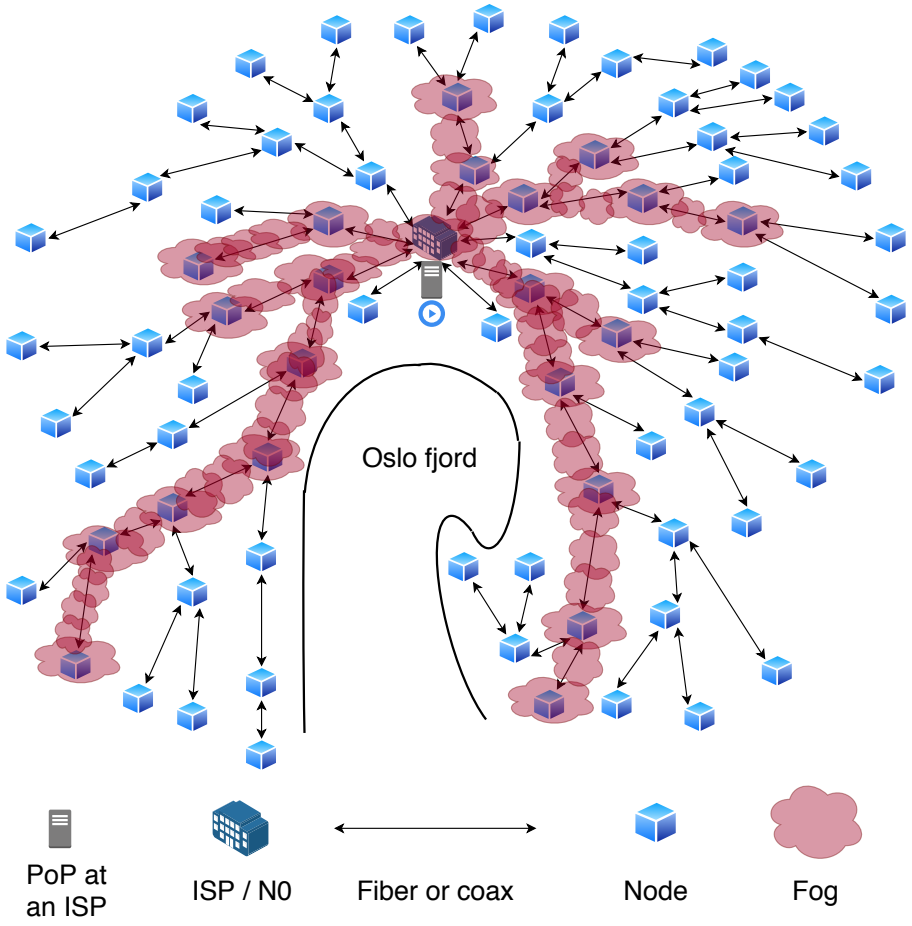

Figure 4.13: The Fortnite use case. This figure shows a fog that covers 22 nodes including the ISP/N0. Mostly southwest and southeast, but also around 10 nodes northeast and west.

Chapter 5

Implementation

Regarding hardware specifications, there is naturally a difference between what would be implemented in a real-life ISP infrastructure, and what is realistic in this chapter. Therefore, a technical framework of what would be realistic in real life will be presented, before we look at our models and a prototype.

5.1 Technical Framework

The ISP infrastructure is complex, and in real life we would need powerful hardware to cope with challenges such as traffic spikes, and cutting of cables. Regarding the technical specifications of the infrastructure, there are a few things we have to keep in mind and take into consideration. The ISP root node (N0) is more powerful than the rest of the nodes, because it would reside in the server room of the ISP itself. N0 has more RAM, more hard drive (HDD) space, and a more powerful CPU. Powerful hardware is also very expensive. An implementation of this would cost several million Norwegian kroners (NOK).

The overall capacity of the infrastructure is a balance between physical space, scale and budget. When we deal with simple web traffic and simple applications like newspapers and Facebook, one Raspberry Pi 4 Model B should be powerful enough. But, from our use cases with different types of streaming and gaming, we know that the amount of data to be processed, exceeds what a Raspberry Pi can handle. Raspberry Pi 4 specs can be seen in Table 5.1. Several Raspberry Pies may be able to handle streaming on a certain scale, but gaming requires very powerful CPUs, as well as fast and heavy amounts of RAM.

Model	1 Model B+	2 Model B	3 Model B+	4 Model B
CPU	700Mhz	900MHz Quad Core	1.4GHz Quad Core	1.5GHz Quad Core
HDD	Depend on SD-card	Depend on SD-card	Depend on SD-card	Depend on SD-card
RAM	512MB	1GB	1GB	4GB
OS	Raspian	Raspian	Raspian	Raspian
Price	349	435	479	729

Table 5.1: Specs of Raspberry Pie 1-4

5.1.1 Hardware Capacity of a Physical Node

In a real-life infrastructure, a node is approximately one cubic meter. This means that there is room for at least some, if not several Raspberry Pies. To be capable of processing the data generated from gaming, several CPU cores running at several gigahertz are required. Large amounts of RAM as well. Probably 32GB, maybe even 64GB of fast RAM, are preferred to handle traffic from many players at the same time.

Beforehand a node has network equipment installed as well, which means that it has got some space, but not very much. Small, powerful and quite expensive components are preferred to cover most scenarios with computing power. Table 5.2 shows approximate specs and prices.

	Specs	Price (in NOK)
CPU	Intel i9 14 Core 4GHz	10 000
HDD	2-3TB SSD(Solid State Drive)	2500
RAM	64GB DDR4	10 000
NIC	10Gb ethernet x2	4500
		Total: 27 000

Table 5.2: Approximate real hardware specs of a node.

5.1.2 Applying Requirements to the Infrastructure

If we apply these requirements to the more complex infrastructure with many nodes, we see that it is a very expensive affair to implement in this infrastructure. We see that there are 80 nodes in total in our map, including N0, which generates a cost of over 2 million NOK.

Price of one node:

$$10000 + 2500 + 10000 + 4500 = 27000$$

Price of all nodes:

$$27000 * 80 = 2160000$$

The prise is just in order to buy it. The price of placing it out in our infrastructure is not included. In real life, the infrastructure is much more complex, which means we can multiply the price at least a few times. Costs of ten, if not twenty or thirty million NOK is not unlikely. This is of course just an estimate, but if we say that this does not include salary and other expenses for those who install it, total expenses are even higher.

5.1.3 Raspberry Pies Installed at Nodes

We know that we can get a Raspberry Pi 4 for 700 NOK. This does not cover power cables and other necessities. Adding this, we are at approximately 1000 NOK. Hardware price for one node would then 10 000 NOK if we place 10 Raspberry Pies at one node. That is under half the price of the more powerful hardware, and Raspberry Pies can handle streaming to a certain level. Although, we want to be certain that we cover more demanding services like gaming as well. That is because we know there will be a lot of that as well.

It is important to remember, that even if one installs several servers at one location, our model still considers it as one single node.

5.2 Programming Languages and Data Representation

We need programming languages and a way to represent data to create scripts, programs, and represent data. In this implementation, Python, Bash, and JSON are used.

5.2.1 Python

Python is a very popular, high-level programming language, which also is suitable for scripting. It is easy to become familiar with, and highly applicable. If something is programmable in Python, it is most likely easy to adapt to other technologies as well, which makes it likely to succeed when using Python. Also ISPs.

5.2.2 Bash

Bash is the most common shell interface of Linux, which Ubuntu is based on. Bash can be challenging to use, because one has to be careful with the syntax. Small mistakes can be fatal. However, it is very popular, because Linux is a world-wide popular operating system. Also, once you are familiar with it, many people tend to like it. We have to use bash in this implementation to run our Python-scripts, pass on script arguments, and other small operations.

5.2.3 Data Representation, JSON

SQL is a more formal, relation-driven model of data representation. NoSQL and JavaScript Object Notation (JSON) can be simpler to use when we do not know how the final model looks. SQL often represents a picture of the application logic with primary keys, IDs, what is allowed and what is not. The JSON file format has become a very popular way to represent data. It is dynamic, so it is easy to change when we want, or when it is required. That fits an exploratory thesis like this well.

5.3 Operational Workflow

To change the infrastructure state, two main operations must be performed: (1) data manipulation of the deployment state, and (2) actual deployment. JSON represents the data model. The previous approved manipulated state, is a picture of the next deployment. This means that to get the next state to deploy, we have to manipulate the JSON file containing the current data model. When it is changed, and contains the next state for deployment, it must be verified to make sure everything matches the constraints of the model. After verification, it can be deployed. This workflow is similar to other configuration management tools which deal with state change, such as Puppet. We just want a service that works continuously, at all times. Figure 5.1 illustrates the workflow.

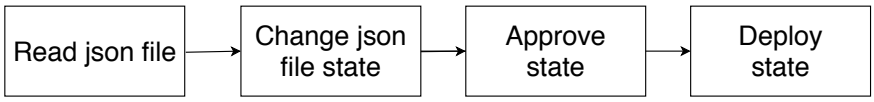

Figure 5.1: Operational workflow. Read json file $->$ Change json file state $->$ Approve state $->$ Deploy state

5.4 Data Model

JSON represents the data model and paints a picture of the infrastructure. Inside the JSON file we find a ”containerlist” and a ”nodelist”. Listing 2 shows this file without any containers or nodes. An example of a container in the containerlist can be seen in the list below. It shows the properties of a container entry.

List of fields in a container entry:

- ”name”. This is the container name.
- ”resource_memory”. This is the amount of memory required by the container, in megabytes

- ”resource_cpu”. This is the amount of CPU required by the container. It is represented as a whole number or a fraction.
- ”customer”. This is the company or service that has content in the content in the container.

Listing 3 shows a container ”c1” represented in JSON format. Resource memory is set to 256, resource CPU is set to 0.2, and the customer is HBO.

```
{
    "containerlist" : [

    ],
    "nodelist" : [

    ]
}
```

Listing 2: The base JSON file with empty containerlist and nodelist.

```
"containerlist" : [
    {
        "name" : "c1",
        "resource_mem" : "256",
        "resource_cpu" : "0.2",
        "customer" : "HBO"
    },
]
```

Listing 3: Containerlist containing a container with the name ”c1”. It requires 256 of memory and 0.2 of CPU. The ”customer” is HBO.

The nodelist can be filled with nodes. A node entry also has properties like a container entry in the containerlist. Properties of a node can be seen in the list below. A node entry is illustrated in Listing 4.

List of fields in a node entry:

- ”name”. This is the node name.
- ”parent”. This is the parent node name.
- ”children”. This is the name, or names of this node’s children.
- ”leaf”. This field is true if the node is a leaf node, and false if it is not.

- ”households”. This is the name or names of households connected to the node.
- ”capacity_mem”. This is the total memory capacity of the node.
- ”capacity_mem_used”. This is the amount of used memory.
- ”capacity_cpu”. This is the total CPU capacity of the node.
- ”capacity_cpu_used”. This is the amount of used CPU.
- ”containers”. This is the name, or names of the containers running at the node.
- ”depth”. This is the node depth. Number of hops away from the root node (N0).

```
"nodelist" : [
    {
        "name" : "N0",
        "parent" : "none",
        "children" : [ "N1", "N2" ],
        "leaf" : "false",
        "households" : [],
        "capacity_mem" : "8192",
        "capacity_mem_used": "256",
        "capacity_cpu" : "16",
        "capacity_cpu_used" : "0.2",
        "containers" : [ "c1" ],
        "depth" : "0"
    }
]
```

Listing 4: Nodelist showing N0 with container c1 running on it. The following node properties are changed in addition to running containers: Used memory and used CPU.

Now we show a version of the JSON file that reflects a figure from an infrastructure state. We use the example from the design chapter of the balloon model ”before”, where a container is running at N0-3. Figure 5.2 illustrates this. Listing 5 shows this infrastructure represented in JSON. The model can be extended to contain more.

5.4.1 Transactional Focus

As previously described, we use JSON to represent the infrastructure, and Python to modify this JSON file. This means that first the state-file is changed, before the state is deployed, and if anything fails or any

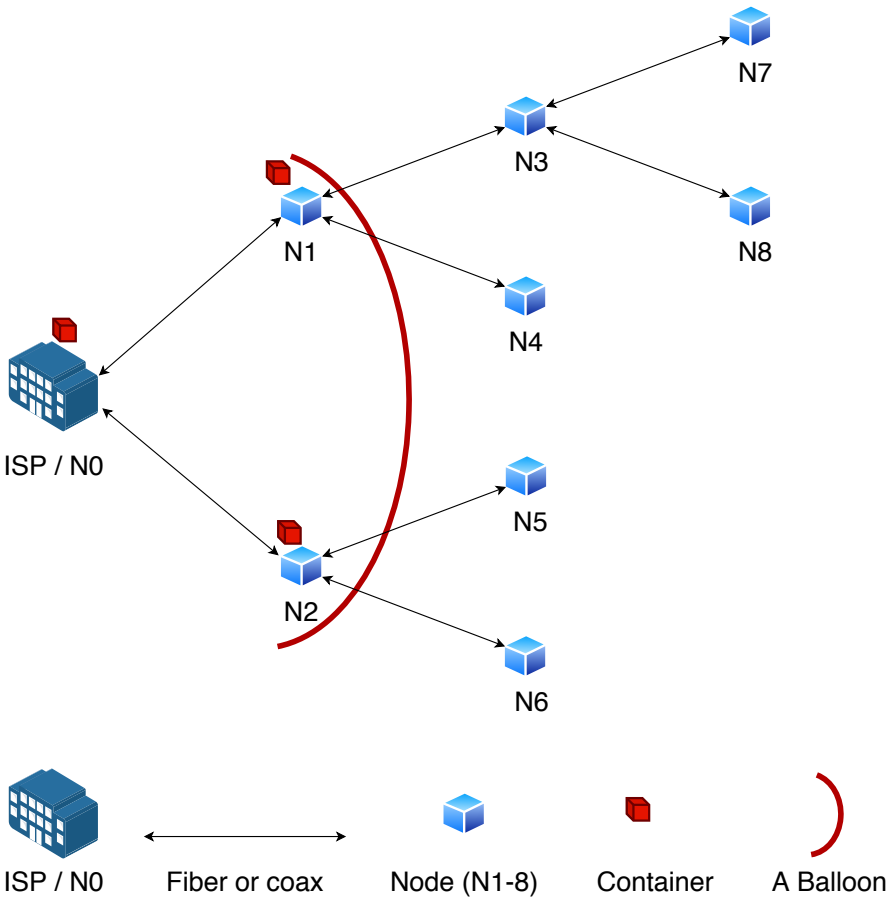

Figure 5.2: This is the same figure that showed the balloon model ”before” in the design chapter. The figure shows the infrastructure state with container c1 running on N0-2. The red line signifies the current depth of the balloon.

exceptions occur, everything is reset and nothing is changed. By doing this, we keep a transactional focus. We either do everything, or nothing, to prevent errors.

In this thesis, we intentionally save every new state as a separate file. This is in order to observe what happens more closely. A realistic system would use a database.

5.5 Prototyping by Doing Operations on the Infrastructure Tree

In the design chapter, we talked about the tree structure and three different models to apply to it. This section covers these actions, and other functions such as non-functional functions.

```
"nodelist" : [
    {
        "name" : "N0",
        "parent" : "none",
        "children" : [ "N1", "N2" ],
        "leaf" : "false",
        "households" : [],
        "capacity_mem" : "8192",
        "capacity_mem_used": "256",
        "capacity_cpu" : "16",
        "capacity_cpu_used" : "0.2",
        "containers" : [ "c1" ],
        "depth" : "0"
    },
    {
        "name" : "N1",
        "parent" : "N0",
        "children" : [ "N3", "N4" ],
        "leaf" : "false",
        "households" : [],
        "capacity_mem" : "4096",
        "capacity_mem_used" : "256",
        "capacity_cpu" : "8",
        "capacity_cpu_used" : "0.2",
        "containers" : [ "c1" ],
        "depth" : "1"
    },
    {
        "name" : "N2",
        "parent" : "N0",
        "children" : [ "N5", "N6" ],
        "leaf" : "false",
        "households" : [],
        "capacity_mem" : "4096",
        "capacity_mem_used" : "256",
        "capacity_cpu" : "8",
        "capacity_cpu_used" : "0.2",
        "containers" : [ "c1" ],
        "depth" : "1"
    },
]
```

Listing 5: N0-2 with containers c1 running on them. CPU and memory capacity used adjusted on all nodes, and ”containers” is filled with ”c1”

All models have one thing in common. The first container is always deployed at N0, which is the root node at the ISP. This means that the container containing the content is deployed at the root node before anything else happens.

We won't use the tree structure used in Figure 4.1 with the total of 9 nodes, because that is only a simplification. Will use the tree structure in Figure 4.10 which was used in our use cases. This way we can better show the magnitude of the infrastructure and what we want to apply to it.

5.5.1 PraaS.py Script Arguments

We have our Python PraaS.py. To easily send different input to this script, command-line arguments have been implemented. In Table 5.3, all these arguments and argument descriptions are listed. This is a very helpful way to work, as arguments can be added if needed.

Optional arguments	Argument description
-h, –help	show this help message and exit
-v, –verbose	Turn verbosity on
-p, –print	Print the current state
-R, –register	Register a new container
-f STATEFILE, –file STATEFILE	File containing state
-j JSON, –json JSON	Container information as JSON
-C CONTAINER, –container CONTAINER	Container to deploy
-B BALLOON, –balloon BALLOON	Balloon to node n
-F FOG, –fog FOG	Fog to node n
-n NODE, –node NODE	Node
-d DEPTH, –depth DEPTH	Depth (used when using Balloon strategy)

Table 5.3: Praas.py arguments. Running Praas.py with "-h" as the only argument, prints out the content of this table.

Read State From File

This function, which is listed in Listing 6 opens, reads the JSON file, and returns the state as a local data structure consisting of nested Python dictionaries. This file contains the infrastructure state, sent as an argument when running the Python-script. "json.load" is used to load the JSON file correctly. The state is used throughout the next functions in the script.

```
# This function reads the JSON file sent as an
# argument. This function is used
# to make the JSON file readable for the other
# functions.
def read_state_from_file(f):
    verbose("read_state_from_file\n")
    with open(f) as json_file:
        state = json.load(json_file)
        return state
```

Listing 6: Function - read_state_from_file(f). Takes a JSON file as an argument ”f”, uses ”with open(f) as json_file” to open JSON as a file. The infrastructure state is returned as JSON.

5.5.2 Container Registration

The first operation that has to be done in order to continue deployment of containers, is to register one in the ”containerlist” of the state file. We register a container by sending a JSON string to our PraaS.py using arguments. We use ”-R” to specify that we want to register a container. To specify that we want to send a JSON string, we add ”–json”, followed by the string. We also need the JSON file containing the infrastructure state. Using ”-f” as argument, followed by the file, the script is now able to read it and apply the JSON string to ”containerlist”. By using the ”> new JSON state file”, we create a new JSON file with the registered container. The operation using ”>” is done to create a new file on each state, and is also done when increasing the fog or easing out the balloon as well. The registration command is listed in Listing 7, and uses HBO as an example. We also specify the amount of RAM and CPU this container requires. When ”-R” is used, our script calls the ”register_container” function listed in Listing 8. The function takes a container which is a JSON string, and appends it to the ”containerlist” in the state file. When this is done, the container is still only registered, not deployed. The function ”deploy_container_at_node” is called right after container registration to deploy it.

```
PraaS.py -R --json '{ "name" : "hbo", "resource_mem" : "256",
↪  "resource_cpu" : "0.2", "customer" : "HBO" }' -f statefile_basic.json
↪  > expHBO-1.json
```

Listing 7: PraaS.py is used in order to register the container ”HBO” at node 0. It will store the new results in the file expHBO-1.json.

```
# This functions registers a container.
# the container argument is appended
# to containerlist.
def register_container(container):
    state['containerlist'].append(container)
```

Listing 8: Function - register_container(container). Appends container argument which is a JSON string to the containerlist.

5.5.3 The Balloon

The requirements to the balloon model is that it expands and contracts based on depth, or hops in the infrastructure. A container is deployed at each node, from depth 0 out to the depth in question, which makes it a good model to deploy many containers with a single broad brush.

When the container is registered and deployed at N0, we may want to expand the balloon. To do that, we need two arguments, ”-B” and ”-d”, in addition to the JSON file we are going to manipulate. The arguments passed along with ”-B” is the container we want the balloon to contain, and ”-d” is used to specify the depth we want to expand the balloon to. For example Listing 10, where ”-B HBO” and ”-d 3” specifies that we want to expand the HBO container to depth 3 in the infrastructure tree. This command makes sure that the ”increase_balloon_to_depth” function in Listing 11 is executed.

The ”increase_balloon_to_depth” in Listing 11 takes two arguments, container and depth. To find which nodes that should have the container, we use a for loop to check the depth of all nodes. If the depth is larger than zero and larger than or equal to the depth of the node in question, we check if there are any containers at the node. If not, we deploy, else we use a for loop to check all containers at the node against the container we want to deploy, so we can check if it already is running there. A variable ”found” is preset to 0. If the container is found, this variable is set to 1. If found equals 0 at the end, the container is found, and we deploy.

HBO Use Case Deployment

In our HBO use case, we first register a container, then we expand the balloon out. The registration process was described in the subsection ”Container registration” above. To expand the balloon after registration and deployment at N0, we set the balloon depth to 4 in four steps. First to depth 1, then to depth 2, then depth 3, and at last depth 4.

Listing 12 shows that we expand the balloon to depth 1, creating a new state in expHBO-2.json.

```
# This function deploys a container as long as ther are enough RAM and CPU
# available at a the node.
def deploy_container_at_node(container,node):
    # using get_container to obtain the container
    new_container = get_container(container)
    # obtaining the memory and CPU used by the container
    container_memory = new_container["resource_mem"]
    container_cpu  = new_container["resource_cpu"]

    # check if sufficient mem and cpu
    if int(int(state['nodelist'][node]['capacity_mem']) -
    ↪  int(state['nodelist'][node]['capacity_mem_used'])) >
    ↪  int(container_memory) and
    ↪  float(float(state['nodelist'][node]['capacity_cpu']) -
    ↪  float(state['nodelist'][node]['capacity_cpu_used'])) >
    ↪  float(container_cpu):
        # We have sufficient memory AND CPU to deploy
        # Adjusting memory and CPU at the node according
        # to what the contianer requires.
        state['nodelist'][node]['capacity_mem_used'] =
        ↪  int(state['nodelist'][node]['capacity_mem_used']) +
        ↪  int(container_memory)
        state['nodelist'][node]['capacity_cpu_used'] =
        ↪  float(state['nodelist'][node]['capacity_cpu_used']) +
        ↪  float(container_cpu)
        # Appending container to nodelist
        state['nodelist'][node]['containers'].append(container)

        return 0
    else :
        return 1
```

Listing 9: deploy_container_at_node(container,node)This function obtains the container we want to deploy at a node from the containerlist. It takes the amounts of RAM and CPU required by the container and adjust RAM and CPU used at the node accordingly before it appends the container to the node.

```
PraaS.py -B "hbo" -d "3" -f expHBO-1.json > expHBO-2.json
```

Listing 10: PraaS.py is used in order to expand the balloon of the container ”HBO” to depth 3. It will store the new results in the file expHBO-2.json.

```python
# increase_baloon_to_depth(container,depth):
# Function for increasing the balloon. When increasing, containers will not be deleted
# from the previous depth
def increase_balloon_to_depth(container,depth):
    # getting depth of parameter node
    for d in state['nodelist']: # looping through nodelist
        # checking if node depth is larger than 0 and
        # if nodedepth is equal or less than the argument depth
        if int(d['depth']) > 0 and int(d['depth']) <= int(depth):
            # checking if there are any containers at the node at all
            if d['containers'] == []:
                # deploying container if there arn't any containers present at the node
                deploy_container_at_node(container, int(get_node_index(d['name'])))
            else:
                # using a found variabel to tell if the container is present when
                # there are containers at the node. This way we eliminate
                # the risk of duplicates.
                found = 0
                for cont in d['containers']: # looping through containers at node.
                    if container == cont:
                        # if the container is present, found -> 1
                        found = 1
                if not found:
                    # container is not found/present. Found -> 0 and we deploy container
                    deploy_container_at_node(container, int(get_node_index(d['name'])))
        else:
            verbose("We need to make sure it's not here")
```

Listing 11: Python function - increase_balloon_to_depth(container, depth). This function expands the balloon out to the desired depth. We iterate through all nodes, and deploy at all nodes with depth larger than 0, and equal or less to the desired depth. However, deployment only happens if the container is not running there already.

Next, Listing 13, 14 and 15 show how we expand the balloon to depth 2, 3 and 4. From registration and deployment to the balloon is expanded to depth 4, we have completed only 5 operations in total. That demonstrates how we can turn something quite extensive and complicated, into an easy

```
PraaS.py -B "hbo" -d "1" -f expHBO-1.json > expHBO-2.json
```

Listing 12: PraaS.py is used in order to expand the balloon of the container ”HBO” to depth 1. It will store the new results in the file expHBO-2.json.

operation of a few lines. PraaS.py, of course, supports going directly to depth 4, making the process even simpler.

```
PraaS.py -B "hbo" -d "2" -f expHBO-2.json > expHBO-3.json
```

Listing 13: PraaS.py is used in order to expand the balloon of the container ”HBO” to depth 2. It will store the new results in the file expHBO-3.json.

```
PraaS.py -B "hbo" -d "3" -f expHBO-3.json > expHBO-4.json
```

Listing 14: PraaS.py is used in order to expand the balloon of the container ”HBO” to depth 3. It will store the new results in the file expHBO-4.json.

```
PraaS.py -B "hbo" -d "4" -f expHBO-4.json > expHBO-5.json
```

Listing 15: PraaS.py is used in order to expand the balloon of the container ”HBO” to depth 4. It will store the new results in the file expHBO-5.json.

5.5.4 The Fog

The requirements for the fog model is that we must be able to use it when we want to be more dynamic and ”picky”. The fog is deployed in a recursive manner, where a container is placed at each node in the shortest path between the root node and the furthest node of the fog. First on the node in question, then we recursively move towards N0 and make sure the container is deployed at every node on the shortest path up until N0. Multiple fogs can exist in the same tree, like branches. But each fog is from its furthest node and the root. Also if two fogs overlap, one would still only have a single container the node where they overlap. As opposed to the balloon, where one container only can be a part of one balloon.

When we increase the fog, we must make sure that the container in question is not already running on the node we are trying to deploy it on. There is no point in having two identical containers running at the same node. That takes up unnecessary resources. The container is deployed if it doesn't already exist on the node.

The first step of the fog, is as it is with the balloon. We have to register the container. When the container is registered and deployed at the first node, we want to increase the fog to another node. To increase the fog we need two arguments, ”-F” and ”-n”. Using ”-F” we specify that we want to increase a fog, not expand a balloon, and ”-n” is used to specify the node we want to increase the fog to.

The ”increase_fog_to_node” function which is listed in Listing 16 is recursive and takes two arguments, container and node. First we check if the node is equal to zero. If it is, we return because node zero is the root and the container is already deployed there. If the node is not zero, we continue to a for-loop. This for-loop loops through the deployed containers at the node to check if the container is already deployed. A boolean ”found” is set to 1 if the container is found at the node. Next we check if found is true or not. If not, we call the function ”deploy container to node”, because if ”found” equals zero, the container is not already deployed. In the last check, which makes this function recursive, we check if the node is not zero. If it is not, we call ”increase_fog_to_node” again, only this time, we send the parent node as an argument. This way we move towards the root node to make sure that all nodes along the path have the container running.

The Lillestrom Use Case

In our Lillestrom use case we first register the container by sending the JSON string in Listing 17 as an argument to the script. The same as with HBO, only name and customer are set to ”Lillestrom”.

When the first container is registered and deployed at N0, we want to increase the fog out to Lillestrom. Lillestrom is represented in the marked area of Figure 4.11 used in the Lillestrom use case of the design chapter. Within this area we find N13, N28-31 and N46-50. We want the fog to cover these nodes.

To increase the fog to these nodes, we use ”-F” and ”-n” with arguments. First, we increase the fog to N13. The operation to do this, is listed in Listing 19. When we do that, the container will also be deployed at N2, because N2 is the node between N0 and N13. The fog is always deployed at all nodes on the shortest path to the node we actually wanted to deploy on in the first place.

To cover all the other nodes we run the same command with some changes. We change the ”-n” argument and what files to use. The file specification is changed from ”expLillestrom-1.json $>$ expLillestrom-2.json” to ”expLillestrom-2.json $>$ expLillestrom-3.json”, because we always want

```python
# increase_fog_to_node(container,node):
# recursive function which starts on the node
# in question and traces back to N0.
def increase_fog_to_node(container,node):
    # step0: if root node , return.
    if node == 0:
        # code for returning none
        return

    found = 0
    # step1: deplay container at node, if not already there.
    # looping through containers in the argument node
    for cont in state['nodelist'][node]['containers']:
        if container == cont: # checing if container is present
            # found -> 1, because container is present
            found = 1

    # if found = 0 -> container not present at node -> deploy!
    if not found:
        #call deploy container to node
        deploy_container_at_node(container,node)

    # step2: call function: increase fog to node (container,parent).
    if node != 0:
        increase_fog_to_node(container,
        ↪  int(get_node_index(state['nodelist'][int(node)]['parent'])))
```

Listing 16: Function - increase_fog_to_node(container, node). This function recursively iterates from the node in question, back towards to N0 to make sure all nodes on the shortest path have the container running.

```
{ "name" : "Lillestrom", "resource_mem" : "256", "resource_cpu" : "0.2",
↪  "customer" : "Lillestrom" }
```

Listing 17: A JSON string that shows the format of how a container is being registered. It contains the name of the container, the name of the customer, and its resource usage parameters.

one state per file. Therefore 1 is changed to 2 and 2 is changed to 3. Listing 19 shows this command.

To cover the rest of the nodes we run the same command with N29-31 and N46-50. The number on the files is also increased by one each time.

```
PraaS.py -F "Lillestrom" -n "13" -f expLillestrom-1.json >
↪ expLillestrom-2.json
```

Listing 18: PraaS.py is used in order to increase the fog of the container ”Lillestrom” to node 13. It will store the new results in the file expFortnite-2.json.

```
PraaS.py -F "Lillestrom" -n "28" -f expLillestrom-2.json >
↪ expLillestrom-3.json
```

Listing 19: PraaS.py is used in order to increase the fog of the container ”Lillestrom” to node 28. It will store the new results in the file expFortnite-3.json.

We end up with 11 state files in total from the container is registered and deployed until we reach N50. The entire process could be accomplished with fewer commands, but this process was displayed here in order to provide more details. The next use case will be more effective.

Fortnite Use Case Deployment

This deployment is probably the most complex, but our model makes it easier. We start with the registration which is listed in Listing 20. Name is set to ”Fortnite” and customer is set to ”Epic games”. Resource CPU and memory are the same as with the other cases, CPU is 0.2 and memory is 256.

```
PraaS.py -R --json '{ "name" : "Fortnite", "resource_mem" : "256",
↪ "resource_cpu" : "0.2", "customer" : "Epic games" }' -f
↪ statefile_basic.json > expFortnite-1.json
```

Listing 20: PraaS.py is used in order to register the container ”Fortnite” at node 0. It will store the new results in the file expFortnite-1.json.

In this deployment we reach several hops (depth) into the infrastructure. To be as effective as possible, we deploy at the deepest nodes right away. This way we also cover all other nodes on the path in between. The nodes in question are N76, N19, N20, N10, N12, N31, N66 and N16. The command for deploying at N75 is listed in Listing 21. It is the same here as

with the Lillestrom use case. We increase the number on the file for each command to end up with one file for each command we execute.

```
1 PraaS.py -F "Fortnite" -n "76" -f expFortnite-1.json > expFortnite-2.json
```

Listing 21: PraaS.py is used in order to increase the fog of the container ”Fortnite” to node 76. It will store the new results in the file expFortnite-2.json.

5.5.5 Micromanagement

As we see that the micromanagement model is just as implementable as the two other models, we use this model to see what is possible, not what is not possible. Therefore it is also out of the scope of this thesis to demonstrate it here.

5.6 PraaS as a Reference Implementation

Our tool, PraaS.py represents a minimalistic implementation of the essential operations that we see is possible. The minimalistic approach is intentional to map out what it would be like to make something like this for an ISP infrastructure.

We have also chosen to save the state as a JSON-file at every step for the same reason. We wanted to follow all steps accurately to find small errors and run tests to fix them along the way. One file per state made that possible. By doing it like this, we avoided to run everything each time if a small error occurred.

This is the foundation, a type of a base, a ”Hello world” for Proximity-as-a-Service.

Chapter 6

Analysis

We want to analyze the model and the data model, say something about the choice of languages and talk about the use cases in regards to our problem statement. We will also look at pricing of the models. In the end we analyze the prototype. Does it lack anything, and could we have done something about it?

6.1 The Model

Our implementation has implemented a prototype based on the designs in the design chapter. A fundamental aspect of this prototype is that the data model is based on a tree structure. In this tree structure each node has a static location that separates it from other clouds. Moreover, a tree structure also makes a lot of sense, because traffic rarely or never is transferred between household directly. The traffic is sent or received from the ISP, because there often are a client and a server, and the server runs at the ISP. The tree model also fits our problem statement in the matter of proximity, in a different way than other clouds today, because the existing ISP infrastructure is created to be close to its customers. Clouds and data centers are created to be as central as possible, to reach extremely broad, across countries, even continents.

The model has been great to reduce complexity. The ISP infrastructure is complex. We have managed to simplify this infrastructure by using our model. Could it have been done differently by not using a tree structure? Yes, nodes could have been placed a little bit more random and several nodes could have been connected, to create redundancy for example. Redundancy is a fairly important feature of the internet today, which could have been taken more into consideration.

6.2 The Data Model

JSON has been used to create the data model. It is a data format that is easy to program with, and it is easy to read. That makes it easy to verify that the information is correct at all times. JSON is also very adaptable in a way that makes it easy to expand without too much work. This way it can be expanded without having to add a lot of new code.

The infrastructure which the model is based on is also very static. We do not make significant manipulation to the nodes, besides adjusting resources and adding containers. What is important is which containers that are running at a node, not which node is running where. That makes the node's relation less dynamic.

6.3 Programming Languages

Python was the choice of programming language to work with the model, which has worked fine. The tool created to manipulate our model does not require any special libraries or a specific language. Integration into other tools programmed in a different language is therefore not a problem.

6.4 Use Cases

We have three realistic use cases that work well with our infrastructure and model. Our problem statement refers to creating proximity as a product, with technologies like containers as a base. Can we use the tool and model as a base for a product? Would it be worth the investment for an ISP? We have seen that it is a possible job to create when you got the programming skills. Our model and idea offer an opportunity that makes the ISP special, which is distance. Realistic cases help to see that we can use the ISP infrastructure to push services and scaling of these, closer to the user in a way we never have seen before. Instead of latency close to 50 milliseconds to reach a service at a data center, we can get it down to a lot less. For example 5 or maybe 10 milliseconds of latency, given that the service is running at the depth and location you are at, or a depth close to you. Increasing from depth 2 to 3 would make a difference. Everyone nearby get lower latency as well. The load on the rest of the infrastructure is also reduced, because there are maybe a few hundred instead of several thousand requesting a file over a short period of time. If we say that the latency between the ISP and depth 5 is 25 milliseconds, we can get it down to 5 milliseconds by pushing the service out to depth 5. Playing a video game, 20 milliseconds can be fatal.

6.4.1 Products

We have used two different models in these cases, the balloon and fog. Using our models, we could create a variety of products using different pricing models. The balloon is simplest and does not require a lot of effort to administrate. Therefore it makes sense that it has the cheapest base price. Secondly, the fog is more dynamic and we can be pickier, which makes it a little more expensive. Lastly, we have the micromanagement model which is the most accurate model and gives the highest level of fidelity, but it takes a lot of resources to administrate and monitor. Common for all the models is that the price increase for each node that is used, and the ”pay as you go” pricing model. ”Pay as you go” is broadly used and an easy model to understand. You pay for what you use, nothing more. Besides that the models have their differences. Each container will cost 10 NOK per hour. More information about this is found in the two subsections below about the pricing of the balloon and fog model.

The ”pay as you go” model is broadly used by cloud providers, but we adapt it to the ISP level. That can be a base for further competition between ISPs and reasons to develop new models in other business areas. One of the reasons why the ”pay as you go” model has become very popular, is because you get the power of a VM if you pay for it, not just 50 percent for example.

Pricing of the Balloon

When we price the balloon model, we need to think about how broadly it can reach, and that a node has quite expensive hardware. The closer we get to the network edge, it is more likely that each node will be used less and therefore more expensive in the long run. Therefore it is smart to increase the price gradually from depth 3 to 6, and let depth 1 and 2 have default prices. Table 6.1 shows an overview.

Depth	Price
Depth 1	10 * 9 = 90
Depth 2	10 * 14 = 140
Depth 3	10 * 1.3 * 22 = 286
Depth 4	10 * 1.5 * 18 = 270
Depth 5	10 * 1.7 * 7 = 119
Depth 6	10 * 1.9 * 9 = 171

Table 6.1: Balloon model pricing

Pricing of the Fog Model

The fog model is different from the balloon model, and therefore need a little bit different pricing. Each node still has expensive hardware, but we can reach nodes on depth 4 without having a container at every single node at depth 1 to 3. We assume that containers use the same resources as with the balloon as well. Anyway, the fog model is more dynamic, so we can accept it to be a little more expensive on depth 1 and 2. So, if we say that the depth level times ten equals the percentage we add on top of the original price per container. For example depth 3, and 3 containers. Price equals 10*1.3*3, which is 39. In Table 6.2 we can see the prices for the Fog model.

Depth	Price
Depth 1	10 * 1.1 * number of containers
Depth 2	10 * 1.2 * number of containers
Depth 3	10 * 1.3 * number of containers
Depth 4	10 * 1.4 * number of containers
Depth 5	10 * 1.5 * number of containers
Depth 6	10 * 1.6 * number of containers

Table 6.2: Balloon model pricing

Does the Prototype Lack Anything?

Could there have been done anything more with the prototype or model? We can see that a cluster of virtual machines could have been created with respect to the JSON model. This way we could have looped through the file and made sure that all nodes were running the correct containers. Another tool could have helped with that, but not Docker Swarm or Kubernetes. Technologies like these are not well suited for our model because of the tree format for example. If they were, we could just have used it, but then we most likely would not have got to where we are now. We would have had to make another tool on top of these technologies, an extension of some sort. Since we see that it is possible to create a cluster of VMs and use the JSON file to decide where to place containers, is it then worth spending time on, or should we use that time on other work?

From a cloud perspective, our implementation lack quite a few functions. For example monitoring and storage clusters. In general, the computing nodes have a large infrastructure behind them. VMs and containers have private networks across data centers as well. Well, our implementation never meant to provide heavy monitoring or anything close to what the cloud offers in those regards. We follow some container principles, like immutable computing where the container has a life cycle. When it is out of date or we need something else, it is replaced. It is running

on a computer, but does not require its own hard drive. A hybrid solution of some kind where the ISP (N0) is the public cloud in that matter would have been something to consider. In the end we want proximity to the end-user.

Chapter 7

Discussion

In this chapter we take a step back, and look at the big picture. Could we have done anything more or different with the ISP modeling? We give our thoughts about the chosen thesis format and try to find our own place in research. The problem statement is discussed. Have we provided any answers? Finally, we discuss Kubernetes shortly and look at future work.

7.1 ISP Modeling

This tree structure reduces complexity, but we could have created an ISP infrastructure on a cloud platform virtually by implementing a lot of VMs for example. All these VMs could have had several virtual network cards. All these network cards would have been possible to configure as one large router or switch. That would have eliminated distance but the complexity would have been the same, because the difference would have been a physical infrastructure created into a virtual one.

We could have used AI to mitigate the risks associated with container migration. [50] talks about container migration and how it is simpler to migrate containers rather than VMs, because containers are very lightweight. It also mentions how Cisco came with the expression fog computing, and container migration in relation to fog computing. Netflix also uses AI to create personalized suggestions, as do Facebook and many others.

Implementing AI like this would have been very dynamic, but also very time consuming, and therefore out of the scope of this thesis. But AI and similar technologies are becoming more and more popular. One way or the other, their usage will expand in the years to come.

7.2 Thoughts About the Exploratory Thesis Format

Creativity is a major part of an exploratory thesis. You do not know where you are heading at all times, and it can be difficult to find relevant literature about the topic itself or related topics. To speak about what we are doing or tell where the work is going, there is often a need for some time to finish what we are doing that day or that week.

In general it is hard to reach a conclusion without knowing all the facts. This fact does not change when working with science and computers. For this thesis it was smart to choose an exploratory approach, because there has been limited literature available. From the beginning we did not know, but had an idea that normal cloud concepts maybe wouldn't fit directly, only indirectly. Because we wanted to give the ISP more power, like cloud and content providers have, by creating something that would give them opportunities only they are in a direct position for.

A thesis or paper, or a survey may seem straight forward. But, there are certain rules to follow, like which chapters that must be included and in which order. These rules can help to structure your work, but they can also challenge certain parts of an exploratory thesis. An exploratory thesis is more iterative compared with a survey, because a survey is written with facts and comparisons from other work. An exploratory thesis may not have that much relevant literature available. That has been the case with this thesis. The work direction has changed a few times. Not totally, but some. Like when we assumptions have been challenged, which is described more detailed in the next paragraph. Some philosophizing and maturation of the pricing model in relation to the prototype has also affected this work. The exploratory thesis format has allowed us to go back and reflect on decisions made and work done, to make small improvements. That has been nice because we not necessarily knew where we were heading at all times.

Some assumptions have also been challenged. For example we have learned that the difference between fog and edge computing not necessary are that big. Fog nodes can be located at the network edge. Latency has also been discussed a lot. Latency is important, but for streaming and related activities that do not happen in real-time, it is not as critical as we might have expected to begin with. That is because the buffering of data is used to cope with data that might be received a few milliseconds too late. For gaming, it is probably more important than originally assumed, because of real-time interaction.

7.3 Our Own Location on "the Map"

There has been a lot of focus on the ISP and its surroundings in this thesis. Data centers, IoT, latency, some CDN, and subtopics have been researched to find relevant literature. When we look back, can we find something that

has a resemblance to our work? Well, there has been an explosion of articles and research within IoT. In [3] they talk about security and heterogeneous software in IoT sensors. These sensors may communicate with software running in a container. CDN with some edge and fog-related to streaming and latency, also in regards to IoT, has also got attention. CDN is talked about in [22] where they look at "The Birth of the Sub-Millisecond Internet" and how CDN has changed the internet. Somehow we have created a CDN for ISPs, because AWS data centers are located around the world and they have PoPs in other data centers and at ISPs closer to their users. Hubs and nodes in an ISP infrastructure are very close to their customers. This becomes a comparison of two things on different scales and worlds, but the principle and desire for closeness remains the same.

If we look at our work in perspective to research the past five or ten years, it is hard to find a field where we can say, we belong here. ISPs have been a part of some telecommunication conferences, and been mentioned in some papers, but rarely got more attention.

In this thesis there has been much focus close to a cloud point of view, and How IT people think in regards to the end-user. We have looked at paradigms that make it easier for people in IT to deal with complex challenges and problems. The user only sees the top of the iceberg. Still, they pay for services, because what they see is most often simple to understand and use.

In the fog of all literature, it is not unheard of, that we can find something to compare ourselves with, but it would mostly be cherry-picking and taking a principle or something into comparison. Like containers for example, but not in the same way. Because we use them to benefit the ISP directly. Facebook, Amazon and others can indirectly benefit from it.

We have talked a lot about latency, real-time interaction, streaming, gaming and so on. Latency and bandwidth versus user experience. [18] works with edge and fog computing, which we can relate to, because we want computation at the edge. We want to increase the user experience by placing services like streaming and gaming on servers some meters down the street, rather than kilometers away or in another city or another country.

In five, or maybe ten years, we will see changes like we have seen in the past ten years. Changes in different directions and more innovation. Those who don't follow and make small and large changes, may get to a point where they need to change in order to keep their customers and survive. Like with retail stores that have been greatly challenged by online shopping. The retail business is far from dead, but some stores have been forced to close their business. Therefore, we should not be surprised if we see our work, or at least nuances of it in the future.

ISPs are in constant change like everything else, but not in the same way like for example clouds. ISPs have for some time offered movies for rent, created their own streaming and content services based on content

from other distributors, which is extremely costly. Online TV can be used and viewed with an app or a browser as well. These changes are very challenging, but necessary. We should not be surprised if we see services offered by an ISP, or totally new players within the next decade, who challenge Netflix and other well-established providers and creators. Because new business models can appear from our work and others.

7.4 The Problem Statement

Now, let us see if we have provided any answers to the problem statement.

The problem statement: *How can recent technologies like containers contribute to facilitate edge-computing towards ISPs delivering Proximity-as-a-Service (PraaS)?*

There has been created a prototype model that decides which node or nodes to deploy a container on. Nodes are placed close to the network edge where we find end-users, which means that we have created a model that is able to provide edge computing.

To deliver a product or a service that customers are willing to pay for and use, financial models that already are familiar, or easy to understand are important. The customer needs to know if there is anything to gain and if the price is worth what can be or what is gained. ”Pay as you go” is an already well-established model used by many others, like cloud providers.

This pricing model is combined with the two other models. These models reduce the infrastructure complexity. Some users and technologists may understand the complex infrastructure and other complex problems, but for the everyday man of the street, friction-free simplicity is key. The average user simply does not care how it all works, as long as the services work when they are used. On the other hand, there is a difference between a container running the service hundred meters down the street, compared to a PoP in a data center somewhere.

In some scenarios, like gaming, or maybe even streaming, the user will probably notice a difference without thinking too much about it. But if noticed, will they speak about it with their friends and family? Will the total package result in more customers for the ISP, or will the difference not be big enough? Some years ago, this would probably have mattered more, but today the infrastructure is well established to handle most traffic.

TVs today support Ultra HD 4K. In five or ten years, devices are likely to be compatible with 8K. Data streams of content in 8K will demand extreme bandwidth, which means that we most likely will see another period where the race of upgrading infrastructure will be costly for ISPs.

Another product that entered the telecommunication business all of a sudden some years ago, was ”data rollover”. This product made it possible for users who didn’t use all data included in the phone subscription every

month, to bring the rest over to the next month. For example, you got 5GB included, and you use 3GB of data. The next month you have got 7GB because you got to bring the 2GB you didn't use with you to the next month. This product somehow revolutionized the telecommunication business within data quotas. One company started with it, and in a short period of time, most companies had it. That shows how quick the business develops and works, and that new ideas and models suddenly can appear and make changes.

Our model enables placing of containers at nodes. The masses which are the people, the end-user, and other service providers, want content as close to the end-user as possible for the best user experience. We have proven that it is possible to create a model where we use containers to push services to run at nodes in the ISP infrastructure. The nodes are running at the edge, only meters away from the user.

7.5 Why Not Kubernetes?

Kubernetes, which was mentioned in the background chapter, is a very popular and robust orchestration technology. Docker swarm was also mentioned. Unfortunately neither of them are directly applicable to the ISP tree model this thesis focuses on. We could maybe make it work with some ”hacks” and clever configuration. It would be ideal to be able to do this without having to worry about these restrictions, which is why we have to find a way to work around it. Create something new, that maybe could be an extension to Kubernetes and other configuration management tools later on.

Kubernetes can deploy and orchestrate, but where to deploy and what decisions to be made is something out of the Kubernetes scope regarding the ISP tree structure. We therefore want, and need to create a model where we can represent and manipulate information and data to fit the tree structure.

7.6 Can the Idea Be Transferred to 5G Radio Masts?

Some ISPs today are already providers of mobile subscriptions and mobile broadband. All you need today to get an internet connection is a SIM-card and a device that supports it. The ISPs that provide these products have already built many base stations for 5G. It is even tested in projects around the world, with success.

5G is much faster and much more responsive than 4G, which is a requirement in emerging paradigms like IoT and edge computing. This thesis aims to provide edge computing close to end-users. Everyone at a certain age today, owns a smartphone. A 5G radio mast is a very decent place for a small computer with high capacity. People already talk about

edge and fog computing in 5G, so it is safe to say that we can discuss our work as a positive contribution to possible ways to go in the future work of 5G.

The inclusion of 5G, or radio masts in general, into our model is not as seamless as one would think. We have worked with a tree-structure in this thesis. The 5G infrastructure is based on the 4G infrastructure, which means that there are several thousand base stations. It could be possible to force this infrastructure into a tree-structure, but there would be challenges to face, because there are often several masts within a few hundred meters or less. However, this infrastructure is not that massive in less populated areas, which means that a tree structure most likely could be an appropriate model here also.

7.7 Future Work

This section will discuss possible future work. Which types of projects can continue where we have left of. Here we present three possible projects. The first one is implementation in a cloud platform, where we discuss how our model can be further developed to cooperate with other tools. Secondly, we have have further exploration of the ISP infrastructure, where we can dive deeper into an ISP infrastructure to get more answers. Last, we look at refinement of business models. Can we explore and find a new way to apply them to the already existing cloud marked?

7.7.1 Implementation in a cloud platform

This thesis has created a model and prototype that has simplified something complex, but we have not created any VMs and deployed containers beside the VM used to program and test our tool programmed in Python.

We know that implementation with simple VMs would be possible. But, can we change and develop the base model to cooperate decently with Kubernetes or a similar tool when running on a cloud platform? OpenStack could also be a valid choice, but Kubernetes would be a more obvious and natural choice as Kubernetes often is associated with container orchestration.

A similar approach like we have had in this thesis would be a natural approach, because it will still be exploratory as compared to a comparative study. It would be necessary to know something about containers, and preferably something about Kubernetes. These technologies are often implemented and used in complex architectures as well. A project to investigate problems like these would therefore be a good fit for a master thesis.

By doing a project like this we could get answers about strengths and weaknesses in the original prototype, which would be of help to improve

our own work. Any new algorithms, changes in existing code, adding a parameter or simplifying it even more. Increasing the complexity a tiny bit to gain functionality would also be interesting to look at.

7.7.2 ISP Infrastructure Exploration

On the bachelor level of many engineering focused computer science educations, it is normal to do a bachelor project as a group of 2-5 persons. The group picks a task or an assignment provided by a principal, which normally is a company or an organization of some sort.

Further exploration of an ISP infrastructure would be exciting. The objective of a study like this could be to provide answers to the rest of the world related to what a real ISP infrastructure looks like. A task like this would be fitting for a bachelor student group in computer science of some sort. They could test equipment, maybe do some simulations or use small machines like Raspberry Pies and other network equipment to recreate a small part of an infrastructure.

This would probably require close collaboration with an ISP. For the student group, this is a good fit if the bachelors project is 20 credits for each student.

From a project like this we could learn more about the tree structure than we already know, like how much more complex it is. We know it is complex, but ISPs do not reveal how their infrastructure looks like. This way we could get a more general picture. Capacity is also of importance. Any traffic logs that could help us to create even more realistic cases and improve our own. All of this could be a base for any further and more complex simulations in the future.

7.7.3 Refinement of Business Models

We have developed a price model, which relates to business as well, not just technology. Can we explore these models deeper and find a way to apply them to the already existing cloud marked? If one uses numbers and statistics in a comparative study, or analyze data conducted from surveys and questionnaires to find trends and patterns, one could create more sophisticated products and models which fit the cloud.

This would require some help and cooperation with for example business students, but would give valuable data. These results and information can then be used to do realistic simulations. If we combine this study with engineering and results from the bachelor's project of the ISP infrastructure, we would have greater grounds to create and adjust for better business and technological models.

Chapter 8

Conclusion

The aim of this thesis was to bring services closer to the user than they already are in existing data centers using novel technologies.

We have developed a model and a prototype that uses JSON, and a tool Programmed in Python to distribute containers in an ISP tree infrastructure. We investigated three major distribution strategies based on the tree model where the two of them, the fog and the balloon were considered to be the central elements of the investigation. This tree model and accompanying distribution strategies have greatly reduced the infrastructure complexity and helped to better manage it.

The exploratory thesis format allowed us to continue something that we did not know the whole story to yet, and get back to it once we knew more. Despite the fact that our assumptions were challenged along the way, we feel confident that the study format was appropriate.

Our work has similarities with previous work. Containers as a topic is covered in lots of other work that have used containers in different situations. Edge computing has also become a hot topic at conferences and in research. We utilize both paradigms, just not in the same way. Because these paradigms rarely are associated with ISPs in the way we have done it.

Our model distributes containers across an ISP infrastructure and provides proximity to a mass of users who are connected at the network edge. Price models and a deeper investigation into the creation of a product is an interesting avenue for future exploration.

The essential result of this project is that the scale of a complex deployment has been successfully abstracted away. By changing one number, one can increase the number of deployed containers from 10 to 80. This demonstrates that advanced distribution of a great number of containers can be done using only simple commands.

Bibliography

[1] *Mapping Netflix: Content Delivery Network Spans 233 Sites*. URL: https://datacenterfrontier.com/mapping-netflix-content-delivery-network/ (visited on 05/08/2020).

[2] Roberto Morabito et al. "Consolidate IoT edge computing with lightweight virtualization". In: *IEEE Network* 32.1 (2018), pp. 102–111.

[3] Nitesh Mor et al. "Toward a global data infrastructure". In: *IEEE Internet Computing* 20.3 (2016), pp. 54–62.

[4] Hwejoo Lee et al. "A data streaming performance evaluation using resource constrained edge device". In: *2017 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE. 2017, pp. 628–633.

[5] Matthew Furlong, Andrew Quinn, and Jason Flinn. "The Case for Determinism on the Edge". In: *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*. 2019.

[6] *USENIX LISA15*. URL: https://www.usenix.org/conference/lisa15/conference-program.

[7] *LISA16 Conference Program*. URL: https://www.usenix.org/conference/lisa16/conference-program.

[8] *LISA17 Conference Program*. URL: https://www.usenix.org/conference/lisa17/conference-program.

[9] *LISA18 Conference Program*. URL: https://www.usenix.org/conference/lisa18/conference-program.

[10] *HotCloud '17 Workshop Program*. URL: https://www.usenix.org/conference/hotcloud17/workshop-program (visited on 03/02/2020).

[11] *HotCloud '18 Workshop Program*. URL: https://www.usenix.org/conference/hotcloud18/workshop-program (visited on 03/02/2020).

[12] *HotCloud '19 Workshop Program*. URL: https://www.usenix.org/conference/hotcloud19/workshop-program (visited on 03/02/2020).

[13] *HotEdge '18 Workshop Program*. URL: https://www.usenix.org/conference/hotedge18/workshop-program (visited on 03/02/2020).

[14] *HotEdge '19 Workshop Program*. URL: https://www.usenix.org/conference/hotedge19/workshop-program (visited on 03/02/2020).

[15] European Commission. *HORIZON 2020 WORK PROGRAMME 2014 – 2015*. 2015. URL: https://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/main/h2020-wp1415-leit-ict_en.pdf.

[16] European Commission. *Funding tender opportunities, Reference Docuemnts*. 2018. URL: https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/how-to-participate/reference-documents;programCode=H2020.

[17] Nelson Mimura Gonzalez et al. "Fog computing: Data analytics and cloud distributed processing on the network edges". In: *2016 35th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE. 2016, pp. 1–9.

[18] Kashif Bilal and Aiman Erbad. "Edge computing for interactive media and video streaming". In: *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE. 2017, pp. 68–73.

[19] *Caching Overview*. URL: https://aws.amazon.com/caching/ (visited on 03/2020).

[20] *What is a CDN?* URL: https://www.keycdn.com/what-is-a-cdn.

[21] *What Is a CDN?* URL: https://www.cloudflare.com/learning/cdn/what-is-a-cdn/ (visited on 03/02/2020).

[22] Martino Trevisan et al. "Five years at the edge: Watching internet from the isp network". In: *IEEE/ACM Transactions on Networking* 28.2 (2020), pp. 561–574.

[23] *Amazon CloudFront*. URL: https://aws.amazon.com/cloudfront/ (visited on 04/2020).

[24] *Amazon S3*. URL: https://aws.amazon.com/s3/ (visited on 04/2020).

[25] *About AWS*. URL: https://aws.amazon.com/about-aws/ (visited on 03/02/2020).

[26] *What's New?* URL: https://aws.amazon.com/cloudfront/whats-new/ (visited on 03/02/2020).

[27] *Amazon CloudFront Infrastructure*. URL: https://aws.amazon.com/cloudfront/features/ (visited on 03/02/2020).

[28] *Fast, Intelligent and Secure at the Edge*. URL: https://www.akamai.com/uk/en/ (visited on 03/02/2020).

[29] Reza Farahbakhsh et al. "How far is Facebook from me? Facebook network infrastructure analysis". In: *IEEE Communications Magazine* 53.9 (2015), pp. 134–142.

[30] *How Netflix Works With ISPs Around the Globe to Deliver a Great Viewing Experience*. URL: https://media.netflix.com/en/company-blog/how-netflix-works-with-isps-around-the-globe-to-deliver-a-great-viewing-experience.

[31] *Mapping Facebook's FNA (CDN) nodes across the world!* URL: https://anuragbhatia.com/2018/03/networking/isp-column/mapping-facebooks-fna-cdn-nodes-across-the-world/ (visited on 03/02/2020).

[32] *Explore where Office 365 stores your customer data*. URL: https://products.office.com/en-us/where-is-your-data-located.

[33] *Discover our data center locations*. URL: https://www.google.com/about/datacenters/locations/.

[34] *Global Infrastructure*. URL: https://aws.amazon.com/about-aws/global-infrastructure/.

[35] *Locations*. URL: https://www.digiplex.com/locations (visited on 04/2020).

[36] URL: https://www.cisco.com/c/en/us/solutions/enterprise-networks/edge-computing.html.

[37] URL: https://www.winsystems.com/cloud-fog-and-edge-computing-whats-the-difference/.

[38] URL: https://www.cmswire.com/information-management/edge-computing-vs-fog-computing-whats-the-difference/.

[39] Michele De Donno, Koen Tange, and Nicola Dragoni. "Foundations and Evolution of Modern Computing Paradigms: Cloud, IoT, Edge, and Fog". In: *Ieee Access* 7 (2019), pp. 150936–150948.

[40] *Service Discovery*. URL: https://ns1.com/dns-service-discovery (visited on 04/10/2020).

[41] *What's the Difference Between DNS and Service Discovery?* URL: https://medium.com/microscaling-systems/whats-the-difference-between-dns-and-service-discovery-dec4055ce4e2 (visited on 05/07/2020).

[42] *Consul*. URL: https://www.consul.io/ (visited on 05/07/2020).

[43] *Service Discovery in a Microservices Architecture*. URL: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ (visited on 04/10/2020).

[44] *Elastic Load Balancing*. URL: https://aws.amazon.com/elasticloadbalancing/ (visited on 04/12/2020).

[45] *Kubernetes vs. Docker: What Does It Really Mean?* URL: https://www.sumologic.com/blog/kubernetes-vs-docker/ (visited on 05/07/2020).

[46] *Kubernetes vs. Docker*. URL: https://azure.microsoft.com/en-us/topic/kubernetes-vs-docker/ (visited on 04/12/2020).

[47] Bikram Bam. “Research as code: Instrumenting scientific computing as executable containers”. MA thesis. 2018.

[48] Kyrre Begnum. “Simplified cloud-oriented virtual machine management with MLN”. In: *The Journal of Supercomputing* 61.2 (2012), pp. 251–266.

[49] K Begnum and Matthew Disney. “Scalable deployment and configuration of high-performance virtual clusters”. In: *CISE/CGCS* (2006).

[50] Zhiqing Tang et al. “Migration modeling and learning algorithms for containers in fog computing”. In: *IEEE Transactions on Services Computing* 12.5 (2018), pp. 712–725.

Appendices

Appendix A

PraaS.py

```
#!/usr/bin/env python
# encoding=utf-8
import json
import argparse
import sys

VERBOSE = 0
state = {}

# parser arguments.
parser = argparse.ArgumentParser(prog='PraaS.py')
parser.add_argument('-v','--verbose',dest="verbose", help='Turn verbosity on', default=False, action="store_true")
parser.add_argument('-p','--print', dest="print_state", help='Print the current state', default=False, action="store_true")
parser.add_argument('-R','--register',dest="register", help='Register a new container', default=False,action="store_true")
parser.add_argument('-f','--file', dest="statefile", help='File containing state')
parser.add_argument('-j','--json', dest="json", help='Container information as JSON')
parser.add_argument('-C','--container', dest="container", help='Container to deploy')
parser.add_argument('-B','--balloon', dest="balloon", help='Balloon to node n')
parser.add_argument('-F','--fog', dest="fog", help='Fog to node n')
parser.add_argument('-n','--node' ,dest="node", help='Node')
parser.add_argument('-d','--depth', dest="depth", help='Depth ( used when using Balloon strategy)')

arguments = parser.parse_args()
VERBOSE = arguments.verbose

```

```
26 # JSON FORMAT
27 # {
28 #    "nodelist" : [
29 #     {
30 #       "name" : "N1",
31 #       "children" : [ "N10","N11" ],
32 #       "households" : [ "h1","h2","h3","h4" ],
33 #       "capacity_mem" : "4096",
34 #       "capacity_mem_used" : "1280",
35 #       "capacity_cpu" : "8",
36 #       "capacity_cpu_used" : "1",
37 #       "containers" : [ "c1","c2","c3","c4","c5" ],
38 #       "parent" : "N0",
39 #       "leaf" : "false"
40 #       "depth" : "1"
41 #     }
42 #     ]
43 # }
44
45 # function to use verbose.
46 # When verbose is called like this -> verbose("some text")
47 # this function is called.
48 # If "-v" is added as an argument when running this script
49 # "VERBOSE" followed by the text
50 def verbose(text):
51     if VERBOSE:
52        print "VERBOSE: " + text
53
54 # This function reads the JSON file sent as an
55 # argument. This function is used
56 # to make the JSON file readable for the other
57 # functions.
58 def read_state_from_file(f):
59     verbose("read_state_from_file\n")
60     with open(f) as json_file:
61         state = json.load(json_file)
62         return state
63
64 # get_container(container_name):
65 # function for to get hold of a container name
66 def get_container(container_name):
67     # looping through containerlist
68     for container in state["containerlist"]:
69         # checking if the container is in containerlist
70         if container["name"] == container_name:
71             return container # returning container
72
73 # deploy_container_at_node(container,node):
```

```
74  # This function deploys a container as
75  # long as ther are enough RAM and CPU
76  # available at a the node.
77  def deploy_container_at_node(container,node):
78      verbose("deploy_container_at_node called for container " + container +
        ↪  " on node " + str(node))
79      # using get_container to obtain the container
80      new_container = get_container(container)
81      # obtaining the memory and CPU used by the container
82      container_memory = new_container["resource_mem"]
83      container_cpu  = new_container["resource_cpu"]
84
85      verbose( "Checking if " +
        ↪  str(int(state['nodelist'][node]['capacity_mem']) -
        ↪  int(state['nodelist'][node]['capacity_mem_used'])) + " > " +
        ↪  str(container_memory) + " and " +
        ↪  str(int(state['nodelist'][node]['capacity_cpu']) -
        ↪  int(state['nodelist'][node]['capacity_cpu_used'])) + " > " +
        ↪  str(container_cpu))
86      # check if sufficient mem and cpu
87      if int(int(state['nodelist'][node]['capacity_mem']) -
        ↪  int(state['nodelist'][node]['capacity_mem_used'])) >
        ↪  int(container_memory) and
        ↪  float(float(state['nodelist'][node]['capacity_cpu']) -
        ↪  float(state['nodelist'][node]['capacity_cpu_used'])) >
        ↪  float(container_cpu):
88          # We have sufficient memory AND CPU to deploy
89          verbose("We are OK to deploy here\n")
90          # Adjusting memory and CPU at the node according
91          # to what the contianer requires.
92          state['nodelist'][node]['capacity_mem_used'] =
            ↪  int(state['nodelist'][node]['capacity_mem_used']) +
            ↪  int(container_memory)
93          state['nodelist'][node]['capacity_cpu_used'] =
            ↪  float(state['nodelist'][node]['capacity_cpu_used']) +
            ↪  float(container_cpu)
94          # Appending container to nodelist
95          state['nodelist'][node]['containers'].append(container)
96
97          return 0
98      else :
99          return 1
100
101 # increase_baloon_to_depth(container,depth):
102 # Function for increasing the balloon.
103 # When increasing, container will not be deleted
104 # from the previous depth
105 def increase_balloon_to_depth(container,depth):
```

```
106     verbose("increase_balloon_to_node called")
107     # getting depth of parameter node
108     for d in state['nodelist']: # looping through nodelist
109         verbose("Investigating node: " + d['name'] + "\n")
110         verbose( "Depth: " + d['depth'] + " target depth: " + depth +
        ↪  "\n")
111         # checking if node depth is larger than 0
112         # and
113         # if nodedepth is equal or less than the argument depth
114         if int(d['depth']) > 0 and int(d['depth']) <= int(depth):
115             verbose("This node has depth -> " + d['depth'] + ", we need to
            ↪  deploy here\n")
116             # checking if there are any containers at the node at all
117             if d['containers'] == []:
118                 # deploying container if there arn't any containers
                ↪  present at the node
119                 deploy_container_at_node(container,
                ↪  int(get_node_index(d['name'])))
120             else:
121                 # using a found variabel to tell if the container is
                ↪  present when
122                 # there are containers at the node. This way we eliminate
123                 # the risk of duplicates.
124                 found = 0
125                 for cont in d['containers']: # looping through containers
                ↪  at node.
126                     if container == cont:
127                         # if the container is present, found -> 1
128                         found = 1
129                 if not found:
130                     # container is not found/present. Found -> 0 and we
                    ↪  deploy container
131                     deploy_container_at_node(container,
                    ↪  int(get_node_index(d['name'])))
132         else:
133             verbose("We need to make sure it's not here")
134
135 # find node depth is used to find the depth of a node
136 def find_node_depth(node):
137     verbose("find_node_depth called")
138     # depth = depth / number of hops away from root/NO.
139     # use argument node to index nodelist and find
140     # the depth of the node in question.
141     depth = state['nodelist'][int(node)]['depth']
142     return depth # returns depth
143
144 # get_node_index
145 def get_node_index(node):
```

```
146     # 1: is used to exclude the "N"
147     # for example: to get 13 out of N13.
148     return node[1:]
149 
150 # increase_fog_to_node(container,node):
151 # recursive function which starts on the node
152 # in question and traces back to N0.
153 def increase_fog_to_node(container,node):
154     verbose("incrase_fog_to_node called")
155     # step0: if root node , return.
156     if node == 0:
157         # code for returning none
158         verbose("We are at the root node and the container is already
            ↪  running there\n")
159         return
160 
161     found = 0
162     # step1: deplay container at node, if not already there.
163     # looping through containers in the arguemnt node
164     for cont in state['nodelist'][node]['containers']:
165         verbose("cont in " + cont )
166         if container == cont: # checing if container is present
167            #code for returning container already at node
168            verbose("Container is already running here, nothing to do.\n")
169            # found -> 1, because container is present
170            found = 1
171 
172     # if found = 0 -> container not present at node -> deploy!
173     if not found:
174         #call deploy container to node
175         deploy_container_at_node(container,node)
176 
177     # step2: call function: increase fog to node (container,parent).
178     if node != 0:
179         increase_fog_to_node(container
            ↪  ,int(get_node_index(state['nodelist'][int(node)]['parent'])))
180 
181     verbose("Deploy on node N has finshed")
182 
183 # This functions registers a container.
184 # the container argument is appended
185 # to containerlist.
186 def register_container(container):
187     state['containerlist'].append(container)
188 
189 #def print_state(state):
190 # this method is only to print the state
191 # no functions except printing.
```

```
192 def print_state(state):
193     verbose("print_state called")
194     # prints the state which is a json file
195     print json.dumps(state,sort_keys=True, indent=4)
196 
197 #
198 ## This is where the program actually starts
199 #
200 # this is to check if a json-file is sent as
201 # a file-argumentthe when running the script.
202 if arguments.statefile:
203     state = read_state_from_file(arguments.statefile)
204 else:
205     print "You have to specify a state file to read from"
206     exit(1) # exits the program if file-argument not is present
207 # -p as arguemnt
208 if arguments.print_state:
209     # we want to read the state from a file and subsequently print it out
210     print_state(state)
211 # -R and -f as arguments
212 elif arguments.register and arguments.json :
213     # we need to register a new type of container
214     # This means adding it to the general list of containers AND
215     # deploying it at the root node
216     container = json.loads(arguments.json)
217     register_container(container)
218     deploy_container_at_node(container["name"],0)
219     print_state(state)
220 elif arguments.container and arguments.node:
221     # we want to deploy a container at a specific location
222     deploy_container_at_node(arguments.container,
        ↪  int(get_node_index(arguments.node)))
223     print_state(state)
224 # -F and -n as arguments to increase a fog
225 elif arguments.fog and arguments.node:
226     increase_fog_to_node(arguments.fog,
        ↪  int(get_node_index(arguments.node)))
227     print_state(state)
228 # -B and -n as arguments to increase a balloon
229 elif arguments.balloon and arguments.depth:
230     increase_balloon_to_depth(arguments.balloon,arguments.depth)
231     print_state(state)
232 else:
233     print arguments.container
234     print arguments.node
```