Ατομική Διπλωματική Εργασία

CHARACTERIZATION OF SYSTEM BEHAVIOR USING PERFORMANCE COUNTERS

Anatoliy Atanasov

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

CHARACTERIZATION OF SYSTEM BEHAVIOR USING PERFORMANCE COUNTERS

Anatoliy Atanasov

Επιβλέπων Καθηγητές

Γιάννος Σαζεΐδης

Χάρης Βώλος

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2022

Acknowledgements

I would like to thank all the people that helped make this thesis project happen by giving their constant support and effort. First a special thank you to my supervisors Mr. Yanos Sazeides and Mr. Haris Volos for their constant help and push week in week out to keep going. Their weekly feedback made me learn many things about a different way of thinking and overall different professional techniques used to make a person stand out. Also, a special thanks to Georgia Antoniou, a PhD student at the University of Cyprus, who was always available when help was needed irrelevant of the time and persistently explored various encountered issues until they were resolved.

Abstract

Cloud computing and generally the use of the cloud for designing systems has gained rapid popularity in the recent years. Specifically, the use of microservices changes the way we perceive applications and cloud services.

In this thesis I explore and learn what a microservice is as well as its different surrounding technologies that are needed for it to function fully.

Research around this topic was done by the team of SAIL group at Cornell University and was shown at the ASPLOS 2019 Conference. In this thesis I deployed one of the five end-to-end services, they created, named Social Network to gain an understanding of how a whole system using microservices is designed and its principles. During this deep dive into the application, I mention the setup needed to deploy the benchmark as it was described, the configuration, the inner workings, the errors faced along the way, and how to fix them. After that I discuss the results of the experiments ran to determine the communication overhead between the microservices and show how the overhead can prove a dramatic factor in latency resulting in more nodes not giving improved latency. In addition to that I analysed the bottleneck of "dropped" queries and showed how after some number of target requests per second the latency has a massive increase and queries are not being completed. The goal is the deep understanding of an end-to-end service implemented using Microservices.

Contents

Chapter 1 Introduction

1.1 Problem

When presented with the task to create a server-side application the traditional monolithic architecture is what comes to mind first, with the User Interface the Business Layer and the Data Interface in one block. This method is simple to develop, deploy, test and scale [1]. But as the application grows when getting updates this approach can become a burden. It can become too complex and too large for developers to be able to comprehend and then find and fix errors.

In addition, as mentioned monolithic applications need to be looked upon, that means that even when a small portion of the application needs to be fixed or tested the whole application needs to be deployed each run [1]. Hence why a new architecture may be needed that deals with those challenges. One such architecture, that is gaining rapid attention in recent years, is the microservices architecture.

Microservices architecture is a new way of application development, which creates a wave of people with mixed feelings about it. To properly evaluate the future and the possible impact of this architecture we must fully understand it. Understanding what microservices are, is a crucial first step. More specifically it is important to learn how a single microservice operates and how it is setup. But microservices cannot work alone they need tools to be able to operate fully and prove to be beneficial. When we gain an image of the singular Microservice and the tools it needs to run, we must move onto learning the function of a cluster of Microservices and their in between communication as well as the tools a cluster needs to operate.

The issue is that Microservices also create a lot of challenges. However, the benefits can prove to be fundamental making it impossible to discard the Microservices as an option for an architecture pattern. Hence why, researchers are attempting to determine what challenges

Microservices create, how to tackle these challenges or at least minimize their effects and overall, how to optimize the design of this architecture so that the positives of it outweigh the issues.
Our motivation for this thesis project is to explore the microservice architecture its implications and validate if there are benefits.

1.2 Methodology

We explore various sources such as websites, articles and papers to gain an understanding of Microservices and the different technologies that exist to support them. We show the differences between those technologies and why one would choose one over the other or a combination of them if possible. We then proceed to understand a specific application so that we learn how the chosen technologies are used to operate, manage, monitor and scale microservices and their communications. One of the many such researches is the Death Star Bench Suite [2] which is an Open-source benchmark suite for cloud microservices. In the Death Star Bench Suite there are five end-to-end services that combine many microservices each using a different set of technologies. We attempt to deploy and run one of those end-to-end services to see the microservice architecture in practice. More specifically we use the Social Network end-to-end service which uses RPCs for communication and Docker for orchestration. We deploy the benchmark both as a single container and as a cluster of containers.

1.3 Contributions

The contribution in this thesis work is focused on explaining what a Microservice is and what its surrounding technologies are, how they work with each other. After giving a general idea we showcase specifically the technologies used in the Social Network by doing a deep-down dive in the system. Meaning learning how the system works, showing the process of the deployment and discussing the issues encountered throughout as well as minor fixes that help to solve the temporal problems. Overall difficulties encountered during the configuration and how the configuration was done and showing the inner workings of the system. We evaluate if thread and/or connection increase proves to be beneficial for the latency, as well as giving a guide on how to validate that the deployment was performed correctly. We validate if the given scripts work as intended. We showcase how contrary to theoretical views using 20 nodes gives worse latency than 5 nodes.

1.4 Outline

First, we are going to explain with more detail how a Microservice architecture works and what are its surrounding technologies, and then what are the benefits as well as challenges of using this architecture. Then we will focus on the Death star Benchmark suite explaining their work and mentioning the various created end-to-end services. From those services we will pick one up (the Social Network) and focus entirely on it. Discuss how that system is implemented then how we attempted to deploy it and what work we did on it after its deployment. Finally, presenting the results gathered from experiments and conclusions made about the benchmark as well as microservices architecture overall.

Chapter 2 Microservice Architecture

Microservice Architecture is an architectural pattern where the distinct functionalities of a system are broken down into smaller pieces known as microservices. As we can see from figure 1 using microservices, the business logic and data access layer are divided into microservices, instead of being in one whole block as in Monolithic Architecture. Each microservice can have its own database that better suits the service. For example, in figure2a we can see how two services use SQL type of database and another service uses a NoSQL type. Also, the types maybe the same but the database schema may be different for each microservice. These pieces are interconnected and communicate with each other and with the client using an API Gateway.

We can see from figure 2a how the client does not communicate directly with the microservices but rather through the API gateway. Two popular API gateways that are often used in combination with microservices are REST API and RPCs, we will explain in detail each approach further on. A microservice can be implemented with different technologies and its own database that's why we put each microservice into containers. As we can see from figure 2b these containers need to be managed/orchestrated using a tool. Two popular tools are Docker and Kubernetes which we will also discuss in this chapter. [3] [4] [5] [6]

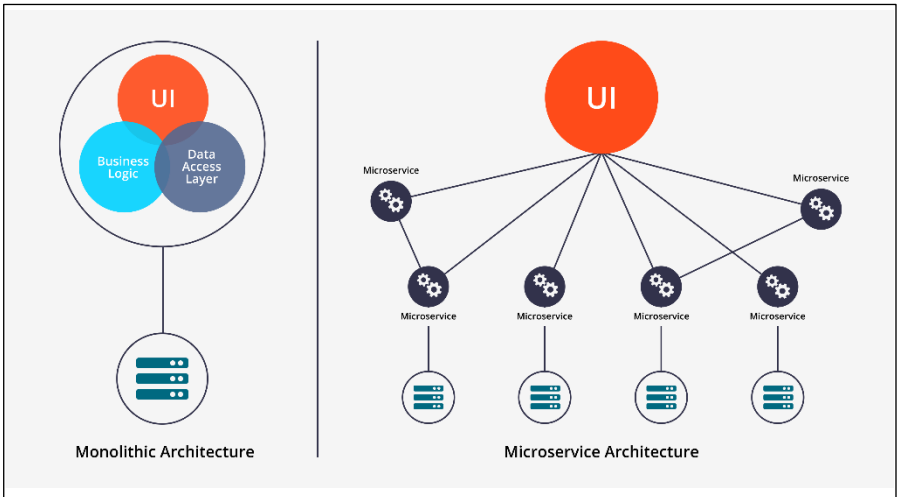

Figure 1. Differences between Monolithic and Microservice Architecture

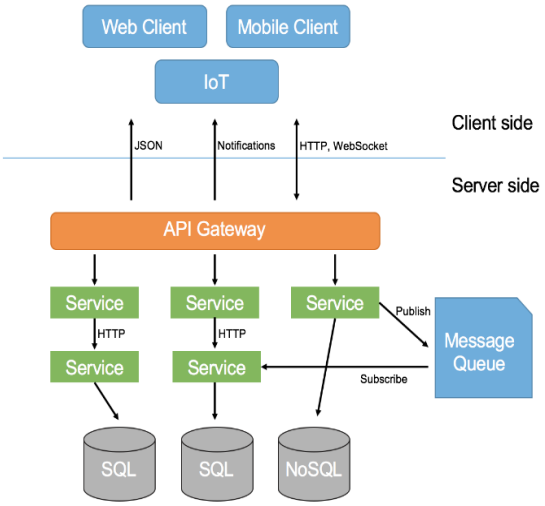

Figure 2a. Microservice Architecture showcase of API gateway to communicate with client.

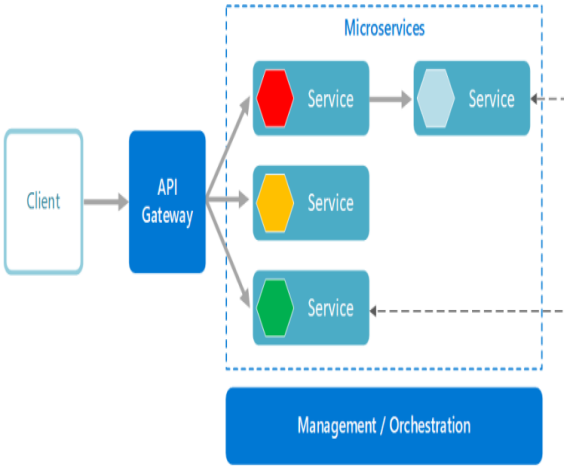

Figure 2b. Microservice Architecture showcase
Of API gateway and need for Orchestration of containers.

2.1 API Gateway

2.1.1 Remote Procedure Calls (RPC)/Apache Thrift

API gateway is needed to have communication between clients and microservices as clients send requests. In the case of microservices also used for the in between communication.

The RPC framework is used to enable developers to call a piece of code in a remote process in another machine or code in different process in the same machine.
Apache Thrift is a set of code-generation tools that allows developers to build RPC clients and servers by just defining the data types and service interfaces in a simple definition file. Apache Thrift was created with the idea that there is a need to have a framework that can be efficient and that works with any OS and any programming language. It adds flexibility for transports and protocols. For example, a server may be setup on a Windows machine written in C and offer a service over HTTP, that service can be accessed via a Linux machine from a program written in Python. And to do so the code is generated from the Thrift IDL file by writing the program logic and putting the different written pieces of code together, as well as configuring the data types needed. Meaning different pieces of code using different programming languages can be used for each microservice. When these pieces of code are imported into the definition file RPCs are generated. As we can see from the figure 3a the RPC services and user types are generated after the IDL file is compiled. [7]
This way each microservice can communicate with each other even if they are implemented with different technologies.

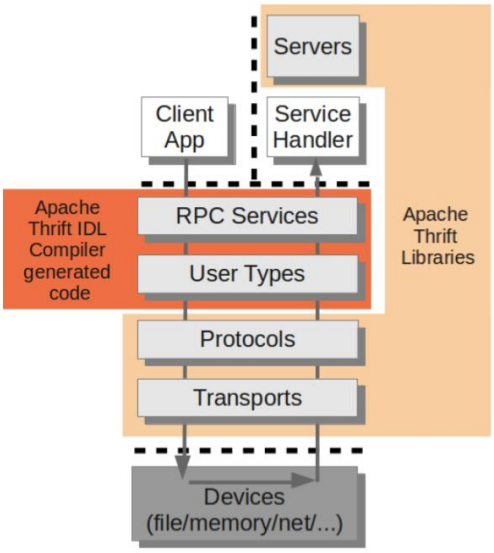

Figure 3a. Apache Thrift Description

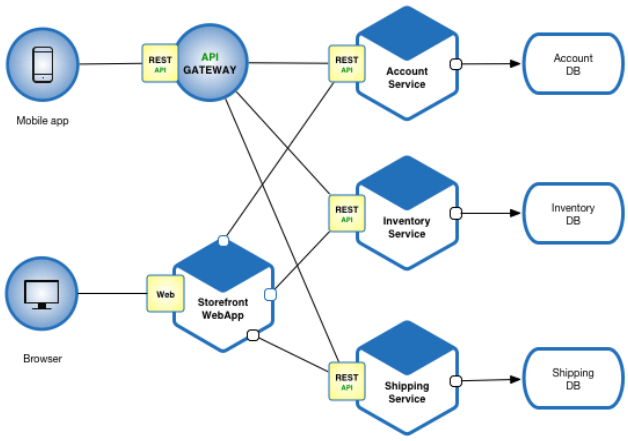

Figure 3b. API gateway configured with REST.

2.1.2 REST API

The REST API is a type of API that provides help for communication between web service applications. It usually uses HTTP and JSON as file transfer format. The REST API uses the

given client request to present back the response. The concept of REST is based on stateless server with the client storing the session states. Also, every file in REST is considered a resource. A service may produce a resource or consume a resource. With microservices the same applies, every microservice may consume or produce a resource which is how the microservices communicate with each and the client. For example, in the figure 3b the microservice named Account Service will produce a resource after a request happens from the webapp service. Then the webapp service will consume that resource and show in the browser the response in the predefined format (JSON, XML). Similarly, if one service requires a resource from another service it will just consume the produced resource. [8]

2.2 Container Orchestration tool

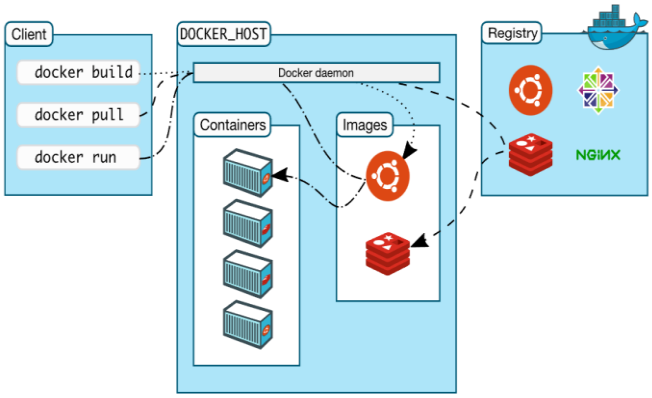

Figure 4. Examples of 3 docker commands in the docker architecture.

A container is a software that packs up different pieces of code with their dependencies so that the application can run efficiently and reliably. Multiple containers can run on the same system as each executes as a unique process. Two famous container orchestration tools are Docker and Kubernetes. It is important to note that it does not mean that you must choose between these two tools as they can work together too.

2.2.1 Docker Architecture

Has become the default orchestration tools used in recent years as it has a small learning curve and it's easier to install. Also, it's lightweight has auto-balancing and it is integrated with Docker CLI. On the other hand, it has limited functionalities, needs manual scaling and third-party tools for monitoring.

The Architecture of Docker consists of the client, remote or local, the docker daemon and the registry. The interaction of the user and the Docker is mainly in the Docker Client. Docker

Compose for example is a Docker Client. From the client the users run commands which will access the daemon, which builds, runs, and distributes the Docker containers it can also communicate with other daemons to manage containers. Lastly in the registry are stored the Docker images, which can be pulled. The HUB is a public registry that is accessible by default.

In Figure 4 we can see 3 examples of commands and how they will run. First docker build is sent from the client and accesses the daemon which builds the images. Docker pull accesses the daemon which accesses the registry to pull the images. Docker run access the daemon which runs the containers with the images. [9]

2.2.2 Kubernetes

Kubernetes is a more complex tool that provides more functionalities than docker. It supports auto scaling it has built in monitoring but needs to be set up manually for load balancing and needs a separate CLI tool.

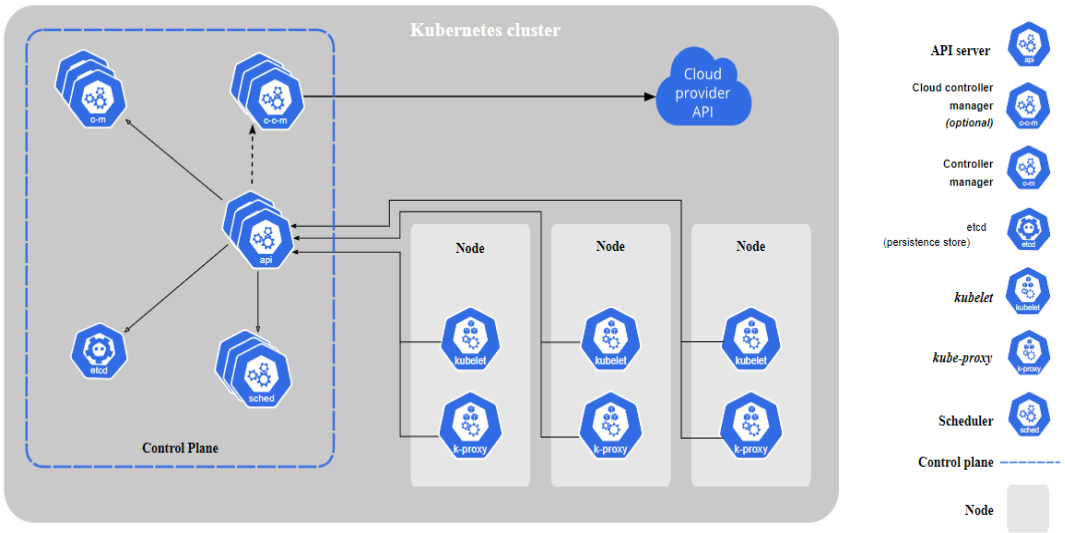

Figure 5. Kubernetes Architecture with its needed components to work.

When deployed Kubernetes creates a cluster. This cluster is made up from worker machines referred to as nodes. In each node there are Pods which are the running containers. The nodes are controlled through a control plane consisting of the API server, the etcd, which acts as backing store, the scheduler and the controller manager. The scheduler assigns newly created pods to nodes. [10]

In figure 5 we can see a diagram of the above showing how the different nodes communicate with the control plane in a Kubernetes cluster.

2.3 Where to deploy Microservices

There are different providers for nodes when you want to run a cluster of nodes. Some of them are AWS lambda, Microsoft Azure Serverless. With AWS lambda you can run your code with the need of servers or creating workload-aware cluster scaling logic, maintaining event integrations, or managing runtimes. Also, you can scale automatically your application and optimize the execution time by choosing the right memory size for your code.
Using Microsoft Azure Serverless you can create serverless applications without the need of infrastructure to run the code and the containers. You can orchestrate the deployment of containers using Kubernetes with Azure Kubernetes Service (AKS) and AKS virtual nodes. [11] [12]

2.4 Benefits of Microservices

A monolithic application can become over time very complex and not understandable. Using microservices architecture pattern can split the whole system to many modules, with each module having its own database and designated hardware. Also, these modules can scale independently and can be implemented using different programming languages. By using microservices legacy core components of a system can be modernized, because as mentioned these modules can be implemented with different programming languages [6]. Lastly, with microservices you can update and understand code easily making detection and fix of errors less of a burden.

2.5 Drawbacks of Microservices

Complexity of intercommunication between the microservices remains a problem as well as the overhead it causes. In addition, a problem in one service may cause problems in another service as sometimes tasks require many services to be accomplished. The overhead mentioned is mostly on network connections which may lead to latency problems and overall QOS violations. Finally, something to think about is that with microservices logging on the client node is not enough and logging for each microservice may be required to have a full visualization of the system’s performance.

However, these drawbacks don’t seem to affect users as during a survey made by IBM 56% of takers said they would likely adopt microservice architecture in their future applications. In the same survey 78% of respondents said they would like to invest either money and/or time on learning about microservices. [6]

Chapter 3 Death star Benchmark Suite

The Death star Bench Suite has certain design principles that were followed. The suite is representative as it uses popular open-source applications and frameworks like NGINX, Memcached, MongoDB, MySQL. These make the system more reliable as it is made up from tested and well-designed components with only a portion of the code being new. Most of that new code is about the Apache thrift RPCs and the http requests. The operations of the services are captured as end-to-end meaning from the moment a request is sent from the client until it returns to the client or reaches the database. The different end-to-end services are implemented using low-level and high-level languages showing the heterogeneity that can be achieved using microservices. The different created end-to-end services are mentioned and explained below. [2]

Service	Total New LoCs	Comm. Protocol	LoCs for RPC/REST Handwritten	LoCs for RPC/REST Autogen	Unique Microservices	Per-language LoC breakdown (end-to-end service)
Social Network	15,198	RPC	9,286	52,863	36	34% C, 23% C++, 18% Java, 7% node.js, 6% Python, 5% Scala, 3% PHP, 2% Javascript, 2% Go
Movie Reviewing	12,155	RPC	9,853	48,001	38	30% C, 21% C++, 20% Java, 10% PHP, 8% Scala, 5% node.js, 3% Python, 3% Javascript
E-commerce Website	16,194	REST RPC	4,798 2,658	- 12,085	41	21% Java, 16% C++, 15% C, 14% Go, 10% Javascript, 7% node.js, 5% Scala, 4% HTML, 3% Ruby
Banking System	13,876	RPC	4,757	31,156	34	29% C, 25% Javascript, 16% Java, 16% node.js, 11% C++, 3% Python
Swarm Cloud	11,283	REST RPC	2,610 4,614	- 21,574	25	36% C, 19% Java, 16% Javascript, 14% node.js, 13% C++, 2% Python
Swarm Edge	13,876	REST	4,757	-	21	29% C, 25% Javascript, 16% Java, 16% node.js, 11% C++, 3% Python

Figure 6. Different end-to-end services showing technologies used, number of lines of code and number of microservices

3.1 End-to-End Services

The social network is a released end-to-end service with follow and follow back connections between users, where these users can make different types of posts, see posts on their home from different other users they follow and access another user's profile to see its posts from there. To simulate the interaction between users registering, composing posts, following other users and reading posts there are created scripts. These scripts generate workload that simulate that. First a social graph is created simulating the registration of users equivalent to the nodes and follow connections between those users equivalent to the edges. The compose posts script creates random posts some may contain only text; some may contain images and/or videos. Reposting a post is the worst kind of query latency wise as it must find a post read it and append to it.

The E-commerce site which is still in progress is an online website for shopping of clothes. Like the open-source Sock shop app. Highest magnitude of request is placing an order as it is needed to add the item to the shopping cart and then login and confirm the payment. This type of request is a lot heavier than browsing the catalogue.
The other services that have similar behaviours like the e-commerce site are the media service, which is released, and it is a system to view movie information, give reviews and ratings to certain movies and rent or stream movies and the Banking System which is progress that is a service for secure banking that has certain methods to process transactions, request loans and balance credit cards. Lastly their Drone coordination system which is in progress is an alternative design of system. This system's microservices run both on cloud and on edge devices. The system's functionality is to coordinate drones so that they can avoid obstacles and process images.
As we can see from figure 6 Social Network uses RPC protocol for communication and has 56863 autogenerated lines of code using Apache Thrift, we notice overall how the other end-to-end services use either REST, RPC or a combination of the two for their communication protocol. We also notice that REST does not have autogenerated lines of code as it does not work with Apache Thrift the way RPC does and is code needs to be handwritten. Lastly the figure 6 shows the number of microservices used for each benchmark and which languages were used.

Chapter 4 The Social Network

The Social Network Service is one of the created end-to-end services mentioned above and is the service that this thesis focuses on. This system uses Docker for its container orchestration tool and Apache Thrift RPCs. There are 36 microservices in this system and each is in a different container. It is important to note that the version of the Social Network used in this work is that of the Social Network before 19th of January 2022. The Social Network Benchmark is still in development and some issues mentioned later on may have been already fixed in a newer version of the benchmark.

4.1 How it works

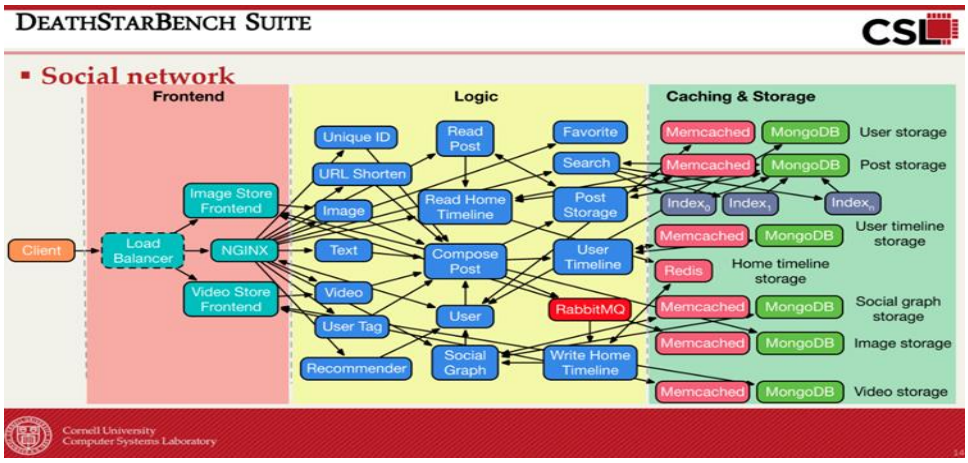

Figure 7a. Diagram of the frontend the logic and the backend of the Social Network.

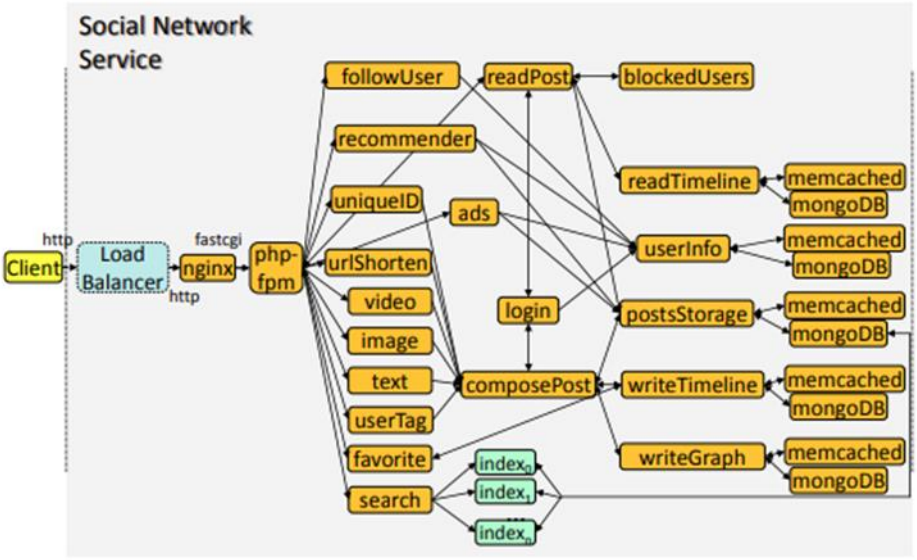

Figure 7b. Diagram of the frontend the logic and the backend of the Social Network.

In figure 7a we see the way the frontend, the logic and the backend are divided and which microservices are used for each part and the in-between connections of these parts. Figure 7b is the same diagram that shows the microservices and their connections, but it is a simpler form of figure 7a which makes sense as this figure was in the paper and the figure 7a was in the presentation of the paper with a few added details. Let us explain the figure 7b in detail. On the far left there is a client (the user of the system) that sends requests over HTTP protocol. These requests reach a load balancer which is implemented using Nginx. Nginx communicates with the microservices (on the right) using php-fpm module. The microservices communicate with each other using Apache Thrift RPCs. It is noticeable that certain microservices have a Memcached service and a MongoDB database attached to them. Almost every entity in these diagrams represents a microservice. In addition to those we have the Jaeger microservices which are used for distributed tracing and the Cassandra microservice which is a database where the traces are stored. There are three different microservices that function together to form the whole Jaeger function. All microservices are configured to depend on the Jaeger microservices as those microservices are needed so that tracing is achieved. Furthermore, those Jaeger microservices depend on the Cassandra microservice to be able to store the traces correctly. [2]

4.2 Functions

This end-to-end Service implements a social network where users are created, they can follow other users and other users can follow them, they can create posts with text, links, media, tags. Those posts can be liked, commented, favourite, report and shared. These posts can be seen in the home timeline where all the posts of the people you follow appear. Alternatively, by clicking on a user's account to see the posts of the specific user.
To simulate the creation of different users and the connection between those created users we create a social graph that is created using a certain dataset. That dataset has a number of nodes and a number of edges that connect those nodes. The nodes represent the registered users in the social graph and the edges are the follow connections between those registered users.
For these functions three different workload generator scripts were created: Compose-Post script, Read Home timeline script and Read user timeline script. Lastly, the mixed-workload script was created to combine the other three. [2]

4.3 Workload Generator Scripts

4.3.1 Compose-Post Script:

This script composes a random post for a random user. To do so first we randomly choose one of the nodes in the social graph meaning an existing registered user. In the current version of the Social Network this random value is hardcoded to be between 1-962 (962 being the nodes of the first dataset used), this can be a problem in the future when different datasets may be added to test the benchmark. However, in this thesis project that exact first dataset is used, as it will be mentioned below in more detail, so this does not affect us.
There are 2 sets to help compose this script. First there is a charset of characters ranging from A-Z and a-z and 0-9. Second, there is a number set with range of 0-9. The charset helps to create random text for the post also there is a random number to determine the number of mentions, urls and media on the post. The charset is also used to create the random url and the number set is used to create the random media ids.
This process is repeated for each request.

4.3.2 Read-Home Timeline Script:

This script is used to read random number of posts of a random user's home timeline. More specifically a random user out of the 962 (same issue with hardcoded number of users) is picked out and from the user's home timeline randomly 10 posts are read. To pick those 10 posts a random number from the range 0-100 is chosen and having that number as a starting point the next 10 posts are read. We can notice how this has the potential to lead to problems such as what if there are not 100 posts to be read and the random number is 90-100 what posts will be read.

4.3.3 Read-User Timeline Script:

This script is used to read random number of posts of a random user. This script works exactly like the Read-Home timeline script with the only difference being how it reads from the randomly chosen user's profile (the posts made by the user). Similar issues arise in this script as well.

4.3.4 Mixed-Workload Script

This script is used to run all the scripts above together but with a weight for each script. It is basically a combination of the three scripts in one with the difference that each time a request is called it will randomly, but with a weight, choose what script to execute.
There is 60% chance for the request to execute the Read-Home Timeline script, 30% for the request to execute the Compose-Post script and 10% for the Read-User Timeline Script.

4.4 Social Graph

As mentioned, to simulate the registration of the users and the follow connections between two registered users we create a Social Graph from a dataset. Initially the dataset used in the Social Network was only one and it contained 962 nodes and 18.8k edges. This dataset is also used to create the Social Graph in this thesis. However, this is the automated way for registering users in the social network, what if we add users manually through the frontend? To validate if the manually created users are inserted into the social graph, we will check the database of the social graph microservice before and after creating manually users.
We manually create 15 users using the frontend. As we can see from Figure 8 the increase in the number of objects corresponds to the number of manually created users. From this we can

conclude that those created users are added into the Social Graph. However, the scripts mentioned above will never reach these users and their posts as they are new and out of bounds.

Maybe a change should be made so that each time the social graph is checked for changes and that number is used for the upper bound of the random user id in the workload generator scripts.

Figure 8. On the left we have the database of the social graph before the creation of users and on the right, we have the database after the creation (Focus is on the number of objects)

Chapter 5 Deployments

In this Chapter we will discuss the two different deployments we have done. The first is on a local machine in the laboratory of the University of Cyprus. The second deployment is that of a swarm of nodes using hardware provided by Cloud Lab.

5.1 Deployment on Local Machine

We show the deployment of the local machine by first showing the specs of the machine then the needed installations as well as some problems that came up and how they were fixed. In the end we discuss firstly how to open and operate the frontend and secondly how to run the workload generator scripts and their results.

5.1.1 Specifications of the Local Machine

Architecture: x86_64CPU

CPU Number:4

CPU Type: Intel(R) Core (TM) i7-6700K CPU @ 4.00GHz

We disable all idle states with an equal or higher latency than 0 and enable all idle states with a latency lower than 0. [13]

We set the governor to be Performance mode.

5.1.2 Needed Installations

First, we download the social network code from GitHub. Then downloaded the required: Docker, Docker-compose, Python 3.5+, Libssl-dev, Libz-dev, Luarocks, Luasocket in that order. [14]
After that we can pull the docker images, create the social graph, build the files and start running scripts.
Needed ports: port 8080 for Nginx frontend, 8081 for media frontend and 16686 for Jaeger.
Before running the workload generator, we need to create a social graph which consists of the users of the system. The python script that is used to create the graph also creates follow connections between the created users. There are 3 different social graphs that can be used to run this application. First is the Soc-fb-Reed98 which has 962 nodes and 18.8k edges. Meaning there are 962 created users in the system and 18.8k follow connections between them. Next is the Ego-twitter with 81306 nodes and 1768149 edges. Last, is the largest social graph Soc-twitter-follows-mun with 465K nodes and 835.4K edges. The graph used for this research was the first of the three.
Last step before running scripts is the build on the wrk2 directory using make.

5.1.3 Problems and Fixes

After the first attempts to setup, we attempted to run the compose-post script to validate if the installation is correct. We encounter the problem that the local host could not connect because of an issue with Docker. The solution to this problem was to simply reinstall docker and docker-compose.
After fixing that issue another problem came up, when we again run the compose-post script. The issue was with the directory of luasocket as the script could not find the required socket. To fix this we run the command $eval $(luarocks path –bin) so that the required socket by the script has its directory defined and works properly. This command needed to be run each time before running the script. Also, we alter the compose-post script's first line of code to be local socket=require("socket"). The above to steps were later performed for all scripts (change of first line of code in each script and running the needed command before running each script).
After these fixes the deployment was corrected and the workload generator scripts were running, and the frontend was accessible. However, after an update of the repository in which the development had added two more graphs that create registered users and redis sharding

the deployment was showing problems. After attempts to update the project by downloading the new version of the Social Network again and resetting the docker containers the scripts did not produce results but rather outputted Internal Server Error. The decision was made to not use the updated version of Social Network but rather the previous one that was functioning.

5.1.4 How to run

To run experiments, we will use the workload generator scripts mentioned before. However, another approach is to access the Frontend and create manually a new user access his profile and start creating posts and adding follow connections. Note that accessing the Frontend successfully can also be an indication of correct initial setup. To access the Frontend its needed to access the localhost on the port 8080. When doing so the signup and login page are produced.

The options here are to either create new users manually and login or to access the http://localhost:8080/main.html page that is used to create default users and from there new posts can be composed and read.
We can create new post and login again to see that it can be read in the user profile to confirm correct functionality, this validation is performed in chapter 6.1.1.
As mentioned, another way to run is to use the workload generation scripts. We have explained what these scripts do. Let us now show how these scripts can be run in a command line client.
Workload Generator with Compose-Post Script
./wrk -D exp -t <num-threads> -c <num-conns> -d <duration> -L -s ./scripts/social-network/compose-post.lua http://localhost:8080/wrk2-api/post/compose -R <reqs-per-sec> [14]
In the example above the flags shown represent different things. Starting from the order we see them in the example let us explain these flags. -D is the distribution of time for each request, -t is the number of the threads which the generator will have, -c are for the total number of connections kept open, this number needs to be higher than the thread number as each thread must have at least one connection. Each thread will have connections equal to num-threads/num-conns. Next, we have the -d flag for the duration of the experiment, the -L flag which is optional and will present a histogram of latency, -s which determines which script will be run from the four mentioned before and finally -R which is the rate of requests

per second. Those flags are the ones used in every following experiment, but some other optional flags exist such as -P to print each request's latency, -p to print 99th latency every 0.2s to file, -H to add header to the request, -T to have a socker or request timeout, -B to measure latency of whole batches of pipelined ops (as opposed to each op) and -v to print the version details.

After running the workload generator, we see at the end some results regarding the average latency for the requests, 99th percentile latency, thread statistics, requests made, and data transferred. An example of the metrics and their values is presented in Figure 9 where we can see how we had a 60 second test of compose-post script with 2 threads, 2 connections and on average the results produced show an average of 10.6 ms latency and 21.53ms 99th percentile. With requests per second set to 24 and total requests done equal to 1452.

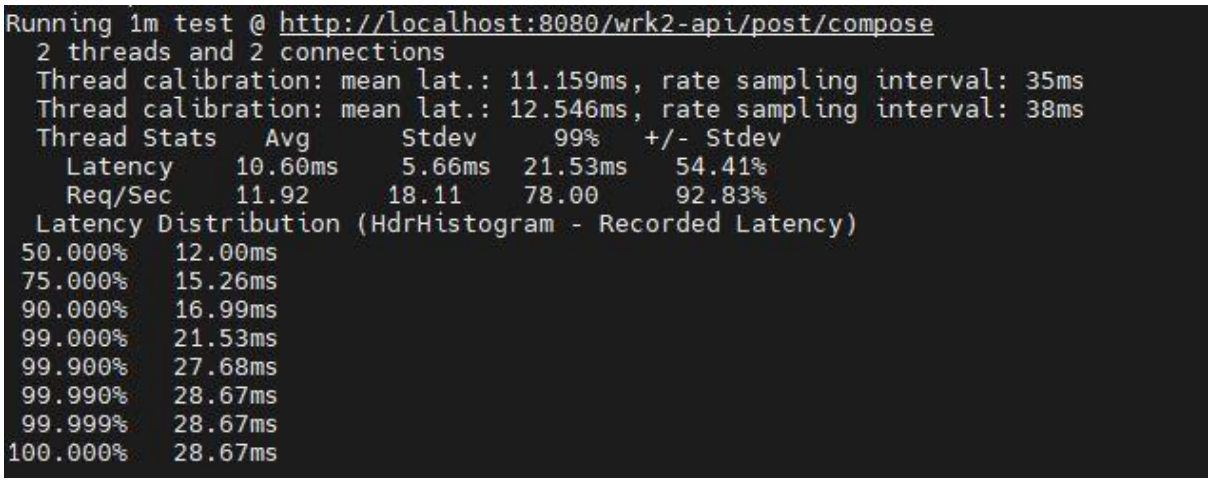

Figure 9. Example of Client Results after a run.

5.2 Deployment as a cluster on Cloudlab

5.2.1 Cloudlab

Cloudlab is a website that provides research groups with nodes to run experiments using the Cloud [15]. It has different functionalities as you can create certain profiles, create specific experiments and try out different architectures. In order to use Cloudlab you need access by requesting an account and being approved. After than you can input your own datasets and create profiles to use in your experiments. You reserve certain nodes for some set period and

utilize them. After the end of your experiments the data from the nodes is lost as they are refreshed. To access Cloudlab from a local ssh agent you will need to create a ssh key, have the key to the local agents in the .ssh file and upload the public key on to Cloudlab.
In our deployment we use the MobaXterm agent and have a profile that grants permission to move in between nodes.

5.2.2 Docker Swarm

Docker swarm is used to manage several docker engines. There is no need for an extra orchestration tool as the management of the swarm can be done through the Docker CLI. In docker swarm you have manager nodes and worker nodes. By default, manager nodes are also workers meaning they can contain containers, but they have authority for other tasks as well. Managers should be stable hosts and not many in number as they need to be available for the swarm to function. The process of creating a swarm is simple you just need to first create a manager node, initializing the swarm, the join to the swarm the different workers and/or other managers. The next step is to deploy the swarm using an yml file that mentions all the services, and their configuration, to be deployed. [16]

5.2.3 Specifications of machines

The machines that we use on Cloudlab are in the cluster of the University of Wisconsin. They are named c220g5 (Intel Skylake, 20 core, 2 disks) with Two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz, RAM that is 192GB ECC DDR4-2666 Memory. The two discs are 1 TB 7200 RPM 6G SAS HDs and Intel DC S3500 480 GB 6G SATA SSD. Lastly, the NICs are Dual-port Intel X520-DA2 10Gb NIC (PCIe v3.0, 8 lanes) and Onboard Intel i350 1Gb. [17]
The machines have disabled P-states and turbo boosting, this is performed by a script we created that is run after the nodes are allocated (Cloudlab experiment initiated).

5.2.3.1 C-states/P-states/Turbo boosting

Let us first mention Core power states (C-states) and performance states (P-states). C-state means one or more subsystems of the CPU is at idle, powered down whereas P-state means a subsystem is running but at a certain lower pair of frequency and voltage.
The states are numbered starting from zero like C0, C1… and P0, P1… The higher the number is the more power is saved.

C0: Active, CPU/Core is executing instructions (P-state relevant).
C1: Halt, nothing is being executed, but it can return to C0 instantaneously.
C1E: Like C1 but with lowest frequency and voltage operating point.
C6: Execution cores save their architectural state before removing core voltage. [18]
Turbo boosting is used when the maximum power a CPU can sustain is not reached. Turbo boosting increases the CPU frequency from the base frequency set if the maximum power threshold won't be surpassed or at least not for long. [19]

5.2.4 Setup

As mentioned, to run experiments first we create the swarm. In our experiments we have 1 manager node which is the first node of the swarm (node0) and one client node, which is not added as a worker up until we deploy the swarm, on the last node (node n-1 where n=total nodes). To do the above we created a script that adds the first node as a manager and the technologies needed which are Docker, Docker-compose, Python 3.5+, Libssl-dev, Libz-dev, Luarocks, Luasocket in that order. When we added the swarm manager, we are given a command with a token that is used from the workers to be able to join the swarm. We access all next nodes download on them the same technologies and add them to the swarm with the saved command and token [16]. After all nodes are connected, except the last one, we return to the manager node to deploy the swarm and create the defined services. It is important to note that it is crucial to be on the first manager node when deploying the swarm. Even if there are multiple instances of managers there is one that is above all, the one that started the initialization, and from which the deployment is done. But what does this deployment mean? The manager will read the yml file and create the services and allocate them evenly along the worker/manager nodes. The distribution of the containers is done using the round-robin method. That is by default it is also possible to have some restrictions in the yml file that manually allocate where certain microservices will be and other characteristics such as CPU limitations and replication of each microservice.
After that deployment is finished, we access the last node, the client. We add the client as worker to the swarm afterwards so that he has no microservices assigned but is connected to the swarm. On the client we again download and install all the needed technologies. Also, we create the social graph and build the dependencies of the code. We can now run the workload generator script from the client and get results after the runs for the client.

Chapter 6 Results

In this chapter we will discuss the different runs done on the two different sets of deployments and their results. Also, we will comment on those results as to if they were expected and what caused them.

6.1 Results of Local Machine Deployment

In this subsection we will discuss the results of the local machine deployment. Our first step is to determine whether the initial deployment was done correctly. We will access the frontend once and create a new user using the interface.

6.1.1 Frontend

When accessing the register interface as shown in figure 10, we must ensure that we enter a first name, a last name, a username and password to sign up. As we can see from figure 11. the home timeline has no posts as the newly created user has no followees. There are some

recommendations to follow and your frequent contacts that when the user is new are set to the default values. We can access the user's profile and see an about section and that the user has no posts created from figure 12. We can see that the frontend is accessible which is a good first step but to fully validate the setup we will create a post as the registered user and then try to access it using the login. In figure 13 we see the results of login and we validate that the login was successfully done, and the posts made by that user saved correctly.

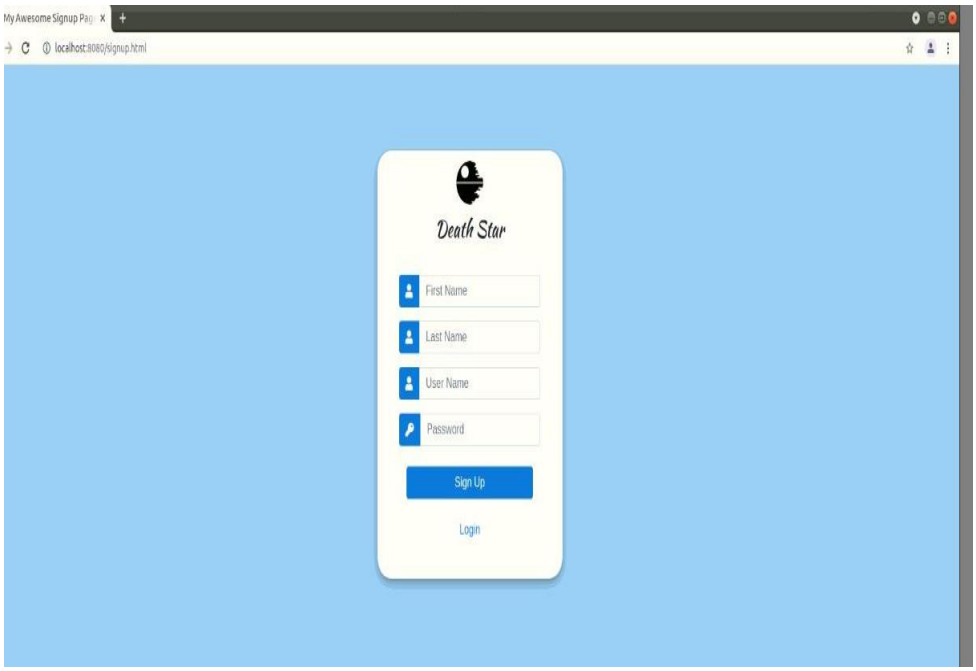

Figure 10. Social Network Registration interface

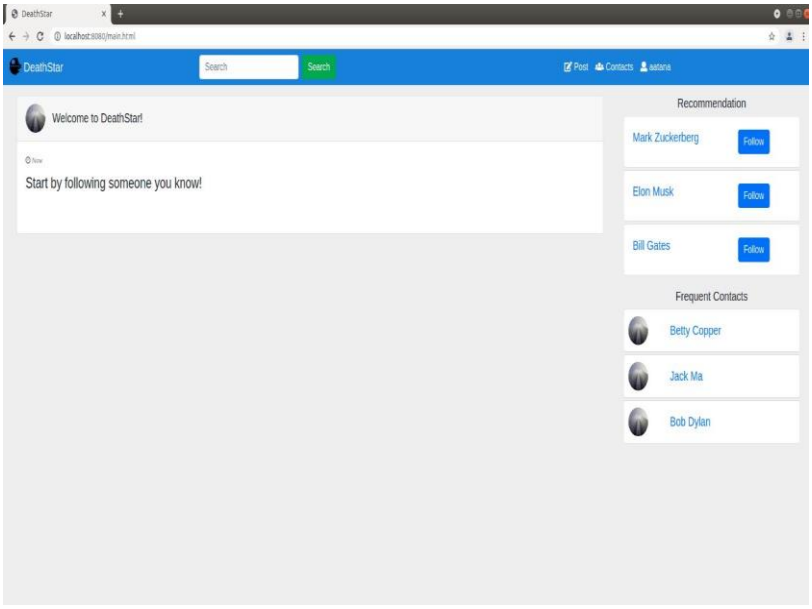

Figure 11. Social Network frontend home timeline

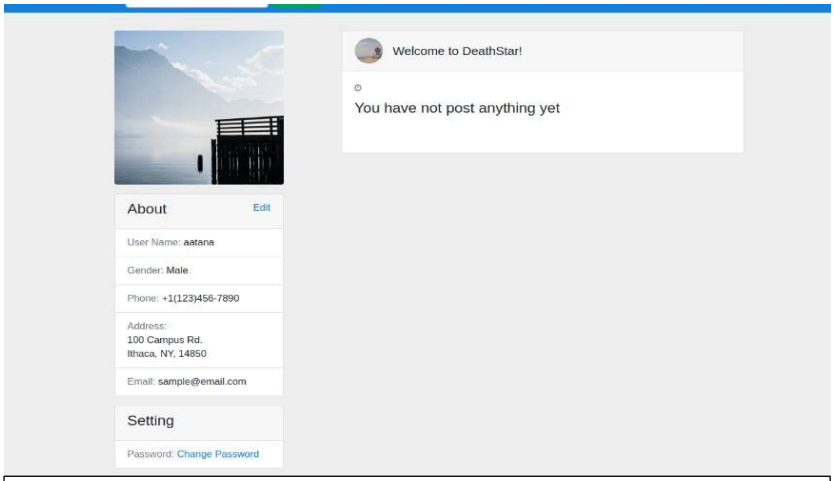

Figure 12. Social Network frontend user timeline

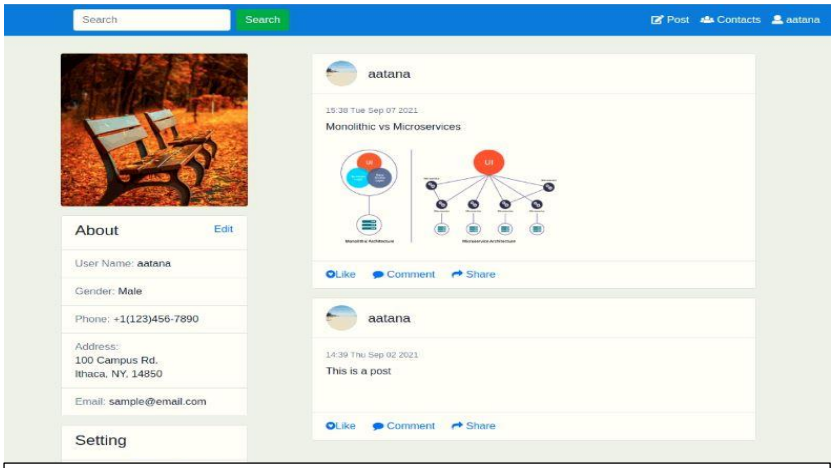

Figure 13. Social Network frontend user timeline after posts and login

6.1.2 Workload Generate Scripts

When running the workload generator there are various flags to give. Some of them are distribution (-D), connections (-c) and threads (-t). We can run different experiments changing one of these flags to see how it affects average latency and 99th percentile.

6.1.2.1 Checking Distribution

-D fixed, exp, zipf

We have the other flags fixed meaning we run for 100 seconds 100R/S with 4 threads and 8 connections the compose-post script each time. The command for this example would be -D exp -t 4 -c 8 -d 100 -L -s ./directory_script/compose-post.lua http://localhost:8080/wrk2-api/post/compose -R 100

We run the experiment 3 times with each -D value (fixed or exp or zipf) and get the following results:

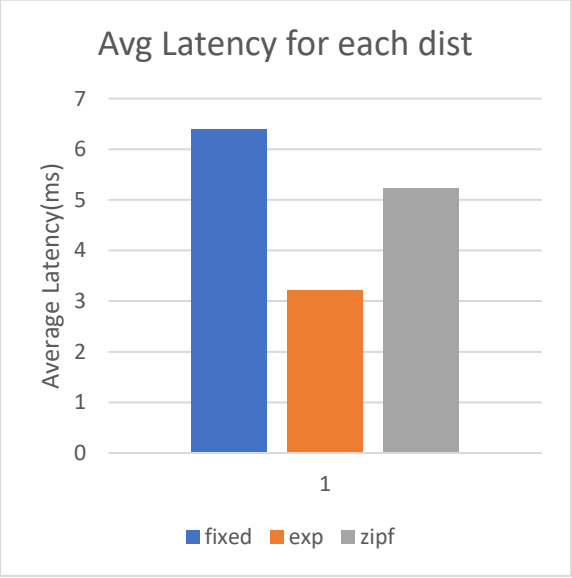

Figure 14a. Graph presenting the average latency for each distribution.

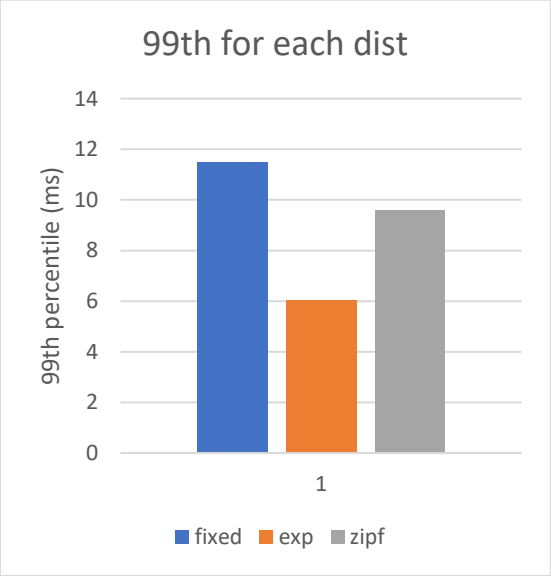

Figure 14b. Graph presenting the 99th percentile for each distribution.

We notice from figure 14a and 14b how exp has the lowest latency and the lowest 99th percentile out of the three distributions. This result may be an anomaly as exp is implemented to distribute by a random factor. A small note is that there is also a fourth distribution called normal which is not actually implemented although it is mentioned as an option.

For all future experiments exp distribution is used.

6.1.2.2 Checking threads and connections

The script doesn't allow us to have more connections than threads that is because each threads has several connections equal to threads/connections. Example if we have for the thread (-t) flag 4 and for the connections flag (-c) 4 each thread will have one connection to manage requests. Next, we will attempt to understand how increasing either the thread number or the connection number could potentially have any impact on the client latency.

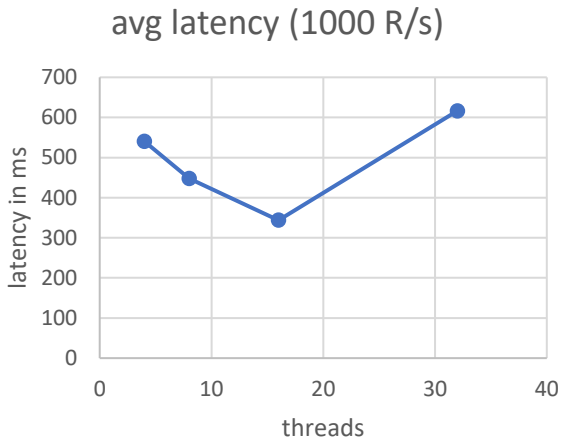

Figure 15a. Graph presenting the average latency for 4 8 16 and 32 threads Running 1000r/s for 10 seconds.

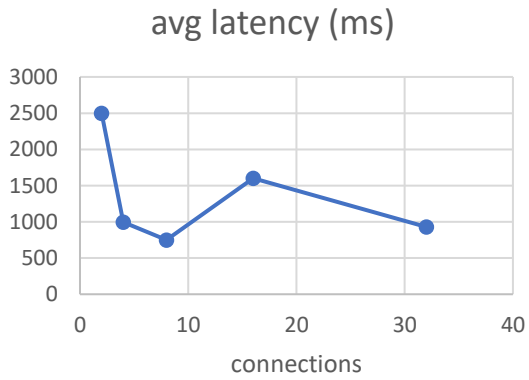

Figure 15b. Graph presenting the average latency for 4 8 16 and 32 connections Running 1000r/s for 10 seconds.

We run an experiment for 10 seconds with 1000 target requests per second (-R) running the compost-post script with 32 connections and changing the number of threads. We notice from that we have optimal latency time on 16 threads by looking at figure15a in which 16 threads has the lowest average time of 343ms.

After experimenting with threads, we move on with testing the connections. We have fixed 2 threads and we increase the number of connections running like previously 1000 requests over 10 seconds.

We notice that the optimal latency time comes with 8 connections by observing the results in figure 15b. Because we have the restriction that connections must be more than threads and because the optimal number of threads is 16, we can use 16 connections.

But from this experiment we noticed that the increase of threads and of the connections does not always have a positive outcome when it comes to latency.

6.1.3 Validation of execute threads

We can use the docker engine which has a stats command to have visualization of the containers and to analyse how the executing threads of different types of microservices scale [9].

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS	PIDS
dad9fd94df31	socialnetwork_social-graph-service_1	0.00%	2.215MiB / 7.514GiB	0.03%	9.77kB / 2.79kB	0B / 0B	3	5
31badfb45e21	socialnetwork_media-service_1	0.00%	1.922MiB / 7.514GiB	0.02%	6.22kB / 0B	426kB / 0B	2	4
496625fe02f8	socialnetwork_post-storage-service_1	0.00%	2.648MiB / 7.514GiB	0.03%	9.9kB / 2.77kB	295kB / 0B	3	11
922dd7ddcbaa	socialnetwork_user-service_1	0.00%	2.902MiB / 7.514GiB	0.04%	10.2kB / 2.83kB	508kB / 0B	3	127
1b382ae7d509	socialnetwork_user-timeline-service_1	0.00%	2.375MiB / 7.514GiB	0.03%	10.3kB / 2.98kB	393kB / 0B	3	5
8983b90540a5	socialnetwork_url-shorten-service_1	0.00%	2.629MiB / 7.514GiB	0.03%	9.78kB / 2.75kB	500kB / 0B	3	5
27b2b36209dc	socialnetwork_user-mention-service_1	0.00%	2.309MiB / 7.514GiB	0.03%	6.43kB / 0B	209kB / 0B	2	4
0d32da59983f	socialnetwork_compose-post-service_1	0.00%	1.672MiB / 7.514GiB	0.02%	6.4kB / 0B	8.19kB / 0B	2	2
5f595d3ccaf4	socialnetwork_text-service_1	0.00%	2.008MiB / 7.514GiB	0.03%	7.41kB / 0B	1.02MB / 0B	2	4
e3b7d7f45666	socialnetwork_home-timeline-service_1	0.00%	1.594MiB / 7.514GiB	0.02%	6.43kB / 0B	197kB / 0B	2	4
e994a8860454	socialnetwork_unique-id-service_1	0.00%	1.699MiB / 7.514GiB	0.02%	6.43kB / 0B	270kB / 0B	2	4
87431fe21049	socialnetwork_social-graph-mongodb_1	0.00%	166.6MiB / 7.514GiB	2.17%	8.86kB / 3.83kB	217kB / 311kB	35	280
9474ad8b9eee	socialnetwork_nginx-thrift_1	0.00%	8.895MiB / 7.514GiB	0.12%	6.78kB / 0B	8.54MB / 4.1kB	9	9
43a891890ab2	socialnetwork_media-frontend_1	0.00%	17MiB / 7.514GiB	0.22%	6.15kB / 0B	10.1MB / 4.1kB	17	17
11751ca79ec2	socialnetwork_home-timeline-redis_1	0.00%	1.305GiB / 7.514GiB	17.36%	6.27kB / 0B	97.4MB / 0B	5	5
5bebfb923447	socialnetwork_user-mongodb_1	0.00%	167.9MiB / 7.514GiB	2.18%	9.85kB / 3.89kB	13.4MB / 479kB	35	434
307d8fbbc9bd	socialnetwork_user-timeline-redis_1	0.00%	35.77MiB / 7.514GiB	0.46%	6.11kB / 0B	1.96MB / 0B	5	5
2e7c221f18e1	socialnetwork_user-timeline-mongodb_1	0.00%	167.3MiB / 7.514GiB	2.17%	9.93kB / 4.22kB	8.4MB / 483kB	35	46
991def139c4a	socialnetwork_media-mongodb_1	0.00%	208.1MiB / 7.514GiB	2.70%	6.99kB / 0B	55.3MB / 537kB	33	33
822b4fc9fe30	socialnetwork_post-storage-mongodb_1	0.00%	166.8MiB / 7.514GiB	2.17%	8.93kB / 3.83kB	197kB / 393kB	35	66
8529fc80f4a9	socialnetwork_url-shorten-mongodb_1	0.00%	167.3MiB / 7.514GiB	2.17%	9.3kB / 3.94kB	205kB / 475kB	35	42
75db1b11577e	socialnetwork_social-graph-redis_1	0.00%	5.09MiB / 7.514GiB	0.07%	6.18kB / 0B	729kB / 0B	5	5
5db0e6bd4a55	socialnetwork_post-storage-memcached_1	0.00%	2.348MiB / 7.514GiB	0.03%	6.85kB / 0B	0B / 0B	10	10
988597e78de0	socialnetwork_url-shorten-memcached_1	0.00%	2.344MiB / 7.514GiB	0.03%	6.11kB / 0B	0B / 0B	10	10
df033cab2ebb	socialnetwork_media-memcached_1	0.00%	2.453MiB / 7.514GiB	0.03%	7.21kB / 0B	2.19MB / 0B	10	10
fab585a786d8	socialnetwork_user-memcached_1	0.00%	2.281MiB / 7.514GiB	0.03%	6.85kB / 0B	0B / 0B	10	10
ee858e0827e4	socialnetwork_jaeger_1	0.01%	10.2MiB / 7.514GiB	0.13%	7.83kB / 0B	22.4MB / 0B	9	

Figure 16a. Initial State of docker containers stats where pids are the executing threads for each microservice.

Figure 16b. State of only PIDs after running experiments.

We can see the IDs, names, CPU usage, memory usage and pids in the docker stats. An example of sush showcase is in figure 16a where we see all the microservices and their statistics. Pids are the executing threads for each container and the metric on which we will focus. We can use this tool to see how scripts affect the executing threads. In figure 16a we have the initial state of the containers before running any scripts, in figure 16b we see how the pids change after running workload generator scripts. We can also notice that the executing threads of mongodb containers increase by a lot more than the rest of the containers. Every script combines a different number of microservices to complete its function (compose post uses all 31, read-home timeline uses 8 microservices, read-user timeline uses 7 microservices). Which could explain why for different runs of scripts some stats may change differently, for example when running compose post we expect the compose service container to have increased pids in comparison with the increase of pids if

we ran read-home timeline. However, by running various experiments we don't notice a pattern in which the threads increase.

6.1.4 Validation of Post

Furthermore, we can also examine the databases where posts are stored to detect any changes after we run the workload generator scripts. We want to validate that compose post creates posts which are stored in the post database, and which are equivalent to the number of total expected requests. By looking at figure 17a we can notice how we start with 321902 objects, and we run the Read-Home Timeline script to see if there is any change on the stats. The result is expected Read-Home timeline does not make changes on the post database as the objects remain 321902. We then run the Compose-post script and check again expecting now a change in the object number relevant to the requests we made. We confirm this by seeing how in figure 17b the objects reach 322775.

Figure 17a. Contains two results on the top we have a state of the post database and on the bottom the state of the database after running Read-Home timeline

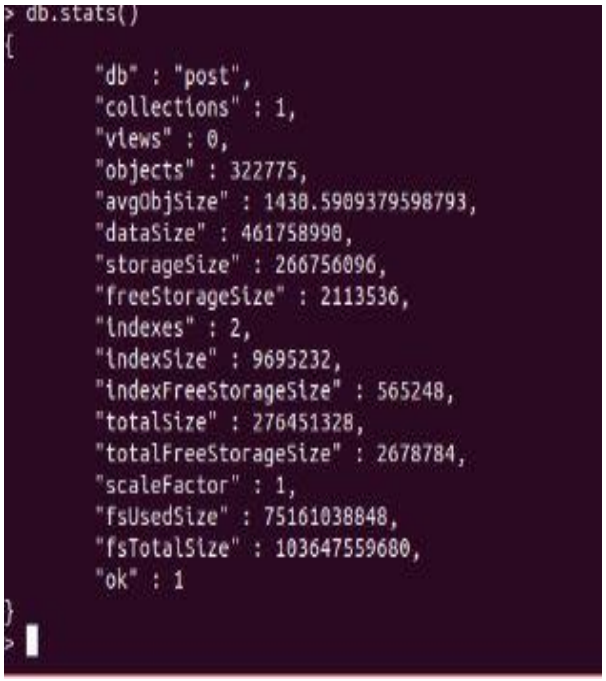

Figure 17b.State of Post database after running compose-post script

We run more tests to check if requests are equal to object increase. First, we try using 100 R/S for 10 seconds, then we run 10R/s for 100 seconds.

From figure 18 we deduce that if we run scripts as 10 R/S for 100 seconds, objects and requests are equal. To further examine why if we have more duration and less R/s obj=req but if the opposite the case is not the same. We attempt to run 200 R/s for 5 seconds and 1000 R/s for 1 second to see the difference. By consulting figure 18 we can see how in the experiments with duration of 1 second and 5 seconds the requests as well as the object are a lot less than the expected total requests but on 100s the number is equal. Why is this happening? It needs some time to initialize the process. This conclusion is also explained in the README file of the wrk2 which states “It's important to note that wrk2 extends the initial calibration period to 10 seconds (from wrk's 0.5 second), so runs shorter than 10-20 seconds may not present useful information” [14]. That can explain why our sometimes short in duration experiments didn’t manage to create enough requests on time as we were below the calibration period. We can also note that sometimes the reason of requests being less than objects is how requests need to return to the client and be traced but object accesses the database. There could be an inequality caused by the tracing not being completed and some request not been accounted for.

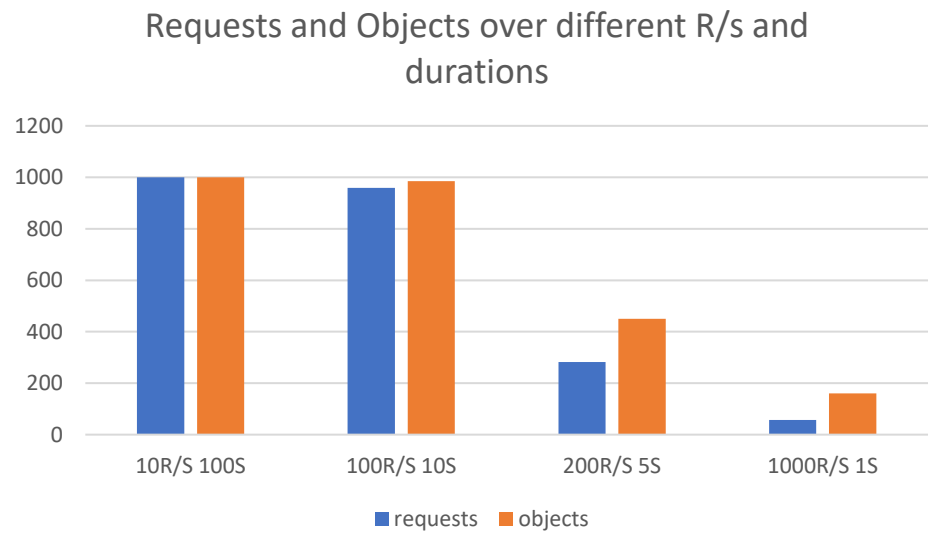

Figure 18. Show how requests and objects change when changing the R/s and the duration but by keeping the expected total query number the same.

6.2 Results of cluster of Machines Deployment

6.2.1 Initial Topology

We create a Cloudlab experiment with 20 machines. 19 machines containing microservices and one client machine. Mapping using Round-Robin around the nodes. We have 31 Microservices meaning each node will have two or one microservices assigned to it. Queries 100,200,300 with constant duration of 60seconds.

6.2.2 Initial Methodology and Results

We get the CPU usage for each node by accessing the node and getting the docker stats of the node for 2 seconds. Meaning each node is monitored for at least 2 seconds during the run of the compose postscript.

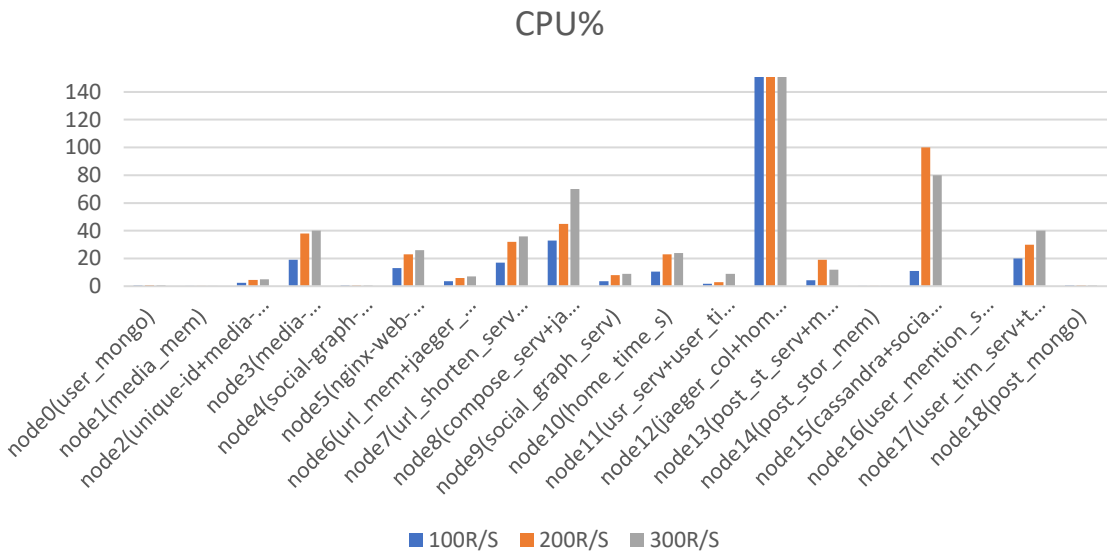

Figure 19. CPU usage across all nodes when running different R/s with fixed duration of 60 seconds

By observing figure 19 we notice very high percentage of CPU usage in the nodes consisting of Jaeger and Cassandra. Also, after those two next up is the compose service which makes sense as we are running the compose-post script. Jaeger's high utilization could be explained by the need to monitor and give tracing for every microservice, but not to such a high extent. Lastly, Cassandra is the database that saves the traces and since the jaeger client is not in the same node as Cassandra there could be networking overhead between the microservices. An

idea to be tested would be to check if forcing Cassandra and Jaeger into the same node reduces average latency.

Except the CPU usage we can examine the client's stats for different number of nodes.

In the following experiments we run the compose postscript for 60seconds on 5 nodes and on 20 nodes. Running 100 200 300 400 Requests each time. The two topologies are 4 nodes with the containers and one client and 19 nodes with containers and one client.

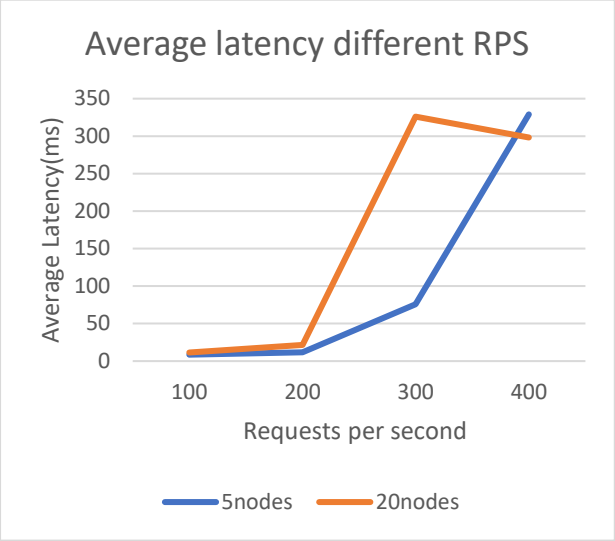

Figure 20a. Average Latency for different requests per second (100,200,300,400) for both 5 and 20 nodes.

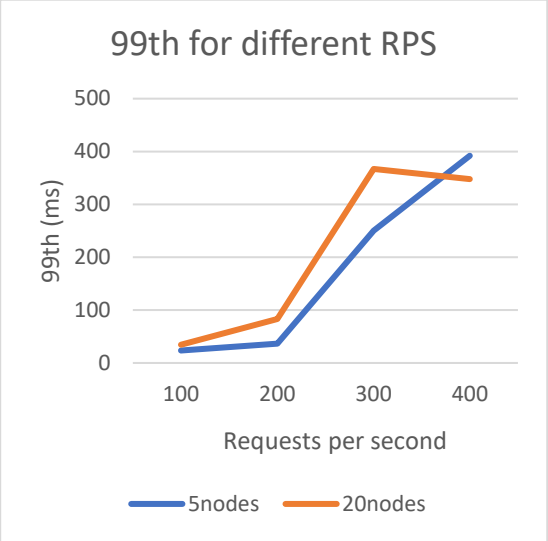

Figure 20b. 99th Percentile for different requests per second (100,200,300,400) for both 5 and 20 nodes.

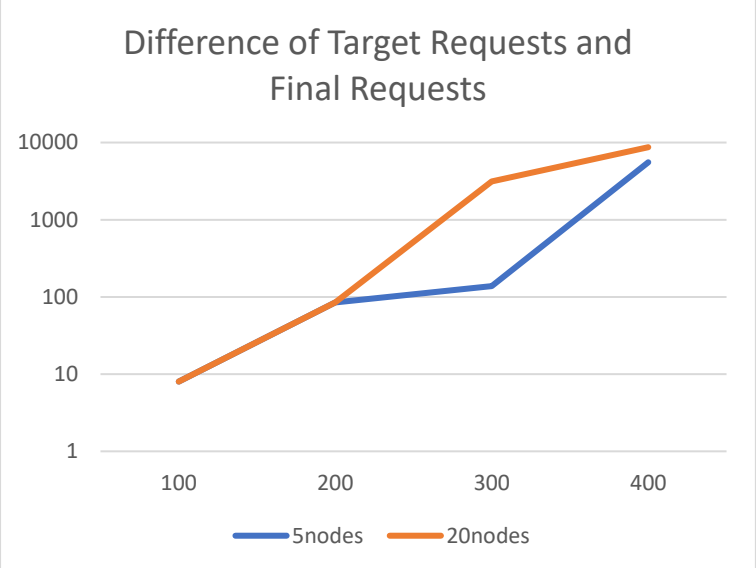

Figure 20c. Dropped Requests for different requests per second (100,200,300,400) for both 5 and 20 nodes.

We notice in figure 20a that during 300 Requests per second there is a big difference between the average latency using 5 nodes and 20 nodes with the 5 nodes having significantly lower average latency, more specifically the average latency for 5 nodes reaches 75.6 ms and for 20 nodes 326ms. That may be possible to be explained by the big communication overhead caused by having 20 nodes. Like the comment above the 99th percentile latency also is lower on 5nodes than 20 nodes in the 300requests per second as seen in figure 20b. However, after on the 400requests per second we see the times to be closer to each other, but to appreciate this we need to also acknowledge the difference in "dropped" requests. We define dropped requests as the difference between total target requests, QPS we input times the duration of the experiment, and final requests, the reported number of completed requests by the client. We notice the number of requests not completed which can be seen in figure 20c is abnormally high both for 5 nodes and for 20 nodes. That leads us to the decision to attempt and debug why so many requests are not completed.

6.2.3 Jaeger debugging

We attempt to debug the "dropped" requests through Jaeger interface. On the machines that we are running there is a trend for the "drops" to happen around 300 requests per second and more. When trying to find the missing requests we notice the limitations of Jaeger.

6.2.3.1 Jaeger limitations

Jaeger is a user interface and at the beginning of working with the interface we had some misconceptions about it. First, Jaeger does not show all the made requests from the experiment as there is a limitation on the UI to present only a maximum of 1500 requests. Another wrong idea was that Jaeger traces every request as well as the "dropped" ones. The tracer was setup to save the traces of 20% of the total requests.
However, Jaeger provides an interface to see the breakdown for each request of those 1500 with the time needed for each microservice as we can see in figure21a, where we have a request of the type home-timeline read and the nginx web server taking up 5.76 of the total 8.45 ms, home-timeline service taking up 2.43 and post storage service taking up 2.25 ms. In figure 21b we see how this breakdown can be presented differently using a table with statistic for how much each microservices takes up.

Figure 21a. Jaeger example of one request of the kind home-timeline read

Total	Avg	Min	Max
15.20ms	1.90ms	0.26ms	6.10ms
25.36ms	8.45ms	6.65ms	11.92ms
4.56ms	1.52ms	1.03ms	2.24ms
1.92ms	0.64ms	0.28ms	0.99ms
4.73ms	1.58ms	0.79ms	2.40ms
1.53ms	0.77ms	0.70ms	0.84ms
1.17ms	0.58ms	0.57ms	0.60ms
1.30ms	0.65ms	0.62ms	0.68ms
0.93ms	0.47ms	0.44ms	0.49ms
0.01ms	0.01ms	0.01ms	0.01ms
0.01ms	0.01ms	0.01ms	0.01ms
0.02ms	0.02ms	0.02ms	0.02ms

Figure 21b. Jaeger example of one request of the kind home-timeline read when selecting the produce its statistics.

6.2.3.2 Attempting to remove Jaeger

In a further attempt to debug the reason for “dropped” requests we attempted to remove the tracing tool from the benchmark but that proved to be unsuccessful. The approaches and ideas for removing jaeger were first to remove the jaeger and Cassandra services from the yml file so that they are not created to begin with. The issue with this was that the setup was causing problems sometimes and could not start correctly. Another idea was to deploy normally but kill the corresponding containers of Cassandra and jaeger after the deployment, this didn’t work as the containers are set to restart. The last idea was to change the configuration files for the sampling so that no samples are taken, this causes the data given at the end of the experiment from the client to be inaccurate as it is derived from the tracing. Concluding removing Jaeger may not be a viable option for the Social Network.

6.2.3.3 Changing Jaeger tracing

To have fully validate that we cannot use the jaeger interface to debug performance and where bottleneck may occur and to find the reason for the “dropped” requests we attempt to change the sampling rate of jaeger to 1 instead of 0.2 so that all requests are traced. We run an experiment of 400R/s for 60 seconds and as we can see from the figure 22b that run produces requests that need more that a second to complete, we then search in the Jaeger interface using the min duration filter with 1 second as we can see in the 22a figure, and no result is shown.

This proves that we cannot use Jaeger interface and if we want to continue debugging, we will need to access the Cassandra database.

While running various experiments we noticed something extra about the experiments some have something called non-2xx or 3xx responses. We run an experiment of 24R/s for 60seconds to have less than 1500 requests so that all fit in the jaeger interface. We then see in figure 23a how there are 6 such responses and by also observing figure 23b we see 5 requests that causes an internal server error and that also have time of 300ms and more which cannot be seen in figure 23a in the client average latency and 99th results. Also, these requests occur first in the experiment, there are first these 5 error requests and then all the rest work normally. Non-2xx or 3xx responses means that the code in the response given by http is not of the format 2_ _ or 3_ _ which makes sense here as a 500 response of internal server error is neither [20].

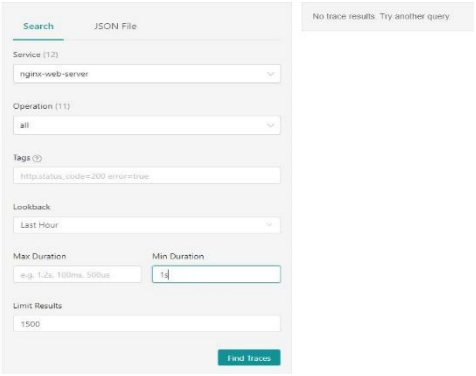

Figure 22a. Jaeger interface result of filter

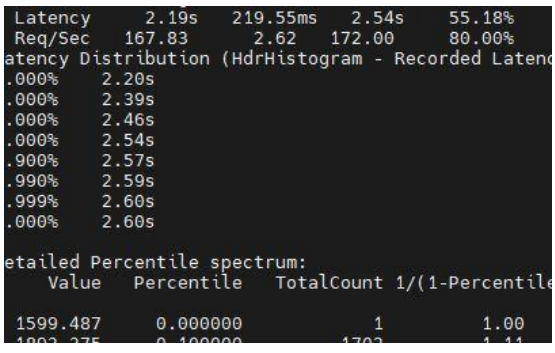

Figure 22b. Client experiment run results

```
Running 1m test @ http://localhost:8080/wrk2-api/post/compose
  2 threads and 2 connections
  Thread calibration: mean lat.: 11.159ms, rate sampling interval: 35ms
  Thread calibration: mean lat.: 12.546ms, rate sampling interval: 38ms
  Thread Stats   Avg      Stdev     99%   +/- Stdev
    Latency    10.60ms    5.66ms  21.53ms   54.41%
    Req/Sec    11.92     18.11    78.00     92.83%
  Latency Distribution (HdrHistogram - Recorded Latency)
 50.000%   12.00ms
 75.000%   15.26ms
 90.000%   16.99ms
 99.000%   21.53ms
 99.900%   27.68ms
 99.990%   28.67ms
 99.999%   28.67ms
100.000%   28.67ms

----------------------------------------------------------
  1452 requests in 1.00m, 10.86MB read
  Non-2xx or 3xx responses: 6
Requests/sec:     24.20
Transfer/sec:    185.35KB
```

Figure 23a. Client Result after run of 24R/s for 60 seconds

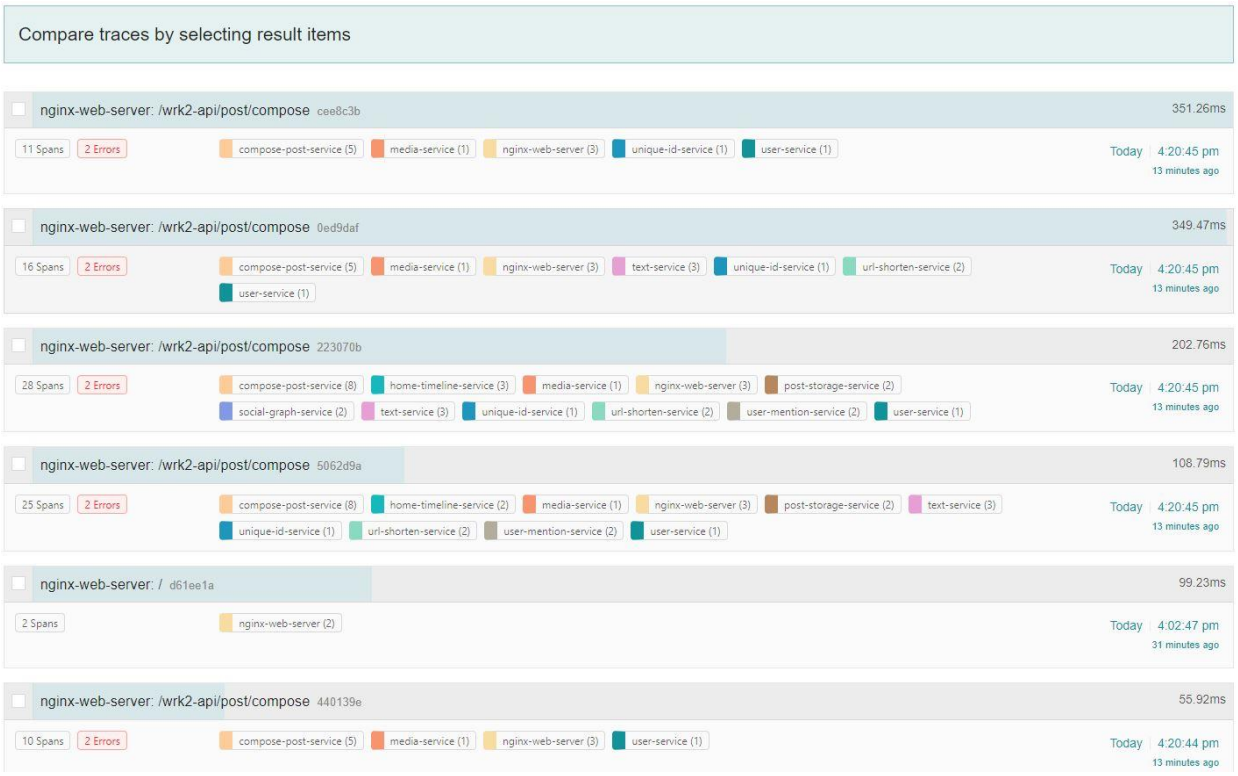

Figure 23b. Jaeger Result after sorting requests by longest time.

6.2.4 Cassandra

We access the Cassandra Database [21] to further analyse the possible bottleneck and "dropped" queries. We validate how the Cassandra table of traces changes when we run different scripts and if the "dropped" requests are captured in Cassandra. We run an experiment with 400R/s for 60s with expected total request number of 24000. However, we notice from figure 24a that the made requests are 20061, we then check with the Cassandra trace table. In the trace table there is saved for each request the time of each microservice. That is why we use a query on the database that gives us the count of the unique ids, meaning each time the newly made requests. The starting value of that count result is 120974 as seen from the figure 24b we then ran the described experiment, and the new count result is 141041 from figure 24c by subtracting the two we see that we have 20067 which in comparison to the 20061 value is very close. This can prove to us that the "dropped" requests meaning the expected 24000 requests are not stored in Cassandra.

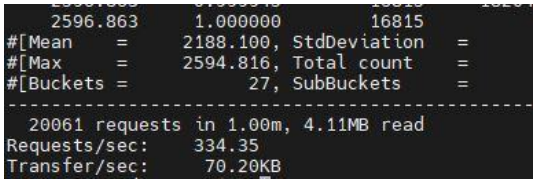

Figure 24a. Client Run number of requests is 20061

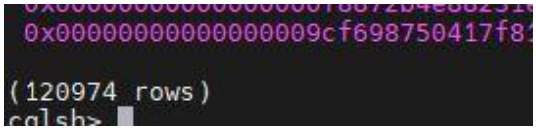

Figure 24b. Trace table unique ids before run

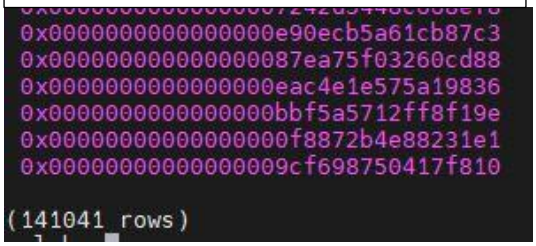

Figure 24c. Trace table unique ids after run

Chapter 7 Related Work

The paper named: “Microservices: Considerations before implementation” by Könönen [22] is a similar work as mine as it analyses what microservices are, the challenges of microservices and the positives as well, that work is also similar in the way that it is a thesis work for a bachelor’s degree. But that paper does not explain the whole microservice architecture and does not then implement a benchmark showing results something that is done in this thesis project.

Another paper is the “Performance Evaluation of Microservices Architectures using Containers” [23] in which the different ways to use containers are explored. They compare the performance of the CPU using master-slave containers and nested containers. In our thesis project only the master-slave containers configuration is used.

Lastly, we can compare our work with the work of the SAIL team at Cornell University on the Social Network. As we can see from figure 12 (in the Deathstar paper) in the graph of Social Network the SAIL team similarly to our runs used 100 200 300 400 QPS for their experiments and noticed big increases in latency after 200QPS. The difference is that in our work there was the factor of frequency change, we set the frequency to be fixed, but we found also that after 200QPS there are QOS violations that cause the tail latency to increase.

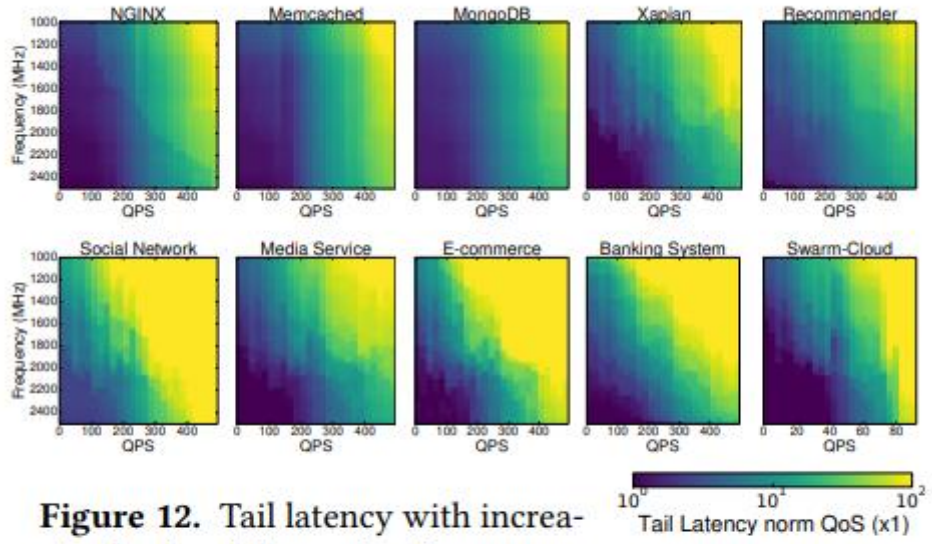

Figure 12. Tail latency with increasing load and decreasing frequency (RAPL) for traditional monolithic cloud applications, and the five end-to-end DeathStarBench services. Lighter colors (yellow) denote QoS violations.

Chapter 8 Conclusions

We can conclude that microservice architecture has a lot of different aspects to be taken in account and needs to be researched more but it has potential using the correct optimizations in the future. We found for example that 20 nodes perform worse than 5 nodes which is the opposite of the expected result for microservices.
Also, more specifically we found out how for the Social Network the Jaeger Interface may prove not very helpful in analysing all requests of an experiment, and if there are less than expected total requests, they will not be stored in the Cassandra database.

8.1 Future Work

We can use the Cassandra Database to further analyse the bottleneck of the network and the reason for the existence of "dropped" requests. Also, by using the database more we can find which microservices saturates and create a heatmap using the CRISP technology [24].
To characterize the performance overhead we can use the profiler [25] which will give us the CPU usage and transition states. To do the above we can use a topology which has 20 nodes for the containers one node for the client and another node for the jaeger and Cassandra microservices. The application for that could be that the mixed workload script is run and by keeping steady the total QPS and changing the time for example we could have 400R/s for 60 seconds 300R/s for 80 seconds 200R/s for 120 seconds and 100R/s for 240 seconds.

In addition, there could be a deployment of another end-to-end service from the Benchmark Suite that uses different technologies like Kubernetes and REST.
Important to note that the Social Network is in development so just deployment of a newer version can also be beneficial for research.

References

[1] H. Sanjaya, “monolith-vs-microservices-b3953650dfd,” 11 March 2020. [Online]. Available: https://medium.com/hengky-sanjaya-blog/monolith-vs-microservices-b3953650dfd.

[2] S. team, “An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems,” in *In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, New York, 2019.

[3] Chris Richardson , “introduction-to-microservices,” Eventuate, Inc., 19 May 2015. [Online]. Available: https://www.nginx.com/blog/introduction-to-microservices/. [Accessed 2022].

[4] C. Richardson, “https://microservices.io/,” 2021. [Online]. Available: https://microservices.io/patterns/microservices.html. [Accessed 2022].

[5] Wikipedia, “Microservices,” Wikipedia, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Microservices.

[6] IBM Cloud Education, “learn/microservices,” IBM, 30 March 2021. [Online]. Available: https://www.ibm.com/cloud/learn/microservices.

[7] A. A. a. M. K. Mark Slee, “Thrift Apache,” 2007. [Online]. Available: https://thrift.apache.org/static/files/thrift-20070401.pdf.

[8] P. Chandel, “The Software Design Blog,” 12 Septermber 2020. [Online]. Available: https://medium.com/the-software-design-blog/an-introduction-to-rest-apis-and-microservices-ea9477699b73.

[9] Docker Documentation, “Overview docker,” Docker, [Online]. Available: https://docs.docker.com/get-started/overview/.

[10] Kubernetes, “Kubernetes-io-components,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/.

[11] Amazon, “aws.amazon,” amazon, [Online]. Available: https://aws.amazon.com/lambda/.

[12] Microsoft, “Azure Microsoft serverless,” Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/solutions/serverless/#overview.

[13] debian, “linuxcpupower,” [Online]. Available: https://manpages.debian.org/testing/linux-cpupower/cpupower-idle-set.1.en.html.

[14] delimitrou, “SocialNetwork,” [Online]. Available: https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork.

[15] Cloudlab, “Cloudlab,” [Online]. Available: https://cloudlab.us/.

[16] Docker, “Docker docs swarm,” Docker, [Online]. Available: https://docs.docker.com/engine/swarm.

[17] Cloudlab, “Cloudlab Docs Hardware,” Cloudlab, [Online]. Available: https://docs.cloudlab.us/hardware.html.

[18] T. Krenn, “Processor P states and Cstates,” [Online]. Available: https://www.thomas-krenn.com/en/wiki/Processor_P-states_and_C-states.

[19] Intel, “Turbo boost,” Intel, [Online]. Available: https://www.intel.com/content/www/us/en/gaming/resources/turbo-boost.html.

[20] Nitrox, “Redme Boards,” 2010. [Online]. Available: https://redmine.lighttpd.net/boards/2/topics/4036.

[21] instaclustr, “Connect to cassandra using cqlsh,” [Online]. Available: https://www.instaclustr.com/support/documentation/cassandra/using-cassandra/connect-to-cassandra-using-cqlsh/.

[22] H. Könönen, “Microservices:Considerations before implementation,” 2020. [Online]. Available: https://aaltodoc.aalto.fi/bitstream/handle/123456789/32232/bachelor_K%c3%b6n%c3%b6nen_Heini_2018.pdf?sequence=1&isAllowed=y.

[23] IEEE, “ieeexplore.ieee.org,” 2015. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7371699.

[24] C. Z. a. M. K. R. Milind Chabbi, “CRISP: Critical Path Analysis for Microservice Architectures,” Uber, 18 Novermber 2021. [Online]. Available: https://eng.uber.com/crisp-critical-path-analysis-for-microservice-architectures/.

[25] H. Volos, “Profiler,” [Online]. Available: https://github.com/hvolos/profiler.