Ατομική Διπλωματική Εργασία

Cloud Based Microservices

Στυλιανός Βασιλείου

Επιβλέπων Καθηγητές
Γιάννος Σαζεΐδης
Χάρης Βώλος

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2022

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

Acknowledgements

I would like to express my heartfelt gratitude to all the individuals who played a crucial role in the successful completion of this thesis project. Their unwavering support and guidance have been instrumental in its realization.

First and foremost, I am deeply thankful to my supervisor, Mr. Yanos Sazeides, for his invaluable guidance and unwavering encouragement throughout the entire journey. His continuous assistance and constructive feedback have not only helped me overcome challenges but also enriched my understanding of different perspectives and professional techniques that set one apart.

Furthermore, I am deeply thankful to Mr. Haris Volos for his extensive knowledge in the field and his willingness to assist me whenever I encountered difficulties in comprehension or implementation. His assistance has been indispensable in successfully completing this work, and I am grateful for his contribution to its accomplishment.

I would also like to extend a special thanks to Georgia Antoniou, a PhD student at the University of Cyprus, whose constant availability and relentless efforts in resolving various issues and guiding me in more technical aspects of the thesis, have been immensely helpful.

The support and contributions of all these individuals have been indispensable, and I am truly grateful for their constant presence and assistance.

Abstract

In recent years, cloud computing, through the use of microservices and other technologies, has shaken the traditional development of an application. Many big companies have abandoned the rigid and inflexible monolithic approach for the more flexible microservices. However, even this novel technology has constraints and weaknesses.

In this Thesis, I explore one such application to deepen my understanding of such systems in general, how different parameters of the system they run on, as well as how they were set up, affect their performance.

The benchmark I will be using is one of the five end-to-end services developed as part of the DeathStarBench benchmark from the team of SAIL group at Cornell University [1]. I will be exploring its inner workings, how to deploy it, and the different configurations I will be using, both for the benchmark setup and the systems it will be running on.

Afterwards, I will discuss the results of the different experiments I run and compare them to derive how more nodes in a cluster and different topologies of the microservices affect the latency of the benchmark as a whole. Additionally, I will test how c-states, which were made to decrease a system's power consumption when not in use, actually affect the microservices.

Contents

Chapter 1. Introduction

1.1 Problem

As more and more app companies move from a monolithic approach of building their applications, to a microservice one, it is imperative to deepen our understanding of this new approach. As such, we will have to find out the capabilities of a system built on microservices, its limits and how to maximize its efficiency. Our aim will be to examine how different system configurations such as c-states affect the performance and the power consumption of a microservice. Specifically we will aim to firstly analyze and understand the functionality and the architecture of our microservice benchmark (DeathStarBench's social network). We use this understanding of the benchmark later on when we try to understand and explain the results of the experimental evaluation. For the second part of our analysis we

investigate the effects of the size of the cluster, its deployment configurations, and the c-states that are enabled on the performance and the power consumption of the benchmark. Before that though, we will have to explore a few different aspects that affect a system's performance, mainly for our thesis, the c-states. C-states give us an answer on saving power when a CPU is not being in use, by entering a sleeping state and switching off different parts of a core. Still, that comes with its own drawbacks that we will explain, in order to gain a good understanding on how they should affect a swarm of microservices. After having understood the workings of the c-states and the microservices, we will continue with running experiments in one such system.

1.2 Contributions

With this work here, I'm contributing by automating the setup process through the creation of different scripts, as well as the processing of the resault. The deployment scripts make it easy to set up the system to a predetermined baseline, aiming to minimize the number of variables involved. The processing algorithm I have developed a Python program takes the output of the Social Network and compiles it into easily comparable graphs. The program also provides the option to combine the output from multiple runs of the Social Network, facilitating the comparison of different setups.

Commented [1]: Maybe combine these and say something: Automate the deployment procedure of the application and the processing of the results

Furthermore, I investigate the impact of grouping or separating specific services within a microservice on the application's performance. I explore how keeping certain services together may enhance performance, while dividing them may have detrimental effects.

Moreover, I examine the influence of different levels of c-states on the application's performance, specifically focusing on how they can help optimize power consumption. consumption on different levels of workloads.

Commented [2]: I would also add something like provide insides into the functionality of the benchmark because you analyse the code before running experiments. This is also a contribution. Also put the contributions into bullets so that it is easier for the reader to follow

1.3 Outline

First though, in chapter 2, we will have to explain what c-states are, their usages, and effects, as well as how they work and, in chapter 3, what microservices are, their different

technologies, and why a lot of the big app companies nowadays prefer them over a monolithic approach. We will also, in chapter 4, have to understand the DeathStarBench benchmark, its aim for its creation, and its multiple benchmarks. Then we will delve deeper to gain a more intimate understanding of the Social network in particular. After we understand what the microservices are, as well as the benchmark we will be using, in chapter 5, we will explain the methodology of the experiments, such as what capabilities the servers we are running them on have, what technologies we will be using, and how to run the benchmark in a cluster of servers. When we have a complete understanding of the entire system and the methodology used, in chapter6, we will show our findings from the experiments, and how different setups in the aspects we've mentioned previously in this thesis, affect the performance and power consumption of the system. In the end, in chapter 7 we will mention different works related to our own, and how ours differentiates from theirs, our conclusions to the problem we set in chapter 8, and in chapter 9, what can be done to further our understanding of such systems.

Chapter 2 Latency critical systems and Microservices

2.1 Latency critical systems

Latency-critical systems are applications with strict response time requirements, where the system must react within a specified time frame upon receiving an input. These systems are commonly found in time-sensitive applications, such as telecommunications and networking, where real-time voice and video communication, as well as data transmission, need to be performed with minimal delay to avoid serious consequences.

Various metrics are used to measure the efficiency of such systems, with the primary one being the tail latency, also known as the 99th percentile. Latency-critical systems must deliver fast responses for the majority of queries, ensuring that most requests are served with

low to moderate latency. However, a small portion of requests may experience significantly higher latency due to factors like resource limitations, network delays, or software bottlenecks. The objective of these systems is to ensure that these high-latency requests constitute less than 1% of the total requests, meaning that 99% of requests are served within an acceptable time frame. In certain cases, systems may require even higher reliability, aiming for 99.9% of requests to be executed timely.

To calculate the tail latency, all request latencies are measured, and the latency of the slowest request at the 99th percentile is determined. This value is then assessed to determine if it falls within an acceptable range, or if measures need to be taken to reduce it.

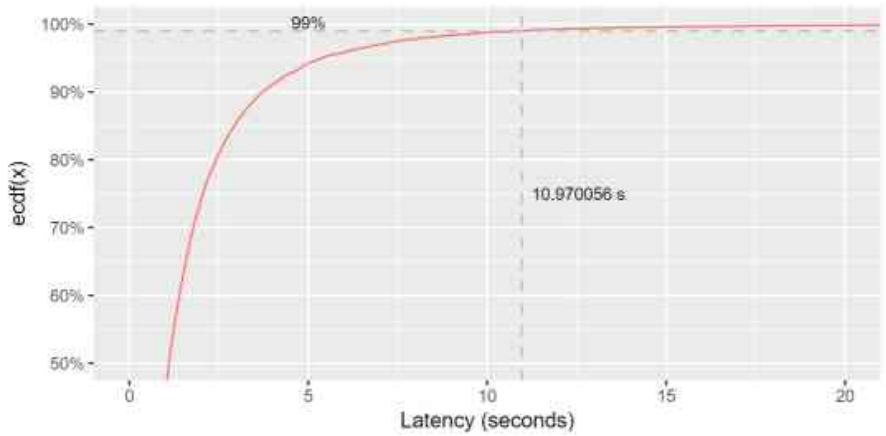

figure 2.1 latency graph on percentage of requests.

Another test that can be conducted in latency-critical systems is a strain test, which involves gradually increasing the system's workload until it reaches its limit and can no longer respond effectively. In latency-critical applications, workload is typically measured in terms of queries per second (QPS). By incrementally increasing the QPS, one can determine the maximum workload the system can reliably handle before encountering failure.

Failure in these systems can manifest in various ways, such as the tail latency exceeding the acceptable threshold or requests being dropped without being serviced. If the identified QPS at which problems start to arise is relatively low, and it is anticipated that such workload

levels will be encountered when the system is deployed in real-world scenarios, optimization measures must be implemented to ensure the system's performance meets the requirements. Such measures can be duplicating the application to divide the workload, or in microservices duplicate only the services that create the bottleneck.

Commented [3]: maybe give some examples of measures, like duplications, partitioning, load balancing etc. Also you didn't mention any examples of such applications. Like fore example web search. It will help the reader understand what you are talking about

2.2 Microservices

Microservices are a software architectural style that involves breaking down an application into a collection of small, loosely coupled, and independently deployable services. Each microservice is dedicated to performing a specific business function or capability and can be developed, deployed, and scaled independently of other services within the system. The degree of granularity in dividing microservices is not rigidly defined. It can range from a simple division between front-end and back-end, maintaining a certain level of unity while allowing independent development, to a fine-grained approach where each function becomes a separate service. This flexibility enables teams to make decisions based on their specific requirements and organizational context.

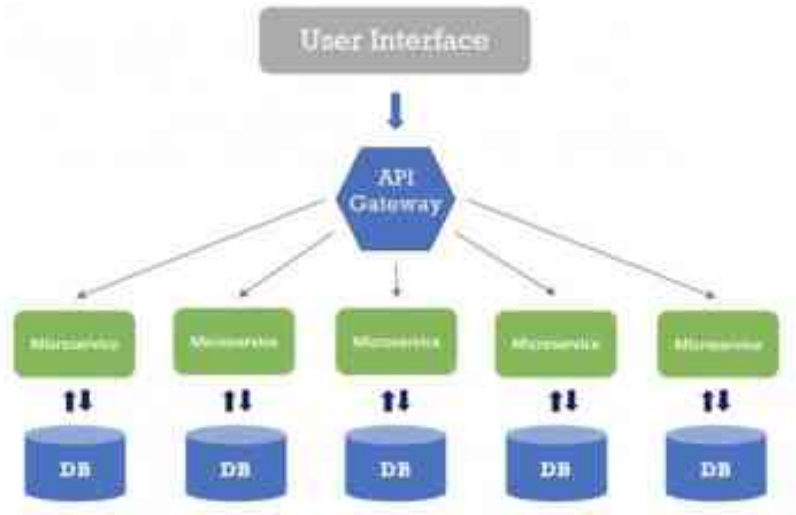

figure 2.2 Microservice Architecture showcasing its granularity [5]

To ensure isolation and consistency, each microservice can be encapsulated within a container that provides its own runtime, libraries, and dependencies. This isolation allows for a consistent and reproducible execution environment, ensuring that a microservice behaves predictably across different deployment environments. Additionally, this containerization enables easy portability and scalability of microservices.

Since microservices are isolated from each other, direct communication between them is not possible. As a result, an API (Application Programming Interface) is necessary to facilitate communication. Two widely used technologies for this purpose are REST API (Representational State Transfer) and RPC (Remote Procedure Call).

2.2.1 Rest API

REST API, short for Representational State Transfer API, is a communication interface utilized for facilitating the interaction between various web applications. It operates through HTTP-based requests, often employing the JSON (JavaScript Object Notation) format for

standardized data transfer. REST API is favored for its simplicity, scalability, and seamless integration capabilities with diverse client applications, making it a popular choice in the realm of microservices.

Within a microservices architecture, each individual microservice typically represents a specific business capability or function. Consequently, each microservice can have its own REST API, which exposes endpoints corresponding to the operations and resources it offers. These endpoints adhere to standard HTTP methods (GET, POST, PUT, DELETE), enabling actions such as resource retrieval, creation, updating, and deletion. Microservices communicate with one another by making HTTP requests to the appropriate endpoints exposed by the respective microservices.

2.2.2 RPC

RPC, which stands for Remote Procedure Call, is a communication protocol utilized for facilitating inter-process communication (IPC) between microservices that are running on separate systems. It enables a service to invoke a procedure or function in another service located on a remote machine or in a different address space, as if it were a local procedure call.

In an RPC, the client, which is the calling program, sends a request to the server, which is the remote program. The request includes the specific procedure to be executed and the necessary parameters. The server processes the request and sends the result back to the client.

One of the key advantages of RPC is that it abstracts the underlying network communication details, providing developers with a high-level interface to invoke remote procedures. This abstraction simplifies the development of microservices by concealing the complexities associated with network protocols and serialization from the programmer.

2.2.3 Docker

Docker is an open-source platform designed to streamline the deployment, scaling, and management of applications, including microservices, through the utilization of containerization. With Docker, developers can automate these processes by encapsulating applications and their dependencies into standardized units known as containers.

Containers offer several advantages. Firstly, they provide isolation, ensuring that applications run independently without interference from other processes. Additionally, containers are lightweight and portable, allowing for consistent application execution across diverse environments. This portability enables developers to create, package, and distribute applications seamlessly.

At the core of Docker is the Docker Engine, responsible for container creation and execution. The Docker Engine runs on the host operating system and manages various aspects of containers, such as their lifecycle, resource allocation, and networking. By leveraging Docker, developers can simplify application management and deployment while enjoying the benefits of containerization.

2.2.4 Microservices compared to Monolithic development.

The Monolithic architecture is a traditional software architecture pattern in which the entire application is constructed as a single, self-contained unit. In this approach, all application components, including the user interface, business logic, and data access, are tightly coupled and operate within a single runtime process.

In comparison, the Microservice architecture offers several advantages. It allows for easier scalability since each service can be scaled independently based on the specific requirements of the overall system. After the initial setup, development becomes simpler as each service can be understood, maintained, and developed in isolation without significant impact on other services. Different technologies, programming languages, and frameworks can be employed for different services based on their individual needs. Moreover, most failures remain isolated within the service that generated them, making it easier to locate, fix, or handle issues. This

architecture also enables each team to focus on developing a single service without heavily impacting the work of other teams.

However, Microservices also introduce the complexities of distributed systems, including aspects such as network communication, service discovery, and data consistency. Managing and deploying multiple services requires additional operational efforts compared to a monolithic architecture. It necessitates the implementation of tools and processes for service discovery, load balancing, monitoring, and deployment. Additionally, multiple front-end and database services may be involved, requiring mechanisms to maintain data consistency, which adds complexity to the development process. Finally, the quality of service (QoS) is harder to enforce in this architecture, as communication between Microservices over the network can introduce additional latency, becoming a dominant factor in developing them, compared to in-memory function calls within a monolithic system.

Commented [4]: maybe worth adding that the QoS is stricter for microservices than monolithic applications. Because the network is dominant factor is microservices

Chapter 3 DeathStarBench Suite

The Death Star Bench Suite was developed with the primary objective of creating an open-source benchmark tailored for Microservices, accurately reflecting the key benchmarks commonly utilized in the market, particularly for research purposes. To achieve this goal, the Suite makes use of well-known and open-source applications such as NGINX, Memcached, MongoDB, and MySQL. The majority of the codebase relies heavily on these applications, while new code primarily focuses on the interfaces between services, such as Apache Thrift, gRPC, or HTTP requests.

The Suite encompasses various independent end-to-end operations to demonstrate the microservices' ability to employ different technologies. Notably, each service is written in a different programming language, showcasing the versatility of the microservices. The benchmark encompasses the entire functionality of such systems, starting from the client's request creation and concluding with its return or delivery to the database, depending on the specific query[1].

Commented [5]: maybe worth mentioning that these benchmarks are not 2-tier benchmarks they reach deeper call flows

3.1 End-to-end services

The suite consists of five distinct end-to-end services, each serving a specific purpose. The first service is the Social Network, which was utilized for the experiments conducted in this thesis and will be explored in greater detail later on. The second service is the Media Service, designed for browsing, reviewing, rating, streaming, or renting movies. It incorporates NGINX for load balancing, a MySQL database for storage, and memcached and MongoDB as secondary databases and caches.

The third service is the E-Commerce Service, focused on clothing purchases and drawing inspiration, as well as code reuse, from other open-source shop applications. It employs Node.js for load balancing and shares the same secondary databases as the Media Service. However, it is implemented using Go and Java programming languages.

The fourth service is the Banking System, responsible for secure payment processing, loan requests, and credit card balance management. It utilizes similar technologies to the E-Commerce Service but is predominantly written in Java and JavaScript.

Lastly, there is the Swarm Coordination service, which encompasses two versions. This service primarily focuses on coordinating a swarm of drones, performing tasks such as image recognition and obstacle avoidance. The first version operates the majority of services directly on the drones, with only the initial route mapping and primary storage occurring in the cloud. This approach minimizes network overhead but imposes resource limitations on the drones. The image recognition component employs a Node.js library, while the overall implementation involves JavaScript and C++.

The second version of the Swarm Coordination service primarily runs in the cloud, with the drones primarily responsible for collecting sensor data, transmitting it to the cloud, and locally logging diagnostics. This version presents the opposite trade-offs, with increased network overhead but virtually unlimited resources. It is implemented using OpenCV and JavaScript, with logging facilitated by Node.js.

3.2 Social Network

The Social Network service operates on a broadcast-style model, allowing users to share content with a wide audience rather than engaging in one-on-one or small group interactions. Users can follow others without requiring reciprocal follow-backs, enabling them to see all the posts of the users they follow. Load balancing for this service is facilitated by NGINX, similar to the E-Commerce service, and it leverages MongoDB, Memcached, and Redis for data storage and caching purposes. Redis serves as a fast intermediary between a cache and a database, offering superior performance but with limited storage capacity.

The Social Network comprises 36 individual services, each deployed in its own container. It offers five primary functions, including user creation, user following, post creation, viewing posts on the user's homepage, and browsing other users' posts on their timelines. The service offers three distinct levels of initial user sizes, providing convenient options for experimentation. These sizes are categorized as small, medium, and large datasets, each with its corresponding number of nodes and edges. The small dataset comprises 962 nodes and 18.8k edges and is the set we used for our eperiments, the medium dataset consists of 81,306 nodes and 1,768,149 edges, while the large dataset encompasses 465,000 nodes and 835.4K edges. These varying sizes allow researchers and users to explore the system's behavior and performance across different scales and assess their relationships to one another. It also supports three distinct automatic workloads, generating random posts from random users, sequentially viewing random posts from a user's homepage, and sequentially viewing random posts from a user's timeline.

In terms of architecture, the Social Network employs a load balancer implemented with NGINX to handle incoming HTTP requests from users (clients). The load balancer directs the requests to specific web servers running NGINX, which then communicate with the microservices responsible for various tasks, and if multiple instances of a service were to exist, at which one it should be sent to. For our setup, we don’t have multiple instances. These microservices utilize php-fpm for communication and rely on Apache Thrift RPCs for inter-service communication.

Many of the microservices within the Social Network that require data retrieval have dedicated Memcached and MongoDB instances assigned to them. Memcached serves as a

data cache specific to each microservice, while MongoDB stores all the relevant data associated with the microservice's functionality. Additionally, the service incorporates Cassandra and Jaeger to capture statistical data in the form of request traces. Jaeger consists of three interdependent services and provides a user-friendly interface for visualizing and analyzing the traces, albeit with a limited capacity. On the other hand, Cassandra can store as many traces as the system's storage allows but has a more SQL-focused interface accessible via a terminal. Most microservices depend on Jaeger to correctly store their traces, with the exception of Cassandra, which acts as a backup database for Jaeger.

3.3 Workloads

As mentioned earlier, the benchmark allows the creation of posts containing various types of content such as text, links, media, and tags. Users have the ability to interact with these posts through actions like liking, commenting, favoriting, reporting, and sharing. Posts can be viewed either by accessing the home timeline, which shows the posts of the users being followed, or by visiting individual user accounts to view their specific posts. The benchmark provides pre-defined queries that can simulate these actions on a large scale. Each query requires the input of the desired number of requests to be executed per second (QPS) and the duration for which the workload should run.

3.3.1 Compose-Post Script

The Compose-post script is responsible for generating random posts from random users. It selects a user at random from the available user dataset (by default, the smallest dataset is used, but this can be modified in the open-source code). The script then appends a 256-character text, consisting of English language characters and numbers, to the creator's name to form the content of the post. Additionally, it includes 0 to 5 user mentions and URLs, as well as 0 to 4 other media elements. All options are chosen randomly. This process is repeated for each request, meaning that if the workload is set to 100 queries per second (QPS) for a duration of 10 seconds, the script will repeat this process 100 times per second for 10 seconds.

Commented [6]: how does it chooses this? randomly?

3.3.2 Read Home-Timeline Script

The Read Home Timeline script is responsible for retrieving a random array of 10 posts from a randomly selected user. Similar to the Compose Post script, it chooses a user randomly from the available dataset, which can be modified by editing the code for larger datasets. From the selected user's timeline, the script randomly selects a post from the first 100 posts and retrieves 10 consecutive posts starting from the selected post.

3.3.3 Read User-Timeline Script

The Read User Timeline script functions similarly to the previous script, but instead of reading posts from the user's home timeline (i.e., posts from users they follow), it reads posts from the user's own timeline (i.e., posts created by that user). As this workload was chosen to be used, to ensure better repeatability of experiments, the random aspect of this query has been modified to always use the same random number generator seed.

3.3.4 Mixed-Workload Script

The Mixed Workload script is designed to combine all three previously mentioned scripts. For each request, it randomly selects which script to execute. Each script has a weight assigned to it: the Read Home Timeline script has a 60% chance of being executed, the Compose Post script has a 30% chance, and the Read User Timeline script has a 10% chance. This mixed workload allows for a more diverse range of operations to be simulated.

Commented [7]: how did you choose the weights?

Chapter 4 Server configuration

4.1 C-States

In the last few decades, the construction of server farms, comprising hundreds or even thousands of servers, has significantly increased. These server farms are frequently utilized to host microservices developed by various individuals or organizations.One prevalent pricing model is to charge users solely for the duration their services are actively running. However, server farms continue to consume power even during idle periods, resulting in additional costs for the owners. To minimize downtime and optimize resource utilization, multiple services are often deployed on the same server. However, many of these applications have stringent performance requirements, necessitating the ability to handle large workloads with minimal latency. Consequently, the hardware used must be capable of supporting significantly larger workloads than typical usage scenarios, leading to idle periods where resources remain unutilized. To mitigate power consumption during idle periods, a commonly employed practice is the utilization of c-states.

C-States, also referred to as CPU power states or idle states, encompass various levels of power-saving modes available in modern computer processors. These states enable processors to reduce power consumption and conserve energy when they are not actively executing tasks or are underutilized. By entering different power-saving modes, processors progressively reduce power consumption by selectively powering down or reducing the voltage supplied to specific components within the processor. However, this renders the processor temporarily non-functional until it returns to a fully operational state. Deeper c-states require more time to restore the powered-down components and bring them back to an operational state, resulting in increased latency. To assess the potential benefits of entering a c-state, the processor continuously monitors its activity and workload. During periods of

idleness or underutilization, the processor considers entering a lower power state. The operating system and the CPU's power management features collaborate to manage the transition between c-states. The operating system can adjust power management settings and issue commands to the processor, governing its power state transitions.

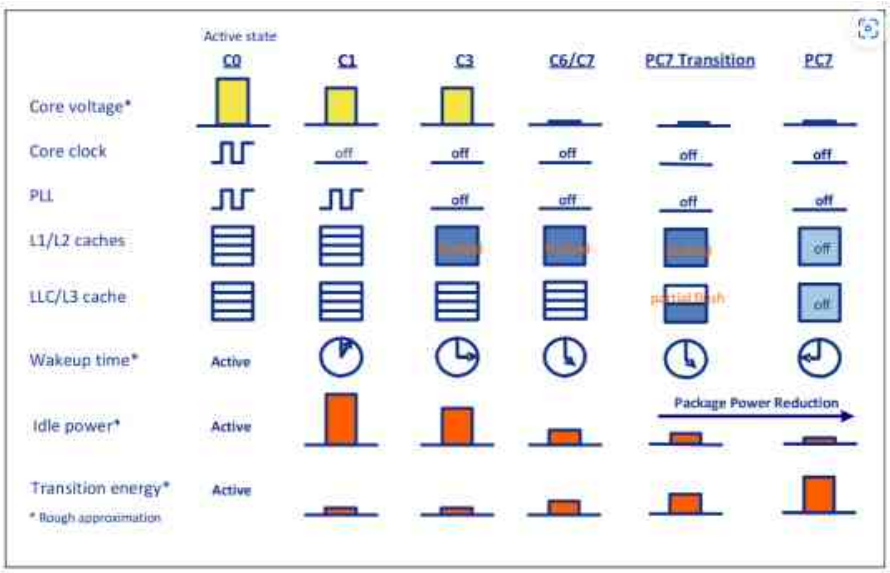

figure 4-1 overview of different C-States[6]

Commented [8]: put the source you get the image

The systems I will be using support 4 levels of c-states. C0 refers to the active state or the highest power state of the processor. When a processor is in C0 state, it is fully powered on and actively executing instructions, performing tasks, and handling the workload. C1 is the first level of idle state or power-saving state in the C-states hierarchy. In C1 state, the processor is idle, but it remains powered on and can quickly resume processing when needed. The purpose of the C1 state is to reduce power consumption while maintaining responsiveness to handle tasks as they arise. When the CPU enters the C1 state, it halts the execution of instructions and stops the clock signal to the CPU cores. However, the processor retains its context and can quickly resume processing without any significant delay. This allows the CPU to respond promptly to incoming tasks or interrupts. C1E is an enhanced version of the C1 state in the C-states hierarchy. It stands for C1 Enhanced or C1 Enhanced

Halt State. C1E is an optional power-saving state that provides deeper power savings than the standard C1 state. When a processor enters the C1E state, it reduces power consumption even further by lowering the voltage and frequency supplied to the CPU cores. This lower voltage results in additional power savings while the processor is idle or underutilized. C6 is a deeper power-saving state in the C-states hierarchy. It involves powering down or significantly reducing power to the processor cores. In the C6 state, the processor takes more aggressive power-saving measures by effectively shutting down the cores. It reduces power consumption by turning off or greatly reducing the voltage supplied to the cores, effectively stopping their operation.

Commented [9]: also frequency

Commented [10]: worth mentioning that each cpu supports different c-states

4.2 Topology and mapping

Microservices are deployed to clusters of servers, forming a swarm. The more servers there are in a swarm, the greater the available resources for the microservices to utilize. However, as the services become more divided among different servers, the need for communication through the network increases, which can introduce potential latency and slowdowns. Therefore, a service that heavily relies on communication may suffer when divided into smaller parts, whereas a service focused on computational tasks may benefit from having more resources.

It is crucial to assess the overall performance and efficiency of the microservice when considering different divisions and numbers of nodes.

4.3 Other Parameters

A server has many other parameters that we will not look into in this thesis, but we will touch upon them briefly.

The Dynamic Voltage and Frequency Scaling (DVFS) is a power management technique used in processors to adjust the voltage and frequency of the CPU in real-time based on the workload and performance requirements. DVFS aims to balance power consumption and performance, optimizing energy efficiency. The CPU achieves this by adjusting the voltage supplied to the processor cores and their clock frequency. Lowering the voltage and frequency can reduce the power consumption of the processor, as the relationship between voltage and power consumption is typically nonlinear.

Additionally, lowering the frequency reduces the number of instructions executed per unit of time, resulting in reduced power consumption. However, lower frequencies generally lead to slower performance, while higher frequencies offer faster performance but at the expense of higher power consumption.

The voltage and frequency adjustments are made dynamically based on the CPU workload and performance requirements. DVFS algorithms monitor factors such as CPU utilization, temperature, and others to determine the optimal voltage and frequency settings. These techniques aim to strike a balance between performance and power consumption. During periods of high CPU utilization or demanding tasks, the voltage and frequency may be increased to maximize performance. Conversely, during periods of low CPU utilization or when power efficiency is prioritized, the voltage and frequency may be decreased to conserve energy. DVFS algorithms typically define a set of operating points, which are specific combinations of voltage and frequency settings. The algorithms dynamically select the appropriate operating point based on workload and power-performance tradeoff considerations. DVFS is typically implemented using a combination of hardware and software mechanisms, involving voltage regulators and clock generators for hardware adjustments, and power management algorithms implemented in the operating system or firmware for software control.

Simultaneous Multithreading (SMT) is a technology used in processors to improve overall system performance and resource utilization. SMT allows a single physical processor core to behave as multiple logical cores, enabling concurrent execution of multiple software threads on the same physical core. When multiple threads are running on an SMT-enabled core, the processor dynamically switches between them, providing the illusion of simultaneous execution. This technique exploits thread-level parallelism by interleaving the execution of instructions from different threads, effectively utilizing available resources and reducing resource idle time. For the experiments conducted in this study, SMT was disabled to gain a clear understanding of the behavior of each core independently, without the potential impact of one virtual core affecting another if they share the same physical core.

Chapter 5 Experimental Methodology

For the experiments to be streamlined, several helpful secondary scripts were created to

automate the setup process. These additional scripts, along with the initial codebase of the Social Network, can be found on its dedicated GitHub repository: https://github.com/svassi04/Ptix-Fork

Once I gained an understanding of the benchmark to be used, the next step was to determine the capabilities of the system on which the experiments would be conducted. All servers, or nodes, utilized in the experiments shared identical characteristics to ensure consistency and eliminate any potential impact on the data collected. Each node consists of 20 physical cores, and since SMT is disabled for our experiments, there are also 20 virtual cores. Each core runs only one thread. The cores are distributed across two CPU sockets, with each socket containing ten cores.

5.1 Cloudlab

The nodes we use are from cloudlab. Cloudlab[7] is a website that gives easier management of clouds, from the hardware side, with most of said hardware coming from the universities of Wisconsin, Utah, and Clemson, that is to be used for research purposes. From the developer's side, it gives an easy to manage setup for a cluster of nodes, as you can choose what types of nodes you want, how many, their location, their OS, and more. You can also make a profile where you can customize more advanced characteristics of the systems, and later use that profile that you made previously. To gain access you need to make an account and then wait to be approved.

Commented [11]: cite cloudlab

Commented [12]: the

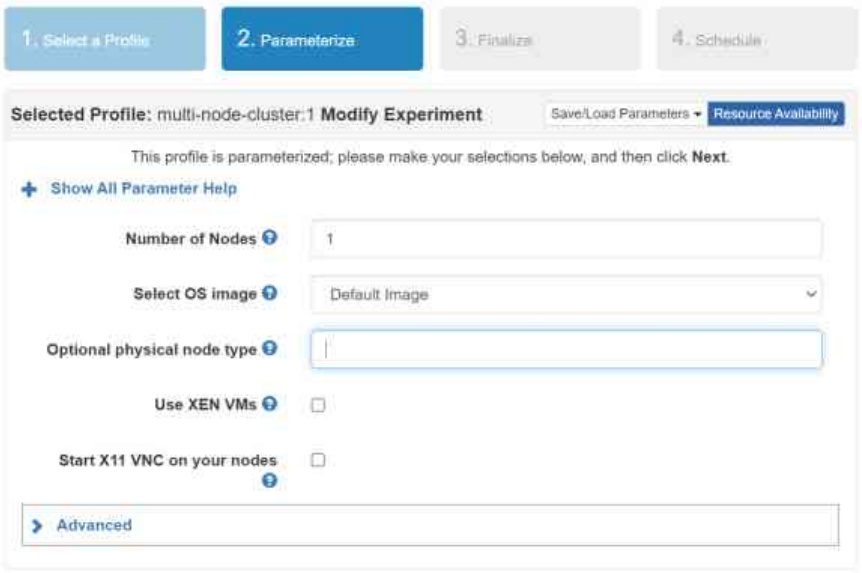

figure 5.1 cloudlab setup[7]

After running an experiment, you will receive an SSH command to connect to the nodes via the terminal. Through this SSH connection, you will have complete control over the nodes, but interaction is limited to the terminal. All data generated or stored within a node is kept locally, and when an experiment is terminated, all data stored in the nodes are lost.

By default, each experiment is allocated a maximum runtime of 16 hours, after which it is automatically terminated. If additional time is required, an extension can be requested. However, the approval process for extensions involves higher-level personnel, and the longer the extension, the more scrutiny it receives. When requesting an extension, it is necessary to provide a justification that will be reviewed.

As the experiment data is deleted upon termination, it is crucial to have a means of storing the results elsewhere. For convenience, one option is to connect the SSH address to Visual Studio Code. This provides an easy way to download desired files for storage and allows for code editing of the benchmark.

5.2 Benchmark Setup

I have mostly automated the setup of the microservice, so to set it up on the you will have to pull the GitHub repository into the last node, which will act as the client node. Inside the folder, you will want to run the script baseline.sh with the number of nodes minus one as input. This file sets the nodes into their baseline, for our experiments, state that we will explain later.

When they are configured, it will run another script to the first node, that will act as the swarm master. This script will pull the repository to all the other nodes and also install the necessary libraries for the benchmark to run correctly.

After that, it makes every other node except the client node and the master node into workers, and assigns to the worker and the master microservices for them to run. It uses a .yml file to decide which microservices to run, what resources they need, and if stated, to which node to run each service. If it is not stated, then Docker decides where to set them up.

For changes to the baseline, it is required to edit the baseline.sh file, and to change the mapping of the microservices, edit the .yml file.

e5.3 Baseline

In this thesis, the baseline was defined with the following characteristics:

SMT (Simultaneous Multithreading) is disabled to ensure that two virtual cores running on the same physical core do not interfere with each other and compete for resources. This helps isolate the impact of individual cores and provides a clearer understanding of their behavior.

Turbo is disabled, preventing the processor from increasing its clock speed beyond the normal frequency. This decision ensures that the performance remains consistent throughout the experiments, without any potential fluctuations caused by automatic frequency boosts.

The Scaling Governor is set to Performance mode, which locks the CPU frequency at the normal level. By doing so, the processor operates at a stable frequency, eliminating variations that could affect the results and allowing for accurate comparisons between different experiments.

The CPU frequency is specifically set to 2.2GHz, providing a reliable baseline for performance evaluation. This fixed frequency allows for consistent measurements and enables precise analysis of the effects of other variables on the system.

The uncore frequency, including components such as the memory controller, is set to 2GHz. This ensures that all components operate at a synchronized frequency, maintaining a balanced performance across the system.

P-States (Performance States), which allow the CPU to dynamically adjust its frequency and voltage, are disabled. This decision keeps the frequency fixed and prevents any unintentional changes during the experiments, maintaining a controlled environment for accurate observations.

For power-saving features, only C-States C0 and C1 are enabled. These states keep the processor active and responsive while minimizing power consumption. Later in the research, different C-States may be explored to investigate their effects.

These decisions were made to establish a consistent baseline, ensuring that the only variables being examined in the experiments are the specific factors under investigation. By maintaining a controlled environment, any differences observed can be attributed to the changes being studied, allowing for meaningful analysis and conclusions.

5.4 Statistics taken

In the experiment, several statistics are collected and compared to the QPS (Queries Per Second) metric. These statistics provide insights into different aspects of the system's performance.

The first metric is the average latency of the requests, which measures the average time taken for requests to be processed. It provides an indication of the overall responsiveness of the system. Since the Social Network is a latency-critical system, tail latency is also measured with up to 99th percentile. Tail latency focuses on the worst-case latency scenarios and is a crucial statistic for evaluating the effectiveness of the system.

Additionally, the experiment logs the number of requests executed and the number of requests dropped. This information helps assess the system's capacity and performance. The QPS level at which requests start to be dropped serves as a limit, indicating the system's threshold.

To understand the utilization of the system's resources, the experiment tracks the percentage of time that the processor cores spend in each C-State through the profiler[8], a tool developed by Mr Volos. C-States represent different power-saving states, and monitoring their utilization provides insights into the distribution of power consumption across the system. Furthermore, the number of transitions to each C-State is recorded, contributing to a better understanding of their behavior and efficiency.

Lastly, the experiment measures the average power consumption during its duration. This value is divided by the power consumption of each component to gain insights into their individual contributions to the overall power consumption.

By analyzing these statistics alongside the QPS metric, a comprehensive assessment of the system's performance, latency characteristics, resource utilization, and power consumption can be obtained.

Chapter 6 Results

In our experiments, we explore three different aspects that we suspect affect the performance of a microservice architecture. First we will explore how different sizes of the cluster affect it, starting from only one node, and then going to two nodes and finally four nodes. Then we look into different mappings for the two and four nodes. We will explain specific mappings

later on. Finally we will check how when different C-States affect the latency of the benchmark.

Commented [13]: how different C-States affect the latency of the benchmark

When we mention the cluster size, we always mean that the services are distributed in as many nodes as the size of the cluster. But we always have one extra node that acts as the client. From the client we run all the queries, so there is always the network overhead from the client to the cluster, and back. Finally, each experiment consists of a series of queries that start from 100 QPS up to 1000 QPS, increasing by 100 QPS each time, running for 30 seconds each. For more accurate data, each experiment is repeated five times.

Commented [14]: do you present the results for each run along with the average?

6.1 One Node Cluster

The one node cluster was used mainly to get more familiar with running the benchmark and gain more familiarity with the statistics I will be looking into. Still, when compared to other cluster sizes, or different C-States enabled, we get some useful data too. As this setup has only one node for the microservices (and one for the client), all the services are necessarily together.

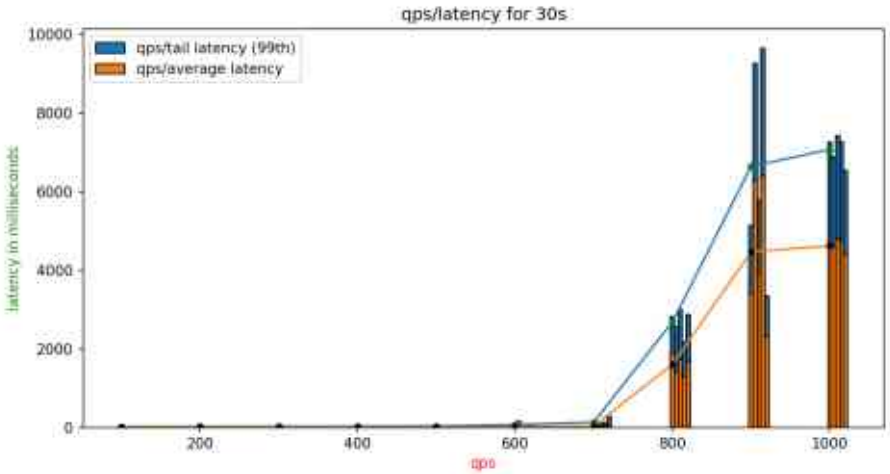

figure 6.1 average latency, and tail latency compared to QPS. Each bar is one repeat.

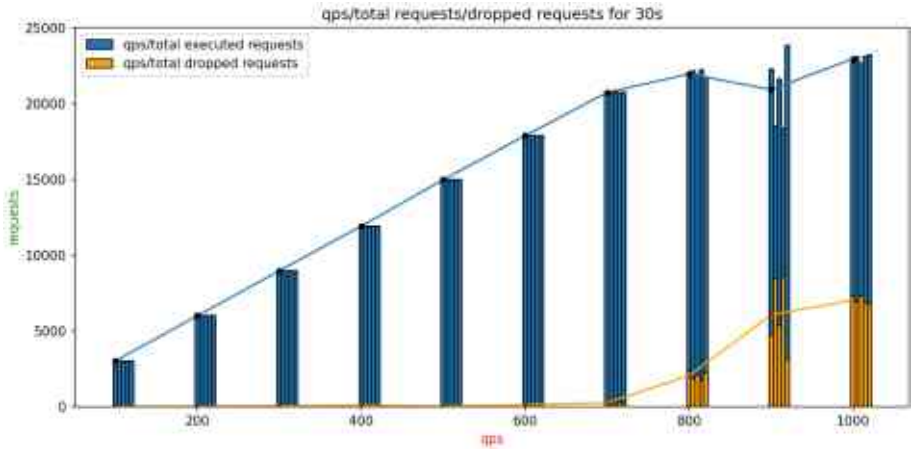

figure 6.2 executed requests and dropped requests compared to QPS. Each bar is one repeat.

As we can see from the two figures above, up to 600 QPS, the system handles the requests fast, and without any drops. But at 700 QPS, they become too many to respond fast, and has a dramatic increase in latency, as well as a few drops. After that, with more QPS the problem only worsens, though the requests executed stay the same. For this thesis I will focus on only the rate of QPS that the system can handle, as afterwards its responses become unstable, and so to an extent unusable.

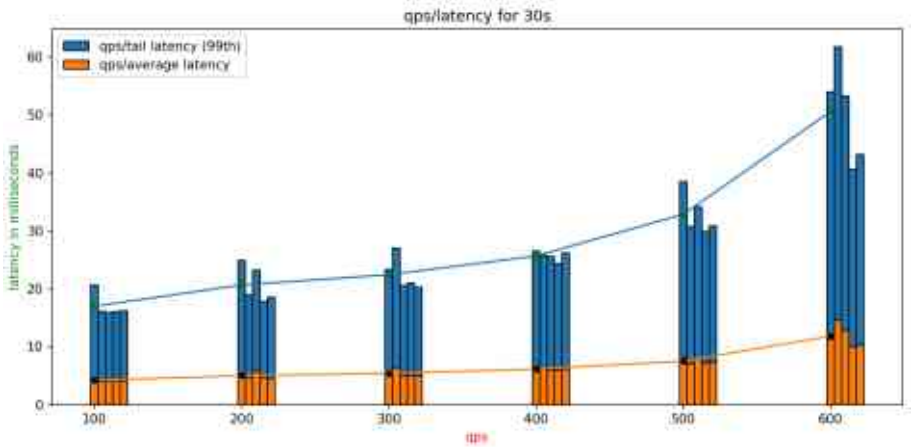

figure 6.3 average latency, and tail latency compared to QPS up to 600 QPS. Each bar is one repeat.

As we can see from above, while the system is still healthy, it responds in less than 50ms for

99% of the requests, with an average around 10ms. It starts even faster than that for few QPS, but as we increase it, the latency, both tail and average, increases non-linearly, with the tail latency seeing more rapid increase. Finally it is interesting to see that on average the tail latency is five times that of the average latency.

Commented [15]: interesting to see that the average response time is 5x the 99th tail latency time

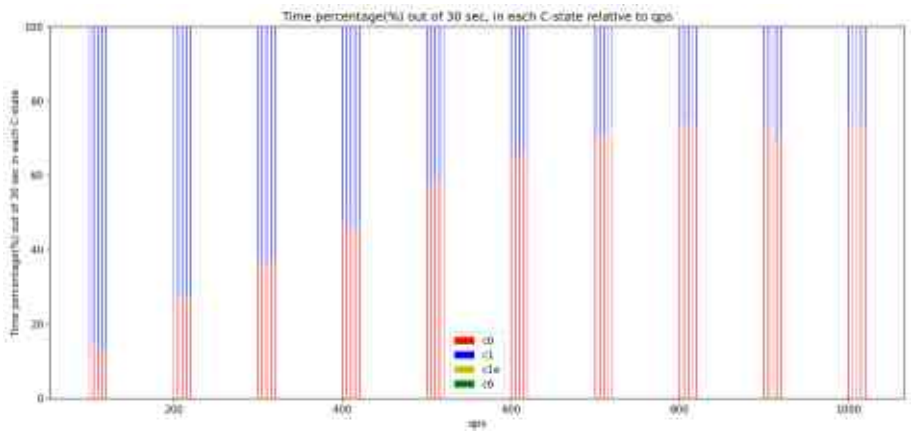

figure 6.4 percentage of the experiment time it spent on each C-State compared to QPS. Each bar is one repeat.

The above graph keeps track of the percentage of the 30 seconds each experiment lasted, the node took in each C-State. For example, for the first experiment it was in C0 for around 15% of the time, or only 4.5 seconds out of the 30. As the workload increased, we can see that the time it sent on C0, meaning it had work to do, increased. At around 700 QPS, the level where it started to have dropped requests, it reached the highest time spent in C0, and when we continued to push it, the time spent on C0 didn't increase, because as we saw from figure 6.2, as we increase the QPS beyond what it can handle, it drops more requests without increasing how many it handles, meaning that the work it does stays the same.

Commented [16]: why? It still serves the same amount of work, it just drops more queries at each QPS, this is evident from graph 6.2

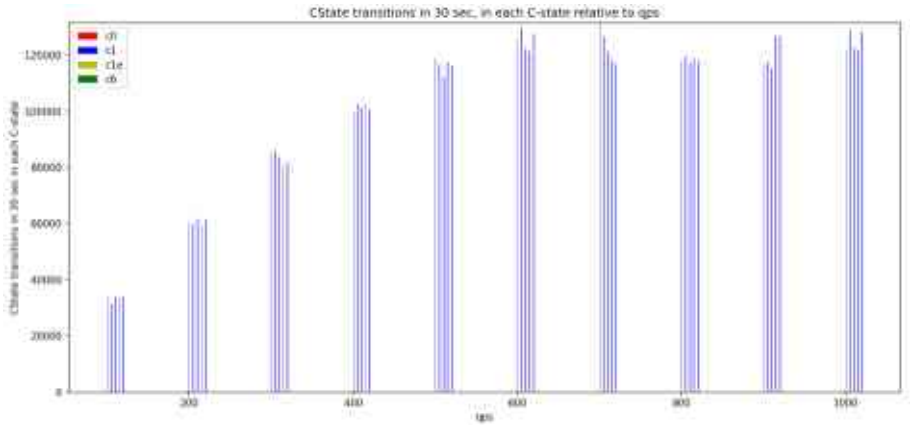

figure 6.4 transitions into each C-State compared to QPS. Each bar is one repeat.

With the above graph, which shows how many transitions the system did to each C-State, we can surmise that since the C1 resitensy is reduced with the increase of the QPS, but the C1 transitions increase, then the duration the system stays in C1 each time it enters, decreases too. Meaning that it will often enter C1 but it will exit it much sooner than if it had lower QPS. Another hypothesis is that, the system wrongly found calm portions that it thought it could enter a deeper C-State and took it, but had to quickly exit that state. As to exit the state takes time, it stayed for a substantial part if the time into C1.

Commented [17]: This is an assumption. We dont know whether the system wrongly predicts the duration of the idle periods for sure(you can include it as an assumptions). You can say that since the C1 residency reduces with the increase of the qps and the c1 transitions increases for sure the idle periods have smaller length with high qps.

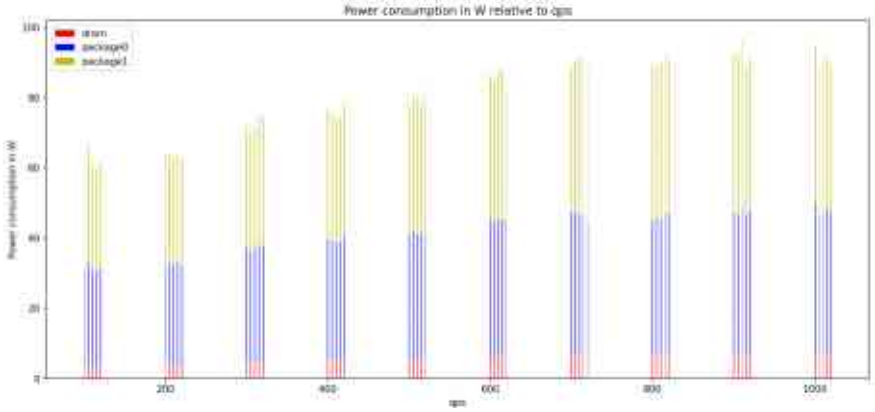

figure 6.5 power consumption compared to QPS. Each bar is one repeat.

Finally, as we can see, the main contributor of power are the two packages, which as the workload increases, most increases in power consumption happen to these two. Also the

power consumption follows the same trend as the C0 resitency, as with more requests it had to run more commands in the core, which consumes power, as well as the fact that it had less opportunities to stay in C1 and save power.

Commented [18]: You can say tha dram power does not increase dramatically and power consumption among the sockets 0 - 1 is the same meaning that the work is equally distributed among the sockets

6.2 One Node Cluster C-State experimentation

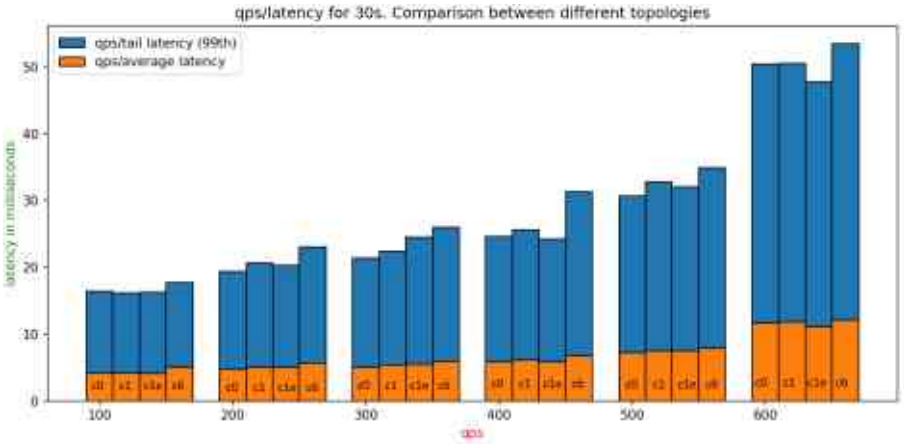

figure 6.6 average latency, and tail latency compared to QPS up to 600 QPS. Each bar is up to which C-State is enabled, and the average of the five repeats.

When different C-States are enabled, it is decided how deep in an idle state the system can go, with C0 being the active state, and C6 the deepest this system has. From the figure 6.6, it is interesting to note that up to 600 QPS, when the C6 is enabled, the system has the biggest latency, especially compared to the tail latency. While the other three states are more or less the same, with sometimes one configuration having better performance than the other two, and other times worse. That is because the underlying system sometimes might put on hold the microservices for a few milliseconds to run a function of the system, damaging the performance. Still, as with C6, it is always slower, we can determine that having this C-State enabled damages the performance. Additionally, it is odd that when C1E is enabled it doesn't affect the performance. An assumption to why that happens would be that even though the Idle Governor, a software, suggests for the system to enter C1E, the hardware might not actually follow the suggestion.

Commented [19]: C6 is always worse due to the transition overhead and due to the fact that the system enters C6. t is interesting to notice that C1E does not affect neither the average or the tail latency performance. This is because even though the idle governor suggest that the system enters C1E the system might not actually do it. but this is an assumption

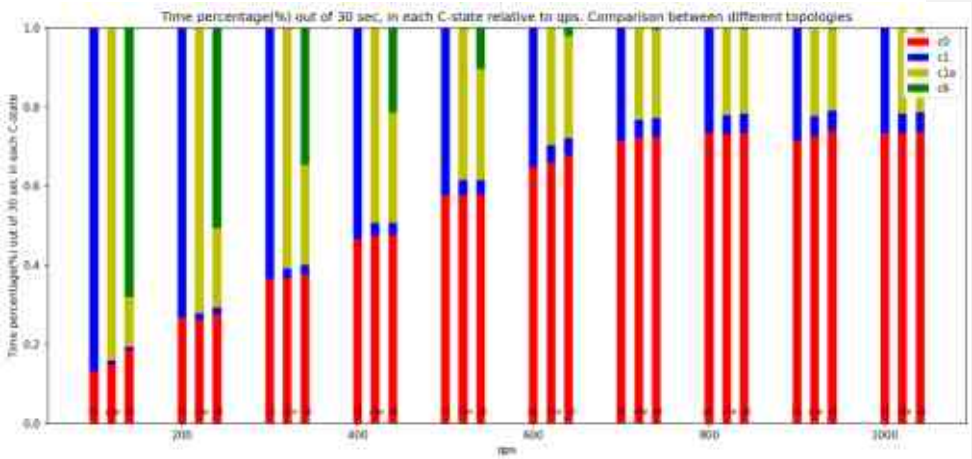

figure 6.7 percentage of the experiment time it spent on each C-State compared to QPS. Each bar is up to which C-State is enabled, and the average of the five repeats.

Because when only C0 is enabled, it has no deeper C-States to go to, so the entire time it would stay in C0. Because of that, in the above graph, it is excluded. As we can see from the other three configurations though, they stayed for nearly the same time in C0 for all configurations, with deeper states taking the time from shallower ones. That is expected, because the system has more C-State options to choose from when it decides that it is idle. Instead of just seeing if it is idle to enter C1, now it calculates for how long it is expected to stay there, and based on the enabled C-States, it chooses accordingly.For example, if it were to go to C1 and stay there for half a second, then it might be better to enter C1E or C6 instead. Finally we can understand, that since they stay in C0 for the same time, then the extra latency comes from transitioning from C6 back to C0, and that transitioning from C1E or C1 to C0 is way faster, or despite being suggested, it didn't actually enter them.

Commented [20]: What you describe here is auto - promotion. We do not know whether this functionality is enabled in our system. Auto - demotion maybe. Basicaly with auto-promotion the hardware might decide to put the system into a deeper c-state as you described. With auto - demotion the hardware might decide to put the system into a shallower c-state as you describe. But in this case just say that because we gradually enable more c-states the idle governor has more options to choose for and when allowed by the sleep duration it chooses deeper c-states. While we increase the qps the duration of the idle times decreases and so it has fewer opportunities to enter deep states

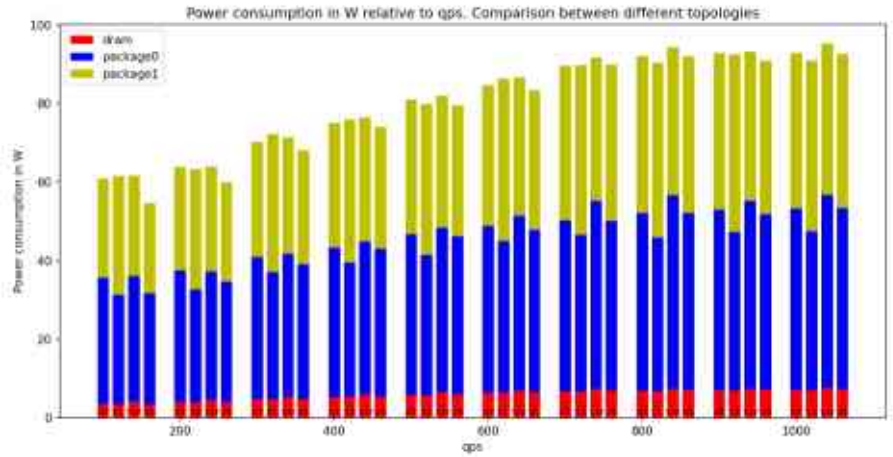

figure 6.8 power consumption compared to QPS. Each bar is up to which C-State is enabled, and the average of the five repeats.

On the other hand, while it still has enough opportunities to enter C6, as we can see from figure 6.8, it has a noteworthy decrease in power consumption, but when compared figures 6.7 and 6.6, as it enters C6 less often with higher QPS, the gains from C6 fall, but the latency generated by entering it stays. Additionally, we can see that there is not a noteworthy reduction of power by entering C1 or C1E.

Commented [21]: entering c6 rescues power consumption but the power savings reduce for higher qps

6.3 Two Nodes Cluster

Here we expand the size of the cluster to two nodes for the microservices, and one for the client. As we have two nodes to divide the microservices into, we can now experiment with how different mappings will affect the benchmark. We will experiment with four different mappings by dividing the microservices to heuristic types. Those types are the nginx, meaning the load balancer, the jaeger that saves the traces from the requests, the Logic that execute the requests, the databases mongoDB and Redis, for storing data, and finally the MemCached, which are as the name implies, caches.

With our five types defined, for testing the mappings, I will divide them between the two nodes. In this thesis I will explore four different mappings. Those are the following. Jaeger alone in one node, and the rest in the other. This will test how heavy Jaeger is on the system,

and if it requires too much communication with the rest. The next mapping is isolating nginx alone, and the rest together, checking how the system reacts without nginx together with the rest. Afterwards, we leave the databases alone, and the rest together, to see how it will react, needing to communicate through the net to get any non-cached data. Finally we see how it will fair if it needs to communicate through the net for any data, by having the databases and the caches together, and the rest to the second node.

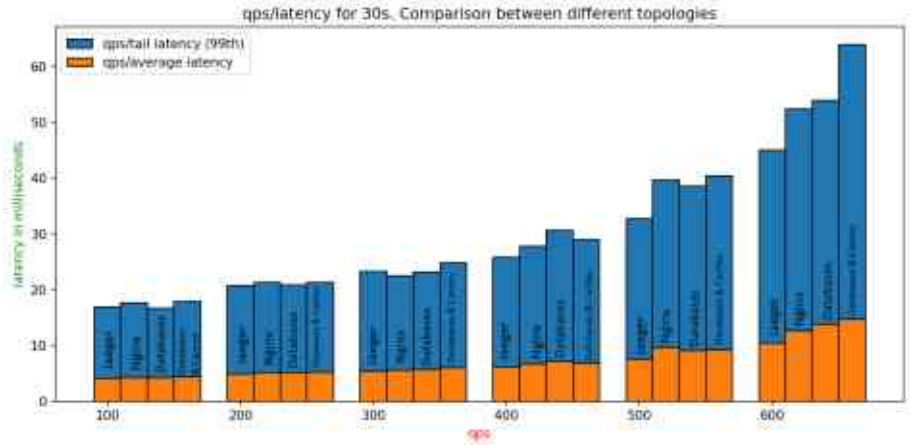

figure 6.9 average latency, and tail latency compared to QPS up to 600 QPS. Each bar is a different mapping, and the average of the five repeats. 1st bar: Jaeger alone, 2nd bar: Nginx alone, 3rd bar: Databases alone, 4th bar: Databases and Caches together

As we can see from the figure 6.9 there is no major difference in the average latency of all the mappings, and for low QPS the tail latency is the same too. At higher QPS, 300 to 600 QPS, the tail latency of isolating the Databases and the Caches is worse, That can be explained by having to transport all data through the network, creating communication overhead. On the other hand, having Jaeger alone has the lowest latency, suggesting that despite being called regularly, this activity is not in the critical path of the system.

Commented [22]: Jaeger has the lowest latency because even though it is called regularly, the amount od data transferred towards it is small and it is not on the critical path like the rest of the microservices(is the communication asynchronous?). Database has better performance than database and cache because big percentage of the database requests is being served locally through the caches. Fianlly nginx performs better than database and cache because even though it is in the critical path, it only transfers queries which have smaller size. And performs worst than niginx because is in the critical path

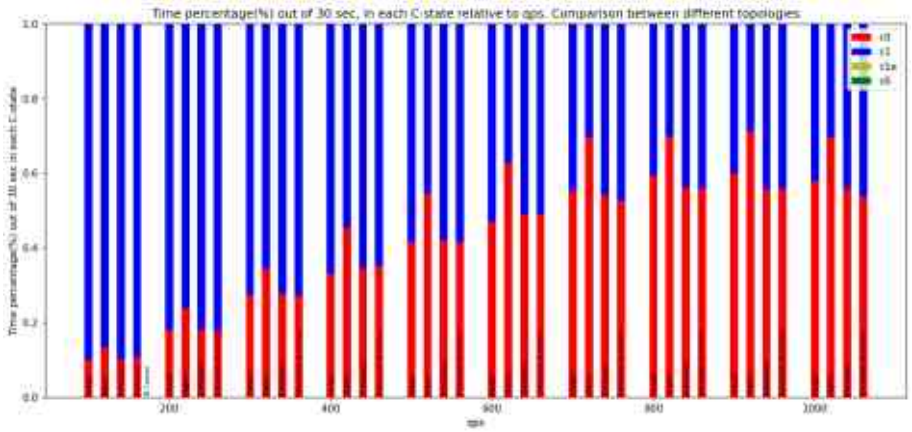

figure 6.10 percentage of the experiment time it spent on each C-State compared to QPS. Each bar is a different mapping, and the average of the five repeats. Of the two nodes, it shows the one with the most activity. 1st bar: Jaeger alone, 2nd bar: Nginx alone, 3rd bar: Databases alone, 4th bar: Databases and Caches together

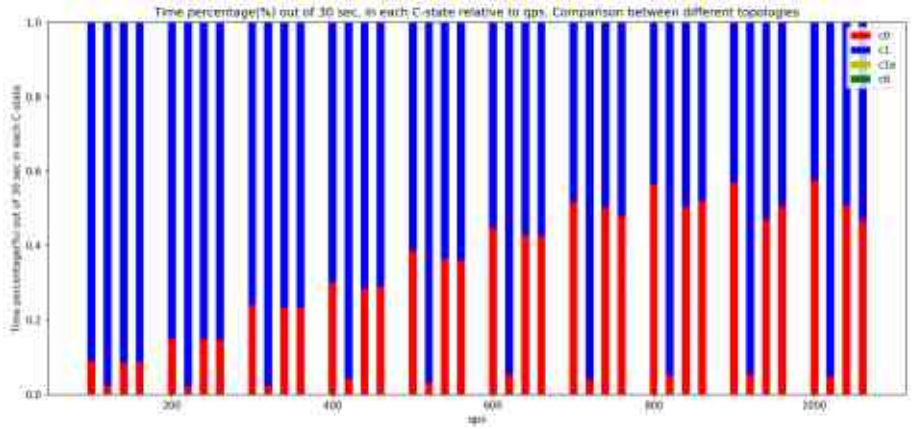

figure 6.11 percentage of the experiment time it spent on each C-State compared to QPS. Each bar is a different mapping, and the average of the five repeats. Of the two nodes, it shows the one with the least activity. 1st bar: Jaeger alone, 2nd bar: Nginx alone, 3rd bar: Databases alone, 4th bar: Databases and Caches together

From figure 6.10 we can see that the mapping where the nginx is alone, we can note that the server that has the rest of the services, stayed at C0 substantially longer without having improved latency. When we compare it with the 6.11 figure, it becomes clear that nginx doesn't have much work to do, and so all of it happens at the other node. Also we can note that as the latency is at the same level as that of having the databases alone. That could happen if the average communication overhead of all the requests from Nginx is of the same

Commented [23]: Good point. Also mention that the trends are as expected the higher the qps the more the C0 residency.

level as that of the databases, as while Nginx has constant but lightweight communication, the databases are used more rarely, due to the caches, but the communication is heavier. To test that, we could construct a workload that would generate requests that needed the databases and not the cashes

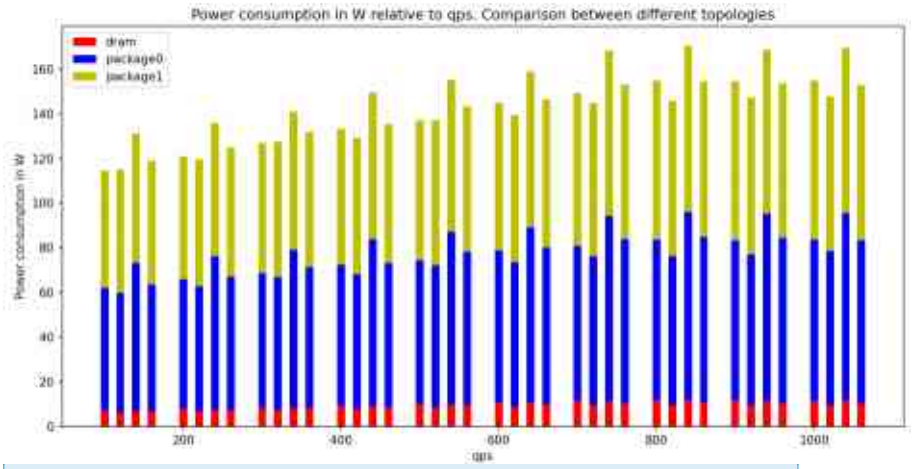

figure 6.12 power consumption compared to QPS. Each bar is a different mapping, and the average of the five repeats. The power consumption of both nodes is added. 1st bar: Jaeger alone, 2nd bar: Nginx alone, 3rd bar: Databases alone, 4th bar: Databases and Caches together

Commented [24]: why databases have higher power consumption? do you have any assumptions?. Is it an outlier in the measurements?

Commented [25]: why databases have higher power consumption? do you have any assumptions?. Is it an outlier in the measurements?

In my research nothing indicated that by isolating the databases would have a dramatic increase in power consumption, and neither do the C-State residency graphs suggest that. An idea on why that is happening is that sometimes, in my experiments, I had to change on which machines I run the benchmark. It is possible that the machines for that one experiment had a different static power consumption due to a problem in it. A test to make sure that such things could be checked would be to run the power analysis program without anything running on the system.

6.4 Two Nodes Cluster C-State Experimentation

For testing the effects of different C-States on two nodes we use the mapping where the databases and the caches are together, and the rest are on the second node.

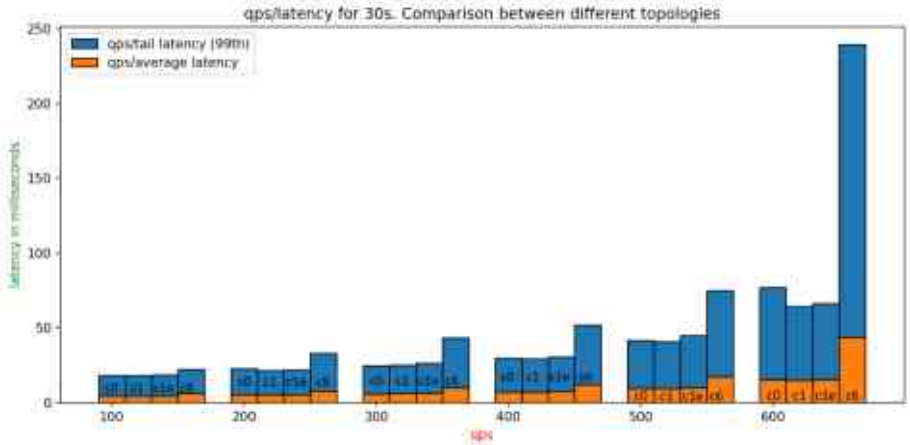

figure 6.13 average latency, and tail latency compared to QPS up to 600 QPS. Each bar is up to which C-State is enabled, and the average of the five repeats.

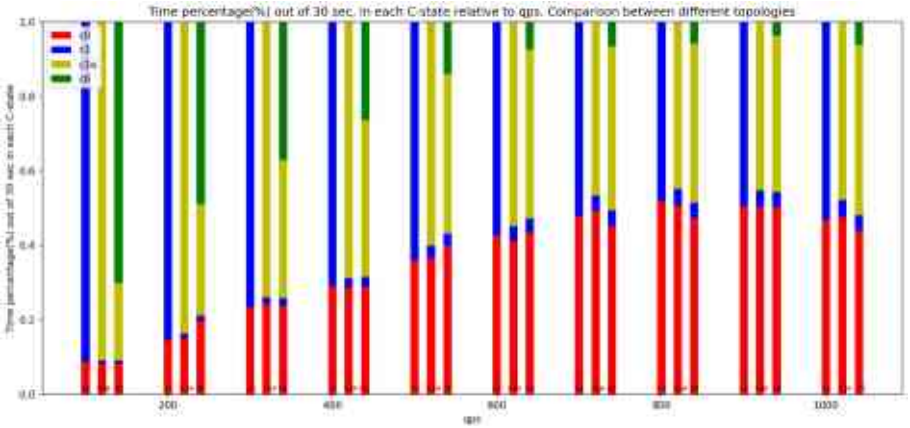

figure 6.14 percentage of the experiment time it spent on each C-State compared to QPS. Each bar is up to which C-State is enabled, and the average of the five repeats. Of the two nodes, it shows the one with the most activity.

As we can see from figures 6.13 and 6.14, it supports our findings that C6 worsens the latency even though it stays at C0 for relatively the same time. Even though it stays for only 10% of the time at C6 when the QPS is 600, the latency of exiting from C6 is too big. It is worth mentioning that as the services are in two nodes now, when one node waits for an answer from the other, it has more opportunities to enter C6, which might explain the tremendous increase inlatency, compared to only one node. As a result the query will experience the C6 transition overhead multiple times.

Commented [26]: why the impact of c6 is worst at two nodes? Did you expect it? I think that it is additive when you have two nodes. Another thing is that in a multi-node setup when one node experiences variability then it impacts tremendously the execution because is slowdowns the execution for all the nodes

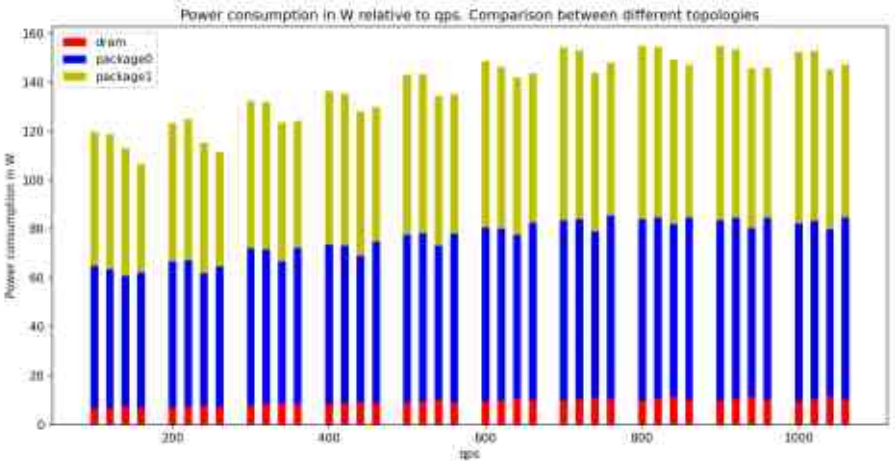

figure 6.15 power consumption compared to QPS. Each bar is up to which C-State is enabled, and the average of the five repeats.

We also support our findings about the decreased power consumption for C6 when the QPS is low, but as it has more opportunities to enter C6 even with more QPS, It maintains some reduction in power consumption even then. In contrast to one node though, we see that with two nodes, C1E gives us good reduction in power too.

6.5 Four Nodes Cluster

For four nodes we will again explore different mappings between the types that we used previously. As we have four nodes and five types, each node will have one type, except one that will have two. The mapping we will explore is when we use the Databases and the Caches together, having together the Logic and the Caches, having the Jaeger with the Logic, and finally having the Logic with Nginx.

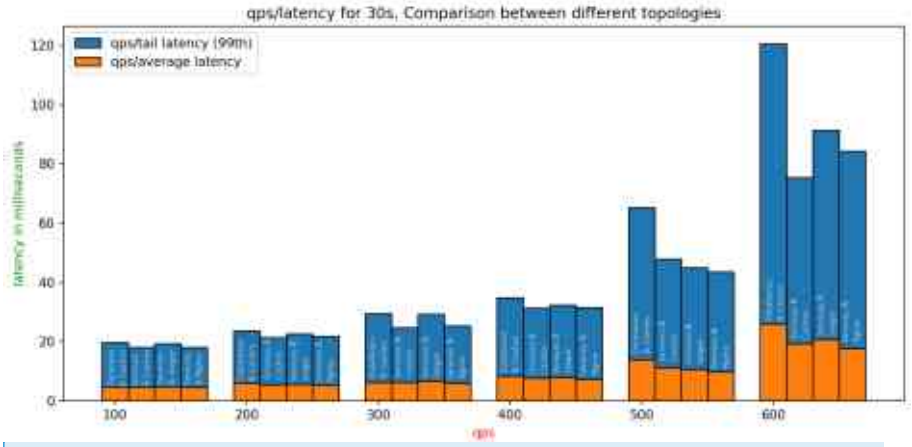

figure 6.16 average latency, and tail latency compared to QPS up to 600 QPS. Each bar is a different mapping, and the average of the five repeats. 1st bar: Databases and Caches together, 2nd bar: Logic and Caches together, 3rd bar:Logic and Jaeger together, 4th bar: Logic and Nginx together

Commented [27]: we cannot see the mapping in the bars. maybe consider different colour

Commented [28]: I am not sure why frontend alone is worst.

Commented [29]: we cannot see the mapping in the bars. maybe consider different colour

Commented [30]: I am not sure why frontend alone is worst.

From the graph above we can see that when the Logic is alone, the latency increases, while when it has any other types of services with them, it decreases. That is because the Logic is the central part of the benchmark, meaning that they send often requests to the other types of services. By having another type with them, it means that part of the communication overhead can be avoided, while when they are alone, all requests have that overhead.

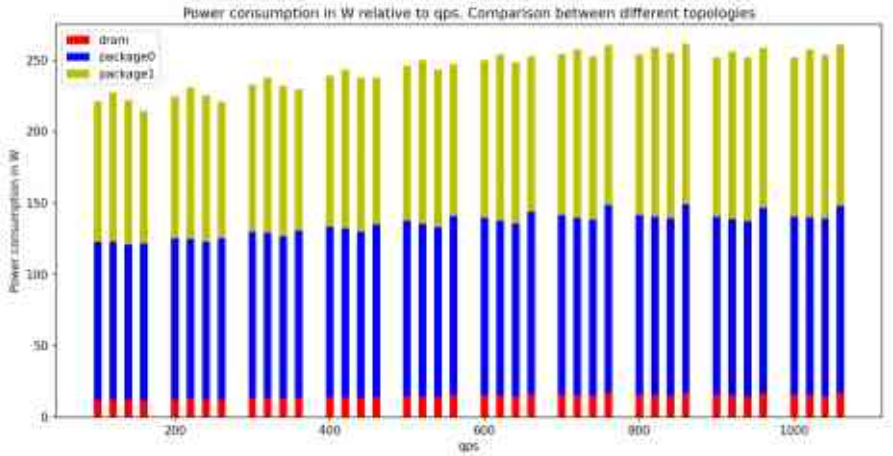

figure 6.17 power consumption compared to QPS. Each bar is a different mapping, and the average of the five repeats. The power consumption of all nodes is added. 1st bar: Databases and Caches together, 2nd bar: Logic and Caches together, 3rd bar:Logic and Jaeger together,

4th bar: Logic and Nginx together

From this graph we can see that the mappings chosen give no clear indication of which has better consumption.

6.6 Comparing Cluster Sizes

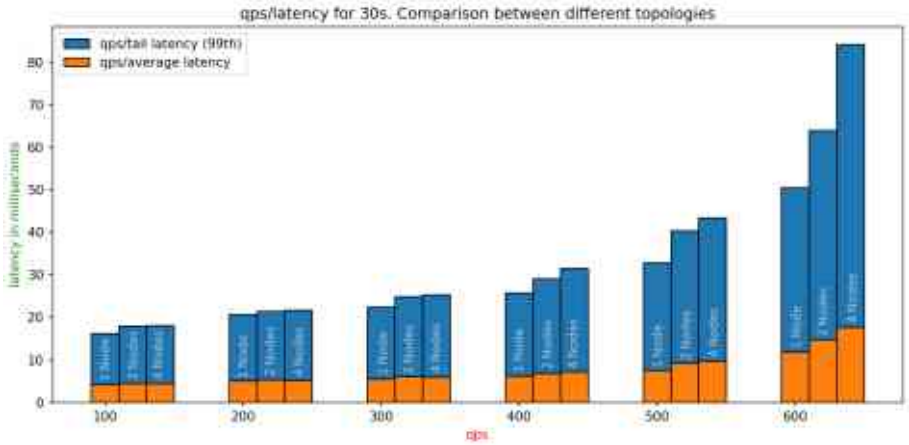

figure 6.18 average latency, and tail latency compared to QPS up to 600 QPS. Each bar is a different cluster size, and the average of the five repeats.

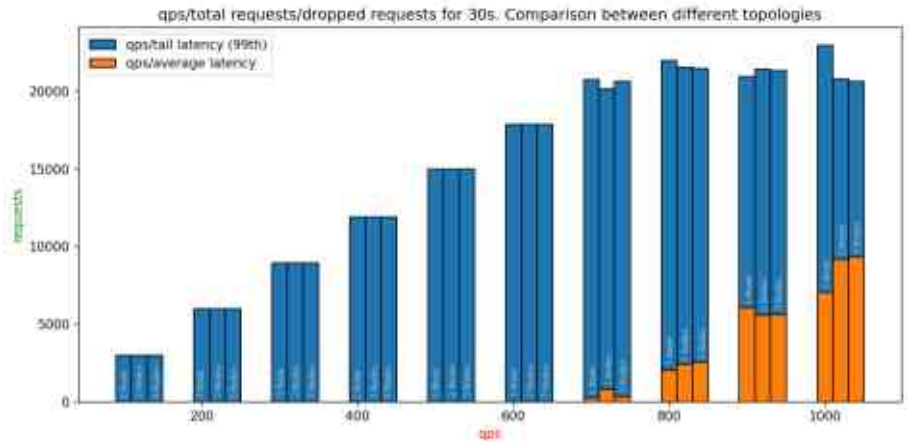

figure 6.19 executed requests and dropped requests compared to QPS. Each bar is a different cluster size, and the average of the five repeats.

As we can see from the graph above, as we increase the number of nodes in a cluster, we notice that the latency increases. With more nodes in a cluster, and the microservices distributed across them, it is normal that communication between different nodes will have an overhead. That overhead creates latency that is added to the latency for the processes to run. As such the benchmark suffers from more nodes in it.

Commented [31]: What about dropped queries?

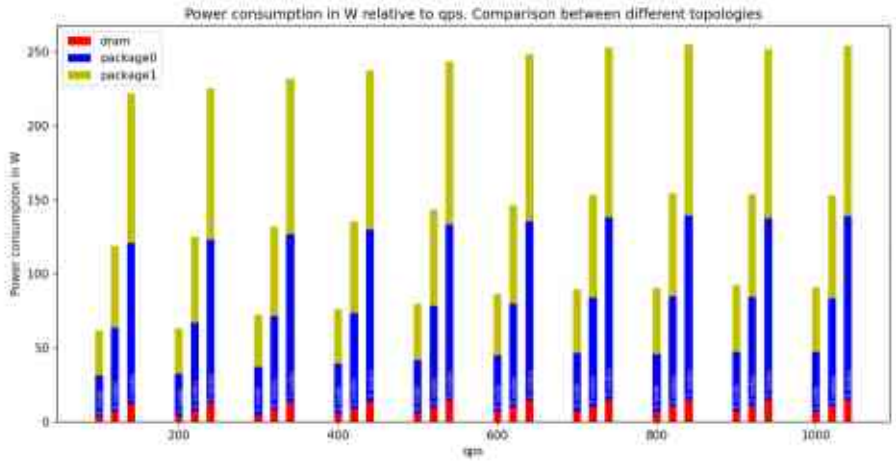

figure 6.20 power consumption compared to QPS. Each bar is a different cluster size, and the average of the five repeats. The power consumption of all nodes is added.

Finally, when we compare the power consumption of the entire system, we can see that by increasing the number of nodes in a cluster, as each server of the cluster is dedicated to running our microservices, the power consumption doubles, by doubling the servers. As each one only runs our benchmark and nothing else, the power overhead to keep the system running is around 55w for each server. By doubling the servers we double the power overhead to 110w, and by quadrupling the number of notes, the overhead rises to 220w. The increase of power consumption due to the activity of the nodes though, increases at the same rate even with more nodes, meaning that increase is divided between the different nodes of the cluster.

Chapter 7 Related work

The paper by Anatoli Atanasov titled "Characterization of system behavior using performance counters" shares many similarities with my own research. Both works delve into the analysis of microservices, their associated technologies, and their respective strengths and weaknesses. Additionally, we both examine the same benchmark, providing a common ground for comparison. However, Atanasov's work focuses on elucidating the correct execution of the benchmark and compares the performance between clusters consisting of 5 and 20 nodes.

In contrast, my research builds upon Atanasov's findings by exploring additional dimensions. Specifically, I investigate the impact of different mappings and varying levels of C-States on performance and power consumption. By examining these aspects, my work expands upon the understanding of microservice behavior and provides insights into optimizing their configurations.

Moreover, it is important to acknowledge the significant influence of the SAIL team at Cornell University, particularly their work on the Social Network. The SAIL team conducted experiments with query rates ranging from 100 to 400 QPS and observed substantial increases in latency beyond 200 QPS. While our research shares a similar focus on the Social Network, we extend the investigation by exploring a wider range of QPS values and analyzing how latency is affected by diverse system and microservice configurations.

In summary, my work aligns with Anatoli Atanasov's research in terms of analyzing microservices and employing a common benchmark. However, my research expands upon his findings by exploring additional factors such as mappings and C-States. Additionally, both our works draw inspiration from the SAIL team's experiments on the Social Network, but my research further examines the effects of different QPS rates and diverse system configurations on latency.

Chapter 8 Conclusion

The microservice architecture presents numerous aspects that require careful consideration and further research. However, it holds immense potential for future optimization when the correct strategies are

implemented. Through my investigation, I have discovered several important findings that shed light on the performance of microservices.

Firstly, the choice of mapping microservices to different nodes can have a significant impact on their latency. It is understandable that when critical path microservices are distributed across separate nodes, the added communication latency becomes a contributing factor. This highlights the importance of thoughtful mapping decisions to minimize latency in the microservice architecture.

Furthermore, I have observed that enabling deep C-States, particularly in systems with multiple nodes, introduces overhead during the exit process, resulting in increased latency. This implies that while deep C-States may offer power-saving benefits, there is a trade-off in terms of increased latency, especially in scenarios where multiple nodes are involved, as the system has more opportunities to enter deep C-States while waiting for an answer from other nodes.

Lastly, I have noticed a relationship between the power gains achieved from enabling deeper C-States and the workload size. As the workload increases, the power-saving advantages diminish, while the latency experiences a corresponding increase. This finding underscores the need to carefully balance power optimization with the performance requirements of microservices, particularly in scenarios with larger workloads.

In conclusion, the microservice architecture demands comprehensive exploration and research to unlock its full potential. By leveraging appropriate optimizations and considering factors such as mapping strategies and C-State configurations, it is possible to enhance the overall performance and efficiency of microservices in the future.

8.1 Further work

In this work, my focus was primarily on a limited set of parameters that directly impact the performance of the microservices. However, there is still a vast scope for further exploration and investigation.

One area of interest lies in exploring configurations such as enabling Turbo or increasing the frequency to potentially enhance performance. It would be valuable to assess the extent of performance improvement achieved by these configurations and weigh it against the corresponding increase in power consumption. Finding the right balance between performance and power efficiency is a crucial aspect to consider.

Additionally, investigating the impact of enabling P-States in the system is another avenue worth exploring. It would be interesting to observe the potential power consumption gains that can be achieved through P-States while assessing their potential effects on the Quality of Service (QoS) provided by the Social Network microservices.

Furthermore, the utilization of system parallelization when Simultaneous Multithreading (SMT) is enabled is a topic that deserves attention. Exploring how the microservices leverage the parallel capabilities of the system can provide valuable insights into optimizing their performance and resource utilization.

By conducting these experiments, it is possible to further refine and optimize the microservices without significantly increasing their power consumption. This approach aims to strike a balance between performance enhancement and efficient resource utilization.

References

[1] 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.

[2] L. Barroso, U. Hoelzle, and P. Ranganathan, The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Morgan & Claypool: San Rafael, CA, USA, 2018

[3] System Monitoring with lo2s: Power and Runtime Impact of C-State Transitions

[4] Paul J. Kühn, Engineering of DVFS-PowerManagement for Cloud Data Center Clusters, University of Stuttgart, Germany

[5] Image from javarevisited.blogspot.com microservices design patterns principles

[6] image from interl’s Energy-Efficient Platforms – Considerations for Application Software and Services

[7] Cloudlab site: https://www.cloudlab.us/

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