UMEÅ UNIVERSITY
Department of Computing Science

January 13, 2024

Evaluation of distributed tracing frameworks for microservice observability

Name	Kristoffer Gilliusson
Supervisor	Johan Tordsson
Examiner	Henrik Björklund
	Michael Minock

Abstract

A current trend in software is the transition from large monolithic systems to smaller, loosely coupled microservices. This leads to a lot of benefits but it also has its fair share of drawbacks. Microservices are especially hard to debug due to their decentralized nature. This thesis investigates the technology of distributed tracing which is a technology to help debug and monitor distributed applications. It compares the two tracing frameworks Jaeger and Zipkin with a testbed consisting of Kubernetes, a microservice benchmarking program (TeaStore), Prometheus and Grafana for monitoring and JMeter for stress testing. The frameworks are tested on the aspects of how much overhead tracing adds to microservices, how easy tracing is to implement, how easy they are to use, its project vitality and how mature Jaeger and Zipkin is.

The results showed that Jaeger is the better tracer in this situation in almost every case except that it draws more CPU than Zipkin and the conclusion is that it is wise to utilize a distributed tracing framework for microservices.

Glossary

BCrypt Bcrypt is a password hashing function designed by Niels Provos and David Mazières . 14

CNCF The Cloud Native Computing Foundation (CNCF) is a Linux Foundation project that was founded in 2015 to help advance container technology . 10

container orchestration Container orchestration is the automatic process of managing or scheduling the work of individual containers for applications based on microservices within multiple clusters . 7

pods Pods are the smallest deployable units of computing that can be created and managed in Kubernetes. A pod consist of one or more containers . 9

REST REST stands for REpresentational State Transfer. It means when a RESTful API is called, the server will transfer to the client a representation of the state of the requested resource . 9

SHA-512 SHA-512 is a cryptographic hash function that is designed by the United States National Security Agency (NSA) . 14

SQL Structured Query Language (SQL) is a standardized programming language for retrieving and modifying data in a relational database . 14

YAML YAML is a human friendly data serialization standard for all programming languages and it is used to describe the deployment of pods in k8s(Kubernetes) . 9

Contents

Chapter 1

Problem description

The new trend in IT infrastructure is the transition from one large monolithic system to many smaller, loosely coupled services called microservices. This adds further complexity as the memory is no longer shared and messages has to be sent to communicate with other machines. This is why distributed tracing is introduced, to help the programmer in the large world of microservices.

1.1 Motivation

It is pretty clear that Cloud Computing was one of the biggest deals in IT in the 2010s. It was in this decade the giants, Amazon, Google and Microsoft really began to expand on their already operational cloud business, AWS for Amazon (2006)[1], GKE for Google (2008)[2] and Azure (2010)[3] for Microsoft. This decade also gave birth to OpenStack, the top open-source cloud software platform. Worldwide spending on cloud platforms was 77 billion dollars and according to the statistical company Statista the world would spend around 411 billion dollars at the end of the decade. Container technology grew rapidly in the early beginning but really took of with the launch of the container orchestrator Kubernetes in 2014. Containers also generated an increase in microservice applications in the world. The cloud native services exploded under the 2010s and in 2018 the Cloud Native Computing Foundation (CNCF) had around 50 members including all major companies in the cloud industry. The cloud native ecosystem gave companies access to software technology that would have taken years to develop in-house.[4] This is just the beginning of the cloud era and what 2020s will bring will be very interesting to see.

1.2 Goal

Distributed tracing is a technology to help debug and monitor distributed applications. This thesis investigates how complex it is to use distributed tracing, as well as quantify the overhead such frameworks add to microservice execution. The maturity of distributed tracing implementations is also be assessed. Maturity in the sense of how well established the framework is in its community, how respected it is as a product and so on.

The research questions are the following:

- How much overhead do tracing frameworks add to microservices?
- How hard is it to implement tracing onto your microservices?
- How mature are the chosen tracing frameworks?

1.3 Report structure

Chapter 2 explains all the basic concepts used in this thesis including what microservices is, how to observe microservices, what distributed tracing is, Kubernetes, Jaeger and Zipkin. Chapter 3 explains how the evaluation is done using the testbed with the TeaStore benchmark, what monitoring tools were used and how to stress test the TeaStore. Chapter 3 describes both the experimental results and the maturity analysis. Lastly Chapter 4 discusses the thesis and some conclusions about the results are drawn.

Chapter 2

Background

This section focuses on all the important technologies and ideas that this Master Thesis approaches. It starts with describing what microservices are and the importance of observability in microservice applications, then leans its focus on distributed tracing which this thesis is mainly about and lastly explain concepts that are used in the evaluation of different distributed tracing frameworks.

2.1 Microservices

Microservices is an architecture style that allows a single application to be split up into smaller parts instead of being one large application. These small parts are loosely coupled and independent of each other. They communicate with themselves using messages over an interface like REST or similar. Each microservice can be seen as its own program. It can be hard to grasp exactly what microservice architecture is but the easy comparison is between a monolith and microservices. A monolith is a large application that is tightly coupled. There is another architecture that is often compared to microservices and that is SOA (service-oriented architecture). The difference between SOA and Microservices are less clear but the main difference is that SOA is a enterprise effort to standardize the way services communicates with each other, it is a way to reuse code without having to rewrite everything while microservices is more application specific. Microservices are not about reusing code, they are more about splitting it up into smaller parts.[5]

The good parts of using a microservice architecture is that it is really easy to update one service of the application without breaking the entire thing. Individual components can also be scaled according to demand. For example, if a bottleneck is identified in a login service, that service can be scaled to ensure more capacity. Understanding of the code is also enhanced because each service is supposed to do only one thing[6].

The downside is that it becomes more complex to communicate between services, it requires more resources to handle many services (multiple databases and so on) and it is harder to test and debug microservices[6].

2.2 Three pillars of observability

It is not a surprise for anyone that observing microservices is hard but there are ways to do it, according to the big companies like Google and Facebook.[7] They use metrics, logging and distributed tracing to observe their systems.

2.2.1 Metrics

Metrics are visual numbers aggregated over time. They are easy for a human to understand and the overhead of metric generation is constant. The amount of data storage does not change over different system loads like logs do. The problem with metrics is that they do not give exact data. A good program to get all kinds of metrics in microservice systems is Prometheus. Prometheus scrapes metrics of applications that allows it. For example Prometheus can scrape the nodes of a Kubernetes cluster to get CPU, memory and more. Prometheus itself is not optimised to display the metrics but there is another program that is, Grafana. Grafana and Prometheus are often used together to visualise good metrics.[8]

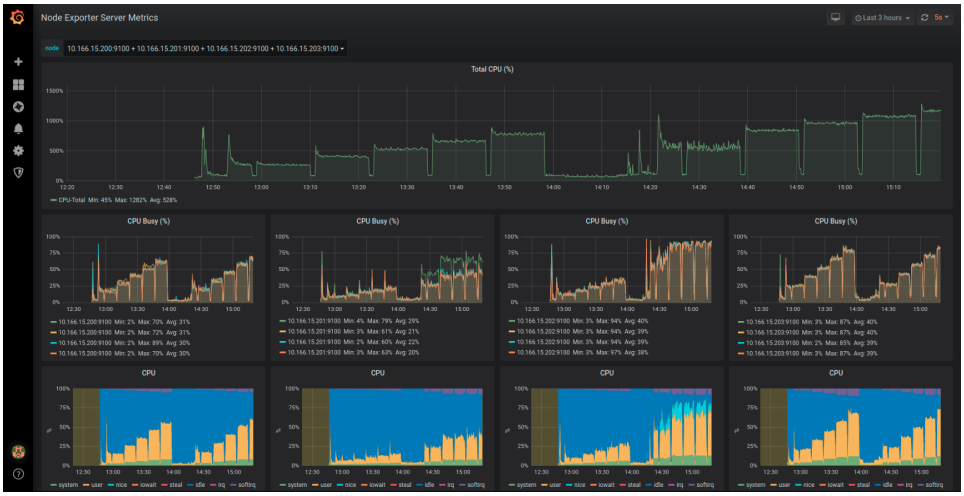

Figure 2.1: Grafana dashboard taken directly from this master thesis when results are gathered.

2.2.2 Logging

Logs can be though of as machine data generated by any application or infrastructure used by that application. They are detailed events of what happens in a system. As they are so detailed they are very good for finding a specific problem. A system under load can generate an enormous amount of logs under a short amount of time. This can be a problem since it can be expensive to store and process these logs. [8]

2.2.3 Distributed tracing

Distributed tracing is what metrics and logging has a hard time to do. It is a method used for monitoring and debugging distributed programs, especially programs built using a microservice architecture. It can help the programmer to pinpoint poor performance locations in the system and to find the root cause of a problem. [9] The architecture of distributed tracing differs depending on what framework you use, but most of them has their origin in the original Dapper paper [10] from Google.

The main question distributed tracing solves is:

- How to understand the behavior of an application and diagnose problems in a distributed system?

The way of solving the problem is :

- Let each request have a unique id.
- Transmit the id to all services that are involved in handling the request.
- Let the request id be a part of all log messages.
- Save information about start and end times of the request.

This information when used correctly allows the programmer follow requests in a microservice execution and visualize it.[11]

2.3 Distributed tracing fundamentals

Currently there exist two standards for implementing distributed tracing, OpenTracing and OpenCensus. These APIs work towards the goal of making tracing available for the masses but the problem that there is two standards still persist. Currently OpenTracing and OpenCensus are planning to merge into one large open-source API, OpenTelemetry but that has not been released yet.[12]

2.3.1 Tracing frameworks

The two tracing frameworks chosen in this thesis is Jaeger and Zipkin. They were chosen because that they are both an OpenTracing implementation which makes it possible to compare them, for example with respect to how hard it is to implement two different tracing frameworks but still see some similarities due to OpenTracing. Jaeger and Zipkin are also opensource and has over ten thousand Github stars each.[13][14] They are both OpenTracing implementations so the rest of the thesis focuses on that implementation.

2.3.2 Spans

A span is the most essential part of distributed tracing. It represent a particular unit of work in a distributed system. It can be thought of like the workflow of a specific component in the system.[15]

According to the OpenTracing specification, a span consist and encapsulates these states:

- An operation name
- A start timestamp and finish timestamp
- Tags
- Logs
- A span context

The operation name is self explanatory, it is the name of the operation executed by the distributed system and the same goes for the **timestamp**. It is the start time and end time of the operation.[15]

Tags are key;value pairs that apply for the whole span and it allows a user to filter and search spans using this tag. It is useful because with many spans it is necessary to be able to comprehend the data available. An example is the tag ***http.status_code*** which represent a http response code, for example 404.[15] There is a convention for writing tag names so that the user easily knows what to search for. [16]

Logs can be thought of as the opposite of tags because tags apply to the whole span and logs do not. Logs are also key;value pair that are useful for documenting a specific event within the span.[15]

The span contexts' purpose is to carry data over different processes. A span context consist of two things, an implementation dependant state and baggage items. The dependant state refers to the distinct span within a trace, i.e., the implemented definition of trace id and span id. The baggage item is a key;value pair that can cross processes. This may be used to access some data across a whole trace.[15]

2.3.3 Trace

Spans often contain a reference that refer to a parent span. The first created span is often called the root span. This reference allows multiple spans to be grouped and compared to each other which is called a trace. The trace can be described as a visualisation of the life of a request sent through a distributed system.

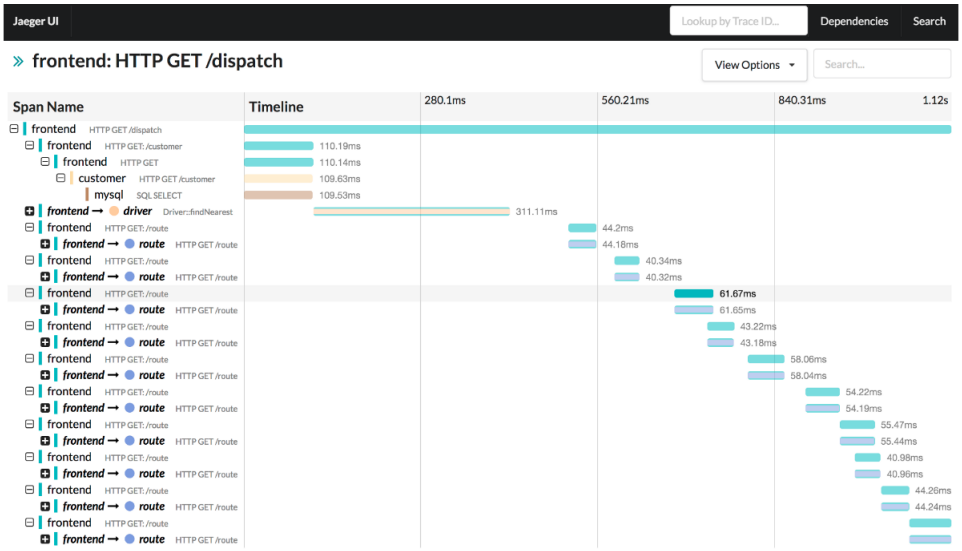

Figure 2.2: A normal Jaeger trace from the application HotRod. Taken from a medium post describing how to use HotRod and Jaeger [17].

2.4 Kubernetes

Kubernetes (also known as K8s) is an open source platform for container orchestration that automates a lot of the tasks in containerization that require human effort. For example, managing, scaling and deploying containerized applications[18].

Kubernetes originated from Googles internal platform Borg and many of the developers on Borg also worked on Kubernetes. Since they already had developed one orchestrator the lessons learned from developing Borg over the years became the primary influence when developing Kubernetes[19].

2.4.1 Containers

To understand why a container orchestrator is needed today, it is important to understand the differences between how applications used to run and how they can be run these days in containers.

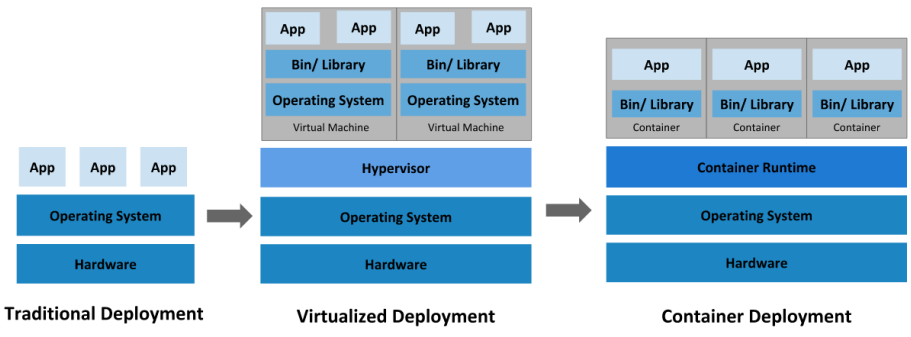

Figure 2.3: Picture of the differences between traditional deployment, virtual deployment and container deployment from Kubernetes own website [20].

Traditional deployment (shown in figure 2.3) is when an organisation uses physical servers to run their applications. This way has its fair share of problems and one of the most crucial is that there is no way to define what resources an application can use on the server. For example, if two different applications run on one server and one of the applications start to hog more resources than it normally uses, then the other application underperforms. The not so good, but commonly used, solution to this is to use two servers and run the different applications on each server. This has its own problems as this does not scale well. Resources can be underutilised and it is also very expensive for companies to run many physical servers. [20]

Virtualized deployment (shown in figure 2.3) is a solution to the problems that the programmer faces when using traditional deployment. It creates the opportunity to run different virtual machines (VMs) on a single server. These VMs uses the same CPU but are isolated from each other. This allows for greater security as applications that run on two different VMs on the same server cannot communicate directly. It also enhances the resource management on the server as the VMs resources can be set to what the application needs.[20]A thing to notice with VMs is that each VM run its own operating system (OS) and the hypervisor is what allows multiple OSs to run on the same machine at the same time. It is the hypervisors job to decide what system resources the different OSs can use.[21]

Container deployment (shown in figure 2.3) is a lightweight way to separate applications from each other on the same machine. It is lightweight because containers have focused less on total separation as a container does not require an OS but instead relies on the underlying OS to provide the basic services to the container. This allows for more space on each container for the application. Although it differs from VMs in the way that there is no OS they still have a lot of similarities. They both uses virtual memory, CPU, process space and more. The things that have made containers popular is the benefits they provide:

- It is easy to create and deploy a container compared to a VM image.
- The environmental stays the same on different computers. It runs the same on your local PC as it does in the cloud.
- It is perfect for microservices as every small piece of an application can be deployed independent.

There are many more advantages and all these together is why containers these days are in the center of distributed programming.[20]

2.4.2 Kubernetes components

A Kubernetes cluster is split into two components, the worker nodes and the control plane.

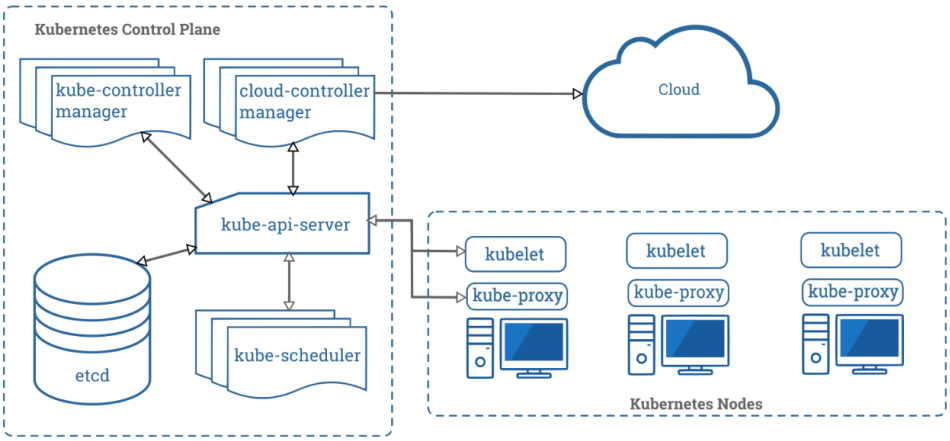

Figure 2.4: Image of a typical Kubernetes cluster taken from Kubernetes own documentation [22]

The worker nodes hosts the pods which are the components of the application and there must always be one worker node available[22].The kubelet agent controls the pods in each node. This agent uses a PodSpec which is a specification file in the form of YAML or JSON. The PodSpecs task is to describe the pod itself and make sure that the described containers are is healthy and running.[23] The second important part in each worker node is the kube-proxy. The kube-proxy implements the service concept in Kubernetes.[22] The service concept in Kubernetes is a way to separate a pod from its network components. It is through a service it is defined what type of internet access the pod should be given. The three most common types of network access a service can have is:[24]

- **ClusterIP** - The service can only access other pods within the cluster.
- **NodePort** - Exposes the service on a static NodePort. This allows for traffic outside the cluster.
- **LoadBalancer** - Exposes the service outside of the cluster through a cloud provider load balancer.

The control plane is the master of the Kubernetes cluster. It is the part of the cluster that makes the universal decisions such as where a pod should be scheduled, when to start a pod and so on. The control plane can be run on any machine in the cluster and all parts of the control plane are started on the same machine.[22] One of these parts is the kube-api-server. The kube-api-server serves as the front of the control plane. Its task is to validate and configure data from pods, services and so on. It is the connection point between the worker nodes and the control plane. It communicates using REST and all the other parts inside the control plane communicates with the kube-api-server.[25]

The kube-scheduler is the part in the control plane that assign pods to appropriate nodes. The kube-scheduler takes into consideration pods individual settings such as resource requirements, software constraints, data locality and more. The

etcd is a key-value store that Kubernetes uses to store important cluster data. It implements the RAFT consensus algorithm for data replication. A consensus algorithm is a process to achieve agreement on data value among distributed system. There are two more parts in the control plane that needs explanation, the kube-controller manager and the cloud-controller manager. These two, work similarly as they both are compiled into a single binary and run in a single process but their tasks are different. The kube-controller manager operates Kubernetes specific controller processes such as Node controller (checks the health of the nodes), Replication controller (responsible for maintaining the correct number of pods according to the number of replications) and more. The cloud-controller manager have similar tasks. It has a Node controller as well but it is responsible of checking if a node has been deleted from the cloud or stopped working.[22]

2.5 Jaeger

Jaeger is a distributed tracing framework developed by Uber Technologies.[26] It is open source and it is a CNCF graduated project and is mostly developed in Go. The first commit on Jaeger was in 2016.[13]

The architecture of Jaeger consist of the Jaeger client, agent, collector query and a database.

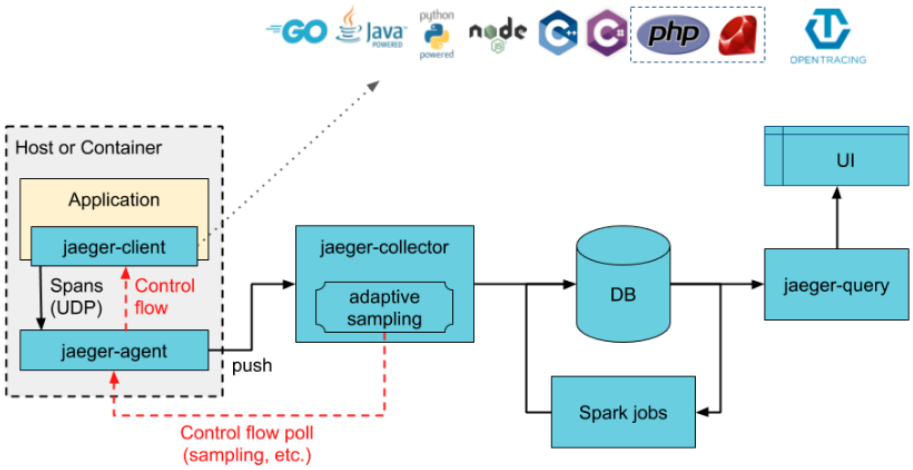

Figure 2.5: Image explaining the architecture of Jaeger taken from Jaeger's website.[27]

2.5.1 Jaeger Client

The Jaeger client is an implementation of Open Tracing and there exists many different clients for different programming languages, such as Java, Go, Python and more. The Jaeger client is responsible for creating spans for each request and sending it to the Jaeger Agent. Tracing every request to an application increases the overhead, in particular if there are many requests. Therefor most tracing systems (Jaeger included) allows a setting called sampling. Sampling is the fractions of requests. The default sampling rate in Jaeger is 1/1000, i.e., Jaeger traces one in thousand requests unless specified different.[27]

2.5.2 Jaeger Agent

The Jaeger Agent is a network deamon responsible of collecting spans from the Jaeger Client via UDP, batch them together to a trace and sending them to the Jaeger Collector. In Kubernetes the Jager Agent is deployed as a deamonSet or as a sidecar.[27] A deamonSet is a protocol that ensures that there always is one pod on each node. If a node is added or deleted from the cluster the pod is added or removed.[28] In contrast, a sidecar is a pod that is deployed alongside another pod to help it out in different ways. In the Jaeger example the sidecar would be deployed in company with the application which should be traced.[29]

2.5.3 Jaeger Collector

The Jaeger Collector has a processing pipeline that runs traces received from the Jaeger Agent. The pipeline validates, indexes and at the end saves the traces in the attached database. The database can be either an in memory, Cassandra or Elasticsearch database.[27]

2.5.4 Jaeger Query

The Jaeger Query is the UI in which traces can be watched by a person. See Figure 2.2 for one example of how Jaeger Query displays a trace. A trace can be either be gathered by service, operation or tags. This allows for easy trace lookup.

2.6 Zipkin

Zipkin is another distributed tracing framework originally developed by Twitter and presently managed by the OpenZipkin Community.[30] It is mostly developed in Java and JavaScript and its first commit occured in 2012 which means Zipkin was released before Jaeger.[14]

Zipkin's architecture is made up by the Zipkin Collector, an API called Zipkin Query Service and the Zipkin UI. Zipkin is also dependent on a Zipkin Reporter and a storage to place traces.

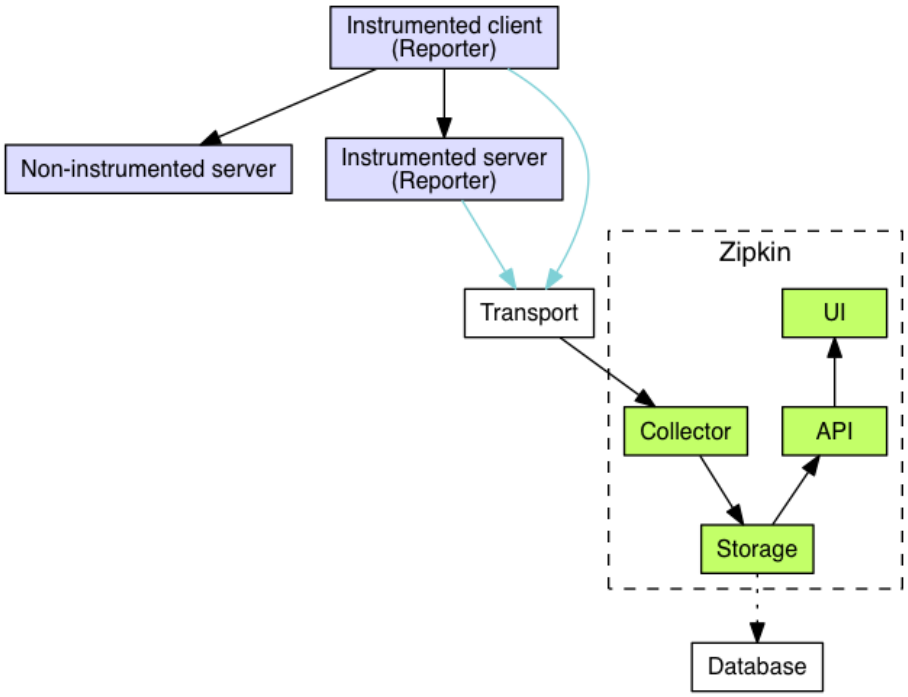

Figure 2.6: Image explaining the architecture of Zipkin taken from Zipkin's website.[31]

2.6.1 Zipkin Reporter

The Zipkin Reporter is the tracing instrumentation of an application. Depending on programming language there exist a lot of Reporters. There are seven that are officially supported by Zipkin and many more supported by the community. One example of a Zipkin Reporter is Brave (Java Reporter).[32] Brave works with JRE6+ and it gathers tracing data such as timing data, span contexts and so on. Reporters forward data to the Zipkin Collector.[33]

2.6.2 Zipkin Collector

The Zipkin Collector is a deamon that collects traces generated by the Reporters. It validates traces, indexes them and stores them into a database. The databases that Zipkin recommends using is Cassandra, ElasticSearch and MySql.[31]

2.6.3 Zipkin API

There needs to be an easy way to extract traces from the database to the UI. This is the purpose of the Zipkin API. This is deamon that exposes a simple JSON API which retrieves traces.[31]

2.6.4 Zipkin UI

The Zipkin UI is where traces can be observed. It is very similar to the Jaeger Query in what information can be seen but the Zipkin UI is a bit different in its looks and usage.

Chapter 3

Evaluation

This chapter is about how the different tracing frameworks is used in the testbed and also what kind of test application was used for evaluation. The way the test application was tweaked to work with tracing is also be assessed. It also contains a part where each distributed tracing framework is objectively assessed on how easy it is to use, its documentation and more. In other words how mature that framework is.

3.1 Used benchmark: TeaStore

The application used in the experimental evaluation is called TeaStore. It is a microservice application specifically built for benchmarking and testing. Its primary task is to emulate a basic tea store in which a user can browse, buy and do all other things a normal user would be able to do in a web shop.[34]

The TeaStore consists of 5 distinct microservices and one registry.

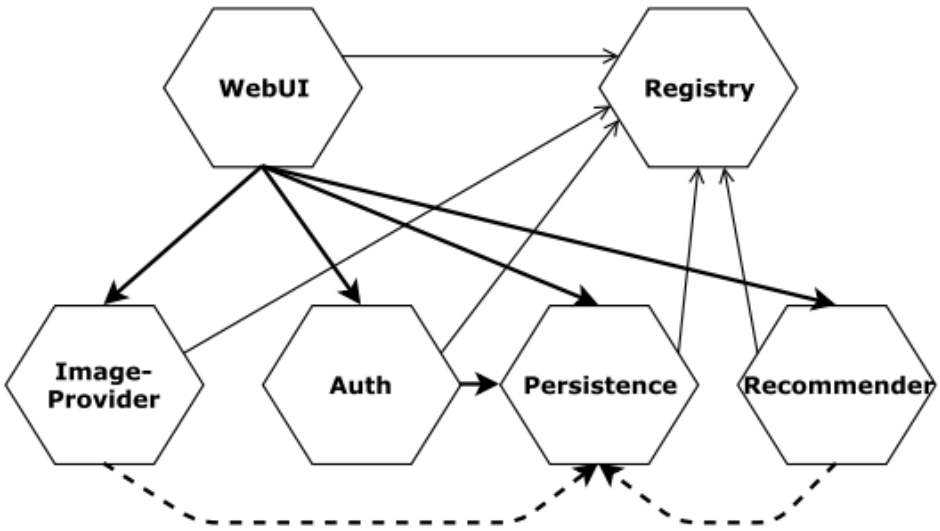

Figure 3.1: Image explaining the architecture of the TeaStore taken from the paper.[34]

3.1.1 WebUI

This service is where the user interface is provided. Its primary focus is to supply a nice looking UI with frontend operations. One of its tasks is to carry out validation checks on the user input before passing this input to the Persistence service. All the data, teas, categories of teas, images and more is provided to WebUI from both the Image Provider and Persistence service[34].

3.1.2 Image Provider

The Image Provider, when queried, provides images of different sizes to the WebUI depending on the WebUi's target size. It does this using an internal cache that stores the image with the target size and returns it if available. If not, the Image Provider takes the largest image of corresponding product or category, scales it to the requested target size and enters it in the cache. This leads to that the response time of the Image Provider depends on whether an image is cached or not[34].

3.1.3 Authentication

This service handles login and session data for the user. It uses BCrypt for the login verification and SHA-512 hashes for the session data. The session data carry information like the shopping cart, old orders and if the user is logged in or not. This means that the authentication service performance depends on how much of that information is available[34].

3.1.4 Recommender

The recommender is a CPU expensive part of the TeaStore as it uses a rating algorithm to evaluate what a customer might want to buy depending on products in a users shopping cart, what other users bought and on what product the customer is viewing right now. When a new TeaStore is started the Recommender uses a generated dataset from the Persistence service for training. If there are more than one instance of a Recommender service the new Recommender obtains its training data from an existing Recommender. This ensures that the recommended data stays identical over different recommenders. This is important because when a TeaStore is used for benchmarking it is crucial that the performance stays the same[34].

3.1.5 Persistence

It is the persistence service that have access to TeaStore's relational database. The database is of the type SQL and it stores all the products, categories, purchases and users. When starting a persistence service for the first time it populates the database with all necessary data. The persistence service also uses caching to decrease load on the database and improve stability[34].

3.1.6 Registry

The registry is special because it has no function under benchmarking but it serves another important role. It is a support service for the TeaStore. It keeps track of all IP addresses, if a service is online, what ports are available and host names. All the other services in the TeaStore sends a heartbeat to the registry to show that they are alive and well. If one dies, the registry will remove it from the available service list which all other services can query at any time[34].

3.2 Testbed

The testbed used in this thesis consists of a modified TeaStore, a Kubernetes cluster, monitoring using Prometheus and Grafana, Jaeger, Zipkin and JMeter for stress testing the TeaStore which is all run inside Google Kubernetes Engine (GKE) with 4 nodes and the vCPUs are N1-Standard-4 which has 15 GB memory per node.

3.2.1 Modified TeaStore

The TeaStore comes with some tracing out of the box if used with the service mesh Istio.[35] This is not ideal in this thesis because the main focus is to quantify the overhead tracing frameworks add to microservice execution. By letting the service mesh proxy handle the tracing it would create extra overhead and thus impact the results. Therefor TeaStore had to be modified to use tracing without Istio. In this case with Jaeger and Zipkin.

The TeaStore comes with pre built docker images for running them on a Kubernetes cluster. To modify the TeaStore, the images have to be rebuilt and then added to a container registry in which the cluster can fetch the images. This is done using the Google Container Registry because then it is easy for a GKE cluster to fetch the images as it is part of the same project on Google Cloud. The modified TeaStore can be found here[1]

3.2.2 Tracing TeaStore

The TeaStore uses the functionality of a GlobalTracer which acts as a global instance of a tracer that can be acquired anywhere in the program.[36] This GlobalTracer is initiated once for every service in the TeaStore. Depending on which tracer is used there exists two different methods to create a tracer that should be used as a GlobalTracer, createJaegerTracer (3.1) and createZipkinTracer (3.2).

[1] https://github.com/Dzora/TeaStore

createJaegerTracer

Listing 3.1: createJaegerTracer

```
public static Tracer createJaegerTracer(String arg1,
    String arg2, String service) {

    SamplerConfiguration sampleConfig =
        SamplerConfiguration.fromEnv().withType("const")
        .withParam(1);

    SenderConfiguration senderConfig = new
        SenderConfiguration().withAgentHost(arg1)
        .withAgentPort(Integer.decode(arg2));

    ReporterConfiguration reporterConfig =
        ReporterConfiguration.fromEnv().withLogSpans(false)
        .withSender(senderConfig);

    Configuration config = new
        Configuration(service).withSampler(sampleConfig)
        .withReporter(reporterConfig);

    return config.getTracer();
}
```

createZipkinTracer

Listing 3.2: createZipkinTracer

```
 public static Tracer createZipkinTracer(String arg1,
     String arg2, String service) {

    Tracer tracer;

    Sender sender = OkHttpSender.create("http://" +
        arg1 + ":" + arg2 + "/api/v2/spans");

    AsyncReporter<zipkin2.Span> reporter =
        AsyncReporter.builder(sender).build();

    tracer =
        BraveTracer.create(brave.Tracing.newBuilder()
        .localServiceName(service).spanReporter(reporter)
        .build());

    return tracer;
}
```

When using Jaeger there are a few requirements and some optional settings that can be applied. First of all, the Jaeger Client needs to know where to send the traces over UDP. This is done by using the senderConfiguration function (Listing 3.1). It takes arguments like agentHost and agentPort. In Kubernetes the Jaeger Agents are deamonSets and thus are applied on each node the cluster. Kubernetes

deploys TeaStores services across the available nodes as it sees fit, evenly on the nodes. That creates some randomness. So to know which host the JaegerAgent should have, there exists an environment variable in the YAML file where TeaStore is deployed. This sets the environment variable (JAEGER_AGENT_HOST) to whatever IP the node has.

```
- name: JAEGER_AGENT_HOST
  valueFrom:
    fieldRef:
      fieldPath: status.hostIP
```

Zipkin works similarly as Jaeger so TeaStore needs to know where to send the traces. To achieve this there is a sender function (Listing 3.2) that sets this information from a Java properties file.

To easily switch between Jaeger, Zipkin and no tracing, when the TeaStore is running in the Kubernetes cluster a few modifications were made to avoid having to build new docker images every time. The Java properties file exists to easily swap between tracers when running locally and when running in the cluster this properties file exists as an environment variable (TRACER_CONFIG_PROPERTIES) in a configMap.

```
kind: ConfigMap
apiVersion: v1
metadata:
    name: tracing-config
data:
    TRACER_CONFIG_PROPERTIES: |
        // Selector for the below config blocks
        tracer=zipkin
        // Jaeger config
        jaeger.reporter_host=jaeger
        jaeger.reporter_port=6831
        // Zipkin config
        zipkin.reporter_host=zipkin
        zipkin.reporter_port=9411
```

3.2.3 Monitoring

To monitor the whole cluster with the TeaStore on it, Prometheus and Grafana where chosen. Prometheus are deployed with NodeExporter which exposes many interesting kernel metrics like CPU, memory, network and more.[37] Prometheus scrapes metrics every 5th second and sends it to Grafana where it is displayed in 4 important graphs. Total CPU, total memory, total network received and total network transmitted. The total CPU displays the active time of all CPUs in the cluster in percentage. Since it is calculated in total each CPU will be added together thus the max percentage the total CPU graph can show is ((number of virtual CPUs per node)* 100)). In the test case there was 5 vCPUs and 4 nodes so a total of 1600%. In the case of the total memory, total network received and total network transmitted the results are in Bytes.

3.2.4 Stress testing

To simulate a high load of request per seconds on the TeaStore, JMeter[38] was chosen because that is what the official TeaStore wiki recommended. There already existed a (simulate user) script which is used in this thesis. The script consist of 8 different requests to the TeaStore, Home, Login, List Products, Look at Product, Add Product to Cart, List Product with different page, Add Product 2 to Cart and Logout. The exact details of these requests are irrelevant but what is interesting is that they are distinctive operations that takes a different amount of time. JMeter has a option to allow each user to generate requests with a static throughput. This is set to 5 requests per second. This is to not overload the TeaStore. From cold start, it takes 1 minute to reach this throughput which has to be taken into account when testing.

3.3 Evaluation method

To evaluate Jaeger and Zipkin the testbed is used inside a Kubernetes cluster. This means that all components are inside the cluster including JMeter. This is to remove any outside interference which occurs if for example JMeter runs outside the cluster. Each test is run 5 times with different user counts (20, 40, 60, 80, 100 users) and is run for 11 minutes each. The first minute is to make the throughput static and will not be included in the results thus making the test 10 minutes long. There is also a 5 min warmup before each test. This is to ensure that the TeaStore is ready (caches are populated, ect.) and works as well as possible when testing.

The setup for the tests is:

1. Start a GKE cluster with 4 nodes (N1-Standard-4: 4vCPU, 60GB Memory).
2. Taint the nodes to ensure identical pod deployment between runs.
3. Start monitoring.
4. Make a loop.

Foreach tracing method (none, Jaeger, Zipkin):
- Start TeaStore with selected tracer
- Run JMeter inside the cluster to generate load
- Get the results from JMeter and Grafana
- Remove tracer

3.4 Test results

The results from the tests can be summed up in five categories: latency, CPU, memory, network received and network transmitted. Latency is measured for each of the 8 requests types the JMeter script was using. Here, only the Home request type is shown, the rest are in the appendix.

3.4.1 Latency

The latency graph consists of five different user counts on the X axis and latency in milliseconds on the Y axis. The X axis remains the same in each graph henceforth. The boxplots show 99% of the data inside their spans. There existed a few outliers that where to extreme so those were removed in favor of keeping the graph somewhat compressed. Each boxlot consist of many thousands of values as each call from JMeter was counted. (5 calls per user every second in 10 minutes).

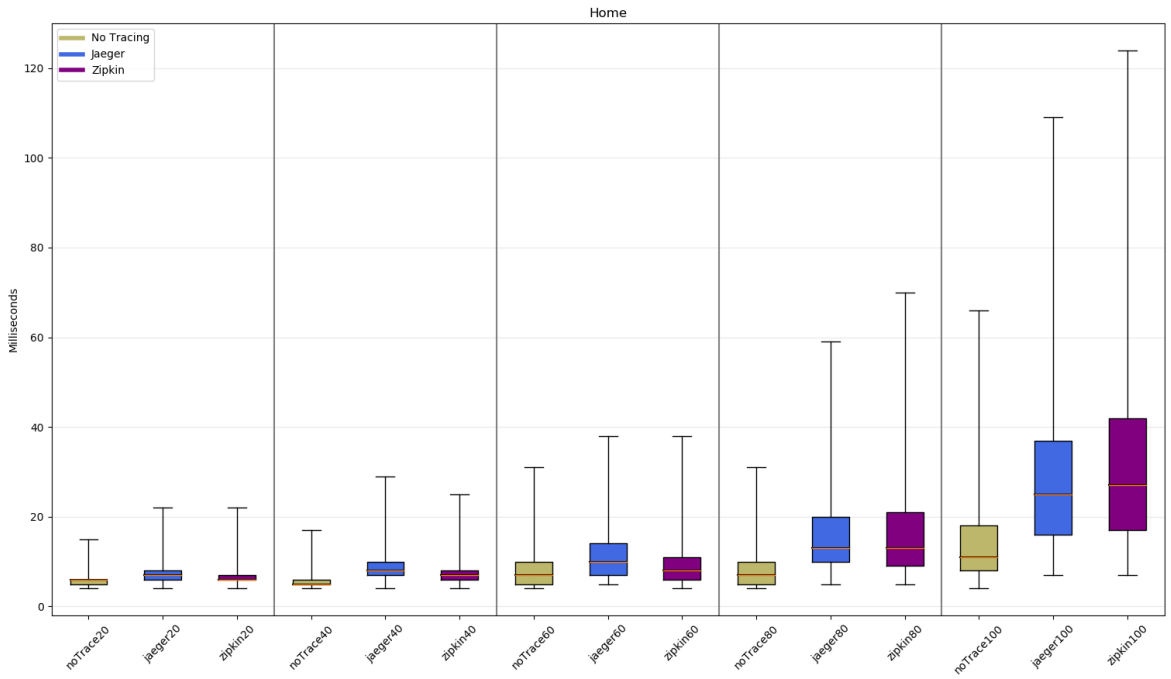

Figure 3.2: Latency of the Home request.

No tracing had the lowest latency, followed by Jaeger with Zipkin a close third. The difference was most prominent for 100 users.

3.4.2 CPU

The CPU graphs boxplots does not consist of as many values as the latency plots. These values are taken from Grafana and exported to a CSV. There should be around 120 values in each boxplot since Prometheus scrapes every 5 seconds and the test lasted for 10 minutes. The Y axis in this plot is in total precentage. As the cluster had 4 nodes with N1-Standard-4 (4 vCPU) there is a total of 16 CPUs in the cluster. This makes the Y axis have a max of 1600%.

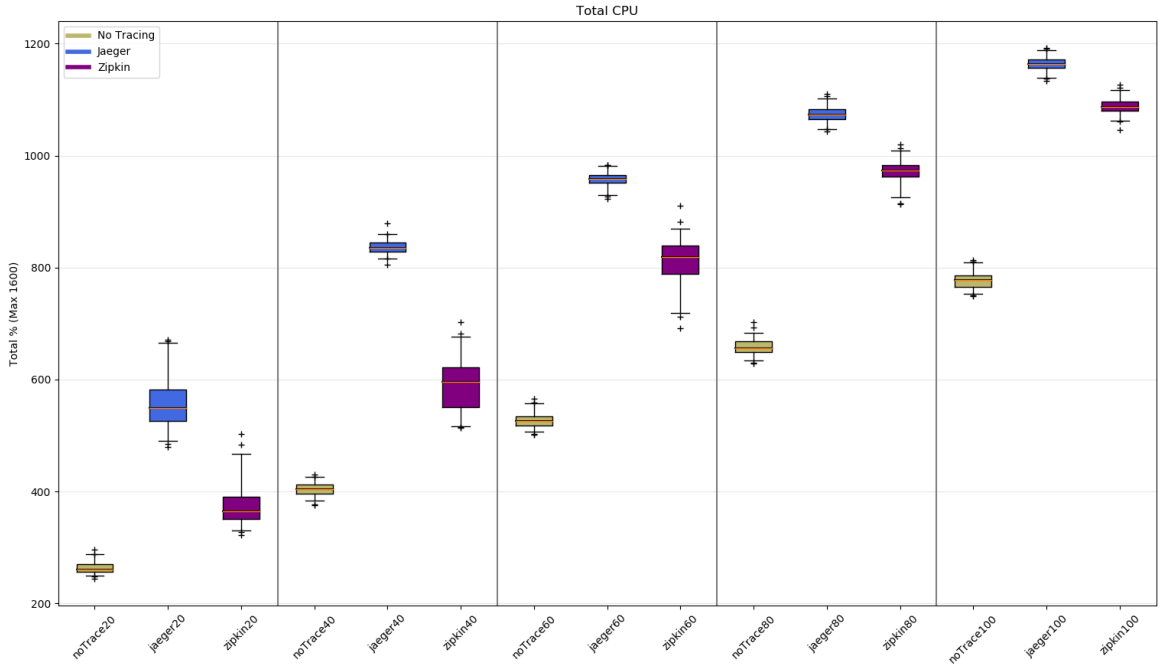

Figure 3.3: Total CPU in percent

No tracing had the lowest CPU percentage followed by Zipkin with Jaeger dominating this field.

3.4.3 Memory

Memory is also taken directly from Grafana. It is the total memory in the cluster. The actual value that Grafana outputted for memory was in Bytes. This is later converted into percentage knowing that the total memory in the cluster is 60GB. This makes the Y axis in this graph have a max value of 100%.

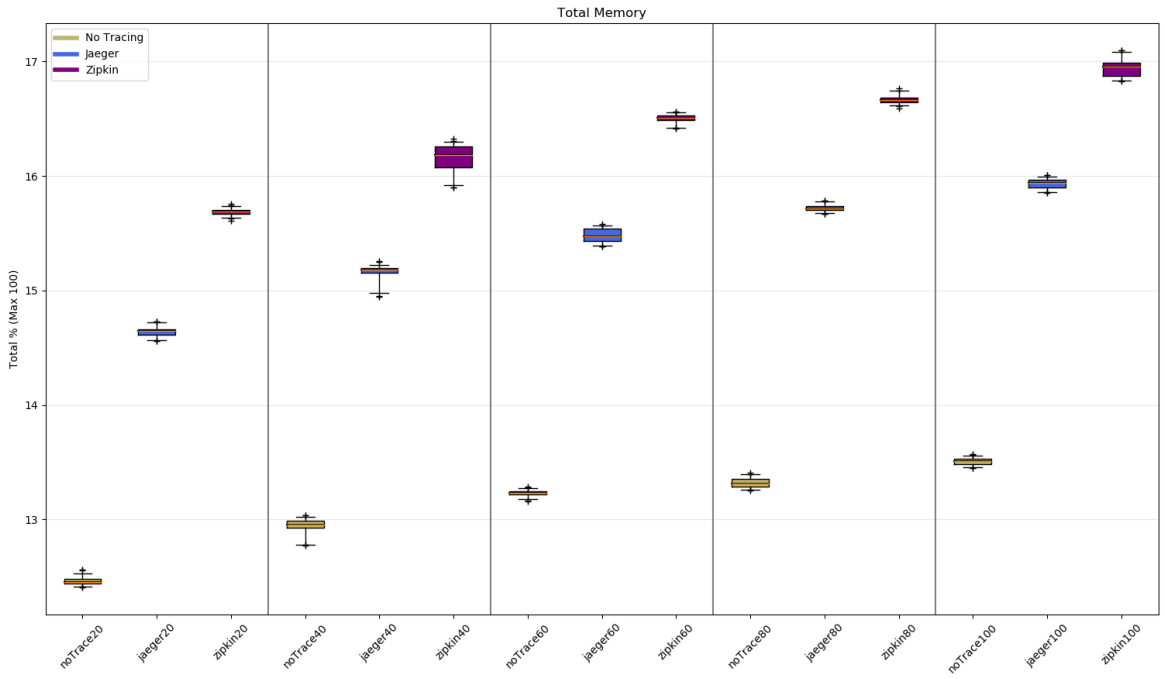

Figure 3.4: Total Memory usage in percent

No tracing has the lowest memory consummation followed by Jaeger with Zipkin as the highest memory consumer.

3.4.4 Network

These two graphs is total network received and total network transmitted over time. Its Y axis is in Bytes and this can not be converted into percent because there is no way of knowing the total network capacity in the cluster. They are very similar but not identical.

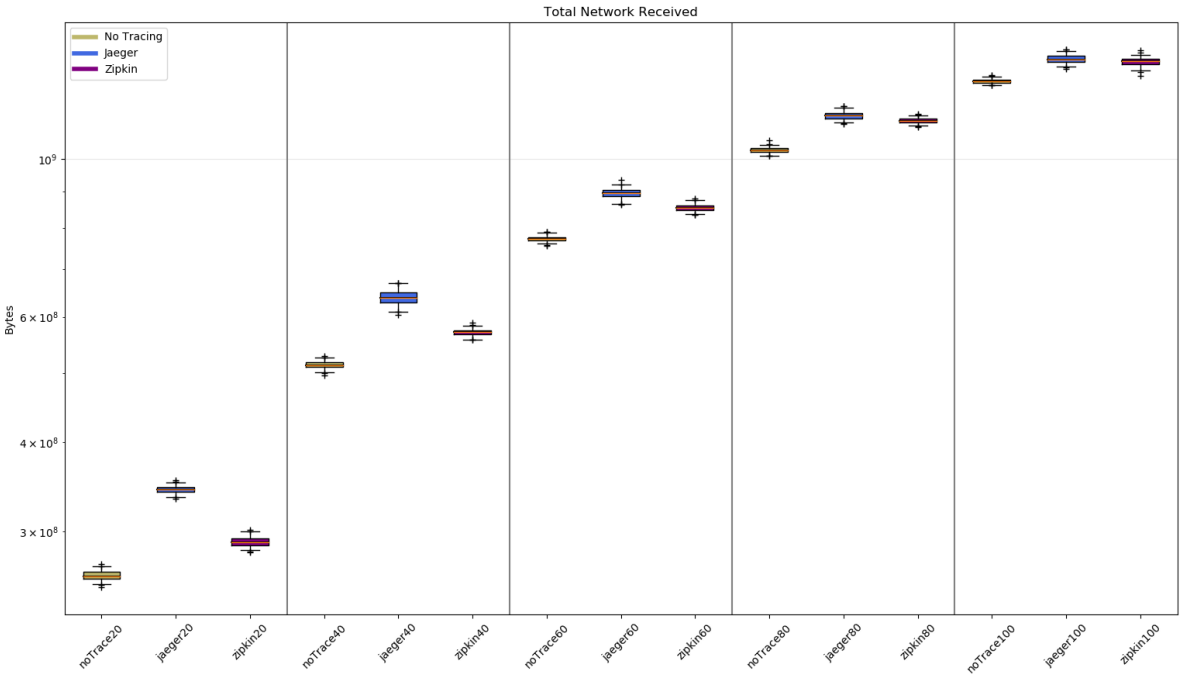

Figure 3.5: Total Network Received

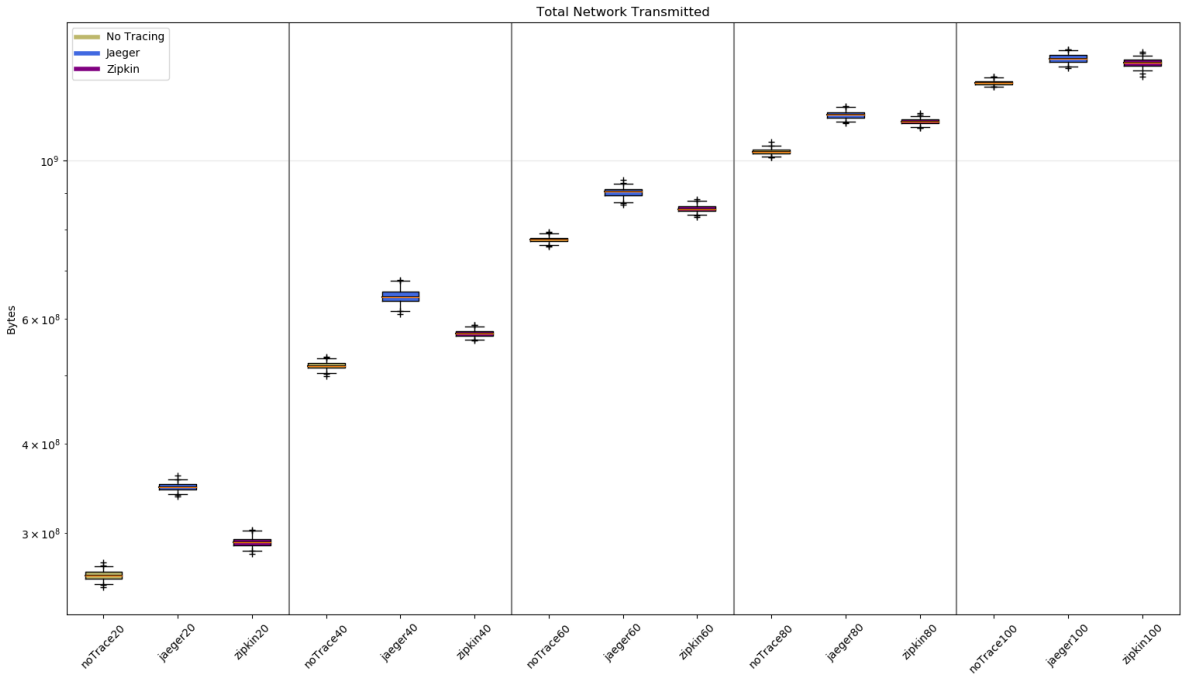

Figure 3.6: Total Network Transmitted

No tracing draws the lowest network capacity with Zipkin as second and Jaeger as a close third in both received and transmitted.

3.5 Maturity

The two distributed tracing frameworks, Jaeger and Zipkin are both open source. They both have five digit Github stars, Jaeger has 10730 and Zipkin has 12722. Jaeger has 134 contributors while Zipkin has 85.[13][14] CNCF has a project level system that ranks their projects in three categories, sandbox, incubated and graduated. There are a lot of different criteria for moving up a level but in short, sandbox is a not necessary a new project but might be, incubated are projects that are used to some extent in the industry and follows the CNCF code of conduct and graduated projects are full fledged projects that are widely used and fills all criteria CNCF has posted[39]. Jaeger is a graduated project which shows that Jaeger is very mature. Zipkin is not trying to be a part of these ranks and thus it is not even sandbox. This does not mean that it is worse, just that Zipkin did not focus on CNCF.

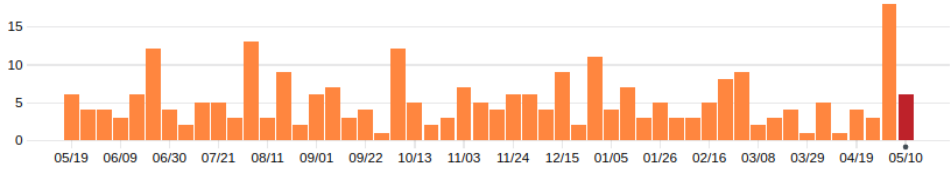

Figure 3.7: Jaeger commits in the last year taken from Jaeger's git[40]

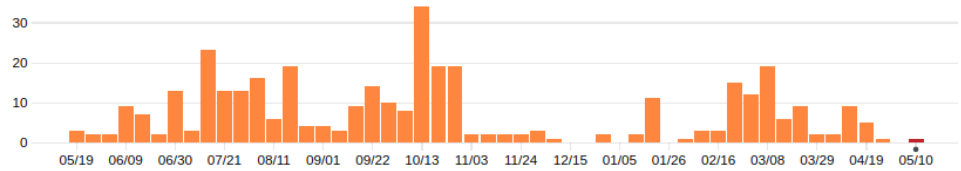

Figure 3.8: Zipkin commits in the last year taken from Zipkin's git[41]

Jaeger has a total of 18 open and 52 closed bugs and Zipkin has 9 open and 103 closed bugs as of May 13. It takes the Jaeger community an average of 12.45 days to solve a bug while it takes the Zipkin community an average of 23.35 days. This data is calculated from the last 20 bugs in their Git history.[42][43]

Chapter 4

Discussion & Conclusion

This chapter discusses the results gathered in Evaluation and draw some conclusions from it. It gives some thoughts on what could have been done different and what future work can be applied based on the thesis. Some parts as which tracer is easier to use will be an objective drawn conclusion from the implementation of each tracer.

4.1 Jaeger vs Zipkin

Jaeger provided more extensive documentation and resulted in an easier implementation compared to Zipkin, which has some outdated documentation with regards to their current API. A prime example of this is the documentation on how to run each tracer in Kubernetes. Jaeger provided a already completed YAML file to run it in the orchestrator but Zipkin did not.

In regards to latency each tracer added to each request the results are inconclusive. A hypothesis test is very hard to do in this situation to show a statistical difference since latency is not normally distributed. It is clear that tracing adds some overhead to the latency but whether Jaeger or Zipkin is better is hard to tell.

It is rather interesting to see that Jaeger uses more CPU in total than Zipkin. This might be because of the Zipkin architecture consist of fewer services than Jaeger. Jaeger in this setup has 1 agent per node in total of 4 agents while the Zipkin client sends data directly to the collector.

In the case of memory consumption, Zipkin uses more memory than Jaeger and that is quite expected as Zipkin has some extra headers in each trace. When tracing all requests with no sampling over a long time this will make a significant difference.

Lastly there is the cases of received traffic and transmitted traffic in the cluster. These results show that Jaeger sends a bit more than Zipkin but it's not really a big difference. Most likely this is because of the same reasons as why Jaeger uses more CPU, the agents. Jaeger sends all the data to a agent which then sends it on to the Jaeger Collector which should lead to higher network load.

When considering all these results and also the results of the maturity part Jaeger seems like the best choice because of being more active, in both documentation,

bug fixing and having almost as many Github stars as Zipkin despite being almost 4 years younger. Jaeger is also a CNCF graduated project and was much easier to use in Kubernetes than Zipkin. The drawbacks is that is uses a bit more CPU than Zipkin but that is an worthy sacrifice to make.

4.2 Summary

The conclusions from this thesis is that it is very good to utilize a distributed tracing framework for your microservices. It is also very good to use Kubernetes to manage the containers as it is a easy and powerful tool. As there are no big drawbacks the conclusion is to always use a container orchestrator when working with containers. That Jaeger is the better tracer overall is an objective conclusion drawn from the results from the testbed and maturity but depending on the situation there exist times when Zipkin might be the better choice. For example when CPU is of great importance.

4.3 Future work

OpenTelemetry is a CNCF sandbox project that will merge the existing APIs for tracing such as OpenTracing and OpenCensus into one project. When it is released and a framework that uses that API comes forth an possible extension of this Master Thesis is to include that framework in the tests. It should be interesting to see if the new API is better or worse than the old standards using the evaluation method used in this evaluation.

References

[1] aws.amazon.com. About AWS. https://aws.amazon.com/about-aws/ (visited on 2020-05-24).

[2] googleappengine.blogspot.com. Introducing Google App Engine + our new blog. http://googleappengine.blogspot.com/2008/04/introducing-google-app-engine-our-new.html (visited on 2020-05-24).

[3] Ray Ozzie. Ray Ozzie announces Windows Azure. https://www.zdnet.com/article/ray-ozzie-announces-windows-azure/ (visited on 2020-05-24).

[4] Esther Shein. The most important cloud advances of the decade. https://www.techrepublic.com/article/the-most-important-cloud-advances-of-the-decade/ (visited on 2020-05-15), November 2019.

[5] Ibm.com. Microservices. https://www.ibm.com/cloud/learn/microservices (visited on 2020-05-11).

[6] Joe Nemer. Advantages and Disadvantages of Microservices Architecture. https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/ (visited on 2020-05-11).

[7] CNCF [Cloud Native Computing Foundation]. Three Pillars, Zero Answers: We Need to Rethink Observability - Ben Sigelman, LightStep. https://www.youtube.com/watch?v=EJV_CgiqlOE&list=PLwb6qBrEgFgOCOyAsVci18R0uWSqaP68g (visited on 2020-03-08), December 2018.

[8] Cengiz Han. 3 Pillars of Observability. https://medium.com/hepsiburadatech/3-pillars-of-observability-d458c765dd26 (visited on 2020-05-04).

[9] OpenTracing.io. What is distributed tracing? https://opentracing.io/docs/overview/what-is-tracing/ (visited on 2020-03-08).

[10] Benjamin H. Sigelman and Luiz André Barroso and Mike Burrows and Pat Stephenson and Manoj Plakal and Donald Beaver and Saul Jaspan and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

https://research.google.com/archive/papers/dapper-2010-1.pdf (visited on 2020-03-08).

[11] Microservices.io. Pattern: Distributed tracing. https://microservices.io/patterns/observability/distributed-tracing.html (visited on 2020-05-19).

[12] Kevin Chu. OpenTracing, OpenCensus, OpenTelemetry, and New Relic. https://blog.newrelic.com/engineering/opentelemetry-opentracing-opencensus/ (visited on 2020-03-08).

[13] CNCF.io. Jaeger. https://landscape.cncf.io/category=tracing&format=card-mode&grouping=category&selected=jaeger (visited on 2020-05-05).

[14] CNCF.io. Zipkin. https://landscape.cncf.io/category=tracing&format=card-mode&grouping=category&selected=zipkin (visited on 2020-05-07).

[15] OpenTracing.io. Spans. https://opentracing.io/docs/overview/spans/ (visited on 2020-03-10).

[16] OpenTracing.io-Git. Semantic Conventions. https://github.com/opentracing/specification/blob/master/semantic_conventions.md (visited on 2020-03-10).

[17] Yuri Shkuro. Take OpenTracing for a HotROD ride. https://medium.com/opentracing/take-opentracing-for-a-hotrod-ride-f6e3141f7941 (visited on 2020-05-04).

[18] Red Hat, Inc. What is Kubernetes? https://www.redhat.com/en/topics/containers/what-is-kubernetes (visited on 2020-04-14).

[19] Kubernetes.io. Borg: The Predecessor to Kubernetes. https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/ (visited on 2020-04-14).

[20] Kubernetes.io. Going back in time. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 2020-04-14).

[21] IBM.com. hypervisors. https://www.ibm.com/cloud/learn/hypervisors (visited on 2020-04-14).

[22] Kubernetes.io. Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/ (visited on 2020-04-14).

[23] Kubernetes.io. Kubelet. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/ (visited on 2020-04-14).

[24] Kubernetes.io. Service. https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 2020-04-14).

[25] Kubernetes.io. Service. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/ (visited on 2020-05-04).

[26] JaegerTracing.io. Introduction. https://www.jaegertracing.io/docs/1.17/ (visited on 2020-05-05).

[27] JaegerTracing.io. Architecture. https://www.jaegertracing.io/docs/1.17/architecture/ (visited on 2020-05-05).

[28] Kubernetes.io. DeamonSet. https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/ (visited on 2020-05-07).

[29] Scott Rahner. How did that sidecar get there. https://medium.com/dowjones/how-did-that-sidecar-get-there-4dcd73f1a0a4 (visited on 2020-05-07).

[30] zipkin.io. Zipkin Community. https://zipkin.io/pages/community.html (visited on 2020-05-07).

[31] Zipkin.io. Zipkin architecture. https://zipkin.io/pages/architecture.html (visited on 2020-05-07).

[32] Zipkin.io. Tracers and Instrumentation. https://zipkin.io/pages/tracers_instrumentation.html (visited on 2020-05-08).

[33] openzipkin/brave-Git. Brave. https://github.com/openzipkin/brave (visited on 2020-05-08).

[34] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '18, September 2018.

[35] DescartesResearch/TeaStore-Git. Testing and Benchmarking. https://github.com/DescartesResearch/TeaStore/wiki/Testing-and-Benchmarking#3-instrumenting-the-teastore (visited on 2020-05-08).

[36] Opentracing.io. Tracers. https://opentracing.io/guides/java/tracers/ (visited on 2020-05-09).

[37] Prometheus.io. Monitoring Linux host metrics with the node exporter. https://prometheus.io/docs/guides/node-exporter/ (visited on 2020-05-10).

[38] Jmeter.apache.org. Apache JMeter. https://jmeter.apache.org/ (visited on 2020-05-10).

[39] CNCF.io. graduation_criteria. https://github.com/cncf/toc/blob/master/process/graduation_criteria.adoc (visited on 2020-05-25).

[40] Jaegertracing.io-Git. Insights. https://github.com/jaegertracing/jaeger/graphs/commit-activity (visited on 2020-05-13).

[41] OpenZipkin.io-Git. Insights. https://github.com/openzipkin/zipkin/graphs/commit-activity (visited on 2020-05-13).

[42] Jaegertracing.io-Git. Bugs. https://github.com/jaegertracing/jaeger/issues?q=label%3Abug+is%3Aclosed (visited on 2020-05-13).

[43] OpenZipkin.io-Git. Bugs. https://github.com/openzipkin/zipkin/issues?q=is%3Aissue+is%3Aclosed+label%3Abug (visited on 2020-05-13).

Appendices

A.1 Latency

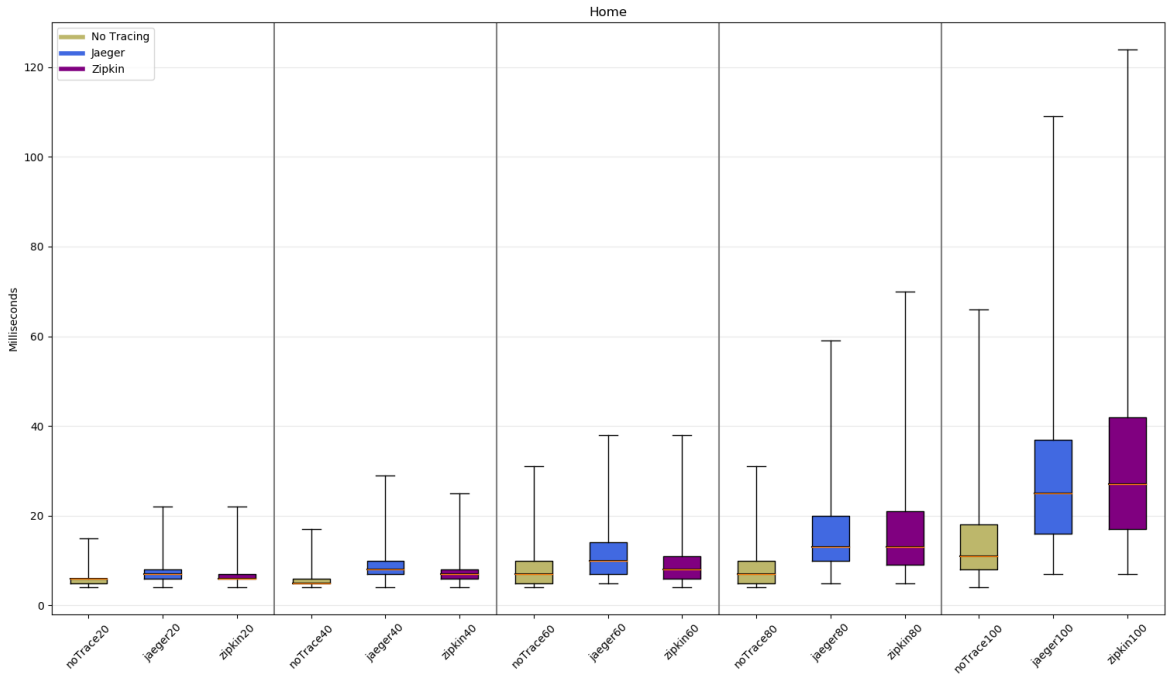

Figure 1: Latency of request Home.

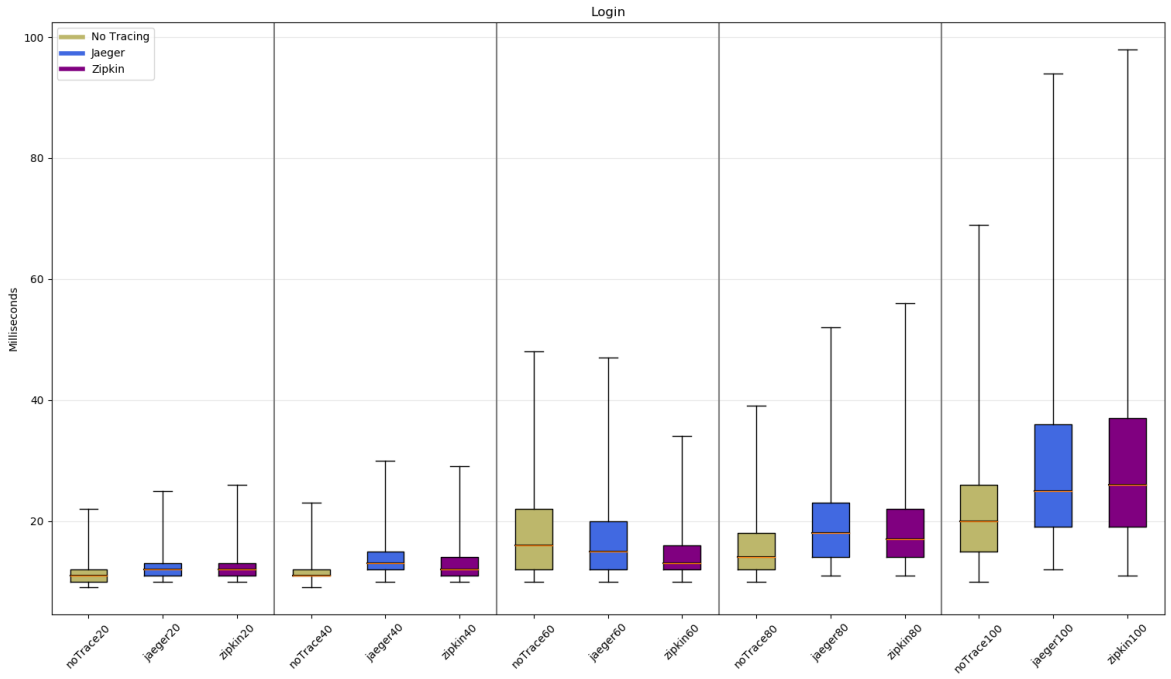

Figure 2: Latency of the request Login.

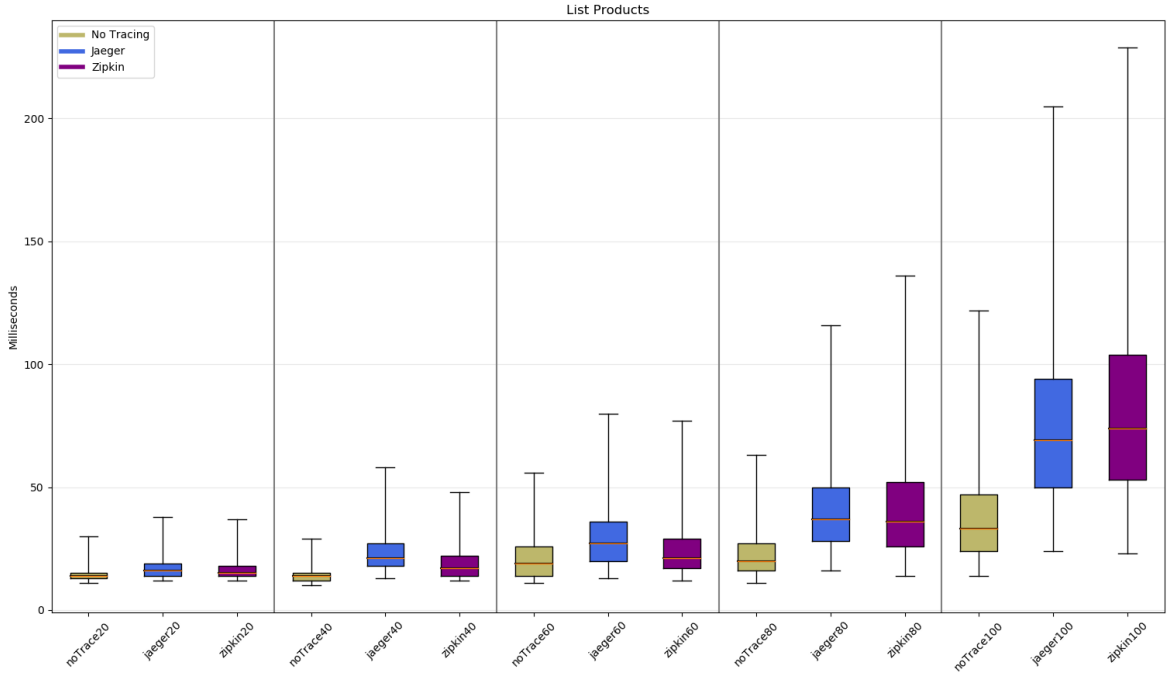

Figure 3: Latency of the request List Products.

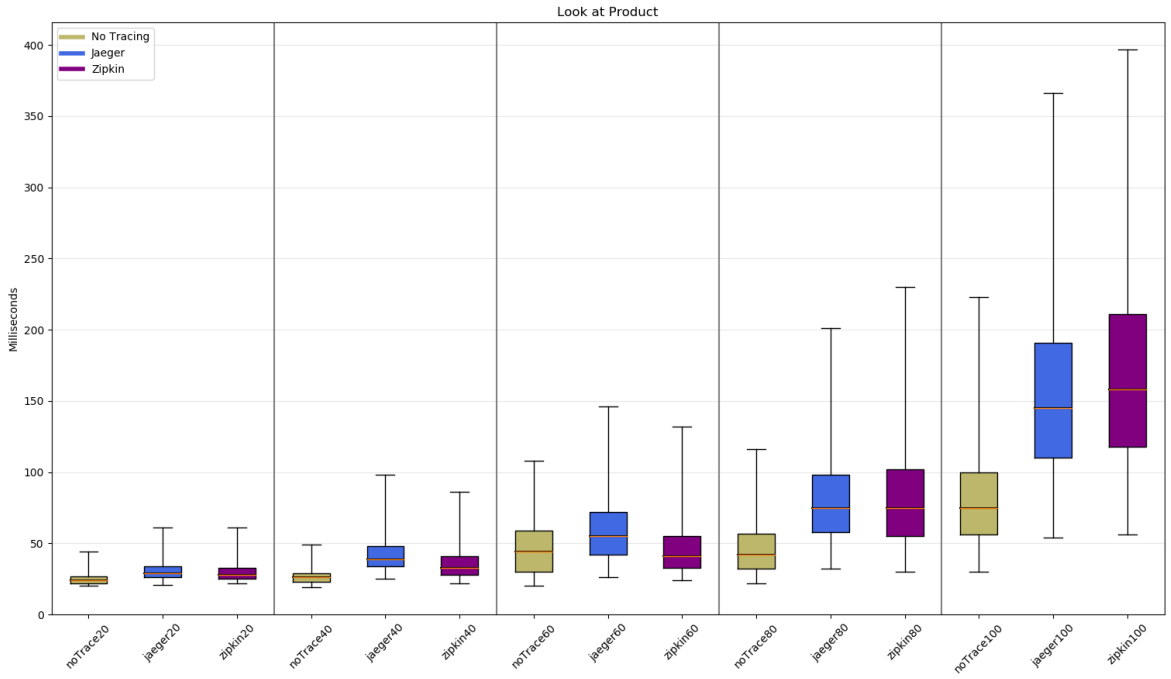

Figure 4: Latency of the request Look at Product.

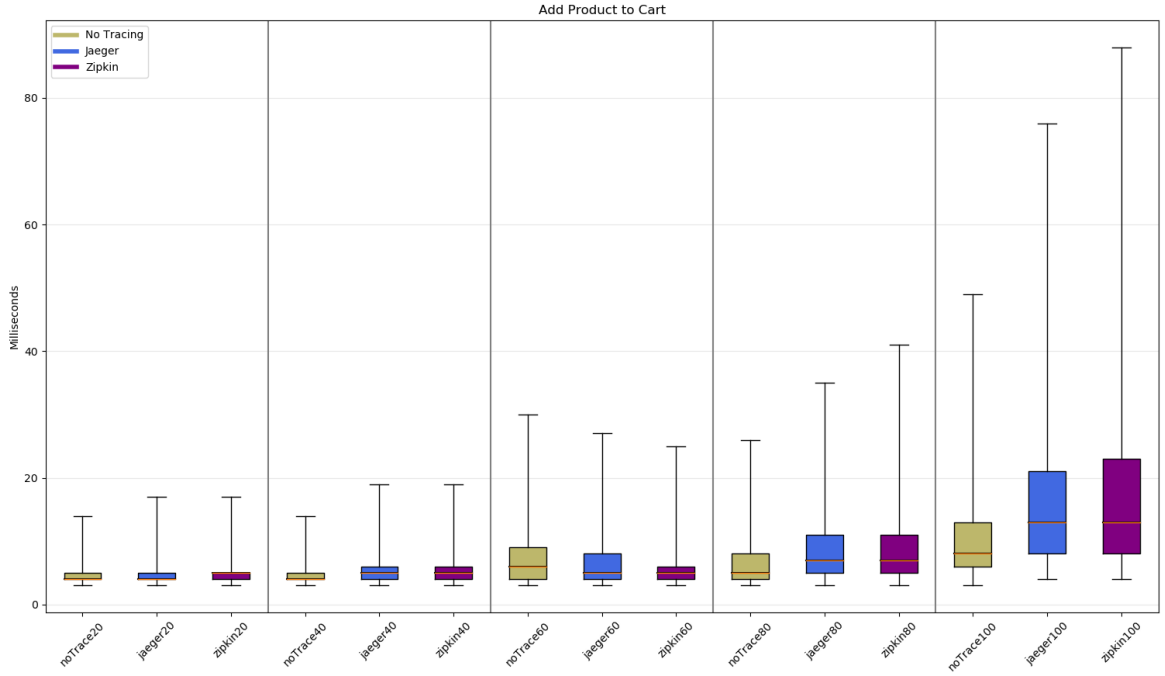

Figure 5: Latency of the request Add Product to Cart.

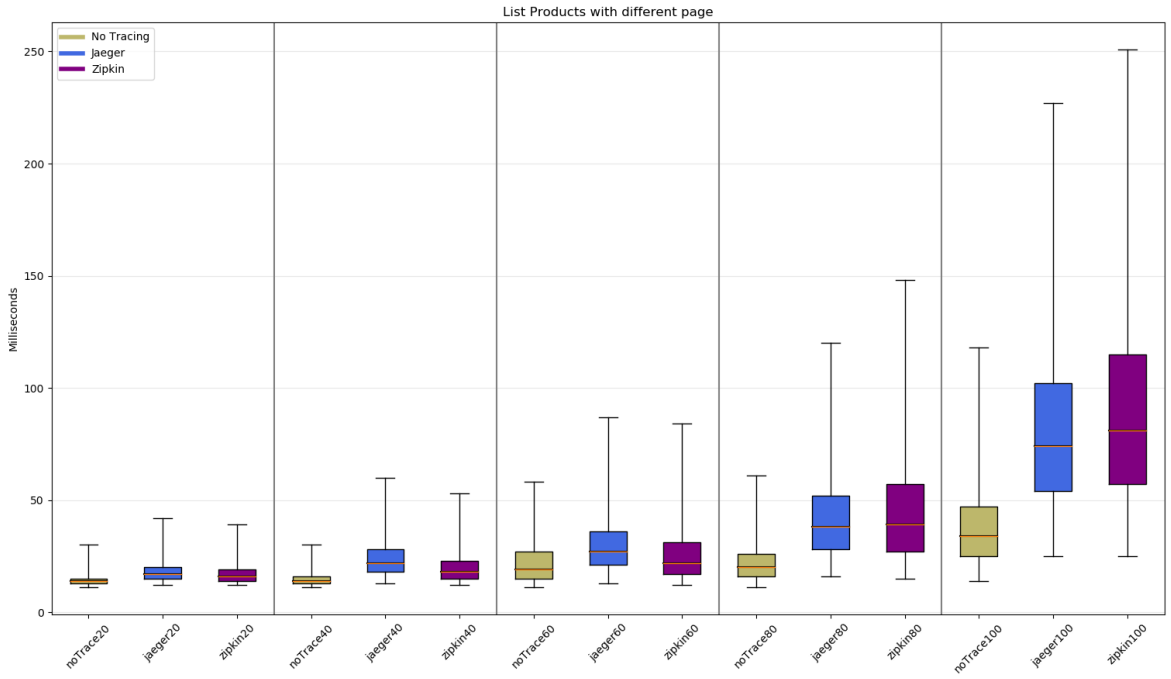

Figure 6: Latency of the request List Products with different page.

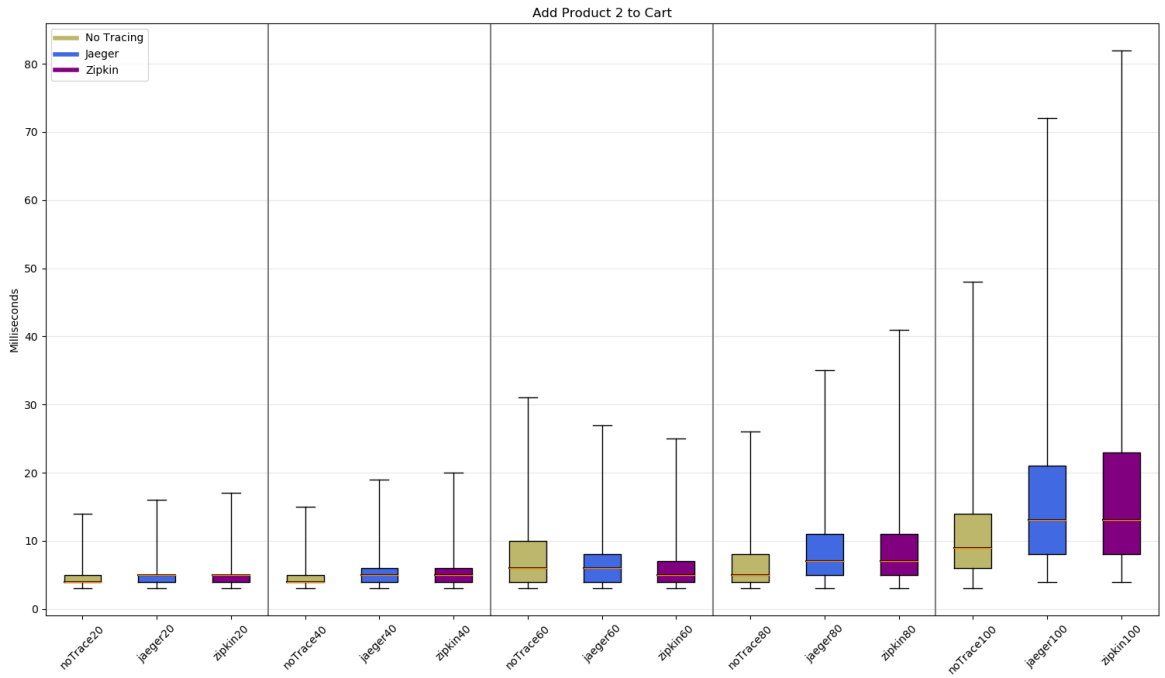

Figure 7: Latency of the request Add Product 2 to Cart.

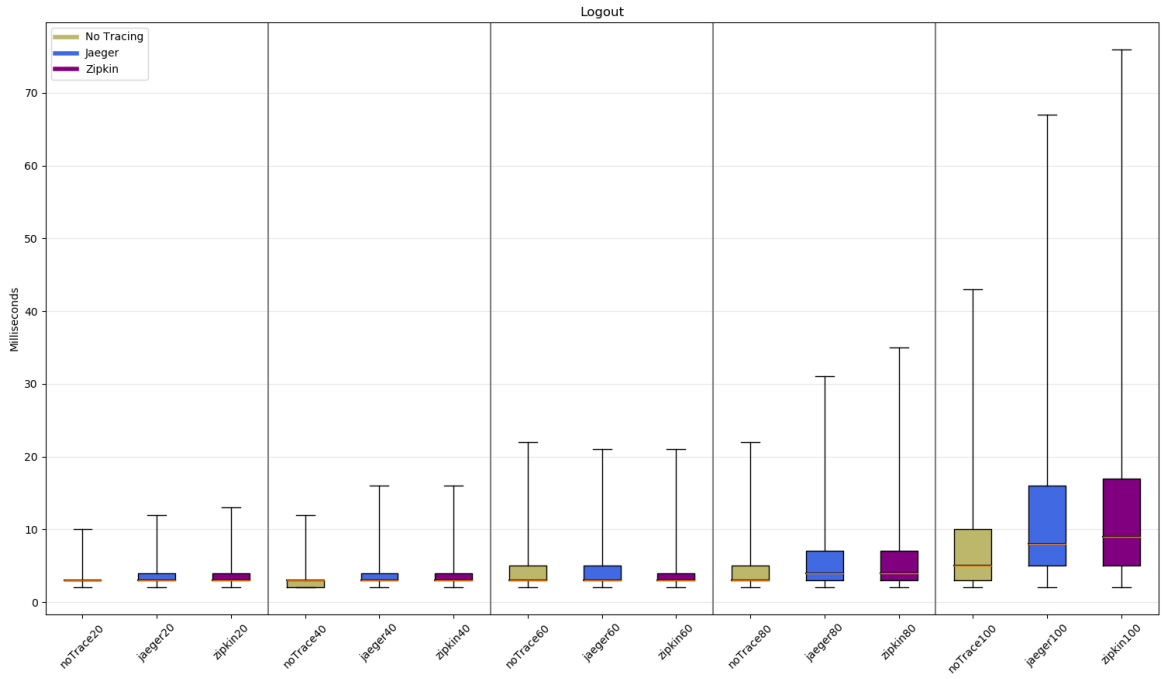

Figure 8: Latency of the request Logout.