Traffic analysis of a service mesh

Swen Nijboer, Thesis ICA-5641381

Department of Information and Computing Sciences,
Utrecht University

January 2021

Abstract

In this thesis we will look into the configuration of microservices. Over the years the software landscape has evolved from monolithic applications to microservice architectures. For these microservice architectures that configuration has become increasingly important. As the microservice architectures grew larger the amount of configuration has also increased which causes the developers to not fully grasp the mesh they are developing. This service mesh configuration allows the developer to automatically scale the application when demand increases. This configuration can also handle cases where a part of the network becomes unavailable or unreliable. By not exactly knowing how the network functions and what a small change can cause further down the network, errors can easily be introduced. We are investigating a way to give the developer more insight into the consequences of his changes. In a world where the software landscape is becoming increasingly important and large this can help developers to seriously improve the quality of their configurations. We looked into data from the configuration files for the service mesh and the logging that is produced by the service mesh. We found that with the information it is possible to create realistic simulations. Under the right circumstances it is even possible to make good predictions about the traffic and load within a service mesh after a change.

Contents

1 Introduction

Software and the infrastructure on which they run have become larger and more complex over the years. As a result the servers that have to provide the computational power for these applications have also become larger and more powerful. At first it was feasible to add more hardware to these systems to make them more powerful; this is called vertical scaling or scaling up [20]. But with these ever growing systems just adding more hardware does not scale anymore, as this has limitations on the maximum power for a single server and causes downtime during the upgrade. When a single server is not sufficient to run an application multiple servers are required; this is called horizontal scaling. This does require the application to run on multiple servers, this has a lot of caveats. The data for the application has to be replicated over all servers, incoming requests have to be balanced and the application has to be changed in order to support running on multiple servers. The splitting of a single application over multiple servers (or instances) has been brought to the next level by microservices.

With microservices each piece of the application is placed within a different server. These servers operate and scale independently of each other, this allows the application to scale specific parts as the demand for them grows. This has many advantages beyond just allowing the application to scale to an increased load. It also allows developers to test new versions of their application for a small percentage of their users and deploy new versions without ever having to bring the system down.

These features do come with a cost: configuration. All these services need to work together in a coordinated fashion which requires a serious amount of configuration which grows with the size of the network. For small networks the developers may still have an overview of all the services and how they work together, for larger networks this becomes a near impossible task. For these larger networks it is beneficial to make the microservice architecture into a service mesh. Within a service mesh some of these functions, like load balancing and resiliency, are taken over. The developer can configure thresholds for the application and the service mesh will scale the microservices within the service mesh to meet these thresholds. This behavior causes the service mesh to be very dynamic, but also hard to understand.

InfoSupport [30] is a company that builds software solutions for their customers. Within their customers they see the increase in use of microservices and the configuration complexity that comes with this. This research has been conducted to build a tool that helps developers with building these large scale applications. We want to do this by increasing the developers understanding of the microservice network by analysing the configuration and creating a visualisation of the network. By doing this we hope that the developer can make better choices and quickly see the consequences of changes he makes in the configuration files. By gaining more insight into the service mesh, the developers can prevent simple mistakes that can cause huge outages. Configuration errors are quickly made as seen by one of the recent outages at Google [18]. These

outages show the importance of good understanding and testing of service mesh configuration. This is also important for the clients of InfoSupport.

1.1 Research questions

For our research we want to look into preventing errors that developers can make while they are working on service meshes. The use of service meshes is increasing with the more agile way of working that has been introduced over the years. Because of the amount of configuration that is involved with structuring these meshes the complexity has also increased which has caused problems as shown by the Google outage [18]. The current tools allow for visualisation of meshes that are in production but do not allow the developer to look into the workings of the mesh without it having been deployed. This can be an issue for larger meshes where the developer might not have a full understanding of all mesh components and has to find out if there will be errors while deploying the service mesh. Analyzing the service mesh before it is deployed also implies that there are only three sources of information which can be used: the configuration files for the service mesh (the YAML files), the source code of the application, and developer knowledge.

The research questions are as follows:

Research Question 1: Is it possible to construct an accurate simulation of a service mesh with just the static configuration files for that network?

Research Question 1.1: Which information is required to create an accurate simulation of a service mesh?

Research Question 1.2: Which information is missing from the configuration files for the service mesh?

Research Question 2: Can we increase the performance of a simulation by adding dynamic information, such as log data?

Research Question 3: What is the accuracy of the simulation of the real service mesh?

1.1.1 Structure of Thesis

We first go into service meshes, what are they made of and how do they operate. We then look at the model that is used for the simulation of a service mesh. After that we look into the analysis of log files that are produced by a real service mesh, and how this logging is stored in our model. Then we look into how the simulation tool is built. We also see how the simulation and the real service mesh compare. Finally we draw conclusions from the conducted research.

2 Related work

2.1 Configuration as code

Application behavior used to be dominated by the application logic, but as the application is shifting from a monolithic application to a microservice architecture the configuration starts to have a bigger influence on the application behavior. Within a microservice architecture there are options to scale the application based on load. Traffic within a microservice architecture is load balanced (traffic is routed to parts of the network that are available). These properties make a microservice architecture dynamic in nature. But being dynamic also means that there are a lot of different states that can be derived from the microservice configuration. Because the configuration has such an influence on the application behavior this configuration should also be managed in the same way that code is. This includes versioning and, most importantly, testing.

2.1.1 The importance of testing

There is no doubt that testing of a code-base is important [11]. Nowadays it is hard to imagine to ship a product without it ever being tested. Especially since large teams of developers are working on a piece of software, which makes it harder to be sure that the internal components are fully compatible with each other. That the configuration of the system should also be tested became clear during the outage of Google [18] where a misconfiguration led to a large piece of the network failing. This shows that the configuration of the network should also be tested before being deployed just like the code-base. But for applications that expect a serious load this can be very hard as there is no good way yet to test how a network will handle a large load without deploying it first.

Cohen et al. [5] showed that configuration can greatly affect the outcome of a test. When software is tested with a certain configuration and is found to be working it does not guarantee that the software will function without errors in another configuration. They showed that depending on the configuration used, the code coverage can vary greatly and thus that which configuration is used for testing is a serious concern. With more configuration options the number of potential configurations (and thus test cases) increases rapidly. This combined with the earlier mentioned configuration dependent code coverage allows for bugs to go undiscovered during testing only to show themselves when a user configures the application in an untested way. This concept of configuration dependent testing is also shown by Dongpu et al. [12]. Those papers focused on the inner configuration of an application (e.g. the behavior that a browser has when opening on a new tab, should it go to a certain web page or open a blank tab). We will focus on the way that microservices within a service mesh interact with each other instead of configuration within a microservice itself. But these papers do show the importance of testing a configuration before deploying it.

2.1.2 Quality metrics

Code quality metrics can say a lot about the general quality of a piece of code. They do not guarantee its correctness in any way but does give an indication of how well the code was written. It can also be used to point towards pieces of code that possibly contain mistakes as shown by Jiang et al. [8]. The importance of quality metrics is also shown by Oliveira et al. [9]. They looked into code quality metrics specifically for the embedded software industry. They felt that the current software quality metrics do not suffice for the embedded software as they usually favor a lower resource footprint over easier and more maintainable code. The authors also showed that the use of code quality metrics improved the quality of the code significantly and decreased the time to market.

The amount of literature on code quality metrics show its importance in software development. Yet there are currently no ways to measure the quality of your microservice configuration or service mesh.

2.1.3 Language expressiveness

To reason about different languages and what they can do it is important to know how expressive these languages are. A framework for reasoning about the expressiveness of a programming language is given in "On the Expressive Power of Programming Languages" by Matthias Felleisen [2]. We will use this to reason about the complexity of the configuration files that are used to determine the behavior of the microservices. The complexity that is possible within configuration greatly determines the number of configuration options and thus the amount of testing that has to be done to cover all scenarios.

2.1.4 Information incompleteness

A lot of information can be gathered from the YAML files that are used to configure the microservice. But it is not always possible to extract all required information from just the static files. For this information we will need additional information which can come from an already running system. Using additional dynamic information has a lot of advantages as stated in "Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information" [3]. In our case the static information will come from the YAML files that are used to configure the network. The dynamic information will come from the logging that is generated by the service mesh. The service mesh allows for distributed tracing which show the path that a request makes through the network. This combined with the static information about the instances that are available can provide valuable insight into the operation of an already running network.

2.2 Visualisation

The relationship between microservices within a microservice architecture can be best described as a graph. The graph can also be used to show potential

connections that receive a high load or connections that will be established or broken when a new version of the configuration is applied. We want to visualize this graph to the developer. The graph used by Istio has already been shown in figure 9.

The visualisation of the network will be very important as this will be the main way to inform the developer about the state of the network. As the network grows larger this will become increasingly difficult especially because when large graphs with lots of nodes and edges are being viewed, it is important that edges do not overlap too often. Excessive edge crossing make the display cluttered and therefore make it hard to read and interpret the graph. In "Geometry-Based Edge Clustering for Graph Visualization" [7] the authors looked into a way of clustering the edges in such a way that the graph is easier to read.

There is also the dilemma of being able to read both the global structure of the network and also be able to view the local details. The authors of Nodetrix [6] looked into a way to show different levels of information through a hybrid view of the network. They combined node-link diagrams with adjacency matrices to make large graphs easier to read.

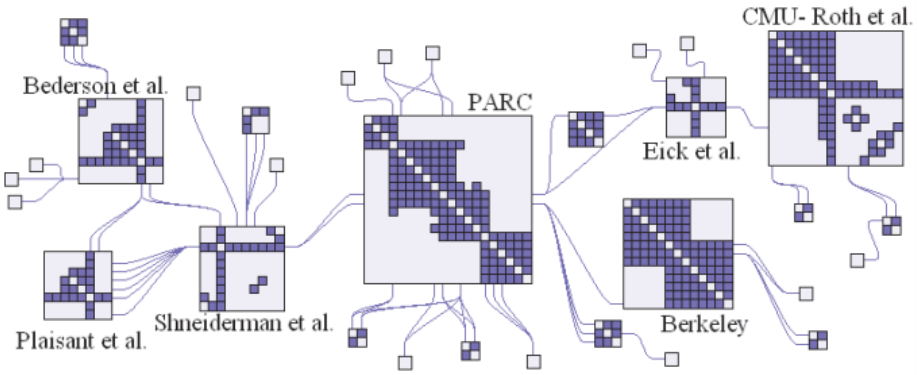

Figure 1: The resulting visualisation of Nodetrix [6]

2.3 Graph analysis

One of the ways that you can look at a computer network is like a flow network. The data flowing through the connections of the microservice architecture is something that should be going without any trouble. Because of this, it is relevant to look at the amount of flow that is possible in any given network [1]. This can be done for different configurations of the network to see which one provides sufficient or even optimal flow. It will also help to test if a node is (still) reachable through the use of graph connectivity. The reachability problem has also been investigated by Xie et al. [4]. They looked into different segments of

an IP network to provide information on situations where a network would be cut loose from other networks. This was done for corporate networks that were accessed through a virtual private network (VPN) connection but it translates very well to containers within a service mesh.

3 Preliminaries

The software landscape has moved from single large monolithic applications to more scalable and modular architectures. These architectures consist of small independent services (*microservices*) which work together. The environment in which such a microservice runs is called a *container*. Each of these microservices runs in its own container.

3.1 Webscale architectures

3.1.1 The problem solved by microservices

As computer usage grew different kinds of architectures were invented to provide the infrastructure for this growing demand. One of the architectures that was developed is the microservice architecture. The microservice architecture replaced the existing monolithic architecture which was just not suitable for the large scale web applications of today [33]. Tools like Istio, which will be explained in more depth later, became available to help manage the load of microservices for an application. New and diverse usages of applications also require more flexibility in scaling specific parts of the application [14]. Each type of user (mobile, desktop, API) may require a different frontend to work with while the application logic might be the same for all users. Having to scale up the application logic because there are a lot of API requests does not mean that there have to be more instances of the desktop frontend. And an update to one of the frontends does not have to cause the application logic to be re-deployed. By moving to the much more modular microservice architecture these components can be scaled and updated individually.

Due to the ability to scale services individually a higher uptime can also be achieved. By deploying the same service multiple times the network is more resilient to failure, if a service fails another instance of that service can take over. This flexibility also allows developers to quickly release new versions of an application without having to go down to do the upgrade [16].

The use of microservice architectures also allows for easier use of polyglot programming. As each component is a self managing piece of the application that uses APIs to expose its data to other parts of the application there is also a new opportunity to choose different languages or programming techniques for different services. Polyglot programming is possible in a large monolithic application, but it will require a lot of effort to couple the individual components to each other. With the microservice architecture the components just need to implement the same API protocol which will allow for inter service communication. By making it easier to use different languages it becomes easier to use the right tool for the job, which reduces the overall project complexity [13].

3.1.2 Problems caused by microservice architectures

Although microservice architectures have a lot of advantages they also come with some drawbacks. This was demonstrated by an outage at Google on the

2nd of June in 2019 [18]. A configuration change that was only intended for a small group of servers was accidentally applied to a large group of servers. This caused the network to only use half its capacity which led to a significant reduction in overall traffic. This misapplication of configuration showed that a change in configuration can have a big effect on the operation of a network. In a system where the entire application is deployed as a monolith there was no need to worry about configuration for inter service communication. By splitting the application into tiny bits and pieces the headache of maintaining all this configuration (and other complexities that come with a distributed system) becomes a bigger challenge [39]. This switch is usually only viable for larger applications as illustrated by figure 2.

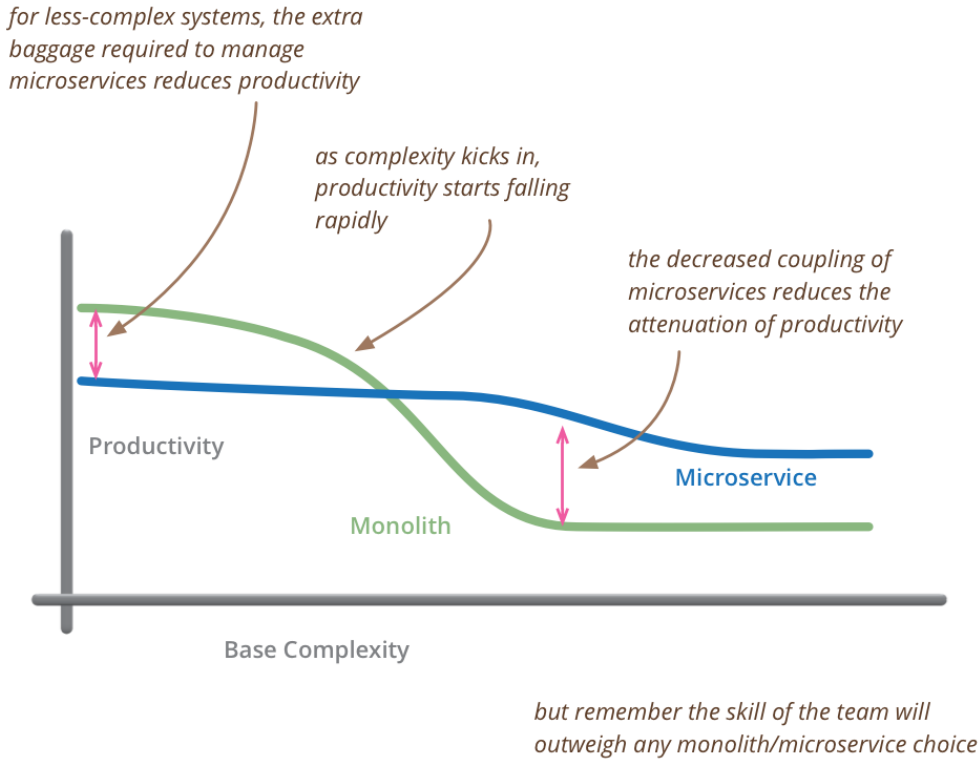

Figure 2: It is only viable to switch to a microservice architecture for large complex applications [39]

The possibility of using polyglot programming is both a strength and a weakness. By using a lot of different languages and paradigms a lot of knowledge is needed within the team to maintain the system, as all these languages and paradigms have to be known in order for the developer to maintain the entire system [32]. It might seem tempting to implement a specific component in a language that might be slightly more efficient, but this will cost more time in

the long term as the maintainers will also need to invest the time and energy into learning that language.

By splitting up the application into different components the difficulty of handling errors also increases significantly. When a service fails it should not cause the entire network to go down and the data in the network should also not be corrupted. Guidi et al. [10] looked into ways of handling errors in Service Oriented Computing (SOC). By deploying the microservices within a service mesh it is also possible to monitor errors within the network communication closely. In the Kiali dashboard it quickly becomes clear when an application instance is malfunctioning.

Microservices need to work tightly together to provide a useful application. This need for communication creates dependencies on other microservices. Having dependencies introduces a series of risks and therefore it is important to manage these dependencies as shown by Esparrachiari et al. [17]. One of the main risks is cyclic dependencies: this is where an application breaks because a series of services depend on each other to function. Other risks involve performance and security issues, where an issue in a single service might cause trouble for all services that depend on that service. Tools like Istio have proven to be a useful tool to mitigate many of these basic security issues. This is done by injecting a piece of code into the microservice which functions as a proxy. This injected piece of code is called a *sidecar* and can handle things like authorization, authentication, and encryption.

3.1.3 Communication patterns

Communication is key for microservices as each microservice only does a small part of the bigger task. This communication can take place in different forms and protocols as described by an article on Reflectoring [38]. One of these forms is a *synchronous call*, where microservice A connects to microservice B, requests some data, waits for it and then returns with that data to microservice A. This form of communication does introduce strong coupling as microservice A has to wait for microservice B to finish. If microservice B does not finish on time it might cause a timeout for microservice A. In order for this to work the microservices also need to know of each others existence. This can be done through service discovery which makes this very easy. When a new service is created it will register itself to a central registry, after that has been done other services can easily find the service by simply doing a DNS lookup for that service.

A better option is the one of *simple messaging*, which is also described by Microsoft [22]. When using this pattern service A fires a message to a message broker and continues working on something else. Service B will see the message if it is subscribed to the topic that service A published the message in. The services do not need to be connected to each other, just to the message broker. If service B is not available it can still receive the message when it becomes available again and service A does not need to worry about having to retry later as this is done by the service broker. This does add a dependency on the message broker, if

the message broker is not available the two services are unable to communicate. Another potential drawback is the message structure. If the application changes and introduces a new message structure then all applications that listen to that type of message will have to be made to understand this new type of message, which works against the idea of independent service deployment.

As any call to a remote service can fail it is important to cope with these failures, Microsoft published an article regarding the design for inter service communication [22]. One way of coping with these failures is by retrying the request after a short time. If the service is unavailable due to a reboot it might not respond for a while and this retry might fail as well. When retrying it is important to have some kind of *circuit breaker* in place to prevent the network from being flooded by these retries. The tools provided by a service mesh can also help to implement systems like the circuit breaker as the proxy in the sidecar can see the requests and find badly behaving instances of a service.

3.1.4 Single responsibility

The idea behind splitting up a large (monolithic) application into multiple smaller components is the idea that each component can focus on a single task [14]. The microservice should be self-contained (it needs to have all the data and logic that is required for its task) so that it does not have to depend on other services. This makes it easy for the microservice to be scaled and deployed independently of other microservices. The microservice can be developed independently of other microservices as long as it still fulfills its specification. As stated by the authors the process of splitting an application into microservices is a daunting task which requires a certain intuition to determine where to draw the line between different microservices. There are a few patterns that can be used to split an application into microservices as described on DZone [21].

The first option is to decompose by business capability. A business capability is something that a business does in order to generate values (for an insurance company capabilities include processing claims or billing) [21]. This decomposition using the business capabilities is a good start but the authors point out that there will still be some "God Classes" which will be too large and have to be decomposed further. For these cases *Domain-Driven design (DDD)* can solve the problem. For example, the "Order" class is a domain which can be divided into subdomains and each subdomain will have a model. The scope of this model is called the *bounded context* and a microservice is developed around such a bounded context. The final division will usually result in something that is very modular which also allows for a great way to scale the application. Each individual component can be scaled depending on the demand for that component.

3.2 Containers and Docker

Containers have enabled developers to focus more on the development of the code instead of having to worry about the environment where the code will

be deployed. A container is essentially the environment that the code is going to run in, this is provided by Docker [24]. Something similar to containers is virtualization, where multiple operating systems run virtually on a single host. Figure 3 shows the difference between a host with multiple virtual machines and a host with multiple Docker containers. Because the Docker containers share the OS and where applicable binaries and libraries, the resource footprint of a container is a lot smaller than the resource footprint of a virtual machine. This allows for quicker deployment and better resource utilization of a single host.

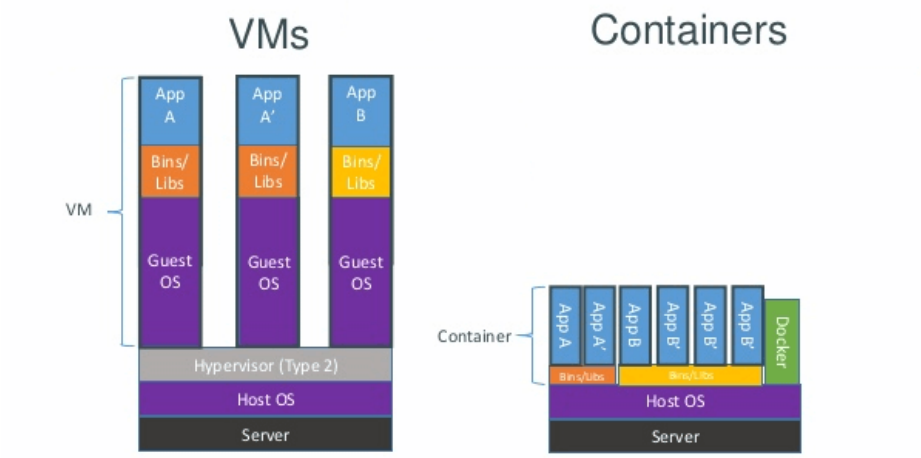

Figure 3: The difference between Docker and a virtual machine (VM) [29]

A Docker environment is defined through a *dockerfile*. In a dockerfile all commands that are needed to build the image for a container are specified. The image is an immutable file which is essentially a snapshot of a container. When you run an image you get an instance of that image which is called a container. These containers can be build from a simple lightweight text file which makes them very portable: it is easy to move a container one from machine to another as only the dockerfile has to be executed to build an identical container on the other machine. This has the added benefit that it is simple to create identical development and production environments (when both run the same docker containers). This greatly reduces the amount of problems while deploying new applications as there is no sudden missing dependency, or a leftover from a previous version that could cause an error. An example of a dockerfile taken from `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/` can be seen below.

```
1 FROM ubuntu:18.04
2 COPY . /app
3 RUN make /app
4 CMD python /app/app.py
```

This file builds further on the Ubuntu image, copies files from the current directory to the 'app' directory, builds the application and then runs the application. This freshly created image can be saved in the *Docker Hub* so that other developers can use it. A lot of images already exist in the Docker Hub (images for MySQL, NGINX, Apache, etc.), developers can easily take these images and use them for their applications.

3.3 Docker compose

Even a simple website will require multiple containers. For a PHP website you will need a container with Apache or NGINX and a container with MySQL. PHP and MySQL can be placed in a single container, but a best practice is to split this over different containers. By splitting applications over different containers you allow them to be scaled and deployed individually. If the PHP and MySQL application were in the same container you will always have to scale them together. When they run in individual containers and your application starts to get more traffic you can easily deploy multiple PHP containers (which do the heavy work) which communicate with a single MySQL container. You can start those two containers manually which will work fine for only two containers. But as you add more containers this will take more time and can result in errors (you could for example forget to spin up a container). This problem is tackled by using *docker compose*, which allows you to define a set of containers that have to be started simultaneously. A part of a docker compose file taken from `https://github.com/nanoninja/docker-nginx-php-mysql/blob/master/docker-compose.yml` can be seen in figure 4.

This compose file will start two containers, a container called "web" (line 3) and a container called "mysqldb" (line 21). The "web" container is based on the nginx:alpine build, which is the source image as can be seen on line 4. On line 6 and 7 the volumes are mounted: this allows the container to access data from the host system (in this case the HTML files for the site and the SSL certificates). After that two ports are exposed. Port 80 and 443 on the container are bound to port 8000 and 3000 on the host (line 9 and 10), which makes the server accessible to the outside world. On line 11-15 the health check is defined. Every 90 seconds Docker will run the command on line 12 to see if the server is online. If it does not respond within 10 seconds three times in a row the container will be killed and a new container will be started. On line 16 the environment variable "NGINX_HOST" is set. The restart policy is set to "always" on line 17, which will cause the container to start if it is terminated for whatever reason. Finally on line 20 it is configured that this container depends on the "mysqldb" container, so that container has to go up before this container can start. By issuing the command "docker-compose up" in the same folder as the Docker compose file both the NGINX and MySQL container will be started.

One of the features of Docker is that the containers can easily connect with each other using their given names (*mysqldb* and *web*). When a container is started it registers itself with Docker using the given name, Docker will register that name with the IP address of the container in its own DNS server. Whenever

```yaml
version: '3'
services:
    web:
        image: nginx:alpine
        volumes:
            - "./etc/ssl:/etc/ssl"
            - "./web:/var/www/html"
        ports:
            - "8000:80"
            - "3000:443"
        healthcheck:
            test: curl --fail -s http://localhost:5000/ || exit 1
            interval: 1m30s
            timeout: 10s
            retries: 3
        environment:
            - NGINX_HOST=${NGINX_HOST}
        restart: always
        depends_on:
            - mysqldb
    mysqldb:
        image: mysql:${MYSQL_VERSION}
        container_name: ${MYSQL_HOST}
        restart: always
        env_file:
            - ".env"
        environment:
            - MYSQL_DATABASE=${MYSQL_DATABASE}
            - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}
            - MYSQL_USER=${MYSQL_USER}
            - MYSQL_PASSWORD=${MYSQL_PASSWORD}
        ports:
            - "8989:3306"
        volumes:
            - "./data/db/mysql:/var/lib/mysql"
```

Figure 4: An example of a Docker Compose file.

another container uses the given name Docker will resolve the name to the IP address of the container. This has the advantage that IP addresses do not have to be hardcoded into the application and thus do not have to be updated.

3.4 Kubernetes

Docker compose is useful for managing a few containers, but once you get to large scale networks Docker compose does not suffice as it only manages a single host. For larger applications it might be necessary to have containers on multiple hosts, which also allows for more resiliency. Google has one and a half decade of experience [15] in running large applications and has developed a tool for this, Kubernetes [35]. Kubernetes is an alternative to Dockers own implementation (Docker Swarm [25], which is easier to use but also has less functionality).

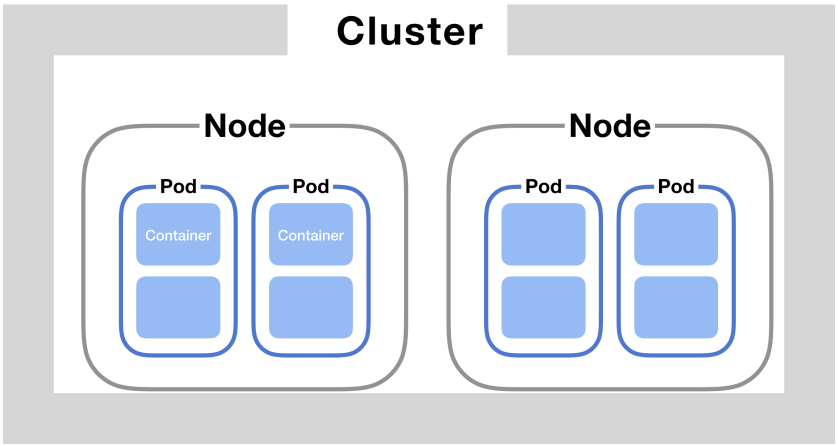

Figure 5: The setup of a Kubernetes network [36]

A Kubernetes setup consists of several elements managed by the master [37]: cluster(s), nodes, pods and containers as seen in figure 5. A *cluster* is a set of worker machines grouped together. We refer to one of these worker machines as a *node*. The node needs to have a kubelet (a process which is responsible for the communication with the master) and a container runtime (e.g. Docker, which is responsible for pulling the container image and running the container). On a single node one or more *pods* can run. A pod is a Kubernetes abstraction that represents a group of one or more *containers* and contains resources like storage, networking, and the information about how to run each container. Containers that run within one pod should be tightly coupled (they share the storage and networking) and will always be deployed together. Containers that run within a pod will be the same as the earlier explained Docker container. An illustration of such a pod can be seen in figure 6.

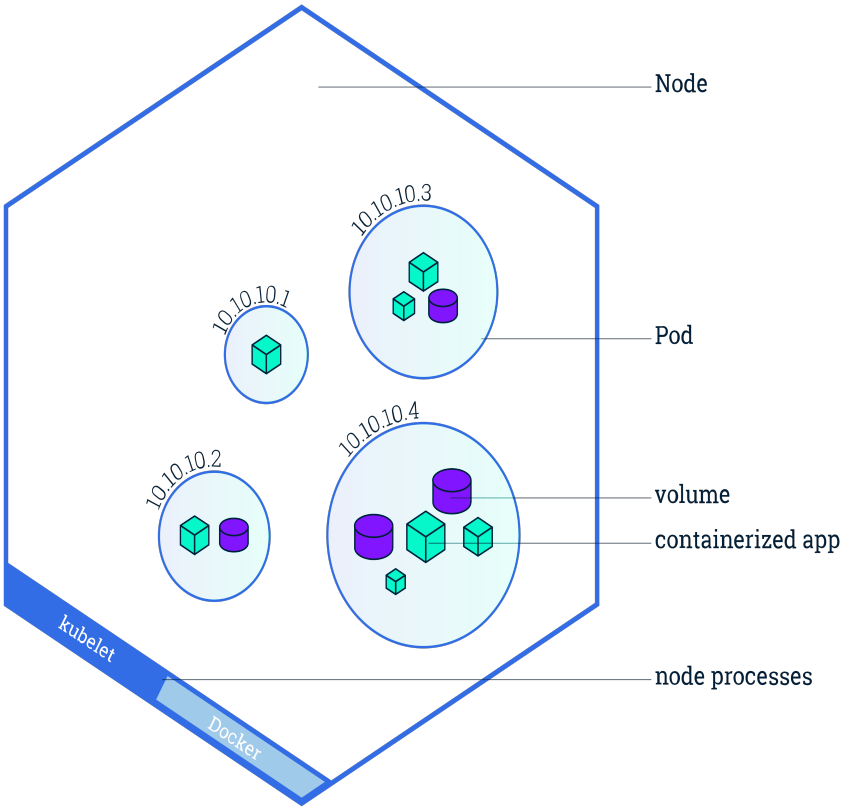

Figure 6: The setup of a Kubernetes pod [42]

Because of the way that Kubernetes manages the full stack from cluster to container it also enables developers to easily scale the application to multiple instances quickly. Kubernetes also detects if a pod goes down or crashes and will start a new pod with the same containers to replace the faulty one. Kubernetes also has the option to run containers on different nodes, which increases resiliency. If a node crashes due to a hardware error (a blown up power supply unit for example) Kubernetes will detect this and will send all traffic to pods that are available. By doing this the uptime of an application can be increased. Kubernetes also offers a *horizontal pod autoscaler* [28], which can automatically scale up the number of pods when it detects that the current pods cannot handle the load. It does this by keeping track of CPU usage for each pod and comparing this to the level set by the developer. If the mean CPU usage of the pods is higher than what is set by the developer Kubernetes will add extra pods (for example, if the current load is 200 and the desired load is 100, the amount

of pods will be doubled as 200 / 100 = 2). If the CPU usage drops below the threshold Kubernetes will stop pods one by one with 5 minute intervals.

3.5 Service mesh

A service mesh builds on top of a container orchestrator (like Kubernetes) to allow for even more control over the network. The service mesh will also take care of interservice communications, monitoring and security-related items. The service mesh does this by adding a *sidecar* to a pod which is used as a proxy, all ingress and egress traffic will be routed through this proxy. This proxy can then monitor all traffic to provide insights and allow for better load balancing. The proxy also enables easier use of encryption as a public key infrastructure (PKI) is set up to distribute the encryption keys to all of the sidecar proxies. The service mesh also handles the dynamic scaling of the instances when load increases or decreases. When the service mesh notices that an instance of type A has a very high load, it will create more instances of type A to distribute the load. When the load drops and instances start to idle the service mesh will terminate those instances to free up system resources. Due to this dynamic behavior service discovery is needed. When a microservice within the service needs to communicate with another microservice it does know what their IP address is. This is handled by the sidecar proxy. The instance communicates with the proxy and the proxy finds an available instance to communicate with.

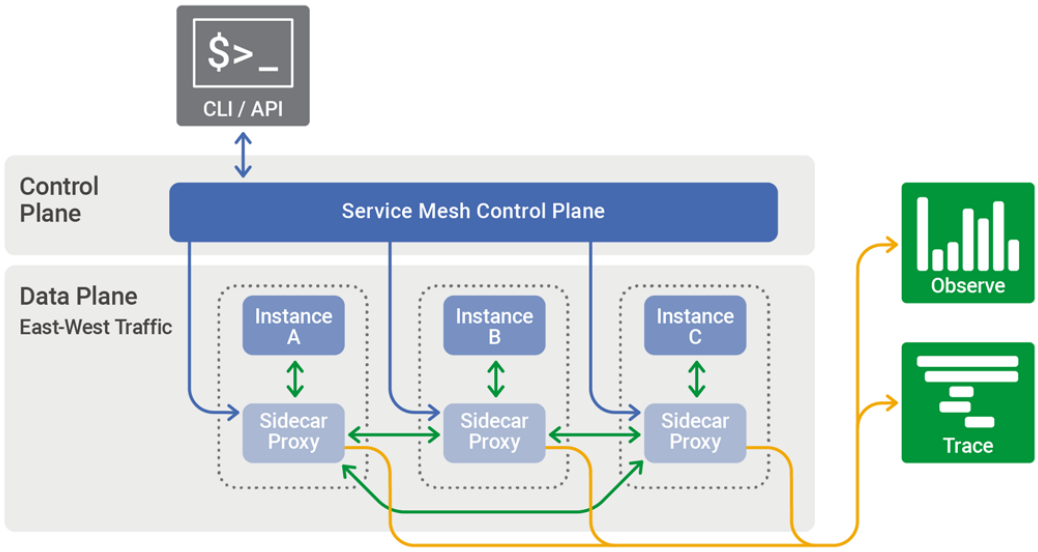

Figure 7: The setup of a service mesh [43]

The service mesh is divided into two main components the *control plane* and the *data plane*. The data plane of a service mesh is the part that manages the

network traffic between pods. The control plane on the other hand generates and deploys the configuration that determines the behavior of the data plane. In figure 7 an example of a service mesh can be seen. The combination of the "Instance" and "Sidecar Proxy" will go into one pod in a Kubernetes based system.

3.6 Istio

There are multiple implementations of a service mesh, Consul, Linkerd and Istio all fulfill the role of a service mesh. However, Istio offers the most flexibility (and thus complexity). Istio is easy to add to an existing microservice architecture as it only requires the deployment of the sidecar in each of the Kubernetes pods. From that moment onward Istio provides a dashboard which allows for detailed logging and fine-grained control over the traffic in the network. This includes fault injection which can be used to see the response of the application to faulty instances. All further sections in this document will use Istio as their base.

3.6.1 Sidecars

The Istio sidecar is extremely easy to install. It can be added to any existing service mesh without any problems as it is configured to allow network communication to all other instances on the network. This allows you to add it easily and then narrow down the configuration for added security. The immediate benefit is the monitoring of all traffic and the injection of errors into the network. The sidecar can be injected using the existing yaml files for the Kubernetes configuration with a single command:

```
1 istioctl kube-inject -f
2     samples/sleep/sleep.yaml | kubectl apply -f -
```

After injection all the benefits of using a service mesh will be available immediately and the instance can be seen and managed in the Istio dashboard.

3.6.2 Envoy

So far the sidecar proxies have been mentioned several times but there have been no specifics on how this is achieved. In the Istio sidecar there is an instance with Envoy. Envoy is a proxy which can also be deployed in a sidecar. Envoy is able to function as a proxy for any TCP protocol and can handle SSL certificates for all connections within the service mesh. This ensures that connection between different nodes of the service mesh are secure. Envoy also records the statistics for network traffic and offers *distributed tracing* [26]. This is done partially by Istio as all the traffic for that instance is routed through the Envoy proxy but it does require some setup. The ability to utilize the distributed tracing is a tool which can be very helpful. Distributed tracing allows the developer to see the flow of a request throughout the network, even if that request 'hits' multiple

different services. This is done by adding a unique ID to all requests as they flow through the mesh.

3.6.3 Logging

The logs which are collected by Envoy contain a lot of information. From the configuration of an individual pod it is not clear which other pods on the network are accessed, but the logging can provide more information on this. Istio keeps track of all connections that are established between pods and can optionally log them. By doing this for an already existing network we can exactly see how the traffic within the network is organized. In the example piece of logging below you can see which kind of data can be extracted from the network log. The "source" and "destination" are the most important to us as they show how the traffic is routed within the network.

```
1 {"level":"warn", "time":"2018-09-15T20:46:36.009801Z",
  ↪ "instance":"newlog.xxxxx.istio-system",
  ↪ "destination":"details", "latency":"13.601485ms",
  ↪ "responseCode":200, "responseSize":178,
  ↪ "source":"productpage", "user":"unknown"}
2 {"level":"warn", "time":"2018-09-15T20:46:36.026993Z",
  ↪ "instance":"newlog.xxxxx.istio-system",
  ↪ "destination":"reviews", "latency":"919.482857ms",
  ↪ "responseCode":200, "responseSize":295,
  ↪ "source":"productpage", "user":"unknown"}
3 {"level":"warn", "time":"2018-09-15T20:46:35.982761Z",
  ↪ "instance":"newlog.xxxxx.istio-system",
  ↪ "destination":"productpage", "latency":"968.030256ms",
  ↪ "responseCode":200, "responseSize":4415,
  ↪ "source":"istio-ingressgateway", "user":"unknown"}
```

One of the drawbacks of logging is that it can only be collected after the fact. This requires a network to be run as the log files cannot be generated statically by Istio.

3.6.4 Distributed tracing

As stated earlier distributed tracing can greatly help a developer to better understand an application. Istio does this by attaching a header to each request that comes in (if it does not have a header yet). When a request passes through the proxy without any of the following headers, they are attached so the request can be tracked: x-request-id, x-b3-traceid, x-b3-spanId, x-b3-parentspanid, x-b3-sampled, x-b3-flags, b3 [23]. Important to note is that a single request gets a x-request-id, x-b3-trace-id, x-b3-spanId and a x-b3-parentspanid. Having multiple of them might seem unnecessary at first but you also have to consider the situation where a single incoming request in a service leads to multiple requests

being sent to other services. By generating new spanid's for the outgoing request but sharing a single parentid the service mesh is able to track exactly where the request branched into multiple requests.

There is one important implementation detail which is necessary in order for the tracing to work correctly. The application that receives the request, processes it, and sends the response, has to handle the HTTP headers as well. This usually leads to the application extracting the headers before it begins processing the request and appending them back to the response (or outgoing requests to other services) after it has processed the request. This is required because there is no implicit way to correlate the incoming requests of a service to the outgoing requests or responses of that service (as a service might receive multiple requests at the same time, which can all trigger different responses to different services).

3.6.5 Desired state configuration

A big advantage of Kubernetes and Istio is the ease of creating environments for your code to run in. It is very easy to create a new container to deploy your code in initially, or even scale them up whenever the application is experiencing more load. You might also want to change some parameters of your container while the application is running, but all this without the application going down. In Kubernetes there is a solution for this, *rolling updates*. The only requirement for this is that the application has been set up so that it can be scaled (as Kubernetes will need to scale pods in order to prevent downtime). By default Kubernetes will add one extra pod with the new configuration to the cluster, wait until that pod comes online and finally remove the old pod from the cluster. This will be repeated until the entire set of pods runs with the new configuration.

An extension of the system above is *canary releasing*. When a new release of the application is available you might not always want to fully switch over immediately. Instead you might want to only allow a small percentage of your traffic to interact with the new version. By doing this you can test whether the new version is stable and without errors before launching it to everyone. The process of doing such a canary release can be seen in figure 8, but it comes down to three steps. At first the application only receives load on the old version of the application, then one or more pods with the new version are created and a part of the traffic is directed to these new pods. If the new pods work as expected all traffic is routed to the new pods and the old ones are removed.

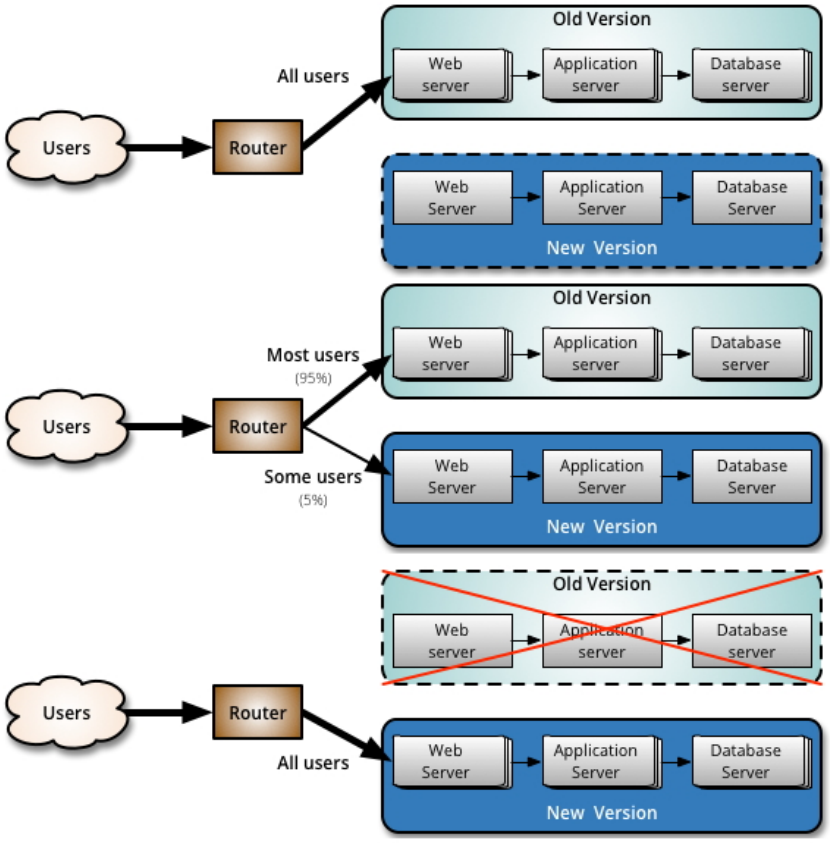

Figure 8: The process of a canary release [19]

3.6.6 Existing visualisation using Kiali

One the useful tools that Istio offers is the ability to visualize your service mesh through Kiali [34]. Going through the logging is useful to get a general idea of what the mesh is doing, but having a visualisation is better as it allows you to quickly see everything that you need to know about the network. In figure 9 an example of such a visualisation as produced by Kiali can be seen. The white squares represent pods and the elements within them are containers. Each line represents a connection between containers. The color of the line can show whether the connection is good (or if errors occur) and can show the traffic flow over that connection. This visualisation can be really helpful when deploying a new version of an application as it is also possible to set the amount of traffic that flow to the canary version of the application. This gives the user a way of comprehending all information regarding the service mesh.

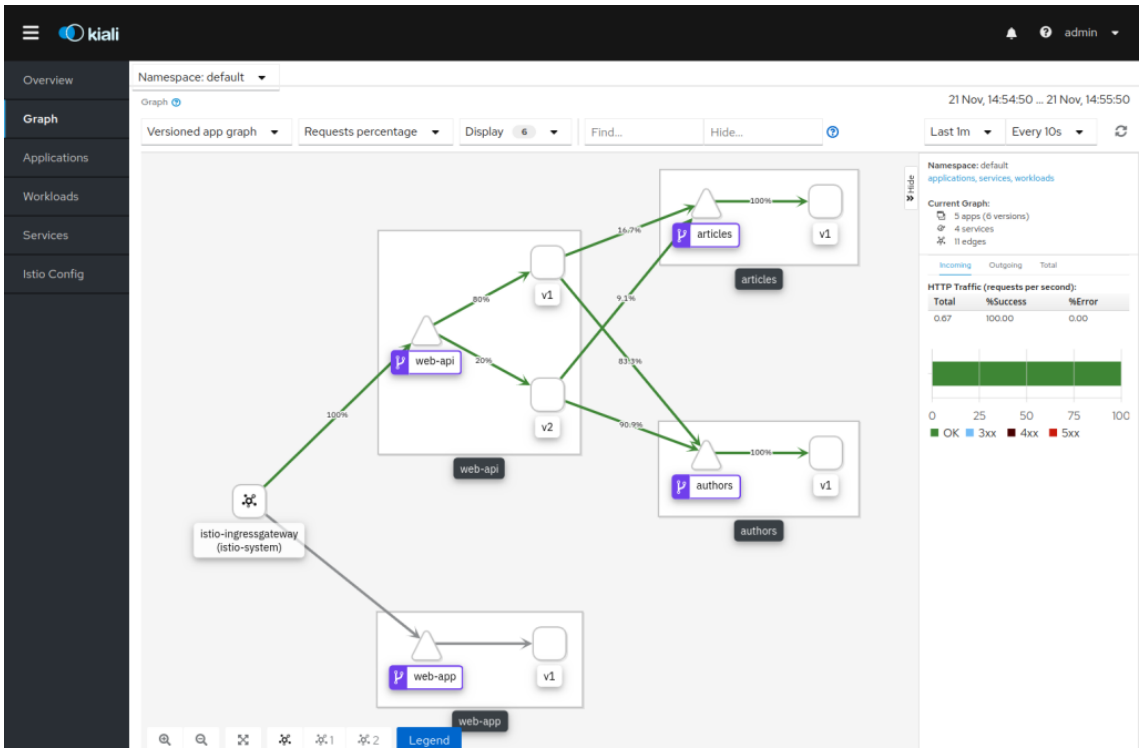

Figure 9: The visualisation provided by Kiali. [31]

3.7 YAML configuration

To setup and configure the network in Istio a set of elements is required. First of all a *deployment* is needed. A deployment is similar to a pod in a Kubernetes cluster. A deployment is a scalable group of containers that does the actual work in a service mesh. When the traffic is routed through the network it ends up at a deployment where it will be processed and the appropriate response will be made. Each deployment has an IP address. Deployments are dynamic, they can be created and destroyed depending on the number of request. The combination of these two things makes it hard to reach the deployments. To overcome this challenge a *service* is used, the service manages the deployments that it matches to (how this matching happens is explained in section 5.3). Applications within the Istio service mesh can connect to the service which will route the traffic to one of the deployments.

Thus far only simple routing of traffic is possible, but sometimes you might want to have more complicated scenarios. With a *destination rule* it is possible to create specific subsets of deployments within all deployments. These subsets allow for more specific routing of traffic within the service mesh. The destination rule is also used to configure a circuit breaker within the service mesh. Finally

a *virtual service* is used to configure the traffic routing to the specific subsets as defined by the destination rule. The virtual service can split traffic over different subsets to allow for canary releasing or mirror traffic to different deployments.

4 Context

This research is conducted within the company InfoSupport [30]. InfoSupport deals with a lot of customers. Because of this they also deal with a lot of problems while developing software for their customers. These problems can be quite complex or require further investigation, this gives students the opportunity to pick one of these problems and conduct the required research.

4.1 The organization

InfoSupport [30] describe themselves on their website as: "InfoSupport is a specialist in tailor made software, data solutions, management and training and is mainly active in the agrifood, energy, fintech, mobility, pension and care. InfoSupport has more than 400 employees with offices in Veenendaal (NL) and Mechelen (BE). The way of working that characterises InfoSupport is build upon four core principles: solidity, integrity, craftsmanship, and passion."

InfoSupport has five business units that focus on different parts of the market. Each of them has their own unit manager. The IT services, professional development center, and knowledge center cross the boundaries of the business units and provide support for hardware and knowledge. The structure of the company can bee seen in figure 10. This thesis is written for the business unit finance.

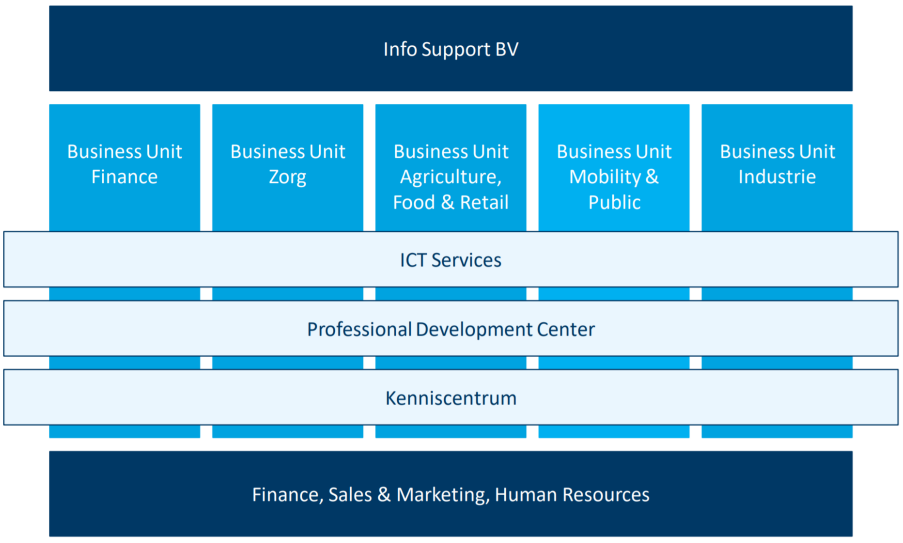

Figure 10: The organization chart of InfoSupport

4.2 The technology stack

There are a few different service mesh providers. The largest three are Istio, Linkerd, and Consul of which Istio is the most well known and has the most features. Furthermore there are two large container orchestrators, Kubernetes and Docker Swarm. Kubernetes offers more functionality and is more suitable for large networks and is used most often because of this.

To prevent a big time investment in setting up an environment where the service mesh analysis tool can be tested the Pitstop demo application [40] is being used. The Pitstop application has been designed to illustrate the software-architecture concepts of microservices, CQRS, event sourcing, Domain Driven Design, and eventual consistency. It also illustrates the usage of technologies like Docker, Kubernetes, Istio, and Linkerd. The application is written in a combination of HTML, CSS, and C#. We will be using the Pitstop version which uses the Istio service mesh.

The architecture diagram can be seen in figure 11. The main interaction with the application is through the *Pitstop Web app* where the user can manage customers, vehicles, and the planning for the workshop. The frontend communicates with the three services that have an API and does not have any knowledge of the other services. The *customer management service* and *vehicle management service* allow for the creation and retrieval of customers and vehicles respectively.

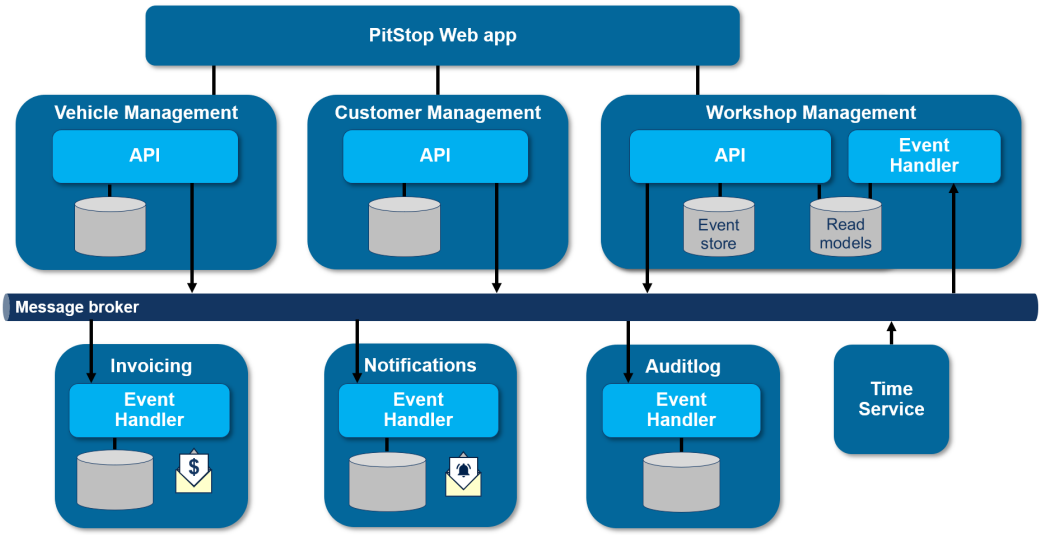

Figure 11: The architecture of the Pitstop application [41]

The *workshop management service* is slightly more complex. It consists of two parts. The first is an API that allows to schedule maintenance jobs and retrieve vehicle and customer data (in case one of these two services goes down).

The second part is an event-handler that handles incoming events and creates a read-model that is used by the API. The event handler receives this information from the message-broker. By building a read-model of all customer and vehicle information the workshop management service is able to schedule new jobs even when the customer of vehicle service is not available.

The *notification service* and *invoice service* listen to events from the message-broker and send notifications and invoices respectively. These two services do not offer an API and only act on the events that are received from the message broker. The *time service* is there to notify the other services through the message broker that a certain time has passed. And finally the *auditlog service* stores all the events that are published on the message broker for later reference.

Each service is deployed in its own pod on a Kubernetes cluster and the Istio serivce mesh is used to observe and manage the network traffic.

5 Model construction

In this chapter the extracted data from the YAML, the way that data is stored, and how that data is processed is explained. The model is the layer in between the YAML files which are used to start the service mesh and the simulation of that service mesh. The model contains only the relevant information from the YAML files and has been adjusted throughout the development of the simulation when extra information was required. To illustrate some of the components parts of YAML files are shown; these are taken from the Pitstop project [40].

5.1 Kubernetes object

```
1  apiVersion: extensions/v1beta1
2  kind: Deployment
3  metadata:
4    labels:
5      system: pitstop
6      app: workshopmanagementapi
7      version: "1.0"
8    name: workshopmanagementapi
9    namespace: pitstop
10 spec:
11     ...
```

Figure 12: An example of the shared variables for all Kubernetes objects.

The Kubernetes object is a class which is not defined by Kubernetes or Istio itself but rather a class that is used as a parent for the deployment, service, destination rule, gateway, and virtual service. The Kubernetes object contains all parameters that each of these child objects have. This group of parameters can be further divided into two subsets, one that contains parameters which are defined in the YAML file of the object and a group of parameters that is used for internal bookkeeping.

The YAML parameters consist of the *kind* and *metadata*. The *kind* is defined on line 2 of figure 12. In this example the kind is a Deployment but it can also be a Service, virtual service or any other type of Kubernetes object. The *metadata* that is defined on lines 3 till 9 has two important parameters, the first is the *name* (line 8) which is used to identify a Kubernetes Object. When Kubernetes Objects need to interact with each other (for example when sending network traffic) this is the identifier that is used to determine where the traffic has to be routed to. The second parameter is the set of *labels* that a Kubernetes object has. These labels are used to distinguish between different versions of a Kubernetes object; this will be explained in more detail later. The metadata

```
1   ...
2   spec:
3   replicas: 1
4   template:
5     metadata:
6       labels:
7         system: pitstop
8         app: workshopmanagementapi
9         version: "1.0"
10    spec:
11      containers:
12      - env:
13        - name: ASPNETCORE_ENVIRONMENT
14          value: Production
15        image: pitstop/workshopmanagementapi:1.0
16        imagePullPolicy: IfNotPresent
17        name: workshopmanagementapi
18        ports:
19        - containerPort: 5200
20      restartPolicy: Always
```

Figure 13: An example of the specification for a Deployment.

also has a *namespace* which is used to group Kubernetes objects together, but as the projects that are used only consist of one namespace we will disregard it.

The parameters that are kept for internal bookkeeping consist of the *links* that a Kubernetes object has. These are used to connect related Kubernetes objects together (e.g. a Service and its corresponding Deployment). In a Kubernetes object *subsets* that apply to that object are also stored (see section 5.4). The *maximum traffic* that a single instance of a Kubernetes object is allowed to have is also stored along with further *traffic data* (see section 5.7). Finally each Kubernetes object gets a unique *id*.

The *spec* part of the YAML file (line 10) is different for each of the different Kubernetes objects and will be explained in the following sections.

5.2 Deployment

An example of the YAML for a Deployment can be seen in figure 13. There are a few interesting parameters in this YAML file. The first is the number of *replicas* as can be seen on line 3. This defines the number of instances of this deployment. The *ports* parameter as defined on line 18 and 19 tell which ports of the deployment can be connected to. Finally the *restartPolicy* is used to determine what has to be done when an instance of the deployment crashes. Information like the *environment variables* (line 12), *image* (line 15),

```
1   ...
2   spec:
3   type: NodePort
4   ports:
5   - name: "http-5200"
6     port: 5200
7     targetPort: 5201
8     nodePort: 30007
9   selector:
10    system: pitstop
11    app: workshopmanagementapi
12    version: "1.0"
```

Figure 14: An example of the specification for a Service.

and *imagePullPolicy* (line 16) could be interesting for additional information in future versions of the model but are currently not used.

5.3 Service

We now look into the YAML of a service, see figure 14. The YAML of the service has two pieces of important information, the *ports* (lines 4-8) and the *selector* (lines 9-12). A service can have multiple exposed ports and each of them can have a *name*, *port*, *targetPort*, and *nodePort*. The *name* is just for convenience but is kept in the model so that it can be used for communication to the end user (e.g. "There is a misconfiguration in the targetPort of http-5200"). The *port* is the port on which the service is exposed within the Kubernetes cluster (that is, for other containers on the cluster). The *targetPort* is the port to which the traffic is forwarded, e.g., if the Service receives traffic on port 5200 from another container it will forward this traffic to port 5201 of the Deployment that matches the selection criteria. The service can also be reached from outside the Kubernetes cluster on the *nodePort*. The *selector* parameter defines to which deployments the Service applies. This is done by matching the labels of the selector (lines 10-12) with the labels that are defined in the metadata of the deployments. This data is stored in the model to link services to their respective deployments.

5.4 Destination rule

The destination rule is simple but important part of the model. The destination rule in the model is used for two things. First of all the *subsets* define subsets within deployments. In the example (figure 15) a subset "V1" is defined on lines 5-7. The labels of subset are matched against the labels of a deployment. If the labels of the subset match the labels of the deployment, the

```
...
spec:
    host: customermanagementapi
    subsets:
      - name: v1
        labels:
          version: "1.0"
      - name: v2
        labels:
          version: "2.0"
    trafficPolicy:
      connectionPool:
        tcp:
          maxConnections: 1
        http:
          http1MaxPendingRequests: 1
          maxRequestsPerConnection: 1
      outlierDetection:
        consecutiveErrors: 1
        interval: 1s
        baseEjectionTime: 3m
        maxEjectionPercent: 100
```

Figure 15: An example of the specification for a destination rule.

```
1  ...
2  spec:
3    hosts:
4      - customermanagementapi
5    http:
6      - route:
7        - destination:
8            host: customermanagementapi
9            subset: v1
10         weight: 75
11       - destination:
12           host: customermanagementapi
13           subset: v2
14         weight: 25
```

Figure 16: An example of the specification for a virtual service.

subset is added to the deployment. Other deployments can specify a subset which they want to use, in this case the "V1" subset. Second, the *trafficPolicy* (lines 11-22) is used to define complex traffic management systems like the circuit breaker. The *trafficPolicy* consists of two elements; the *connectionPool* (lines 12-17) which controls the volume of connections to an upstream service (e.g. a deployment) and the *outlierDetection* which is used to remove unhealthy containers from the load balancing pool.

5.5 Virtual service

To make use of the subsets that are defined by the destination rule in section 5.4, the example YAML in figure 16 allows for canary releasing. All traffic for the host defined on line 4 is routed to the destinations on lines 7-14. In this example 75% of traffic is sent to V1 of customermanagementapi and 25% is sent to the V2 of customermanagementapi.

5.6 Gateway

Request that come into the network need to have an origin. To simulate this, a gateway has been added to the model. The gateway does not come from a YAML file, but is a self created object that allows the end user to specify initial requests. The object does not have any extra parameters, but instead the already existing *trafficData* is used. The end user can specify different sets of traffic as explained in section 5.7.

5.7 Traffic data

```
1  - name: Gateway
2    maxTraffic: 0
3    requestsPerSecond: 1
4    secondsPerRequest: 1
5    updateRequestsPerSecond:
6      2: 5
7    updateSecondsPerRequest:
8      2: 3
9    trafficSets:
10     - probability: 100
11       traffic:
12         logserver: -1
13         vehiclemanagementapi: -1
14         webapp: 1
15 ...
```

Figure 17: An example of the specification for Traffic data.

The traffic data does not inherit from the Kubernetes object, it is a parameter of the Kubernetes object. After all Kubernetes objects have been created and linked to each other, the end user can generate the YAML file for the traffic data, e.g. something like the YAML file in figure 17. In this example the Gateway has been configured. First of all *maxTraffic* is set to 0, which means that there is no limit to the amount of traffic that this instance can receive. It can also be set to any non-zero value which would limit the amount of traffic such an instance is allowed to receive to the given number. Furthermore the Gateway has one trafficSet defined (lines 9-14). A trafficSet has a probability (on a 0 to 100 scale) which determines the chance that this trafficSet is chosen. If this trafficSet is chosen it will send one packet to the webapp (line 14). The trafficSet also defines how many packets a deployment can handle, and how long it takes for a packet to be handled. The "requestsPerSecond" (line 3) determines the number of packets that a deployment can handle each second. The "secondsPerRequest" (line 4) determines how many seconds it takes for the deployment to handle 1 packet. Finally it is also possible to change the "requestsPerSecond" and "secondsPerRequest" based on the time (lines 5-8). This allows us to change the speed of a deployment at runtime. This is all the data we need for defining the traffic within the simulation.

6 Log analysis

An important part of the simulation tool is the log analysis. The log analysis is used to determine how much traffic is sent within the service mesh and also to construct traffic sets from this logging. This data is used to determine the traffic within the simulation of the service mesh. First we explain the differences in logging between the real service mesh and the simulation, after which we explain how this information is used to create the traffic sets and statistics.

6.1 Real service mesh

The real service mesh has log entries that look like figure 18.


```
{
    "downstream_remote_address": "10.1.5.77:45134",
    "authority": "vehiclemanagementapi:5000",
    "path": "/api/vehicles",
    "protocol": "HTTP/1.1",
    "upstream_service_time": "11",
    "upstream_local_address": "10.1.5.77:54684",
    "duration": "11",
    "upstream_transport_failure_reason": "-",
    "route_name": "default",
    "downstream_local_address": "10.104.30.166:5000",
    "user_agent": "-",
    "response_code": "200",
    "response_flags": "-",
    "start_time": "2021-01-02T11:40:09.046Z",
    "method": "GET",
    "request_id": "bbb20802-f5cc-908a-9da2-e3f86ffdaf8e",
    "upstream_host": "10.1.5.75:5000",
    "x_forwarded_for": "-",
    "requested_server_name": "-",
    "bytes_received": "0",
    "istio_policy_status": "-",
    "bytes_sent": "2",
    "upstream_cluster":
    ↪  "outbound|5000||vehiclemanagementapi.pitstop.svc.cluster.local"
}
```


Figure 18: A log entry from the real service mesh.

The log entry contains a lot of information but only a part of it is used. To determine how much traffic is sent within the service mesh we need to know a few things:

1. When a request is sent
2. By whom the request is sent
3. To whom the request is sent

Sadly, only the first two pieces of information are directly present in the log entry. We know when the request is sent by looking at the "start_time" and we know where the request is going by looking at the "authority" and the "downstream_local_address".

When only one replica of a deployment is used the "authority" suffices, but when there are multiple replicas of a deployment we can not see the difference as the "authority" field will be the same for all of them. If multiple replicas of the same deployment are present, the "downstream_local_address" is used to distinguish between them. A dictionary is constructed with the address for each the deployments and a unique identifier is added to the authority based on their address so we can later see the difference in the authority field. A piece of pseudo code for this can bee seen in figure 19.

```
1  LogEntry log;
2  Dictionary<IP, int> unique_id;
3  string authority;
4
5  // This is true when there are multiple replicas
6  if (authority_not_unique)
7      IP downsteam_local_address = log.downsteam_local_address;
8      int ID = unique_id[downsteam_local_address]
9      authority += ID;
10     return authority;
11 else
12     return authority;
```

Figure 19: A piece of pseudo code to distinguish between multiple replicas of a deployment.

The final piece of information is derived from the file name of the log file. The script that extracts the log data from all the pods in the service mesh names the log file after the pod where the log data comes from. We use this information to determine the source of the request.

The log file also contains a lot of log entries that contain information that we do not want to use. Sometimes a pod sends data to itself, or sends data to the service mesh control plane. These entries are filtered by removing log entries that have no "method" set on them.

The log file also contains some log lines that contain information about the state of the pod. These lines are formatted differently (these lines are normal

text instead of JSON) and are filtered by removing all lines that are no JSON. The resulting log entries are converted into log entries that only contain the "authority" (with the unique identifier), "start_time", and "source" fields.

6.2 Simulation

For the log entries of the simulation it is easier to extract the right information. When a request is sent within the simulation the source, destination, and time are recorded. This data is written to a log file which is used for the analysis later on.

6.3 Analysis

As we will see later on in section 8 there are three types of statistics that we generate from the log information:

1. Number of requests within a given block of time.
2. Total number of request from and to any of the services within the service mesh.
3. Average number of request per second from and to any of the services within the service mesh.

The third item (the average number of requests per second) is used to create the traffic data.

First of all, the log entries are filtered. This is done by checking if the source and authority of the log entry are known within the simulation. If a log entry has a source or authority that is not known within the YAML files, we assume that it is not part of the simulation and discard the log entry. The log entries are then sorted by their start_time to put them in chronological order.

A first sweep over all the log entries is made. This sweep takes note of the start_time of the log entries and counts how many log entries there are within seconds 0 to 5 of the log entries. Then we skip one second and look for all the log entries within seconds 1 to 6 and we repeat this process until we get to seconds 55 to 60. This creates the first set of statistics, the number of requests within a given block of time.

Then we do a second sweep over all the log entries and we count how many requests are sent from any source to any given destination. This is done by using a nested dictionary. This is seen in part 1 of figure 20.

Finally we also want to know how many requests are sent on average per second. To calculate this we loop over the dictionary. For each of the entries we divide the total number of request by the number of seconds between the first and last log entry. This is seen in part 2 of figure 20.

All these statistics are saved to files which are later imported into a spreadsheet to analyse them.

```
LogEntry[] logs;
Dictionary<string, Dictionary<string, int> counter;
int runtime = logs.last().start_time - logs.first().start_time;

// 1. Counting the number of requests in total
foreach (LogEntry log in logs)
    counter[log.source][log.destination]++;

// 2. Determining the number of requests per second
foreach (keyValuePair total_requests in counter)
    requests_per_second = total_requests / runtime;
```

Figure 20: A piece of pseudo code to count the number of requests.

7 Tool implementation

To get a good idea of how the simulation tool works it is also important to know what information is fed into the tool and where that information comes from. Therefore the system that we use to provide data for the simulation tool will be discussed first, whereafter the simulation tool is discussed. The developed simulation tool is used to make predictions about the service mesh. For example, when a developer wants to create a new endpoint the simulation can be used to predict how the traffic will influence the entire service mesh.

7.1 The example program, Pitstop

7.1.1 Data gathering

It has already been mentioned earlier that the Pitstop [40] application is used as the example application for the simulation tool. Pitstop is a fully working web application that is build using the Istio service mesh. Pitstop is solely configured using YAML files, all required configuration is specified within these files. These YAML files are written before the network is started and all have to be correct in order for the service mesh to function properly. This is also where the problem comes in, a mistake is easily made in one of the many files and only one mistake can cause the entire application to be slow or malfunction.

When Pitstop is running it generates log entries. These log entries are analysed and used as input for the simulation tool. Later on the log entries of the simulation are compared to the log entries of Pitstop. When the Pitstop application is running the pods make requests to other pods over HTTP. These requests are routed through the sidecars that are injected by the Istio service mesh. These HTTP requests are logged by Envoy. This logging information, combined with the YAML files that are used to start the network, provide enough information for the simulation tool to reconstruct the service mesh and create a simulation.

7.2 The simulation tool

The simulation tool will take the YAML files that are used to start a service mesh (in this specific example the Pitstop application) and the logging files that are generated by the service mesh to create a simulation of that service mesh. This simulation allows developers to test new configuration without having to set-up and run an entire network or alter an existing one.

To use the tool, first of all the YAML files that are used to start the service mesh have to be loaded into the tool. These files are kept in memory and will later on be used to construct the pods of the service mesh. The logging files that are generated by the real service mesh can also be loaded; these files are analysed as specified in section 6. After the log analysis, traffic sets are created and added to the information for each of the pods. If for whatever reason there are no log files you can also manually create these traffic sets for each of the

pods. The downside of this manual creation is the immense amount of work that it takes to correctly configure each of the traffic sets. Manually creating the traffic sets can also result in an inaccurate simulation when the entered traffic information is incorrect.

These first information loading steps can also been seen in figure 21.

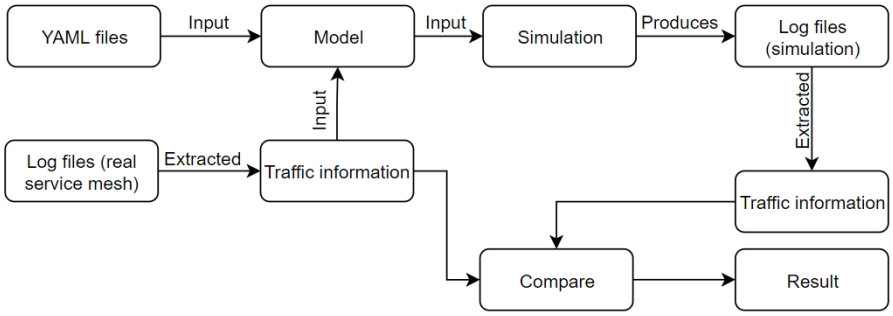

Figure 21: A flow chart for the creation and comparison of the simulation and the real service mesh.

When the YAML are loaded there is a Kubernetes object for each of the services or deployments and one for the gateway (the gateway is used to initiate traffic to the service mesh). All of these Kubernetes objects have the settings from the YAML files and trafficsets that are generated from the logging or entered manually. The developer can start the simulation with the information that is entered into the simulation.

When the simulation starts for each of the Kubernetes objects a simulation instance is made. These instances are there just for the simulation and only contain the information that is needed for a simulation. Each of the Kubernetes objects gets at least one simulation instance. In case of a service and deployment pair, a simulation instance is made for the service first. After that, a simulation instance is made for each of the replicas of the deployment. All the simulation instances that are made for the replicas of the deployment are linked to the simulation instance of the service. At this point all simulation instances are made and have all the information required to start the simulation.

Within the simulation there is a DNS like system for looking up the services and deployments. All traffic for a deployment has to be routed through the accompanying service. This is implemented by keeping a central dictionary with the name of a simulation instance (for example “vehiclemanagementapi”) as the index and the corresponding simulation instance of the service as the value. By doing this, any of the simulation instances can do a lookup of the name that it wants to connect to and receive the simulation instance to which the traffic has to be sent. This DNS like discovery system used by all the simulation instances

within the simulation.

When the simulation starts, a random trafficSet is chosen from the gateway, the traffic in this trafficset is sent to the respective simulation instances. This is always a service (as a deployment is not directly exposed to the other simulation instances). The service will then propagate the traffic to one of the Deployments that are attached to it. This also allows the service to choose which deployment the traffic is being sent to, enabling load distribution within the simulation of the service mesh. The service also keeps track of the circuit breaker of the deployments, if a deployment has a circuit breaker active it won't receive traffic from the service. After the gateway has sent its data all the deployments in the simulation are visited and get the chance to send traffic if they have pending requests.

Any traffic that is being sent from the service to a deployment is added to the "incoming traffic" of that deployment. This is done to separate the incoming traffic from the traffic that was already queued in the deployment. If the traffic was directly added to the traffic of the deployment it could occur that the traffic would go from the service to a deployment and would be handled immediately when the deployment got the chance to handle and sent traffic. This would cause one piece of traffic to be handled twice within one tick of the simulation which is not correct.

Whenever traffic is sent from the gateway to a deployment or from a deployment to another deployment a log entry is made. All these log entries are saved and are used to generate statistics for the simulation. These statistics are used later on to verify and validate the simulation of the service mesh to the real service mesh.

At the end of a simulation tick a snapshot is taken with the distribution of traffic within the simulation. This data is saved to a JSON file and opened in a browser for the user to view. This view essentially gives a visualisation to the simulation of the service mesh.

8 Verifying the simulation

Before doing anything else with the simulation of the service mesh, it is important to know how accurate the simulation is. For this we developed a tool that uses the logging of the service mesh or the service mesh simulation and generates some statistics about the network traffic that took place within the service mesh. Given the logging of the service mesh the tool will output three statistics:

1. The number of requests within a given block of time
2. The total number of request from and to any of the services within the service mesh
3. The average number of request per second from and to any of the services within the service mesh

By running experiments on different kinds of service meshes we see how the simulation performs under varying circumstances. The experiments will vary in the amount that is put on the service mesh or by the configuration of the service mesh itself. For example, the number of deployments or the number of replicas of a deployment is varied. The configuration of the real service mesh will be copied into the simulation and the same load is applied. By doing this we create the same environment within the simulation and the real service mesh. Afterwards we compare the log output and see if the real service mesh and the simulation behaved in the same way. The log output of the real service mesh are used as the ground truth, the log output of the simulation are verified against it.

By varying the load and number of deployments we replicate a wide range of conditions that are possible in the real world. In the real world we see a variety of load and configurations which we mimic in these experiments.

We first explain how the metrics are calculated whereafter we show the results of the experiments.

8.0.1 Number of requests within a give block of time

It is important to know how many requests per second are being sent within the service mesh. If we only look at the total number of request within a given time we won't be able to see any spikes in traffic that could cause the service mesh to be overwhelmed. Therefore we take a sliding window approach over the requests that are sent within the network. The tool starts at the begin of the logging and checks how many request are sent within a five second window. This window slides over the full set of log entries and creates a graph from the result, an example of such a graph can be seen in figure 26. From now on we will refer to this graph as the "traffic graph".

8.0.2 Total number of requests

It is also important to know how many request are sent in total in a given amount of time. When comparing the simulation of the service mesh with a real service mesh we will also take the total number of requests into account. When we use table 1 as an example we see that the gateway sends 10 requests to the webapp and that the webapp in turn sends 10 requests to the customermanagementapi (abbreviated as "customer") and the logserver. All of these values have a standard deviation of 0, which is shown after the "+-". The items in the left most column are the "senders" and the items on the horizontal row are the "receivers". In these tables the names of the senders and receivers of traffic might be abbreviated to decrease the size of the table, below you find the list of abbreviations:

- vehiclemanagementapi → vehicle
- vehiclemanagementapi0 → vehicle0
- vehiclemanagementapi1 → vehicle1
- customermanagementapi → customer

When looking at table 2 you see that there are also values between parentheses "()". These are the values of the simulation. In this table you see that the webapp has send a total of 29,8 request to the logserver, with a standard deviation of 0,4. The simulation has send 60 request to the logserver.

	customer	Gateway	logserver	webapp	Total
customer	0 +- 0	0 +- 0	0 +- 0	0 +- 0	0 +- 0
Gateway	0 +- 0	0 +- 0	0 +- 0	10 +- 0	10 +- 0
logserver	0 +- 0	0 +- 0	0 +- 0	0 +- 0	0 +- 0
webapp	10 +- 0	0 +- 0	10 +- 0	0 +- 0	20 +- 0

Table 1: All requests for the first set of manually constructed log files.

	logserver	vehicle	webapp	Total
logserver	0 +- 0 (0)	0 +- 0 (0)	0 +- 0 (0)	0 +- 0 (0)
vehicle	30 +- 0 (59)	58,1 +- 0,3 (0)	0 +- 0 (0)	88,1 +- 0,3 (59)
webapp	29,8 +- 0,4 (60)	57,8 +- 0,6 (60)	0 +- 0 (0)	87,6 +- 0,66 (120)

Table 2: All requests for the first version of the low traffic verification.

8.0.3 Average number of requests

To get a good feeling of how much traffic is sent per second there is also a representation similar to the previously explained trafficset. When looking at figure 22 you see an example of this. For each of the deployments there is a

specification that shows how many requests are sent to other deployments per second. You also see that there are values between parentheses "()". These are the values of the simulation. In this table you see that the webapp on average sends 0,96 requests per second to the vehiclemanagementapi, with a standard deviation of 0,01. The simulation sends 1 request on average from the webapp to the vehiclemanagementapi.

```
webapp:
    - vehiclemanagementapi: 0,96 +- 0,01 (1)
    - logserver: 0,5 +- 0 (1)
vehiclemanagementapi:
    - vehiclemanagementapi: 0,97 +- 0,01 (0)
    - logserver: 0,5 +- 0 (1)
```

Figure 22: The average number of requests with standard deviation for the first version of the low traffic verification.

8.1 Verifying the tool for verification

Before we use the aforementioned tool to verify the simulation of the service mesh, we will first verify the tool itself. We do this by manually constructing three sets of logging data with predictable outcomes and checking this with the output of the tool. If the outputs of the tool match the expected output for the logging we will assume that the tool is valid. In figure 23 a portion of the first piece of manual logging data is shown. The logging shows the first four seconds of network traffic. Every second The "gateway" sends a request to the "webapp" and then the "webapp" sends a request to "customermanagementapi" and the "logserver". The full logging has a length of ten seconds so we expect the "gateway" to send ten requests, the "webapp", "customermanagementapi", and "logserver" to receive ten request and finally expect the "webapp" to send twenty requests. A small clarification on the contents of the log file: the "source" is the source of the request and the "authority" is the target of the request.

```
1  {..., "start_time":"2020-12-02T00:00:00+01:00",
   ↪  "authority":"webapp", "source":"Gateway"}
2  {..., "start_time":"2020-12-02T00:00:00+01:00",
   ↪  "authority":"customermanagementapi", "source":"webapp"}
3  {..., "start_time":"2020-12-02T00:00:00+01:00",
   ↪  "authority":"logserver", "source":"webapp"}
4  {..., "start_time":"2020-12-02T00:00:01+01:00",
   ↪  "authority":"webapp", "source":"Gateway"}
5  {..., "start_time":"2020-12-02T00:00:01+01:00",
   ↪  "authority":"customermanagementapi", "source":"webapp"}
6  {..., "start_time":"2020-12-02T00:00:01+01:00",
   ↪  "authority":"logserver", "source":"webapp"}
7  {..., "start_time":"2020-12-02T00:00:02+01:00",
   ↪  "authority":"webapp", "source":"Gateway"}
8  {..., "start_time":"2020-12-02T00:00:02+01:00",
   ↪  "authority":"customermanagementapi", "source":"webapp"}
9  {..., "start_time":"2020-12-02T00:00:02+01:00",
   ↪  "authority":"logserver", "source":"webapp"}
10 {..., "start_time":"2020-12-02T00:00:03+01:00",
   ↪  "authority":"webapp", "source":"Gateway"}
11 {..., "start_time":"2020-12-02T00:00:03+01:00",
   ↪  "authority":"customermanagementapi", "source":"webapp"}
12 {..., "start_time":"2020-12-02T00:00:03+01:00",
   ↪  "authority":"logserver", "source":"webapp"}
13 ...
```

Figure 23: A part of manually constructed logging data to verify the verification tool.

The first set of log entries is an extension of figure 23. It has ten seconds of request that are sent and we expect the output as described before. We will verify this with the table output of the developed verification tool. The output can be seen in Table 3. When looking at the table we see that it matches the predicted output. The "Gateway" sends ten requests to the "webapp" which in turn sends ten requests to both the "customermanagementapi" and the "logserver". In figure 24 the number of request with the sliding window is seen. It shows that in the first set of logging, fifteen requests per five seconds are sent. As there is only one dataset in this graph there is no standard deviation and thus only the average line is plotted. Finally, in figure 25 the average number of requests can be seen.

The second set of manually constructed log entries is very similar to the first set, only the amount of traffic per second has been doubled. We thus expect the

traffic in the table with all the traffic to be double of the first set of manually constructed log entries. This can be seen in table 8.1. For the traffic graph and the average number of requests we have used both the first and second set of manually constructed log entries. For the traffic graph we expect to see the average of 15 and 30 (which is 22,5) with a standard deviation of 7,5. This is indeed what we see in figure 26. For the average number of request we expect to see 1,5 requests with a standard deviation of 0,5 for all requests, which is confirmed by figure 28.

For the third and final set we will only look at the total number of requests and average number of requests (because the traffic graph will be exactly the same as the one for the first set of log entries). For the third set of manually constructed log entries we took the first set of constructed log entries and reversed the "source" and "authority". The results can be seen in table 8.1 and figure 8.1. The outputs of the verification tool meet the expected outputs.

	customer	Gateway	logserver	webapp	Total
customer	0 +- 0	0 +- 0	0 +- 0	0 +- 0	0 +- 0
Gateway	0 +- 0	0 +- 0	0 +- 0	10 +- 0	10 +- 0
logserver	0 +- 0	0 +- 0	0 +- 0	0 +- 0	0 +- 0
webapp	10 +- 0	0 +- 0	10 +- 0	0 +- 0	20 +- 0

Table 3: All requests for the first set of manually constructed log files.

Figure 24: Requests per second for the first set of manually constructed log files.

```
Gateway:
    - webapp: 1 +- 0
webapp:
    - customermanagementapi: 1 +- 0
    - logserver: 1 +- 0
```

Figure 25: The average number of requests with standard deviation for the first set of manually constructed log entries.

	customer	**Gateway**	**logserver**	**webapp**	**Total**
customer	0 +- 0	0 +- 0	0 +- 0	0 +- 0	0 +- 0
Gateway	0 +- 0	0 +- 0	0 +- 0	20 +- 0	20 +- 0
logserver	0 +- 0	0 +- 0	0 +- 0	0 +- 0	0 +- 0
webapp	20 +- 0	0 +- 0	20 +- 0	0 +- 0	40 +- 0

Table 4: All requests for the second set of manually constructed log files.

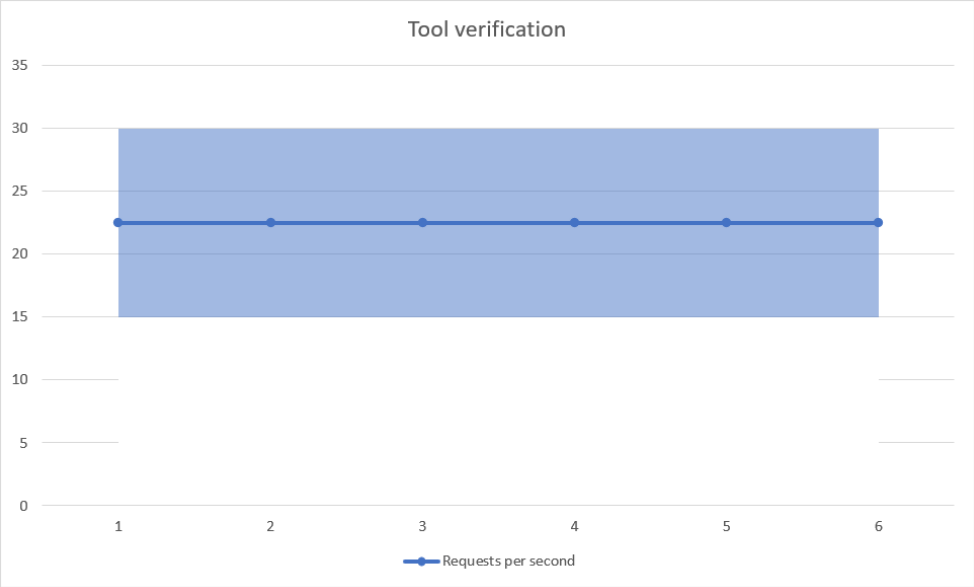

Figure 26: Requests per second for the first and second set of manually constructed log files.

```
Gateway:
    - webapp: 1,5 +- 0,5
webapp:
    - customermanagementapi: 1,5 +- 0,5
    - logserver: 1,5 +- 0,5
```

Figure 27: The average number of requests with standard deviation for the first and second set of manually constructed log entries.

	customer	**Gateway**	**logserver**	**webapp**	**Total**
customer	0 +- 0	0 +- 0	0 +- 0	10 +- 0	10 +- 0
Gateway	0 +- 0	0 +- 0	0 +- 0	0 +- 0	0 +- 0
logserver	0 +- 0	0 +- 0	0 +- 0	10 +- 0	10 +- 0
webapp	0 +- 0	10 +- 0	0 +- 0	0 +- 0	10 +- 0

Table 5: All requests for the third set of manually constructed log files.

```
customermanagementapi:
    - webapp: 1 +- 0
logserver:
    - webapp: 1 +- 0
webapp:
    - gateway: 1 +- 0
```

Figure 28: The average number of requests with standard deviation for the third set of manually constructed log entries.

8.2 Verifying the simulation

All of the experiments are done by executing the following steps:

1. Determine what type of verification is needed.
2. Start the real service mesh with the necessary deployments, services, etc.
3. Start a script that saves the logging of all pods to separate files.
4. Use a load tester with a pre-configured amount of load to put load on the service mesh for 60 seconds.
5. Stop the load test and script that saves the logging files.

The steps above are repeated 10 times. All the results are passed through the log analyser and are then collected in a spreadsheet where the mean values and

standard deviation are calculated. From the resulting average traffic data is constructed by the simulation tool, which is used to start the simulation. The logging from the simulation is also passed through the log analyser and placed in the spreadsheet. This information is used to determine how good the simulation matches the real service mesh.

8.2.1 Verification with low volume traffic V1

The first experiment is done on a simple service mesh. The service mesh consists of a webapp which sends one request per second to the vehiclemanagementapi. The vehiclemanagementapi handles the requests from the webapp and sends some requests to the logserver. We immediately discover a problem with the way that the simulation handles "half requests", which is seen in table 6 and corresponding figure 30. In the 60 seconds that the service mesh received load, 30 requests are made from the webapp and vehiclemanagementapi to the logserver. This comes down to 0,5 request per second, which is rounded to 1 request per second by the simulation tool. As this is behavior that occurs in a lot of simulations we have decided to help the tool with these cases and add two trafficSets to the traffic data. One trafficSet sends 0 requests to the logserver whereas the other set sends 1 request to the logserver. Both of these trafficSets have a probability of 50% which results in an average of 0,5 request per second.

We also see another discrepancy, in the real service mesh the vehiclemanagementapi sends data to itself, as is seen in figure 30. You see that the vehiclemanagementapi sends all requests that it receives to itself. This is due to the fact that a pod consists of two parts: the service and the deployment. When the service receives the request from the webapp the service will forward this request to a deployment. This causes any request to the vehiclemanagementapi or customermanagementapi to be logged twice, once when it is send to the service and a second time when it is forwarded to the corresponding deployment. The simulation only logs these requests once which causes the discrepancy. In all the verification and validation experiment the size of this discrepancy will be given and is accounted for. In the graph a line with the given discrepancy is added for an easier visualisation. This line is called the "simulation corrected".

	logserver	vehicle	webapp	Total
logserver	0 +- 0 (0)	0 +- 0 (0)	0 +- 0 (0)	0 +- 0 (0)
vehicle	30 +- 0 (59)	58,1 +- 0,3 (0)	0 +- 0 (0)	88,1 +- 0,3 (59)
webapp	29,8 +- 0,4 (60)	57,8 +- 0,6 (60)	0 +- 0 (0)	87,6 +- 0,66 (120)

Table 6: All requests for the first version of the low traffic verification.

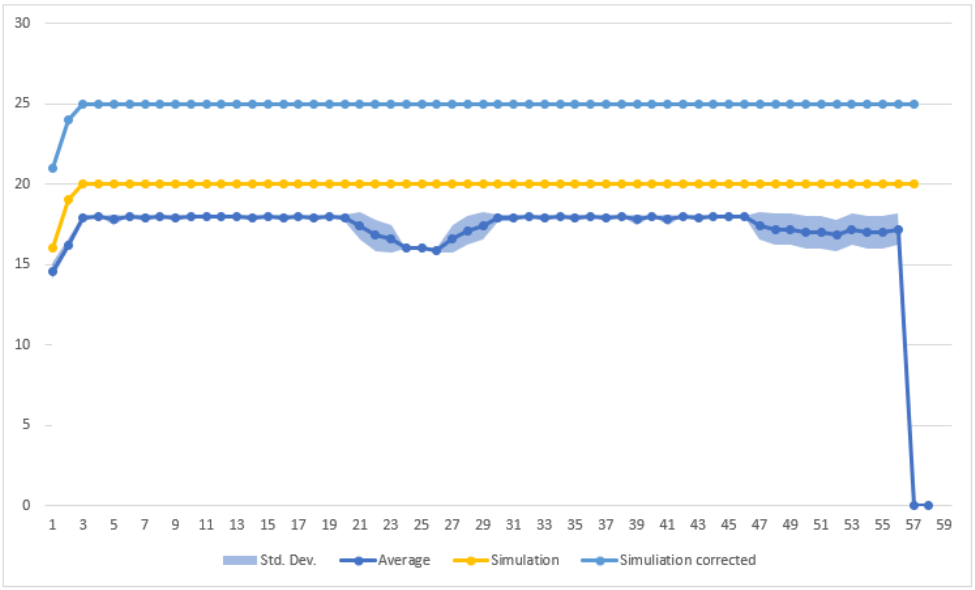

Figure 29: Requests per second for the first version of the low traffic verification.

```
webapp:
    - vehiclemanagementapi: 0,96 +- 0,01 (1)
    - logserver: 0,5 +- 0 (1)
vehiclemanagementapi:
    - vehiclemanagementapi: 0,97 +- 0,01 (0)
    - logserver: 0,5 +- 0 (1)
```

Figure 30: The average number of requests with standard deviation for the first version of the low traffic verification.

8.2.2 Verification with low volume traffic V2

The traffic and load in this experiment are identical to the previous one, except for the change in traffic data (the two trafficSets that are added to account for the "half requests"). When we look at table 7 we see that the simulation is nearly identical to the real service mesh.

When we look at figure 31 we also see that the simulation follows the trend in increasing/decreasing traffic over time. Finally we also see that the traffic data (figure 32) of the simulation is nearly identical to the real service mesh.

	logserver	vehicle	webapp	Total
logserver	0 +- 0 (0)	0 +- 0 (0)	0 +- 0 (0)	0 +- 0 (0)
vehicle	30 +- 0 (29)	58,1 +- 0,3 (0)	0 +- 0 (0)	88,1 +- 0,3 (29)
webapp	29,8 +- 0,4 (29)	57,8 +- 0,6 (60)	0 +- 0 (0)	87,6 +- 0,66 (89)

Table 7: All requests for the second version of the low traffic verification.

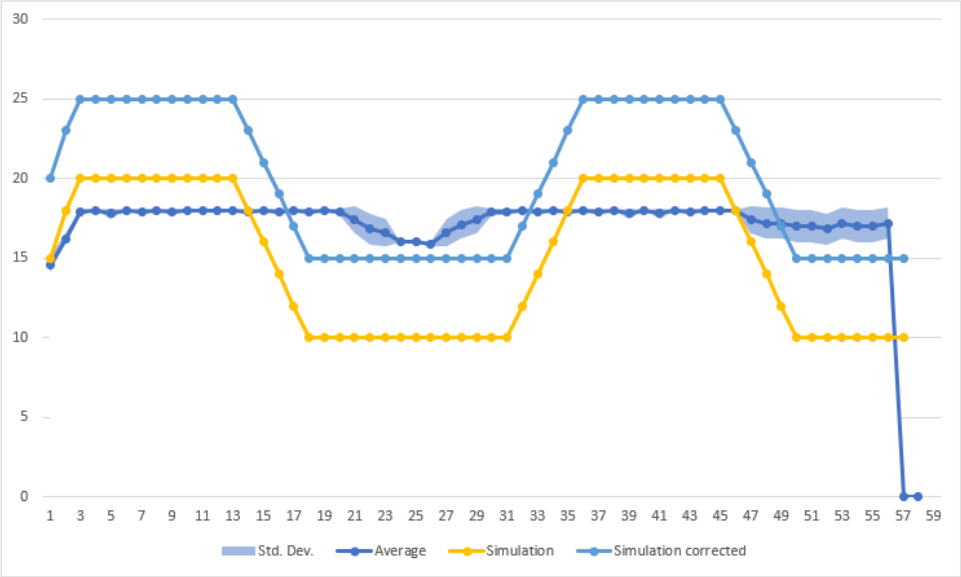

Figure 31: Requests per second for the second version of the low traffic verification.

```
webapp:
    - vehiclemanagementapi: 0,96 +- 0,01 (1)
    - logserver: 0,5 +- 0 (0,5)
vehiclemanagementapi:
    - vehiclemanagementapi: 0,97 +- 0,01 (0)
    - logserver: 0,5 +- 0 (0,5)
```

Figure 32: The average number of requests with standard deviation for the second version of the low traffic verification.

8.2.3 Verification with medium traffic

For the second experiment we have increased the amount of requests that are sent from the webapp to the vehiclemanagementapi. In the previous experiment

1 request per second was sent, whereas in this experiment we send 5 requests per second from the webapp to the vehiclemanagementapi. In figure 34 we see that by putting a load of 5 requests per second on the webapp the service mesh actually only sends 4 request per second. This has to do with the timing within the script that puts the load on the service mesh, leading to a lower actual amount of requests that are send. But because the simulation uses the logging data of the real service mesh to create the traffic data, it also copies this lower amount of requests per second.

When we take a look at figure 33 we see that that the requests per five seconds is significantly lower for the simulation. This is due to the earlier mentioned discrepancy. Because in this simulation 4 requests per second are sent and the size of the sliding window is 5 we expect the traffic graph to have a discrepancy of about 20. In this case the discrepancy is bigger, which is explained by table 8. We see that the vehiclemanagementapi only sends 3 requests in the entire 60 second simulation. After reviewing the logs we found that this was caused by randomness: the trafficSet where no requests to the logserver are sent was chosen far more often than the set where a request to the logserver was sent.

	logserver	vehicle	webapp	Total
logserver	0+-0 (0)	0+-0 (0)	0+-0 (0)	0+-0 (0)
vehicle	29,9+-0,3 (3)	250,5+-0,67 (0)	0+-0 (0)	280,4+-0,8 (3)
webapp	29,8+-0,4 (27)	250,4+-0,66 (240)	0+-0 (0)	280,2+-0,87 (267)

Table 8: All requests for the medium traffic verification.

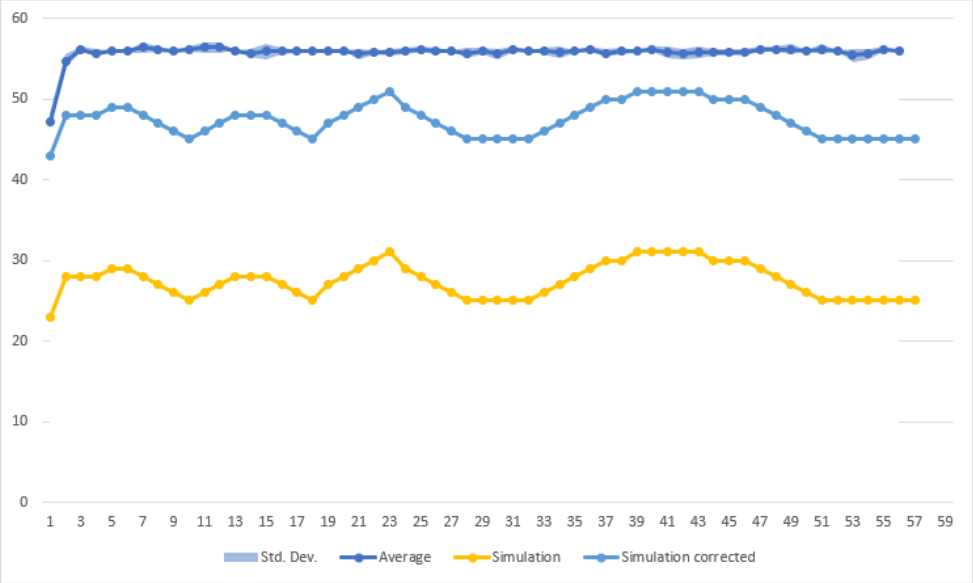

Figure 33: Requests per second for the medium traffic verification.

```
webapp:
    - vehiclemanagementapi: 4,17 +- 0,01 (4)
    - logserver: 0,5 +- 0,01 (0,45)
vehiclemanagementapi:
    - vehiclemanagementapi: 4,18 +- 0,01 (0)
    - logserver: 0,5 +- 0,01 (0,05)
```

Figure 34: The average number of requests with standard deviation for the medium traffic verification.

8.2.4 Verification with high traffic

For this experiment the same network was used as in the previous experiments, but this time the load was increased from 5 requests per second to 20 requests per second. Similar to the last experiment we can see in figure 36 that not 20 requests per second are sent, but only 12. Just like the last time the simulation uses these numbers and also sends only 12 requests per second. We also see that the number of requests that the vehiclemanagementapi sends to the logserver is a lot higher than the expected 0,5. This is caused by the way that the real service mesh sends requests to the logserver. The real service mesh has code that sends a request to the logserver every 2 seconds if the vehiclemanagementapi receives traffic. Because the traffic data for the service mesh is set to a 50% chance

for each of the incoming requests, and 12 requests per seconds are sent to the vehiclemanagementapi, the simulation will send an average of about 6 requests per second to the logserver. The current implementation of the simulation does not allow for these time based requests to be sent and will thus have a discrepancy.

When we take a look at traffic graph 35 we see that the discrepancy between the simulation and the real service mesh is approximately 50, which matches the expected 12 requests multiplied by the 5 second time window.

	logserver	vehicle	webapp	Total
logserver	0+-0 (0)	0+-0 (0)	0+-0 (0)	0+-0 (0)
vehicle	26,8+-0,4 (325)	656,8+-4,5 (0)	0+-0 (0)	683,6+-4,68 (325)
webapp	29,8+-0,4 (29)	730,2+-4,53 (720)	0+-0 (0)	760+-4,49 (732)

Table 9: All requests for the high traffic verification.

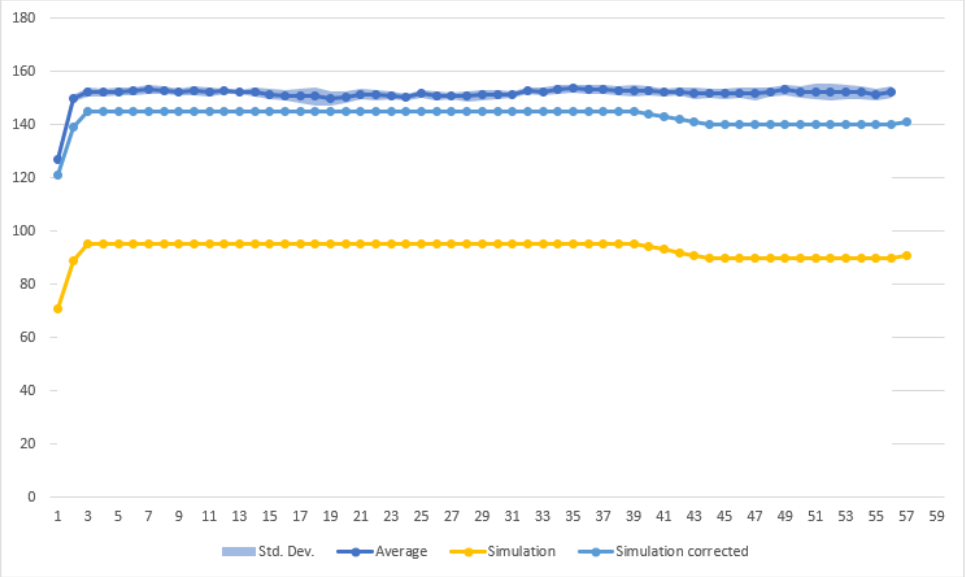

Figure 35: Requests per second for the high traffic verification.

```
webapp:
    - vehiclemanagementapi: 12,25 +- 0,08 (12)
    - logserver: 0,5 +- 0,01 (0,45)
vehiclemanagementapi:
    - vehiclemanagementapi: 10,95 +- 0,08 (0)
    - logserver: 0,45 +- 0,01 (5,4)
```

Figure 36: The average number of requests with standard deviation for the high traffic verification.

8.2.5 Verification with other deployments

In this experiment we will look at the behavior of the simulation with a different deployment. In the previous experiments we looked at the vehiclemanagementapi. To see if the simulation also works well with other deployments the customermanagementapi is used for this experiment. The load on the service mesh is 5 requests per second, which translates to an actual load of 4 requests per second as seen in figure 38. We also see the same discrepancy with the amount of requests that are sent to the logserver. On 50% of the requests to the customermanagementapi a request to the logserver is sent which results in an average of about 2 requests per second that are sent to the logserver.

The traffic graph in figure 37 does have a strange spike between seconds 33 and 53. Similarly to last time this was caused by the random function acting strangely. In the first 30 seconds of the simulation only a few requests were sent to the logserver, but in the last 30 seconds a lot of request were sent to the logserver. If random function would have behaved as expected, then the spike would be smoothed over the entire graph of the simulation, and the average would correspond to the expected 38 requests per 5 seconds (with a discrepancy of 5 seconds multiplied by 4 request).

	customer	logserver	webapp	Total
customer	249,9+-1,14 (0)	29,6+-0,49 (86)	0+-0 (0)	279,5+-1,02 (86)
logserver	0+-0 (0)	0+-0 (0)	0+-0 (0)	0+-0 (0)
webapp	249,6+-1,43 (240)	29,5+-0,5 (21)	0+-0 (0)	279,1+-1,3 (261)

Table 10: All requests for the alternative deployment verification.

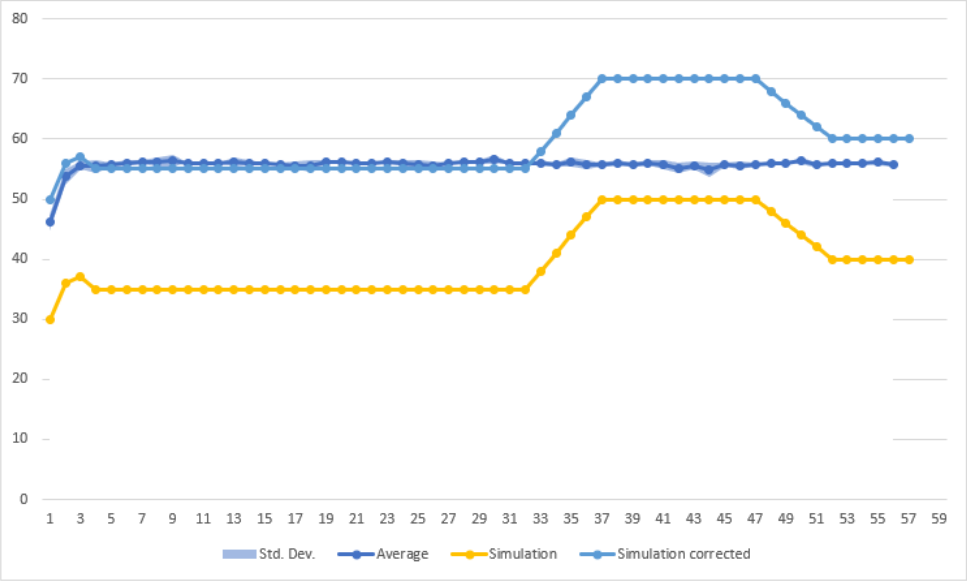

Figure 37: Requests per second for the alternative deployment verification.

```
webapp:
    - customermanagementapi: 4,16 +- 0,02 (4)
    - logserver: 0,49 +- 0,01 (0,35)
customermanagementapi:
    - customermanagementapi: 4,17 +- 0,02 (0)
    - logserver: 0,49 +- 0,01 (1,43)
```

Figure 38: The average number of requests with standard deviation for the alternative deployment verification.

8.2.6 Verification with multiple replicas

In this experiment we look into how the simulation handles replicas. One of the big advantages of a service mesh is the possibility to scale a deployment to multiple replicas, adding resiliency and scalability. For this verification the vehiclemanagementapi was used with 2 replicas. The load on the service mesh was 5 requests per second which translates to the same actual 4 requests per second as in the last experiments. When we take a look at figure 40 we see that the simulation handles multiple replicas nicely. The traffic that the webapp sends is spread evenly across the two replicas of the vehiclemanagementapi, just like for the real service mesh. We also see that similar to the previous experiments each of the vehiclemanagementapis sends one request per second

to the logserver (each of the vehiclemanagementapis receives 2 requests, both with a 50% chance to send a request to the logserver thus averaging to 1 request per second).

When we take look at figure 39 we see that just like in the previous experiments the random function seems to favor sending requests to the logserver in the last 30 seconds of the simulation. The average of the simulation is 40 requests per 5 seconds, which is what we would expect (60 requests per 5 seconds minus the 5 seconds multiplied by the four requests per second).

	logserver	vehicle0	vehicle1	webapp	Total
logserver	0+-0 (0)	0+-0 (0)	0+-0 (0)	0+-0 (0)	0+-0 (0)
vehicle0	29,6+-8,88 (59)	0+-0 (0)	0+-0 (0)	0+-0 (0)	0+-0 (0)
vehicle1	29,9+-0,3 (59)	0+-0 (0)	0+-0 (0)	0+-0 (0)	0+-0 (0)
webapp	29,7+-0,4 (36)	126,1+-37,88 (120)	125,5+-37,71 (120)	0+-0 (0)	281,3+-84,52 (276)

Table 11: All requests for the verification with multiple replicas of a deployment.

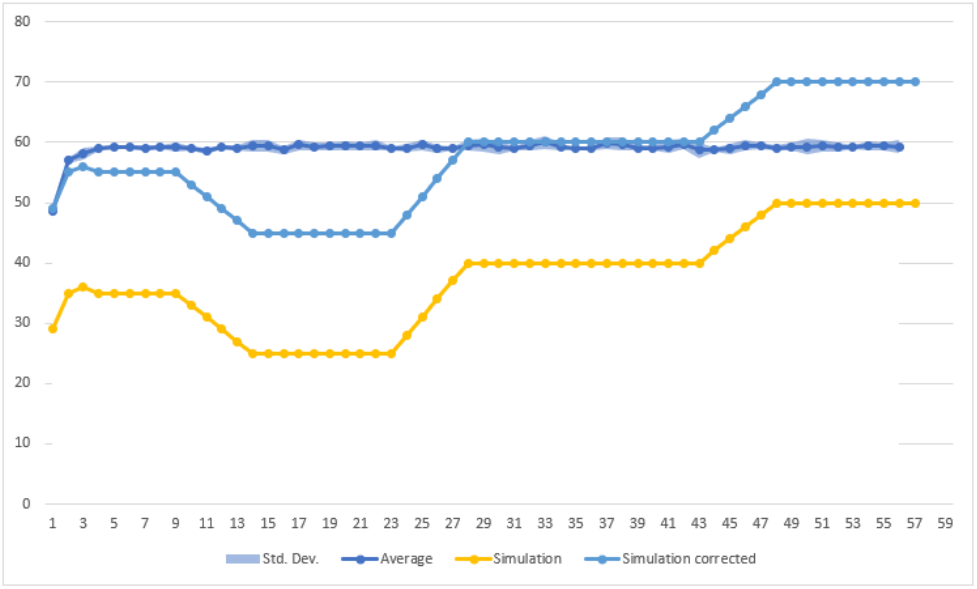

Figure 39: Requests per second for the verification with multiple replicas of a deployment.

```
webapp:
    - vehiclemanagementapi0: 2,1 +- 0,63 (2)
    - vehiclemanagementapi1: 2,09 +- 0,63 (2)
    - logserver: 0,49 +- 0,01 (0,6)
vehiclemanagementapi0:
    - customermanagementapi: 2,1 +- 0,02 (0)
    - logserver: 0,5 +- 0,01 (0,98)
vehiclemanagementapi0:
    - customermanagementapi: 4,17 +- 0,02 (0)
    - logserver: 0,5 +- 0,01 (0,98)
```

Figure 40: The average number of requests with standard deviation for the verification with multiple replicas of a deployment.

8.2.7 Visual verification

The simulation does not just offer the log output, it also has graphical output. In this visual output the developer sees how the pods with the deployments and services are connected to each other. The real service mesh has a similar feature that shows how the pods are connected (and how traffic is sent) when the service mesh is running. We use the real visualisation of the real service mesh to determine whether the visualisation of the simulation is correct.

In figure figure 41 and figure 42 we see a simple network. The "unknown" (figure 41) and "gateway" (figure 42) are the sources of traffic. We see that the connections between the nodes of the real service mesh and the simulation are identical.

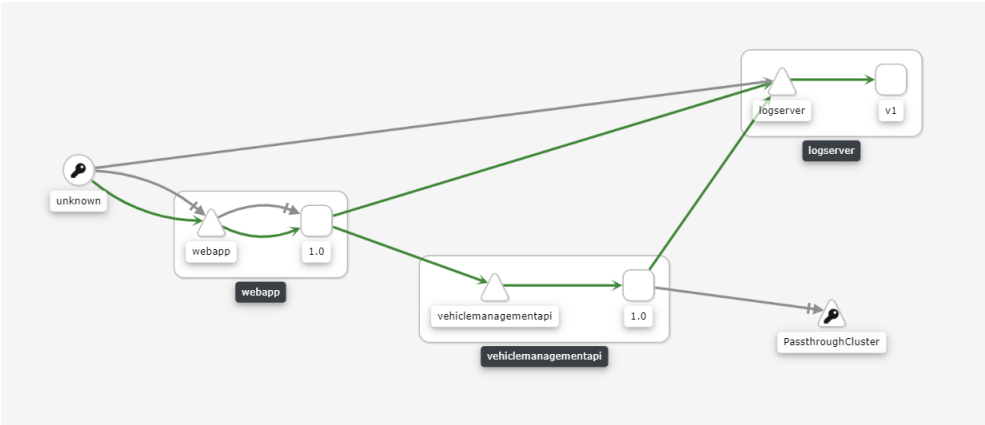

Figure 41: The visual representation of a vehiclemanagementapi with the accompanying deployments in Kiali.

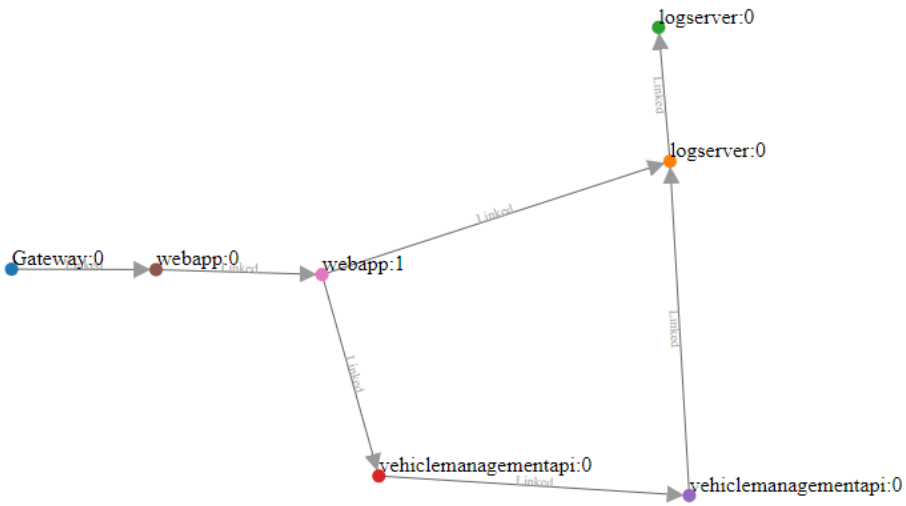

Figure 42: The visual representation of a vehiclemanagementapi with the accompanying deployments in the simulation.

Just like with the other experiments we also use another type of deployment here. You can see this in figure 43 and figure 44. Instead of the vehiclemanagementapi we use the customermanagementapi, and again we see that the links between the nodes are identical.

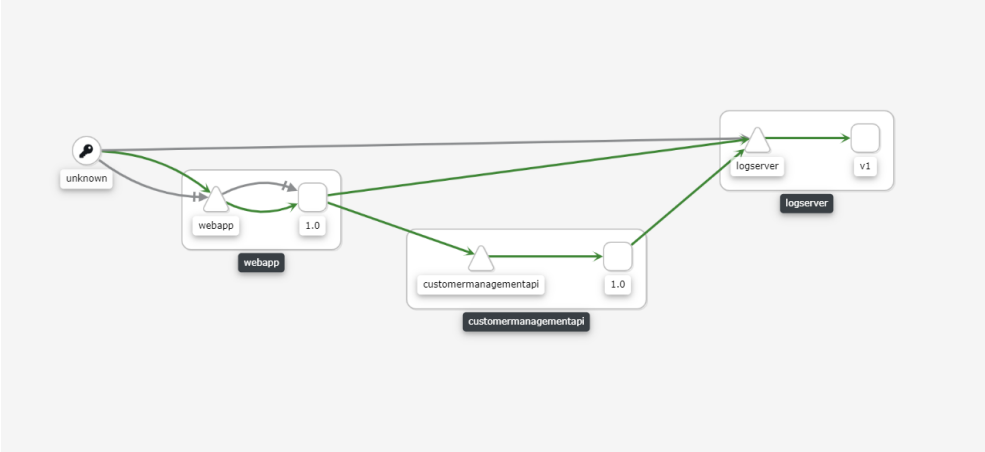

Figure 43: The visual representation of a customermanagementapi with the accompanying deployments in Kiali.

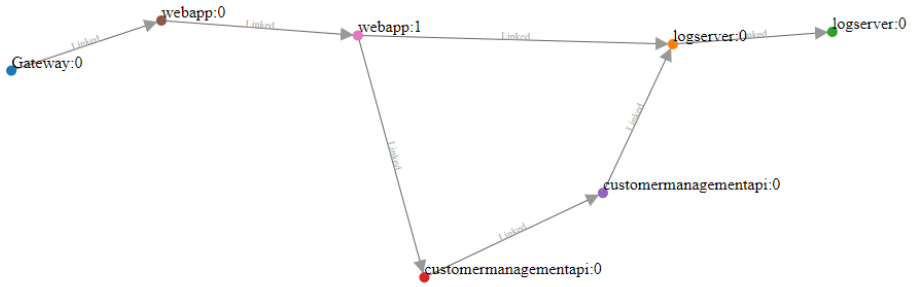

Figure 44: The visual representation of a customermanagementapi with the accompanying deployments in the simulation.

We also take a look at some more complex service meshes in figure 45 and figure 46. Here we use a network that has both the vehiclemanagementapi and the customermanagementapi. Just like the previous two sets of images we see that the links in the simulation are identical. In figure 45 we do see an extra "PassthroughCluster" which was used by the vehiclemanagementapi. A passtrough cluster is added when requests are sent to an undefined service. This can be caused by a missing Istio sidecar. This part was not simulated in the simulation and is thus not visible in the visualisation.

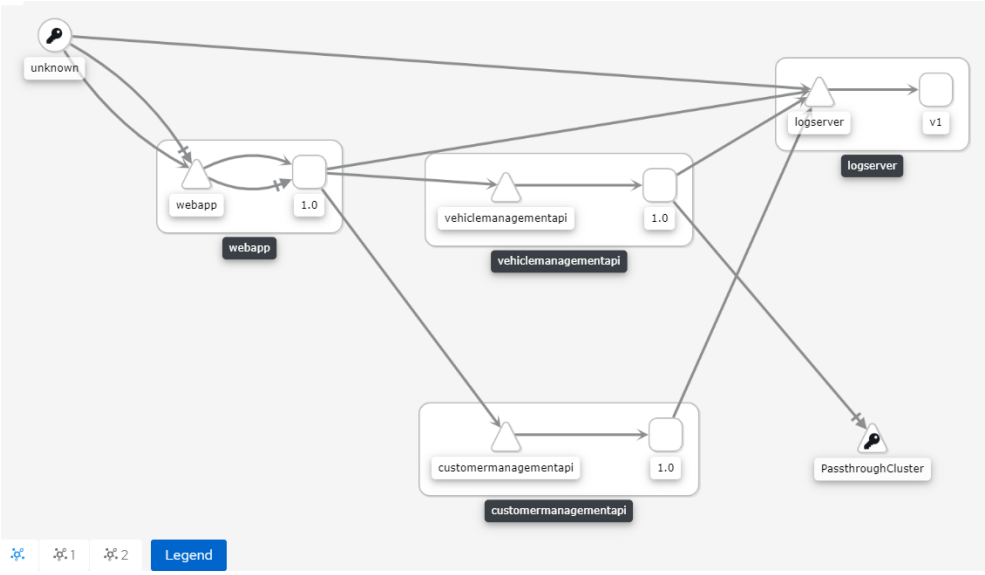

Figure 45: The visual representation of a vehiclemanagementapi and a customermanagementapi with the accompanying deployments in Kiali.

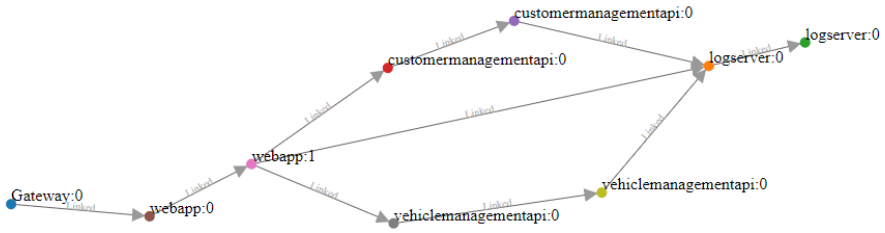

Figure 46: The visual representation of a vehiclemanagementapi and a customermanagementapi with the accompanying deployments in the simulation.

There is an interesting difference between how the amount of replicas is displayed, which is seen in figure 47 and figure 48. In the real service mesh visualisation the developer has to open the pod with the vehiclemanagementapi to see how many replicas there are. In figure 47 you see that on the right there is a small "Pod status" with underneath it "vehiclemanagementapi 2/2". This tells us that the vehiclemanagementapi has two replicas. In our own visualisation you see that the vehiclemanagementapi service is connected to two deployments of the vehiclemanagementapi. The developer does not have to click and search for the information but it is immediately present.

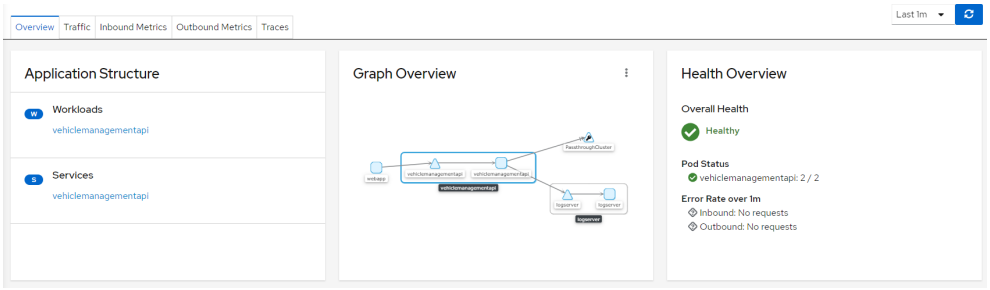

Figure 47: The visual representation of a vehiclemanagementapi with two replicas in Kiali.

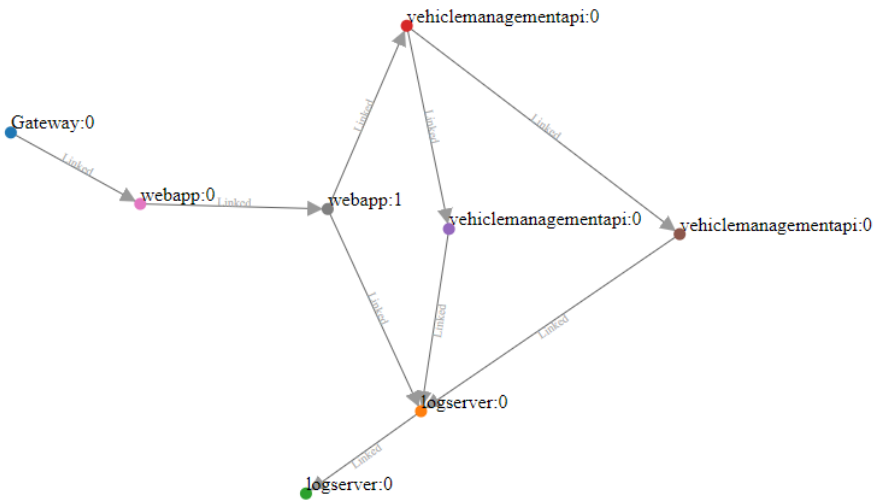

Figure 48: The visual representation of a vehiclemanagementapi with two replicas and the accompanying deployments in the simulation.

8.2.8 Verification of circuit breaking

The final aspect of verification is the circuit breaker. The circuit breaker allows the developer to specify limits for deployments to prevent them from being overloaded. Due to difficulties with setting-up the circuit breaker in the Pitstop project we opted to used a separate small application to test how the simulation handles a circuit breaker. In table 12 the amount of requests that were either accepted or rejected is shown. The circuit breaker was configured to allow only one connection to the deployment, all other connections will be blocked to prevent the deployment from being overloaded. This type of circuit breaker is applied to components that execute heavy operations and can't handle a lot of concurrent requests. After starting the service mesh Fortio [27] was used to send four concurrent request to the service mesh, until 200 request in total are sent.

We see that the real service mesh allows more requests to come through, whereas the simulation allows exactly 25%. This is explained by the nature of the simulation. In the real service mesh requests can come in at any time, but in the simulation all the request are executed in steps. In the real service mesh the request sometimes come in slightly after each other which causes them to be executed after each other instead of concurrently. In the simulation they are all executed at exactly the same time, which causes 75% of the requests to be rejected.

	Real		Simulation	
	200 OK	503 Error	200 OK	503 Error
	71	129	50	150
	74	126	50	150
	74	126	50	150
	77	123	50	150
	71	129	50	150
	71	129	50	150
	78	122	50	150
	77	123	50	150
	74	126	50	150
	71	129	50	150
Average	73,8	126,2	50	150
Std. Dev.	2,638181	2,638181	0	0

Table 12: Overview of requests that are accepted or blocked by the circuit breaker in a real service mesh and the simulation.

9 Predictive Quality of the Simulation

The simulation tool was not just developed to make simulations of already existing service meshes, but also to do predictions of how a service mesh will behave after a change. We have performed two experiments both of which are done by executing the following steps:

1. Start the real service mesh with the deployments, services, etc. that are used for the basis of the simulation.
2. Record the data for this basis and use this for the simulation.
3. Add the modification to the simulation that we want to predict (e.g. increase the amount of traffic).
4. Run the modified simulation and save the data.
5. Run the real service mesh with the same modification.
6. Compare the output of the modified simulation to the output of the real service mesh that has been modified and see if they equal.

9.1 Increasing traffic

For the first experiment we based the simulation on a simple service mesh where 1 request per second is sent to the vehiclemanagementapi. The traffic was modified to 500% resulting in 5 requests per second. We compare the modified version of the simulation to the real service mesh with 5 requests per second. Unlike the verification we see that in traffic graph figure 49 the requests per 5 seconds are higher than the real service mesh. We can see what causes this in figure 50. The simulation scales the requests from the webapp to exactly 500%, even though in the real service mesh we only see a 400% increase due to the timing the script that puts load on the service mesh. We also see that the traffic from the webapp to the logserver is also scaled by 500% whereas this does not happen in the real service mesh.

	logserver	vehicle	webapp	Total
logserver	0+-0 (0)	0+-0 (0)	0+-0 (0)	0+-0 (0)
vehicle	29,9+-0,3 (183)	250,5+-0,67 (0)	0+-0 (0)	280,4+-0,8 (183)
webapp	29,8+-0,4 (186)	250,4+-0,66 (300)	0+-0 (0)	280,2+-0,87 (486)

Table 13: All requests for the simulation where the traffic is increased.

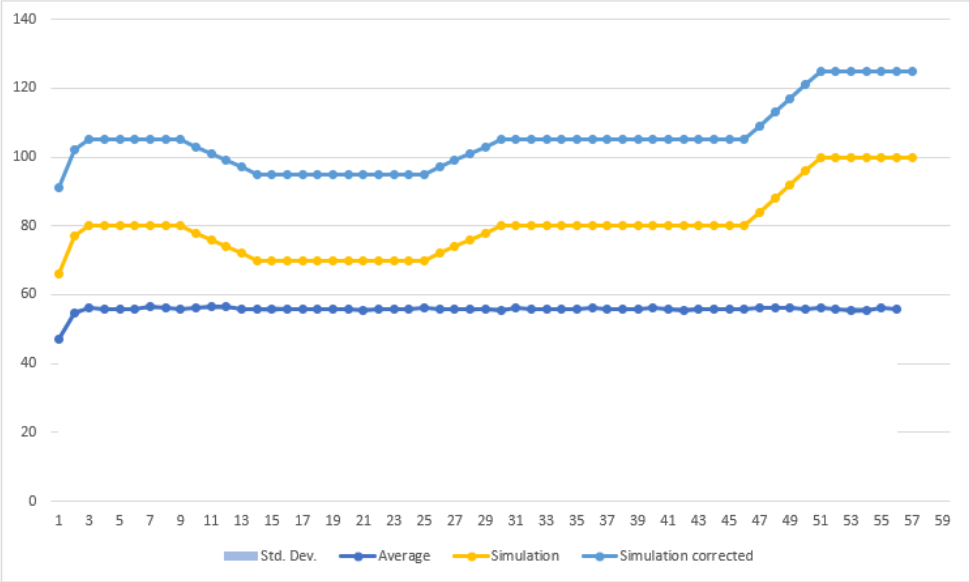

Figure 49: Requests per second for the simulation where the traffic is increased.

```
webapp:
    - vehiclemanagementapi: 4,17 +- 0,01 (5)
    - logserver: 0,5 +- 0,01 (3,1)
vehiclemanagementapi:
    - vehiclemanagementapi: 4,18 +- 0,01 (0)
    - logserver: 0,5 +- 0,01 (3,05)
```

Figure 50: The average number of requests with standard deviation for the simulation where the traffic is increased.

9.2 Adding endpoints

For the second experiment we base the simulation on the same simple service mesh with just the vehiclemanagementapi, but this time 5 requests per second are send. The simulation was modified to also have the customermanagementapi. We compare the modified version of the simulation to the real service mesh with both the vehiclemanagementapi and the customermanagementapi. In figure 52 we see that the simulation is nearly identical to the real service mesh. It has the same discrepancy that we saw in the experiment where the vehiclemanagementapi and customermanagementapi send 50% of the requests they get to the logserver. Furthermore we see in figure 51 that the simulation

on average has the expected discrepancy of 40 (the 5 second window multiplied by the 4 requests per second multiplied by the two deployments).

	customer	logserver	vehicle	webapp	Total
customer	248,5+-0,92 (0)	29,8+-0,4 (121)	0+-0 (0)	0+-0 (0)	278,3+-1 (121)
logserver	0+-0 (0)	0+-0 (0)	0+-0 (0)	0+-0 (0)	0+-0 (0)
vehicle	0+-0 (0)	29,7+-0,46 (121)	248,6+-1,02 (0)	0+-0 (0)	278,3+-1,27 (121)
webapp	248,5+-0,92 (240)	29,7+-0,46 (32)	248,1+-1,22 (240)	0+-0 (0)	526,3+-2,24 (512)

Table 14: All requests for the simulation where the customermanagementapi is added.

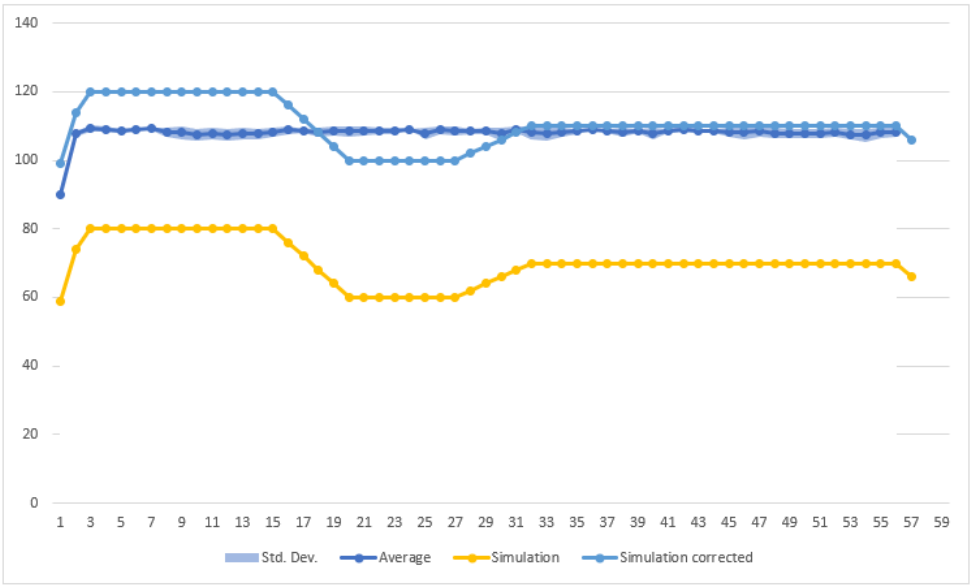

Figure 51: Requests per second for the simulation where the customermanagementapi is added.

```
webapp:
    - vehiclemanagementapi: 4,14 +- 0,02 (4)
    - customermanagementapi: 4,14 +- 0,02 (4)
    - logserver: 0,5 +- 0,01 (0,5)
customermanagementapi:
    - vehiclemanagementapi: 0 +- 0 (0)
    - customermanagementapi: 4,14 +- 0,02 (0)
    - logserver: 0,5 +- 0,01 (2)
vehiclemanagementapi:
    - vehiclemanagementapi: 4,14 +- 0,02 (0)
    - customermanagementapi: 0 +- 0 (0)
    - logserver: 0,5 +- 0,01 (2)
```

Figure 52: The average number of requests with standard deviation for the simulation where the customermanagementapi is added.

10 Conclusion

In this thesis we have looked into the workings of service meshes. How do they work, how much traffic do they send, and whether we are able to simulate this behavior or even make predictions about the effect that changes will have on the service mesh. We go over the research questions and the answers to them, discuss the validity of the work and lastly look into work that could be done to improve upon the work.

10.1 Answers to Research Questions

Research Question 1: Is it possible to construct an accurate simulation of a service mesh with just the static configuration files for that network?

Research Question 1.1: Which information is required to create an accurate simulation of a service mesh?

Research Question 1.2: Which information is missing from the configuration files for the service mesh?

To answer these questions we looked into the workings of a service mesh and created a model with all the information that we require to make a simulation of the service mesh. We determined that the static YAML files do not contain enough information, as there is no data about the traffic within the service mesh. The relation of services within the service mesh (whether they have a connection) also can not be derived from the YAML files. We built log analysis tools that extract this data from the service mesh logging and added this data to the model. We found that with this added information, we can create accurate simulations.

Research Question 2: Can we increase the performance of a simulation by adding dynamic information, such as log data?

As shown in section 6, we were able to extract the traffic information and relation between services from the log entries of the service mesh. With this information we have enhanced the model which allowed us to create a simulation of the service mesh.

Research Question 3: What is the accuracy of the simulation of the real service mesh?

As shown in sections 8 and 9 we can create simulations with varying levels of accuracy. Predictable traffic is simulated accurately, but the current model does not allow for enough parameters to do fully accurate simulations of a service

mesh. When we looked at using the simulation to predict changes in the service mesh we found similar results, when traffic is predictable we can make accurate predictions about the working of the service mesh.

10.2 Discussion

In this section we discuss the quality of the work and possible threats to validity.

10.2.1 Randomness

As we have seen in sections 8 and 9 the randomness of the simulation did not always seem random enough. There were a lot of simulations where trafficSets have been chosen randomly but did not seem to be random enough. This can be solved by running the simulation more often and taking the average of the result. This will smooth out spikes within the individual runs.

10.2.2 Circuit breaker

The circuit breaker was not used within a larger service mesh but was instead tested within an isolated component. It is better to incorporate the circuit breaker in an actual working application and test how the circuit breaker functions within such an application.

10.2.3 Pitstop

For the simulation of the service mesh we have only used the Pitstop [40] demo application. Although the implementation of the log analysis tooling and simulation are general enough to also work for other applications it is better to also create simulations of other service meshes.

10.3 Future work

The current model and tooling can be improved in the following ways:

10.3.1 Half requests

Currently the log analysis tool is not able to create traffic sets with less than one request per second. The tool should be adapted to generate multiple traffic sets with probabilities in such a way that one requests per multiple seconds is sent. This would increase the accuracy of the simulation and predictions.

10.3.2 Requests between the service and deployment

The line with the corrected values is added to each of the graphs. In a future version of the simulation tool it is better to add the additional request from the source to the service to the logging of the service mesh. This prevents the manual correction that has to be done after the collection of the logs.

10.3.3 Time based requests

Currently the tool can only send requests based on incoming requests. It would be better to also add support for time based requests (like we have seen for requests that are sent to the logserver). It will be challenging to find these time based requests automatically, but if they are discovered manually it would be nice if a developer could manually add them to the simulation. This would further increase the accuracy of the simulation and predictions.

10.3.4 Missing requests

The tool only looks at the logs that are collected from the real service mesh and extracts the traffic from them. If there are services within the network that did not sent any traffic in the period that the logging was collected they will not be noticed. In future versions of the simulation tool the developer has to be informed if there are services that do not send traffic.

11 Bibliography

References

[1] S. Even and R. E. Tarjan, "Network flow and testing graph connectivity," *SIAM*, 1974.

[2] M. Felleisen, "On the expressive power of programming languages," in *ESOP '90*, N. Jones, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 134–151, ISBN: 978-3-540-47045-8.

[3] T. Richner and S. Ducasse, "Recovering high-level views of object-oriented applications from static and dynamic information," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, Aug. 1999, pp. 13–22. DOI: 10.1109/ICSM.1999.792487.

[4] G. G. Xie, Jibin Zhan, D. A. Maltz, Hui Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, "On static reachability analysis of ip networks," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 3, Mar. 2005, 2170–2183 vol. 3. DOI: 10.1109/INFCOM.2005.1498492.

[5] M. B. Cohen, J. Snyder, and G. Rothermel, "Testing across configurations: Implications for combinatorial testing," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 6, pp. 1–9, Nov. 2006, ISSN: 0163-5948. DOI: 10.1145/1218776.1218785. [Online]. Available: https://doi.org/10.1145/1218776.1218785.

[6] N. Henry, J.-D. Fekete, and M. J. McGuffin, "Nodetrix: A hybrid visualization of social networks," *IEEE transactions on visualization and computer graphics*, vol. 13, no. 6, pp. 1302–1309, 2007.

[7] W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li, "Geometry-based edge clustering for graph visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1277–1284, Nov. 2008, ISSN: 2160-9306. DOI: 10.1109/TVCG.2008.135.

[8] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow, "Comparing design and code metrics for software quality prediction," in *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '08, Leipzig, Germany: Association for Computing Machinery, 2008, pp. 11–18, ISBN: 9781605580364. DOI: 10.1145/1370788.1370793. [Online]. Available: https://doi.org/10.1145/1370788.1370793.

[9] M. F. S. Oliveira, R. M. Redin, L. Carro, L. d. C. Lamb, and F. R. Wagner, "Software quality metrics and their impact on embedded software," in *2008 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, Apr. 2008, pp. 68–77. DOI: 10.1109/MOMPES.2008.11.

[10] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro, "Dynamic error handling in service oriented applications," *Fundamenta Informaticae*, vol. 95, no. 1, pp. 73–102, 2009.

[11] S. S. R. Ahamed, "Studying the feasibility and importance of software testing: An analysis," *CoRR*, vol. abs/1001.4193, 2010. arXiv: 1001.4193. [Online]. Available: http://arxiv.org/abs/1001.4193.

[12] D. Jin, X. Qu, M. B. Cohen, and B. Robinson, "Configurations everywhere: Implications for testing and debugging in practice," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 215–224, ISBN: 9781450327688. DOI: 10.1145/2591062.2591191. [Online]. Available: https://doi.org/10.1145/2591062.2591191.

[13] A. Krylovskiy, M. Jahn, and E. Patti, "Designing a smart city internet of things platform with microservice architecture," in *2015 3rd International Conference on Future Internet of Things and Cloud*, Aug. 2015, pp. 25–30. DOI: 10.1109/FiCloud.2015.55.

[14] M. Ahmadvand and A. Ibrahim, "Requirements reconciliation for scalable and secure microservice (de)composition," Sep. 2016, pp. 68–73. DOI: 10.1109/REW.2016.026.

[15] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Queue*, vol. 14, no. 1, pp. 70–93, 2016.

[16] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, Apr. 2017, pp. 243–246. DOI: 10.1109/ICSAW.2017.11.

[17] S. Esparrachiaria, T. Reilly, and A. Rentz, "Tracking and controlling microservice dependencies," *ACM*, 2018.

[18] *An update on sunday's service disruption.* [Online]. Available: http://web.archive.org/web/20200207125311/https://cloud.google.com/blog/topics/inside-google-cloud/an-update-on-sundays-service-disruption.

[19] *Canary release.* [Online]. Available: http://web.archive.org/web/20200223185125/https://martinfowler.com/bliki/CanaryRelease.html.

[20] *Computer scaling.* [Online]. Available: https://web.archive.org/save/https://www.esds.co.in/blog/vertical-scaling-horizontal-scaling/.

[21] *Design patterns for microservices.* [Online]. Available: http://web.archive.org/web/20200218194317/https://dzone.com/articles/design-patterns-for-microservices.

[22] *Designing interservice communication for microservices*. [Online]. Available: http://web.archive.org/web/20200218135118/https://docs.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication.

[23] *Distributed tracing faq*. [Online]. Available: http://web.archive.org/web/20200303132748/https://istio.io/faq/distributed-tracing/.

[24] *Docker*. [Online]. Available: http://web.archive.org/web/20200226102603/https://www.docker.com/.

[25] *Docker swarm*. [Online]. Available: http://web.archive.org/web/20200221094811/https://docs.docker.com/engine/swarm/.

[26] *Envoy docs - tracing*. [Online]. Available: http://web.archive.org/web/20200221133825/https://www.envoyproxy.io/docs/envoy/v1.9.0/intro/arch_overview/tracing.

[27] *Fortio*. [Online]. Available: https://fortio.org/.

[28] *Horizontal pod autoscaler*. [Online]. Available: http://web.archive.org/web/20200305195919/https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[29] *How is docker different from a virtual machine?* [Online]. Available: http://web.archive.org/web/20200325213800/https://stackoverflow.com/questions/16047306/how-is-docker-different-from-a-virtual-machine.

[30] *Infosupport*. [Online]. Available: http://web.archive.org/web/20200225120905/https://www.infosupport.com/.

[31] *Installing istio 1.4*. [Online]. Available: http://web.archive.org/web/20200224064434/https://haralduebele.blog/2019/11/21/installing-istio-1-4-new-version-new-methods/.

[32] *Is polyglot programming practical?* [Online]. Available: http://web.archive.org/web/20200217205113/https://dzone.com/articles/polyglot-programmers.

[33] *It's time to move to a four-tier application architecture*. [Online]. Available: http://web.archive.org/web/20200207103900/https://www.nginx.com/blog/time-to-move-to-a-four-tier-application-architecture/.

[34] *Kiali*. [Online]. Available: http://web.archive.org/web/20200325231537/https://kiali.io/.

[35] *Kubernetes*. [Online]. Available: http://web.archive.org/web/20200221094757/https://kubernetes.io/.

[36] *Kubernetes networking guide for beginners*. [Online]. Available: http://web.archive.org/web/20200221102552/https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-networking-guide-beginners.html.

[37] *Kubernetes tutorials - viewing pods and nodes.* [Online]. Available: http://web.archive.org/web/20200221091728/https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/.

[38] *Microservice communication patterns.* [Online]. Available: http://web.archive.org/web/20200207204752/https://reflectoring.io/microservice-communication-patterns/.

[39] *Microservice premium.* [Online]. Available: http://web.archive.org/web/20200227093155/https://martinfowler.com/bliki/MicroservicePremium.html.

[40] *Pitstop - garage management system.* [Online]. Available: https://github.com/EdwinVW/pitstop.

[41] *Solution architecture.* [Online]. Available: http://web.archive.org/web/20200225130507/https://github.com/EdwinVW/pitstop/wiki/Solution%20Architecture.

[42] *Viewing pods and nodes.* [Online]. Available: http://web.archive.org/web/20200221103056/https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/.

[43] *Whats is a service mesh?* [Online]. Available: http://web.archive.org/web/20200221102407/https://www.nginx.com/blog/what-is-a-service-mesh/.