P1 Project Report

In the course of this project I explored the monitoring capabilities of a service mesh, namely Istio, in the context of a microservice deployed on Kubernetes.

Minikube setup

To perform the experiments I decided to use Minikube. This technology allows to simulate a single node Kubernetes cluster within a virtual machine on a single computer instance, but should not be used in a production environment but mainly for testing and development purposes. Similar to conventional virtual machines, Minikube also requires a hypervisor for virtualization. Here you can choose between different driver types, KVM and VirtualBox on an Ubuntu Linux host system. Since the host system itself runs in a virtual machine, it was not possible to use the KVM virtualization driver and therefore had to use the VirtualBox driver, which offers somewhat worse performance since this is a level 2 hypervisor in contrast to KVM.

Istio installation

The installation of Istio as well as of the additionally required tools such as Prometheus, Grafana and Kiali went smoothly as the combination of these technologies is a common scenario and configuration files are already delivered with Istio itself.

Google microservice project

To test the monitoring capabilities it was necessary to use an existing microservice application. The first option I knew of was the microservice application provided by Google. However, it turned out that this application does not work stable. I suspect that this is due to the fact that the application has not been maintained for some time at the time of.
https://github.com/GoogleCloudPlatform/microservices-demo

Teastore reference application

Another possibility for this deployment scenario is the use of Teastore reference application. This was specifically developed to be used as a case study in scientific projects and provides versatile deployment scenarios.
https://github.com/DescartesResearch/TeaStore

Load generator

Since it is necessary to test the functionalities of Istio in the context of this project, it was necessary to develop an own load generator. I decided to use the locust test framework to simulate internet traffic.

Istio Bookinfo reference application

In the end I decided to use the Bookinfo application provided by the Istio developers for the following reasons. On the one hand, the resource consumption is lower than in the microservice architectures evaluated so far, on the other hand, this application has been developed to use features of Istio itself in the best possible way, which additionally suits my exploratory project.

Load generator

This application does not have an integrated load generator either, so I developed my own. The Locust framework that I used here simulates a user behavior. For this a certain number of requests per user is sent to the available endpoints.

Benchmarks

Finally, once I decided on an application, it was time to run the benchmarks. Originally, I ran the load generator and the Minikube cluster inside the same virtual machine. However, since the load generator requires a considerable amount of CPU time, the test itself also affects the results of the benchmark. Therefore, it was necessary to use an application in advance that basically requires only few resources.

Eventually, however, I was able to run these tests from another virtual machine to get more accurate results. However, it is very important to note that I am still using Minikube here, which was not originally intended to provide services other than for testing purposes. Therefore there is no integrated function to open Minikube into the local network.
Here it came in favor that I installed VirtualBox as hypervisor for Minikube. This gave me the opportunity to set up port forwarding on the network adapters of the Minikube VM to provide the services within the network.

I then tested what influence certain features of Istio can have on the performance.
I have decided on 3 scenarios. Generally, I disabled individual features and then compared them against the version with all features.

As a baseline for comparison I used a deployment with load balancing, virtual services, an ingress gateway and all other features that are enabled by default by Istio.
After that I wanted to find out how the influence of the ingress gateway provided by Istio affects the performance by forwarding requests directly to the service and thus bypassing the ingress gateway.
As a third and final scenario, I disabled virtual services in addition to the ingress gateway. Virtual services are another way to create load balancing between the individual microservices within Istio.

Findings

Then I ran the benchmarks with different load scenarios. The first benchmarks showed unreliable results because locust was running on the same machine as the Minikube cluster.

When I finally ran the benchmarks in the optimized setup with multiple virtual machines, it was apparent that the results with the now comparatively low load showed no significant difference regardless of which features of Istio were enabled.

When I increased the load by a factor of 7.5 for another scenario and repeated the benchmarks with the same settings, interestingly, there was only a slight difference between the results. To make sure that I did not make a mistake, I ran the benchmark with high load against the same system, but deactivated Istio. Here the number of executed requests was even the highest. Therefore, I can conclude that the load I chose for my benchmarks was not sufficient to push the tested system to its limits.

I then increased the load further as a probationary measure, which did not result in higher request rates, but only in a higher percentage of failed requests.
This is probably due to the fact that load generation with locust is very computationally intensive and apparently more time-consuming than responding to the requests.
However, it cannot be ruled out that the current setup of the test bed is not suitable for such scenarios.
These points certainly require further research.

A detailed analysis of the data can be found at: https://github.com/whiskeywolke/P1-SS22/blob/main/bookinfo/analysis/bookinfo_analysis.ipynb

How-to

This section lists the commands I used to install and setup Istio, as well as configuring the benchmarks for the chosen Microservice application for reproducibility
An extended version of this list of commands can be found at https://github.com/whiskeywolke/P1-SS22/blob/main/commands2.txt

Setup Minikube & install Istio inside cluster

```
minikube start --cpus 14 --memory 28000 --driver=virtualbox
istioctl install
kubectl label namespace default istio-injection=enabled
```

Installing Istio addons for cluster management

```
kubectl apply -f samples/addons/grafana.yaml
kubectl apply -f samples/addons/prometheus.yaml
kubectl apply -f samples/addons/kiali.yaml

kubectl get all -n istio-system
```

Accessing dashboards on local machine

This is how I configured port forwarding via SSH to my local machine to be able to access the dashboards.

```
ssh -L 20001:localhost:20001 michaelmente@swa-michaelmente.cs.univie.ac.at
kubectl port-forward svc/kiali -n istio-system 20001

ssh -L 3000:localhost:3000 michaelmente@swa-michaelmente.cs.univie.ac.at
kubectl port-forward svc/grafana -n istio-system 3000

ssh -L 9090:localhost:9090 michaelmente@swa-michaelmente.cs.univie.ac.at
kubectl port-forward svc/prometheus -n istio-system 9090
```

Installing Bookinfo Istio reference application

```
kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml
```

Verification of successful deployment:

```
kubectl exec "$(kubectl get pod -l app=ratings -o jsonpath='{.items[0].metadata.name}')" -c
ratings -- curl -sS productpage:9080/productpage | grep -o "<title>.*</title>"
```

Configuring Minikube for access from Network

Publishing port for access via Istio ingress gateway:

This setup is used to access the Minikube cluster from the local network and to access the service by the provided via the Istio ingress gateway.

```
export INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o
jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')
vboxmanage controlvm "minikube" natpf1 "ingress,tcp,,$INGRESS_PORT,,$INGRESS_PORT"
```

Verify if setup is working:

```
curl -s "http://swa-michaelmente.cs.univie.ac.at:$INGRESS_PORT/productpage" | grep -o "
<title>.*</title>"
```

Publishing port for direct service access

This type of setup is used to access the the service directly to bypass the ingress gateway. This is how I configured the system for one of the benchmarks.

```
kubectl expose deployment productpage-v1 --type=NodePort --name=ingress-bypass
kubectl get svc ingress-bypass
export INGRESS_BYPASS=$(kubectl get service ingress-bypass -o
jsonpath='{.spec.ports[0].nodePort}')
vboxmanage controlvm "minikube" natpf1 "ingress-
bypass,tcp,,$INGRESS_BYPASS,,$INGRESS_BYPASS"
```

Verify if setup is working:

```
curl -s "http://swa-michaelmente.cs.univie.ac.at:$INGRESS_BYPASS/productpage" | grep -o "
<title>.*</title>"
```

For the last type of benchmark I disabled the virtual-services which are provided in the same file as the gateway. It does not matter that I do not deactivate the ingress gateway for the previous benchmark as the gateway gets bypassed anyway

```
kubectl delete -f bookinfo-gateway.yaml
```

Benchmarking the system

These are the benchmark runs I used, each of them is limited to the same time, I chose 600 seconds.

Low load scenario

I used the this load scenario with 400 users simultaneously interacting with the system where the user count increases by 80 users every second.

```
locust --host=http://swa-michaelmente.cs.univie.ac.at:32460 -u 400 --spawn-rate=80 --
headless --run-time=600s --csv benchmarks/baseline/$(date "+%d-%m-%y_%H-%M-%S") --csv-full-
history -f load-generator/locustfile.py

locust --host=http://swa-michaelmente.cs.univie.ac.at:32726 -u 400 --spawn-rate=80 --
```

```
headless --run-time=600s --csv benchmarks/noIngressGateway/$(date "+%d-%m-%y_%H-%M-%S") --
csv-full-history -f load-generator/locustfile.py

locust --host=http://swa-michaelmente.cs.univie.ac.at:32726 -u 400 --spawn-rate=80 --
headless --run-time=600s --csv benchmarks/noLoadBalancing/$(date "+%d-%m-%y_%H-%M-%S") --
csv-full-history -f load-generator/locustfile.py
```

High load scenario

As a high load scenario I chose to have 2000 users accessing the system concurrently, where the spawn rate is 100 users per second.
Depending on the machine the limit of open files needs to be increased beforehand which can be achieved with

```
ulimit -Sn 1048576
```

```
locust --host=http://swa-michaelmente.cs.univie.ac.at:32460 -u 1500 --spawn-rate=100 --
headless --run-time=600s --csv benchmarks/baselineHigh/$(date "+%d-%m-%y_%H-%M-%S") --csv-
full-history -f load-generator/locustfile.py

locust --host=http://swa-michaelmente.cs.univie.ac.at:32726 -u 1500 --spawn-rate=100 --
headless --run-time=600s --csv benchmarks/noIngressGatewayHigh/$(date "+%d-%m-%y_%H-%M-%S")
--csv-full-history -f load-generator/locustfile.py

locust --host=http://swa-michaelmente.cs.univie.ac.at:32726 -u 1500 --spawn-rate=100 --
headless --run-time=600s --csv benchmarks/noLoadBalancingHigh/$(date "+%d-%m-%y_%H-%M-%S") -
-csv-full-history -f load-generator/locustfile.py
```

Benchmark used for verification of setup with Istio completely disabled

```
locust --host=http://swa-michaelmente.cs.univie.ac.at:32726 -u 1500 --spawn-rate=100 --
headless --run-time=600s --csv benchmarks/noIstio/$(date "+%d-%m-%y_%H-%M-%S") --csv-full-
history -f load-generator/locustfile.py
```