Orchestration Service-Mesh Testbed

Lars Koenen
Institut für Nachrichtentechnik
Technische Hochschule Köln
Köln, Deutschland
lars.koenen@smail.th-koeln.de

Lars Borggrewe
Institut für Nachrichtentechnik
Technische Hochschule Köln
Köln, Deutschland
lars.borggrewe@smail.th-koeln.de

Philipp Kalytta
Institut für Nachrichtentechnik
Technische Hochschule Köln
Köln, Deutschland
philipp.kalytta@smail.th-koeln.de

***Abstract*—This paper gives a short overview on deploying a functional testbed for orchestrating a service-mesh via Istio on a Kubernetes cluster with Docker Containers.**

Keywords—orchestration, service, mesh, testbed, Kubernetes, Istio, Docker

I. Problem Objective

The objective of this paper is to demonstrate the deployment of a Kubernetes-based service mesh that is orchestrated with the Istio Software. As a beginning, let's look at what a service mesh is. The Istio website defines a service mesh as follows: "The term service mesh is used to describe the network of microservices that make up […] applications and the interactions between them. As a service mesh grows in size and complexity, it can become harder to understand and manage. Its requirements can include discovery, load balancing, failure recovery, metrics, and monitoring." [1]. Istio states to satisfy these requirements by providing control over the complete service mesh architecture. The goal is to test the deployment of Istio and how far it satisfies the proclaimed objective.

A. Problem Sections

The problem objective consists of multiple sections: First, the assembly of the physical hardware part of the testbed. Second, the installation of the Docker, Kubernetes and Istio software. Third, the deployment of a storage provider (needed for the chosen service-mesh, as it is stateful, more on that later). Fourth, the deployment of the actual application as a service-mesh on the infrastructure. And fifth, the testing of the service-mesh and its orchestration tools (Istio and Kubernetes).

B. Problem Section 1 – The Hardware

For the hardware, we were given four desktop computers with a single NIC (Network Interface Card), sixteen Gigabytes of RAM and a 500 Gigabytes HDD. We should use this hardware as the base of our testbed. This also implies some special requirements to the software that is used on top of it, as this is a limited hardware choice, especially given that a usual cluster setup would require far more nodes and would be built onto server hardware.

C. Problem Section 2 – The Software

The Software to use is also fixed (apart from the operating system, which we chose to be Ubuntu Linux 18.04 LTS): We are tasked to deploy and orchestrate a service-mesh on Docker, Kubernetes and Istio. This includes the installation of the named on the four computer nodes and configurating the tools to interact with each other.

D. Problem Section 3 – Storage

As of storage (needed for the stateful containers in the application), we were given no special requirements. But we decided to use a distributed storage solution. Ideally, a storage solution that could be run on Kubernetes. Ceph delivers such a storage solution that can be deployed on Kubernetes [2].

E. Problem Section 4 – The Application

We are free to choose the application that we deploy, it is only limited within some rules:

1. The application must relate to the lecture content
2. The application should be a multimedia application
3. The application should be deployed fully within the Kubernetes cluster

F. Problem Section 5 – The Testing

We are tasked to define some test-cases to evaluate the service-mesh testbed and the used software, especially Istio and its pros and cons. This should include testing of load-balancing, high loads and HTTPS deployment. These are some of the features Istio provides.

II. Solution Approach

A. Basics

1) Container

A container is a standardized unit that contains a process of an application or a microservice and all the dependencies and libraries that the application-process needs, isolated from other processes. Thus, the container is not dependent on the underlying operating system and the application behaves the same in every environment. Only the kernel must be shared by the container with the underlying operating system. One or more containers can run on the same host [3].
These characteristics of containers are possible, because of Linux namespaces and control groups. Each Container has its own namespace to be isolated from other processes on the host and has limited access to resources through control groups [4].

2) Docker

Docker is an open source Software-platform for building, distributing and running containerized applications [5]. The three most important parts of Docker are the container-image, the Registry and the Docker Container. A container-image contains all metadata of a containerized Application and it´s environment [6]. Docker can build an image of a container automatically by reading instructions from a Dockerfile. The Dockerfile is a file the

user of Docker, who wants do containerize an application, writes to define the image [7].
A Docker-Registry is a depository for Docker Images. It can be used to access the stored images by other computers. Docker provides a public Registry for container-images from software vendors, users and open-source projects [5].
A Docker-Container is a running Docker-Image using the Container-Technology.

3) *Kubernetes*

Kubernetes is a software system for provision and management of containers. It helps the developer with the tasks of service localization, scaling, load balancing and self-healing. The architecture of Kubernetes consists of a master node and any number of worker nodes. The master node controls and manages the cluster of worker nodes. The worker nodes execute the applications submitted on the master node. If not specified on which node, the executing worker node is randomly selected. On the working nodes, the applications are executed in the form of containers, for example, Docker Containers. For this to be possible, the container images to be executed must be provided in a registry that Kubernetes can access to pull the images of the container [6].

4) *Service Mesh*

A service mesh is a technology to control interactions and shared data between services. It does not replace networking but is a communication infrastructure that is built into the application.
It is therefore independent of the underlying network structure and is therefore a solution to control the communication in an application that is located in the cloud [8].

5) *Istio*

The functions of Istio can be divided into the three generic terms traffic management, security and observability. In traffic management rules for interaction and routing within the application are defined. As a security function Istio offers management authentication, authorization and encryption of communication. In addition, it is possible to set up and adjust monitoring for this communication in Istio, so that the exchange between the services can be monitored optimally, this is done by deploying so called sidecars with the actual application pods and redirect all traffic through these sidecars. [9]

B. *Testbed Setup*

In the following table all images of containers, that are used in this paper are listed:

Images	Description
php:7.3.11-apache-buster	Webserver
nginx:1.17.6	Cache-Server
mysql:5.6	Database
phpmyadmin	Database-Management
rook/ceph	Shared storage

1) *Apache-Webserver*

The Apache webserver container is responsible for providing the website and videos. The code of the website is directly integrated into the container image, so that the number of webserver containers can be easily scaled by Kubernetes. The metadata necessary for the website is provided to a container instance from a MySQL database. The video data is provided to a webserver instance from the distributed file system, which is mounted as a local folder when the instance is started. The website delivered by the webserver consists of a start page, a playback page and a video upload page. The start page gives an overview of all available videos. On the playback page, the video can be played in an HTML5 video player and the metadata of the video is displayed. On the upload page a new video can be uploaded into the system together with metadata the user provides.

2) *Nginx-Cache-Server*

The nginx container is used in this configuration as a cache server. Besides the content of the website, byte-ranges of the videos are also cached. This reduces the response time of the system as well as the load on the downstream containers. The most important point here is to reduce the load on the shared file system.

3) *MySql-Database*

The MySQL database container handles the persistent storage of the video metadata. For each video the title, description, video categories, number of views and the

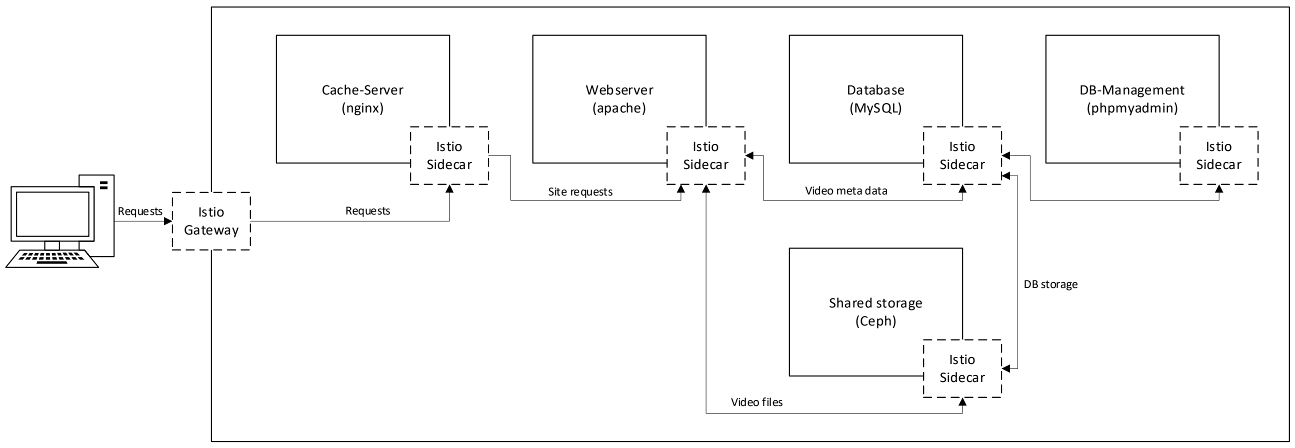

Figure 1: Service Mesh Testbed Architecture

filename of the video in the shared filesystem are stored. The database container instance itself also stores its data in the shared filesystem.

4) PhpMyAdmin

For the administration of the MySQL database a phpMyAdmin instance was also deployed in the Kubernetes cluster. In contrary to the main page it is only accessible internally in the cluster.

5) Ceph

As already mentioned, several times before, a distributed file system is used in this setup. During development, Ceph was chosen because it supports the mandatory ReadWriteMany access mode [10]. This is important because it should be possible for different users to watch and upload videos simultaneously. For the deployment of Ceph in the cluster the storage orchestrator Rook was used. Rook allows the native integration of Ceph into Kubernetes and handles the management, scaling and healing of the storage service [11].

III. Evaluation Results

A. Evaluation of the Istio Software

The evaluation part of this task can be divided into four parts, answering the following questions: 1. How des Istio integrate into a Kubernetes environment? 2. How does Istio provide monitoring of the deployed application? How does traffic routing and traffic control work in Istio? And how can we secure the application?

1) Integration of Istio in Kubernetes

Istio can be installed directly on the Kubernetes cluster, we chose to install it with Helm [3], which can be used to manage Kubernetes applications. But it can also be installed with plain Kubernetes commands (over Kubectl) which means it can be seamlessly integrated into the given Kubernetes deployment. The Istio team provides a set of Helm template files on their GitHub Repository [4], which were used to setup the Software. We then chose to enable the Istio software to automatically deploy sidecars to our applications containers, so that all traffic is send over the Istio software. This integration is robust and straight-forward. It doesn't even require a cluster-restart to work, the sidecars will work right away if deployed on an already existing namespace or a new namespace.

2) Monitoring with Istio

The monitoring on Istio is done over the internal Mixer service and the Prometheus service. The telemetry data can be visualized via Grafana. The sidecars enable Istio to collect the full TCP traffic in the service-mesh (given that the sidecar is deployed on every pod). For the collection of logging data and other metrics, the user must define the log format himself. Istio is heavily focused on HTTP/TCP Traffic for the application traffic, especially regarding monitoring. Apart from that, Istio collects the traffic information without using it. The user cannot define rules or goals for the monitoring and must rely on the monitoring rules that are defined by Istio itself.

3) Traffic Routing and Control

Istio holds its own list containing all endpoints and services in the service mesh. This way, if a pod is deployed with a sidecar, Istio knows how to handle the traffic. This is especially important if you have multiple Versions of a pod or service in your cluster and you want to load balance the pool of services. By default, the traffic is split on a round-robin basis between the members of the service pool. Istio provides a more sophisticated way to split the traffic, by using so called virtual services, destination rules and gateways. A virtual service lets you configure, how requests are routed to the service. A request to a virtual service will trigger the evaluation of some associated destination rules that will specify a real destination in the service mesh. Istio decouples the client form the actual service and is now able to specify routing rules for the workloads. In example, this enables Istio to do a percentage-based destination routing: 5 percent of the traffic should go to service A and 95 percent should go to service B. This is a typical use case in rolling out a new version of a service. The virtual service would be the abstraction layer in front of the real service, that exists in parallel in different versions. The new version could be tested with only 5 percent of the users, this would minimize risk for the developers and discomfort for the users. For high-usage service meshes, it might be necessary to load balance via "Least requests" (The pod with the least requests gets the next request), which is also possible. In HTTP, Istio also is able to split traffic on header fields or the URL. It is possible to let a subset of a web-application be handled by a completely different service behind virtual service (that the user sees and uses). To access virtual services from the outside of your mesh and to manage the inbound and outbound traffic on the edge of the service mesh, an Istio gateway is used. There, you can specify what traffic is allowed into and out of the mesh. The virtual services are bound to a gateway and the rules defined for the gateway then apply to the virtual service as well.

4) Securing the Application - HTTPS over Istio

The Application is accessible via HTTP. To secure the access to the applications incoming traffic, the Istio gateway can be expanded to not only listen for HTTP traffic, but, on a second port, to listen for HTTPS traffic. This is done by modifiyng the gateway configuration and adding a second port section to it (See appendix, "gateways.json"): This port section then specifies HTTPS as protocol and can be further configured by a "TLS" config section. The private key and certificate (those have to be generated beforehand or recevied from a trusted third party) can then be referenced in this section. HTTPS can also be restricted to only relay traffic to certain hosts or to only be deployed for certain parts of the application (mixed content). More options in the TLS part of the configuration enable the redirection of HTTP to HTTPS and definition of minimum and maximum HTTP versions to use as well as setting supported cipher suites. The deployment of HTTPS works on-the-fly, other services won't see an interruption while deploying HTTPS side by side with the existing service. The deployment via an Istio gateway will also obfuscate the underlying service mesh, as an external user will only see the gateway properties (Istio gateway web server fingerprint).

B. *Evaluation of the deployed Service-Mesh*

The service mesh application was evaluated to check how it performs in a real-world use-case. Some video files were uploaded to the application to use them as a reference point. They were then requested by clients together with the associated metadata in the database. At first, some single requests were made by hand using a web browser. While showing that the application works in the desired way, this proved to be too little traffic to test the limits of the service mesh deployment with Istio. It could only show that the flows in the application behaved according to the traffic routing rules that were defined via Istio (i.e. we could see that traffic routed to a different version of our containers would result in an different output to the user). To further test the service mesh, an automated way of testing was deployed.

1) *Performance Evaluation*

An automated shell script was used on several client nodes, that would request a large number of resources from the application at once simultaneously. The script would try to max out the number of concurrent connections and would also request non-existent resources as well as existent ones. This is comprised of the video files (this simulates a video playback on the client) as well as the webpages (this simulates users requesting the video overview page on a web browser), which were generated dynamically. Most of these requests had unique URL-parameters, which means that the caching containers could only cache a part of the requested URLs. This enables the script to better test the service behind the caching containers.

We observed that we never saw more egress traffic that one Gigabit per second in sum over all the client nodes. Every cluster node was connected via a one Gigabit Ethernet link. This led to the observation, that all traffic in the cluster, that came from outside was routed over to one node, although the DNS was configured to serve all three cluster nodes as a connection point for the clients. Internally, Istio deploys its own ingress traffic gateways on a single node and the administrator has no apparent way of choosing where the gateways will be deployed. This limits the traffic handling capabilities to the maximum of the link speed of the chosen node. In contrast, a service that is deployed directly in Kubernetes and load-balanced via DNS, would be able to use the links of all three cluster nodes, giving a total of three Gigabit per second of ingress and egress traffic (minus inter-node traffic).

IV. Discussion and Conclusion

A. *Documentation and API*

Istio was released in 2017 and has been further developed up to the current version 1.4.3 (as of 20.01.2020). Although the version has a major release and Istio has many basic functions for a service mesh, many required functions are still in alpha or beta stage. For example, IPv6 support is still in alpha, locality load balancing and authorization are in beta, although these are features that are of high importance. Users have yet to wait for a stable version of these elements.

As young as the service mesh is the documentation. It describes the implementation of the functions of Istio in small examples or with the example application "Bookinfo". Each feature is represented by at least one YAML file and the possible use-cases are explained. Nevertheless, the documentation has the following weaknesses: "Bookinfo" uses the outdated version 1.3; for some functions, the documentation consists of a YAML file without an exact explanation of the structure and the displayed attributes, it also hardly contains any comments.

B. *Pros and Cons of Istio*

Istio allows for deployment on a new or already existing Kubernetes Cluster that might already contain services. Even the rollout of sidecars to already existing pods didn't interrupt the functionality of the service mesh. The routing of the traffic over Istios Gateways offers both, pros and cons: The granular control over how the traffic should be treated and routed internally in the service mesh enables the administrator to get a much finer control over the network traffic. But the apparent limit for ingress traffic to be routed through only one node limits the ability for high availability and high bandwidth service meshes. Istio tries to centralize the monitoring but fails to deliver a consistent and useful monitoring experience. This all together with the bad documentation and somewhat immature API points to the question if Istio at this point is a useful software to use. In view of large-scale applications Istio certainly will help controlling and securing the traffic, but only to a degree. We leave it to the reader to decide if Istio can be considered for their use-case.

C. *IPv6 with Istio*

Another point to look at is the IPv6 support for Istio, especially since RIPEs IPv4 pool is depleted [13]: Kubernetes supports Ipv6-only clusters since 1.9 and Istio gained IPv6 support with version 1.2 (current version, as of writing of this paper is version 1.4), but the authors stated that the state of the IPv6 implementation is in experimental alpha [14]. We didn't make use of IPv6 in this testbed but it's worth a reminder that if you plan on using Istio in an IPv6 environment, this might cause additional challenges before being able to make use of Istios feature set.

D. *Is Istio worth being implemented?*

Finally, of course, there is the question. Is Istio worthwhile?

We have concluded, yes, but with limitations. The features that Istio offers in the areas of traffic management, security and observability are hardly or not at all covered by Kubernetes, as Kubernetes is mainly focused on infrastructure and resource management. From this point of view, Istio is a clear benefit for a service mesh. after setting up Istio, you quickly realize that Istio requires many containers and thus a considerable amount of resources. Therefore, the use of Istio should only be considered for larger service meshes. But especially for larger product systems a very good reliability and stability is required, which is unfortunately still missing in the current status of Istio. This is reflected in the software as well as in the documentation of Istio. All in all, we see Istio as a good approach, which unfortunately still lacks maturity to be used in a meaningful way.

V. References

[1] Istio.io, „Istio / What is Istio?,“ Istio.io, 14 11 2019. [Online]. Available: https://istio.io/docs/concepts/what-is-istio/#what-is-a-service-mesh. [Zugriff am 27 12 2019].

[2] Red Hat Inc., „Installation (Kubernetes + Helm) - Ceph Documentation,“ 2016. [Online]. Available: https://docs.ceph.com/docs/mimic/start/kube-helm/. [Zugriff am 04 01 2020].

[3] Docker Inc., „What is a Container?,“ [Online]. Available: https://www.docker.com/resources/what-container. [Zugriff am 20 01 2020].

[4] Microsoft Corp., „Introduction to Containers and Docker,“ [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/. [Zugriff am 20 01 2020].

[5] Docker Inc., „Why Docker?,“ [Online]. Available: https://www.docker.com/why-docker. [Zugriff am 20 01 2020].

[6] MarkoLuksa, Kubernetes in Action, Manning Publications, 2018.

[7] Docker Inc., „Dockerfile reference,“ [Online]. Available: https://docs.docker.com/engine/reference/builder/. [Zugriff am 20 01 2020].

[8] Red Hat Inc., „What is a service mesh?,“ [Online]. Available: https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh. [Zugriff am 20 01 2020].

[9] Red Hat Inc., „What is Istio?,“ [Online]. Available: https://www.redhat.com/en/topics/microservices/what-is-istio. [Zugriff am 20 01 2020].

[10] Kubernetes.io, „Persistent Volumes - Kubernetes,“ 20 01 2020. [Online]. Available: https://kubernetes.io/docs/concepts/storage/persistent-volumes/#access-modes. [Zugriff am 20 01 2020].

[11] Rook Authors, „Rook.io,“ 2020. [Online]. Available: https://rook.io/docs/rook/v1.2/. [Zugriff am 19 01 2020].

[12] Cloud Native Computing Foundation, „Helm Docs | Helm,“ 2019. [Online]. Available: https://helm.sh/. [Zugriff am 13 01 2020].

[13] Istio.io, „istio/install/kubernetes/helm/istio/ - istio/istio,“ 13 01 2020. [Online]. Available: https://github.com/istio/istio/tree/master/install/kubernetes/helm/istio. [Zugriff am 16 01 2020].

[14] RIPE NCC, „RIPE IPv4 Pool,“ 20 01 2020. [Online]. Available: https://www.ripe.net/manage-ips-and-asns/ipv4/ipv4-pool. [Zugriff am 20 01 2020].

[15] Istio.io, „Istio / About,“ 2019. [Online]. Available: https://istio.io/about/feature-stages/#core. [Zugriff am 21 01 2020].

VI. Appendix

A. Configs

https://th-koeln.sciebo.de/s/ukYVTQo8qtirVQl

B. Code

https://th-koeln.sciebo.de/s/ukYVTQo8qtirVQl

C. Container

Images	Hyperlink
php:7.3.11-apache-buster	https://hub.docker.com/_/php
nginx:1.17.6	https://hub.docker.com/_/nginx
mysql:5.6	https://hub.docker.com/_/mysql
phpmyadmin	https://hub.docker.com/r/phpmyadmin/phpmyadmin
rook/ceph	https://hub.docker.com/r/rook/ceph