

MASTER'S THESIS 2021

Evaluating Service Mesh as a Network Monitoring Solution

Ennio Mara, Elias Frykholm

Elektroteknik
Datateknik

ISSN 1650-2884

LU-CS-EX: 2023-79

DEPARTMENT OF COMPUTER SCIENCE

LTH | LUND UNIVERSITY



EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-79

**Evaluating Service Mesh as a Network
Monitoring Solution**

Utvärdering av Service Mesh som Verktyg
för Nätverksmonitorering

Ennio Mara, Elias Frykholm

Evaluating Service Mesh as a Network Monitoring Solution

Ennio Mara

`en0067ma-s@student.lu.se`

Elias Frykholm

`el6426fr-s@student.lu.se`

June 14, 2021

Master's thesis work carried out at Axis Communications AB.

Supervisors: Alma Orucevic-Alagic, `alma.orucevic-alagic@cs.lth.se`
Anton Friberg, `Anton.Friberg@axis.com`

Examiner: Emma Söderberg, `emma.soderberg@cs.lth.se`

Abstract

Modern software systems often utilize a distributed architecture, which uses the network as the communication channel. This results in the system being dependent on the network performance to behave as expected. The network performance varies, and network faults can occur, which may negatively affect the system. As a system maintainer, it is difficult to know if the problem lies in the service or the network.

This thesis evaluates service mesh, an infrastructure layer that facilitates service-to-service communication, as a tool for network monitoring without having access to the network infrastructure.

Experiments in a virtualized data-intensive environment running Istio and Kubernetes were performed, with the goal of evaluating how a service mesh aids in identifying network faults and its performance impact on the system.

A latency increase and bandwidth decrease were discovered through the data provided by a service mesh. Packet loss, however, was not discovered. Using a service mesh resulted in a distinct negative performance impact on the system.

Keywords: Kubernetes, Service Mesh, Istio, Network Monitoring, Distributed Systems, Containers

Acknowledgements

We would like to foremost thank our academic supervisor, Alma Orucevic-Alagic from the department of computer science at LTH for her dedication, feedback and support during the many meetings we've had.

We would also like to thank Axis for the opportunity, and specifically our supervisor from Axis, Anton Friberg. Anton has helped us arrange the thesis and provided insights and expertise throughout it.

Lastly we want to thank our examiner, Emma Söderberg, and our colleagues at Axis for their help.

Contents

1	Introduction	7
1.1	Problem Statement	8
1.2	Research Questions	9
1.3	Delimitations	9
1.4	Related Work	9
1.5	Contribution	11
1.6	Distribution of Work	11
2	Background	13
2.1	Microservice Architecture	13
2.2	Containers and Container Orchestration	14
2.3	Monitoring	16
2.4	Fallacies of Distributed Systems	17
2.5	Service Mesh	19
3	Approach	21
3.1	Method	21
3.2	Infrastructure	22
3.2.1	Routers	23
3.2.2	MinIO	23
3.2.3	Kubernetes Cluster	24
3.2.4	Service Mesh	24
3.3	Experiments	25
3.3.1	Network Fault Experiment	26
3.3.2	Application Performance Experiment	27
4	Results	29
4.1	Istio Metrics	29
4.2	Network Fault Discovery	30
4.2.1	Latency	30

4.2.2	Bandwidth	32
4.2.3	Packet Loss	32
4.3	Application Performance Impact	32
4.3.1	Small Files Transfer	33
4.3.2	Large Files Transfer	36
5	Discussion	37
5.1	Istio Metrics	37
5.2	Network Fault Monitoring	38
5.2.1	Latency	39
5.2.2	Bandwidth	40
5.2.3	Packet Loss	40
5.3	Istio Performance Impact	41
5.3.1	Performance Analysis	41
5.3.2	Envoy Throughput Limitations	42
5.4	Future Work	43
6	Conclusion	45
	References	47
	Appendix A Configuration	55
A.1	Quagga Configuration	55
A.1.1	OSPF Configuration	55
A.1.2	Zebra Configuration	56
A.1.3	BGPD Configuration	56
A.2	Istio Operator Configuration	56
A.3	Prometheus Configuration	58
	Appendix B Prometheus Queries	61
B.1	Latency	61
B.2	Bandwidth	63
B.3	Packet Loss	63
	Appendix C Rclone Kubernetes Job	65
	Appendix D Response Loss Experiment	67

Chapter 1

Introduction

In software engineering, a system's lifecycle can be divided into multiple parts, where development and maintenance are two of them [1]. Development is when the system is created, and maintenance is when the system is put in use and enhancements are added and errors are fixed as they are discovered. According to some estimates, 20 to 60 percent (average of 40%) of the time and cost of the system is spent in development, the remaining time and cost is spent on maintenance [1].

When systems are in their maintenance phase (i.e., in production), the engineers in charge need to know how the system behaves to prevent and discover potential faults [2]. The state of the system can usually be determined by monitoring, for example, seeing how high the RAM utilization is and checking the application logs of specific machines. While this might work for smaller systems, as the number of systems increases, this becomes more difficult [3, ch.8]. For larger systems, the collections of telemetry and logs must be automated and stored in a way such that engineers can easily query the data [2].

In recent years, there has been a shift from monolithic applications to a microservice architecture. By utilizing a microservice architecture, an application is split into smaller services, each deployed and developed independently [3, ch. 1]. Alongside the shift towards the microservice architecture was the increasing popularity of running the services as containers with platforms such as *Docker* [4]. Containers are a lightweight environment for running services which enables faster scaling of the systems [4]. With the help of container orchestration tools that manage containers such as Kubernetes [5], containers can automatically scale across multiple servers, making distributed systems more available and popular [6].

Due to the distributed nature of microservices and the possibilities when combined with containers and container orchestration tools, a large part of the network communication in a system is carried out internally between the services [3, ch. 1]. This introduces new problems that need to be considered, which fall under the *fallacies of distributed computing* [7]. They consist of false assumptions often made when developing network-based systems.

Recently a technology called *service-mesh* has gained traction in handling the problems with the fallacies of distributed computing, where one example of a service mesh is Istio [8].

A service mesh introduces an infrastructure layer that handles the communication between services in it. Besides providing features that handle the problems with fallacies of distributed computing, it also provides observability features for inter-service communication [9]. Its observability features are commonly used to monitor the services' health, not the network's.

1.1 Problem Statement

Axis Communications AB [10] is a market leader in surveillance and security systems based in Lund, Sweden. This thesis has been carried at a department in Axis, which is responsible for collecting, visualizing and analyzing system metrics and telemetry from select Axis devices, with the goal to find and fix errors in the devices. They have experienced some problems regarding the reliability of the network, and they are currently in the process of moving some of the services to a Kubernetes cluster. By moving the services to a Kubernetes cluster, it is believed that the network problems will be even more prominent and more difficult to debug.

Anton Friberg is a Data Engineer at Axis Communications and has provided a summary of the problems experienced with the lack of monitoring of network requests for individual data services. Based on his summary, the following problems can be formulated:

1. It is not easy to monitor network performance accurately on a service level. (Bandwidth, Latency, Packet Loss)

In distributed systems composed of services, the communication between the services and users is done through the network. The network, however, is unreliable according to the fallacies of distributed computing. This unreliability is important to understand how it affects a given service, especially over time. For example, a network fault could lead to a decrease in bandwidth, possibly affecting the system's reliability. This, however, is not easy to monitor without having access to the network infrastructure, for example, routers, etc.

2. It is time-consuming to debug a degradation of the network environment.

When a degradation happens due to a network fault, it is unclear to a service owner whether the fault is due to the service or due to the underlying network environment. Therefore, a service owner must first make sure that the service is not at fault, which is a time-consuming process. If the network health can be visible on a service level, the detection time will be lower.

3. Data service performance and limits cannot be fairly estimated or benchmarked since the outside influence on network infrastructure is unknown.
4. Data services are highly dependent on the performance of the network layer, and costly mistakes in network application configurations are common.

For example, a firewall may be misconfigured so that it limits the available bandwidth flowing through it, causing problems for services that transfer large amounts of data. If monitoring solutions are in place, a service owner could discover these problems and alert the network engineers faster.

1.2 Research Questions

The research questions have been formulated to address the stated problems provided by the case company in Section 1.1. However, care has been taken to make the research appealing to a broader audience.

- RQ1 What data can a service mesh provide to aid in network monitoring?
- RQ2 How can the data provided by a service mesh aid in identifying and locating problems with low bandwidth, high latency, and packet loss?
- RQ3 How does applying a proxy in the form of a service mesh affect application performance?

1.3 Delimitations

This thesis work is based on problems experienced at one department at the case company. The techniques and tools used in this study for conducting the experiments were chosen to match the requirements of the case company. This acted as a basis for using Kubernetes and Istio as the foundation in the experiments, along with the use of Rclone and MinIO to model data-intensive applications. Although the chosen configuration was made in order to be easily integrated into the planned setup at the case company, the solution may be applicable in other cases using different applications.

Due to the flexibility of a simulated environment, experiments in this thesis were run on virtualized hardware (in a provided virtual machine from the case company), which can differ from running on real hardware. Since the virtual machines share resources such as CPU and disk with each other, the results could differ from running on dedicated hardware on a real network (comprised of non-virtualized routers).

The way the network faults are introduced in the experiments differs slightly from how real network faults appear in real networks. The tool used in this thesis, *tc*, was chosen due to the need of introducing network faults on-demand. *tc* introduces a constant network fault, such as a constant bandwidth limit for a given time. In a real-world situation, these deterministic faults may not be present.

Due to the scope of the thesis, only the observability part of the service mesh has been of interest, and therefore, other features of the service mesh are not discussed in depth. Some of these features have been disabled to minimize the performance impact for the performance tests and therefore can yield different results using other settings.

A deliberate choice in the experiments was to expand the Istio service-mesh to also include an external virtual machine to better model the use case of the case company. Therefore, the experiments were designed to monitor the traffic between the services running in the Kubernetes cluster and the service running in the virtual machine.

1.4 Related Work

With service mesh being a relatively recent topic, the amount of previous academic research to provide a base for this report is limited.

One related work is a study performed in [11], where similar observability problems were discussed which were experienced in *SigSaude*, an application that handles health data. The main goal was to be able to monitor performance at different stages in the data operation process in order to identify potential problems of services. The following 8 requirements were established to evaluate the outcome:

1. Metrics related to the execution time of each request
2. Request success rate metrics
3. Traceability in the flow of requests
4. Request size metrics
5. Centralized handling of logs for problem debugs
6. Alerts related to the average request time
7. Alerts reporting degradation or failure in microservices
8. Alerts related to host memory usage

Many of the requirements stated in the study are similar to the ones stated in the problem summary of this report. The result of the study managed to meet all the requirements by applying the service-mesh solution Istio. However, the study only covers observability discussions on a service level to discover problems with the services themselves (not the network). This differentiates from this report since the goal of this report is to discuss whether a service mesh solution can be used in order to discover actual network problems. To evaluate this, the service mesh needs to be used differently in order to map the collected metrics to the servers the services are running on.

Several previous research efforts such as [12], [13] and [14] have evaluated service mesh similarly to those [11] mentioned above. They all have performed monitoring on a service level, yielding successful results. However, academic research on using service mesh to discover network problems on a server level has not been found.

The application performance impact of using a proxy-solution like Istio has previously been researched in [15], in which it was shown that there is a distinct performance degradation with Istio applied. However, the goal of the report was mainly to research the performance of large Kubernetes clusters with regard to the number of containers used for each service.

In [16], a way of discovering and locating network anomalies are discussed by using software-defined networking (SDN). The idea was to replace traditional switches with SDN switches in order to collect data at any point in the network topology. SDN makes it possible to add additional layers to the traditional network setup to collect more detailed data. Data could then be polled with a monitoring node in order to calculate metrics over network performance. The network metrics calculated were divided into two parts, the application metrics: response time, throughput, and session number, and the network metrics: packet loss, delay, and available bandwidth. With the collected metrics, an algorithm was implemented to detect anomalies and, if possible, change routes through an SDN controller or alert network administrators. This is in many ways similar to how a service mesh works

and exposes similar metrics but on a network topology level. This approach is used in large cloud platforms such as *Google Cloud* [17] to monitor the network and easily make changes to the network infrastructure. However, this setup requires changes to the current network infrastructure and is therefore not applicable to the problems stated in this report.

1.5 Contribution

When working with distributed applications, the network is an important aspect since all communication between the applications is done through it. According to the supervisor at Axis, network problems in distributed applications are generally hard and time-consuming to debug, which is also supported by the *Cloud Native Foundation's* survey [18] in 2020, showing that 26% of the survey-takers stated networking as the biggest challenge with managing distributed containers. Having a clear view of how the network behaves between applications helps the engineers debug and fix problems faster.

Service mesh is already used today in applications since it provides features such as traffic management, authentication, and observability. This report will mainly cover the observability aspects of a service mesh in order to monitor the network. The observability part for Istio, a service-mesh implementation, covers application service metrics such as latency, traffic, errors, and saturation [19]. A service mesh is traditionally applied to monitor the performance of the services, with the mentioned metrics being collected based on calls between the services and therefore do not provide any information on the network. In the context of this report, this means that an investigation has to be made whether a service mesh can be used to provide metrics based on the network infrastructure instead of the calls between the services. Therefore, the report will contribute to the knowledge around service mesh and potentially find an alternative use-case for the technology.

The results of this master thesis will contribute to the development of the distributed computing infrastructure at Axis. This by hopefully finding a solution to the stated problems that is easily applicable to the current setup without any large changes to the actual service implementations. The thesis will also contribute to the general knowledge regarding problems occurring due to the observability in distributed computing and provide insights into how to solve these problems, focusing on network monitoring. The thesis will serve as a basis for further research along with the previously mentioned articles [20, 12, 11, 15, 16]

Although the project has been formulated around the problems discovered at Axis and made to be implemented in their current setup, the environment used for testing in this problem is made to reflect a typical network, and the resulting solution may apply to other situations. The performance results and measurements provided in this report may also be used to compare other service mesh implementations.

1.6 Distribution of Work

The work behind this thesis has been evenly distributed between the two authors. In the initial phase of the thesis, Ennio Mara focused mostly on setting up the infrastructure in order to perform the experiments. At the same time, Elias Frykholm researched Istio and tested different monitoring setups. Towards the middle and end of the thesis, both authors

worked together to write the report, perform the experiments, document their results, and present them in the report.

Chapter 2

Background

In this chapter, some background and context is provided in order to understand the approach and results presented in this thesis. Some background in how distributed software systems are developed is provided, together with the challenges that arise from it. Monitoring of software systems, as well as background regarding service mesh, is also provided.

2.1 Microservice Architecture

In recent years, developing software using a microservice architecture has been adopted by multiple projects during recent years. A microservice architecture is characterized by splitting an application into different components (services), where they are developed and deployed independently [21].

In order to understand why there has been a shift, one needs to understand how applications were developed and deployed prior to this. Before, a monolith architecture was common, whereby the application was developed as a single unit [21, 22]. A typical monolith application consists of a client-side (which the users interact with), an application layer where the business logic is defined, and a storage layer (for example, a database). The client layer connects to the application layer, which uses the storage layer for persistent storage (for example, serving an article to the user) [22].

Due to the fact that monolithic applications are packaged as a single unit, over time, the size of this unit's codebase will grow [22], which can lead to problems such as difficulty to keep a good modular structure and scaling problems [21]. Scaling problems are caused by the fact that the entire unit must be scaled up, instead of only the component that requires more resources [21].

By moving to a microservice architecture, the application is divided into different services, where each service is small and autonomous [3, ch. 1]. A microservice should be small and focused on doing one thing well. It should also be autonomous, i.e., a separate entity. The communication between services is handled by network calls, where services expose APIs that

other services may use [3, ch. 1].

Since the application now is composed of multiple services decoupled via APIs, the services can now be scaled independently from each other [21]. For example, an application that allows users to store and retrieve their photos, where the photo-storing process is resource hungry. If the application is divided into two services, a store- and a retrieve-service, the store-service can be scaled to accommodate for increased use without affecting the retrieve service.

The microservice architecture provides multiple benefits, but it also comes with costs. One such cost is the move to a distributed system, as the services now will be communicating through network calls. Some common problems to communicating through network calls are performance and reliability [23]. A more thorough description of these problems is given in Section 2.4. These problems the network may introduce need to be handled by each microservice, adding additional complexity.

Another cost of using microservice architecture is the increased operational complexity it introduces [23]. From an observability point of view, monitoring becomes even more important as the number of services grows. If a performance degradation appears in a monolithic application, the only place an engineer has to look at is the monolith, as all the logic is centered there. When using microservices, this task becomes more difficult, as now there are more services to debug, most likely running on different hosts [3, ch. 8].

2.2 Containers and Container Orchestration

Containers and container orchestration in the software context are tools and techniques developed to maintain system reliability and maintainability. A container is a service packaged in a lightweight, reliable environment, including everything needed to run, e.g., code, runtime, system tools, system libraries, and settings [3]. Container orchestration is the process of managing containers, and it is often done through tools used for automation of tasks needed to maintain and scale containers [6].

To understand why the software scene has moved towards the containerized approach, it is important to first understand the difficulties and problems of the traditional ways. The most obvious way of running software is to run it as it is on a system. There are numerous ways a system can be set up, which can affect the running service. This can potentially lead to situations where the software is running perfectly fine on one configuration and not running at all on another. Many system variables can impact the software, such as operating system, operating system version, dependencies, system hardware. These variables should be avoided in order to maintain good reliability.

In order to address these problems, it became more frequent to run services on a Virtual Machine (VM) instead of running the software directly to manage the configuration variables more easily [3]. A VM virtualizes system hardware, therefore, in order to run software on a VM, a full operating system is needed to run software on the VM and a hypervisor [24] is needed to create multiple VMs on one machine. This induces a performance impact on the system due to the resources needed for the VM and long boot times by requiring to start the VM first and then run the service [4].

Containers were created in order to solve these problems. A container is an abstraction at the application layer that packages the code together with the dependencies [4]. The con-

tainers share resources with the host operating system, and therefore, a full operating system and kernel are not needed in the container. Container resources are managed through management software on the host, eliminating the need for a hypervisor since it is not trying to emulate a full system [3]. The most popular container platform is *Docker*, according to the RightScale State of The Cloud Report [25] with 66% of the enterprise respondents and 57% of the total respondents using *Docker*.

Due to their portability, low impact on the system, and low start time, containerized software can be quickly scaled both horizontally and vertically [3]. This can be done manually by starting containers and editing how much of the system's resources they are allowed to use. This is where a container orchestration tool can be useful where popular examples are *Kubernetes* [5], *Docker Swarm* [26], *Nomad* [27], and *Apache Mesos* [28]. *Kubernetes* is the most popular container orchestration tool according to the RightScale State of The Cloud Report [25] with 60% of the enterprise respondents and 48% of the total respondents using *Kubernetes*, which is heavily integrated with *Docker*.

According to the *Cloud Native Foundation* report [18], large software systems can utilize thousands of containers, with 23% of the respondent companies typically run over 5 000 containers. Managing containers in large systems can therefore be very difficult, and this is where container orchestration tools aim to provide a way to do this efficiently. Common features provided by container orchestration tools include the following [6]:

- Scheduling
Scheduling of container deployments, resource allocations, service upgrades, etc.
- Resource management
Allocating system resources for specified containers or services and load balancing.
- Service discovery
Grouping containers based on which service is running on them.
- Automatic horizontal and vertical scaling
Automatically allocate more resources to the containers or deploying more containers running the service based on demand.
- Container health checks
Checking the containers and their running service for errors and restarting containers if needed.
- Updates and upgrades without downtime
Creating new containers running the upgraded service and automatically redirect traffic in order to check the new container is working properly before stopping those running the old version.

Modern container orchestration tools can also operate across multiple physical servers on a distributed system. The workloads can then be divided evenly across them for increased performance through load balancing. The mentioned features can be made on a service level rather than a container level which implies that maintainers do not have to pay much attention to the actual containers [6].

Kubernetes was selected as the container orchestration tool for the setup in this thesis. A *Kubernetes* cluster consists of a set of nodes (e.g., a Linux server), in which the scheduler

can schedule pods to run in. A pod is the smallest unit that can be deployed in a Kubernetes cluster and consists of containers with shared resources such as storage and networking [5]. In the context of the Istio service mesh, a pod could, for example, consist of two containers, one that runs the service (for example an API), and the other runs the service-mesh proxy, which handles in/outgoing traffic.

2.3 Monitoring

Monitoring distributed systems is the continuous process of collecting, processing, aggregating, and displaying quantitative data and is an essential component in detecting and debugging possible faults in the system [22, 29]. Monitoring can be applied at different levels in the system and with different purposes.

By monitoring the growth of a system with regards to the database size, the system can be expanded accordingly or optimized further. It can be possible to detect failures and create alarms to quickly get notified and act accordingly without waiting for reports from the user base. Dashboards can be created based on the monitoring data to provide user-friendly graphical interfaces with an overview of the current state of the system and statistics with tools such as Grafana [30].

Monitoring can be divided into two categories, white-box, and black-box monitoring. White-box monitoring is the metrics exposed by the system itself, which is often the metrics used to detect problems and failures. Black box monitoring on the other hand is symptom-oriented the monitoring of the externally exposed elements which is available to the user. Black-box monitoring represents active instead of predictive problems which sometimes might not be as visible in the data provided by the white-box monitoring [29]. With the two monitoring categories combined, it may be possible to both answer what and why the system behaves a certain way.

With distributed systems, there are four important metrics to monitor in order to discover some of the more critical problems [29]:

1. Latency

In black-box monitoring, the time to service a request can be the total time from a user clicking a button to the point where something happens on screen. With white-box monitoring, it is possible to get measurements on latency from calls between services to know if a certain service is on a slow network or is poorly implemented.

2. Traffic

Measurement on how much demand is placed on the system, such as input/output operations or bandwidth usage. With this data, it can be known if the system should scale across multiple networks, if there is a weak connection to one of the servers, or bad load balancing.

3. Errors

The rate of requests that fail may indicate a broken service, network error, policy break, or network timeout.

4. Saturation

A measurement of the load on a service is often a measurement of the hardware limitations of a service, such as memory, I/O constraints, and CPU usage. This may indicate bad load balancing or that scaling is needed in terms of performance.

There are several ways to implement the monitoring of a system, with the recently most popular being applying a service mesh solution which is similar and somewhat based on the following concepts. Network probing is another way to collect the network data of a system. The idea is to create a virtual user which continuously generates traffic to the services and monitors mainly response times and errors. The probes can be created both on black-box and, to a degree, on white-box levels through the origin and request type. This, however, introduces additional traffic to the system, which can impact the performance depending on probing frequency and system size. A tool to aid in collecting metrics using probing and displaying the metrics is Dynatrace [31]. Another way to monitor a system is by implementing monitoring layers to the services themselves. This approach makes it able to monitor all of the previously mentioned metrics due to access to all the data by implementing data collection for the services incoming and outgoing calls and may provide both white-box and black-box levels of metrics. The problem with this approach is that it is hard to manage both the data and the implementation, induce code intrusion, and needs to be implemented for every service. There are tools and frameworks supporting this approach and making it easier, such as Hystrix [32] and Spring Boot Admin for Spring deployments [33].

In order to monitor data over time, the data needs to be stored in some way. In this thesis, the storage solution of choice is Prometheus [34]. Prometheus is a time series database that stores data with a metric name and value, along with labels in the form of key/value pairs. A time-series database is often used in monitoring because it indexes elements based on timestamps and is optimized for measuring change over time. Metric labels in Prometheus can be used to filter data and uses the query language PromQL to fetch data from the database. Prometheus uses a pull model, which means that Prometheus, at a set interval, e.g., every 30th second, retrieves the data that its targets expose and saves it to its internal database. This means that the targets need to provide data so that Prometheus can scrape it.

2.4 Fallacies of Distributed Systems

With distributed computing comes problems that wouldn't be as prominent on traditional monolithic architecture, such as the network connections between the system endpoints. To address the problems, Peter Deutsch created a list of eight assumptions developers make about distributed systems referred to as the eight fallacies of distributed systems [35]. Monitoring these fallacies is important to identify what is causing the problem, if it's temporary, and decide whether to take action.

The assumptions are the following:

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite

4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

Assuming that the network is reliable as a developer can cause problems since calls over the network will eventually fail. A network request might fail in many ways, such as dead switch, timeout due to network overload, or improper package routing. This must then be taken into consideration when developing to create fail-safes such as retries, acknowledge requests and ignore duplicate requests [7, 36].

A network connection between two systems can theoretically never be zero due to packets not being able to exceed the speed of light [37]. Latency is the time it takes to move data from one place to another and can greatly impact user experience. Latency can be affected by distance to servers, routes, network load, etc. However, there are techniques to mitigate latency by expanding servers and the locations of them or mask latency by caching and mirroring [36, 37].

Network bandwidth continuously seems to increase exponentially according to Nielsen's law [38] and induces fewer problems than a couple of years ago. However, application demand also increases, and depending on the application, assuming bandwidth is infinite can still lead to problems where bandwidth acts as a bottleneck. Ways to prevent this include responsibility segregation and to use domain-driven design patterns [36].

Security is essential when transferring critical data over the internet, and assuming that the network is secure can have large consequences. Therefore, security should always be analyzed and weighed based on the cost, risk, and probability of a potential breach [39]. Developers should be educated in the area, and multilayered security should be implemented in critical applications [36].

In a development environment, network topology rarely changes. However, when the product is deployed, the topology may be altered by node changes, servers being added or removed or faults introducing network changes [39]. Assuming topology does not change may lead to developing the application for the development environment with static routes and ips, abstracting the physical network layer by using a DNS or providing discovery services are ways to mitigate this [7].

Unless a system is in an early stage or is very small, the probability is high that there is more than one administrator in play throughout different levels of the system and external entities [39]. There might be constraints on how the code can be developed and evolved when administrative domains need to be crossed. Solutions to this problem include decoupling parts of the system from each other by making them version-independent [36]. Third-party components should be isolated so that a failure on the third-party component has a small impact as possible on the system.

There are generally two aspects in consideration when stating that transport over the network has a cost. One is the network infrastructure costs, e.g., leasing bandwidth, routers, network security, and maintenance. The second aspect is the CPU time to serialize and de-serialize data to transfer over the network [7, 36, 35, 39]. How large impact this may have de-

depends on the application but to mitigate this, the developers should consider which protocols they use for transferring data to decrease CPU time and analyze if the cost is a problem [36].

The final fallacy is assuming the network is homogeneous. A network almost always contains many different devices and device configurations that can connect to the system. Developers should therefore follow standards and use standard protocols when developing network systems whenever possible and unless there is a good reason not to [36].

Knowing the fallacies of distributed computing, a connection can be seen between the important metrics to monitor previously mentioned. By exposing the metrics, most of the fallacies can be monitored. Monitoring these will give a good overview of the state and performance of the system and provide data to take action and debug as problems arise.

2.5 Service Mesh

The shift to microservices from monolithic applications means that the fallacies of distributed systems [7] must be addressed. One way of accomplishing this is by rearchitecting the applications to be more resilient and flexible. Several open-source libraries exist, such as *resilience4j* [40], *Sentinel* [41], and *failsafe* [42], which provide tools such as retries¹ and circuit breakers².

By re-architecting the application in order to accommodate for the fallacies of distributed systems, added complexity is introduced to the application. This complexity needs to be managed by the application developers, which leads to them spending less time on the business logic [43, ch. 1].

A service-mesh solves these problems by introducing an infrastructure layer that facilitates the communication between services without requiring modifications in the applications [9]. This architecture consists of two planes, a data- and a control-plane. The data-plane consists of proxies, which take the application traffic and route it to its correct destination [9, 43]. The control-plane is the authority that controls the proxies by configuring them, for example, by enforcing routing policies and what metrics to collect [9].

The features a service-mesh provides to an application are traffic control, service discovery, observability, and security, amongst others [43, ch. 1]. The traffic control property makes, for example, canary-deployments possible, where a new version of a service is used by only a subset of users, for example, beta testers [43, ch. 3]. Through the security property, a service mesh allows defining for example access policies, i.e., which services a service can access [9]. The observability property allows an operator to monitor the inter-service traffic, which provides metrics such as latency, request volume and, success and error rates, amongst others [9].

In the data plane, the proxy is the central component. The application is configured to talk directly to this proxy instead of the network interface [9]. This means that all outbound and inbound traffic to the application must pass through the proxy. Traffic passing through the proxy allows the component to monitor traffic and apply traffic-control and access policies [9].

¹If a request to a resource fails, send the request again [3].

²A circuit breaker works by monitoring the sent requests. If they have failed, the circuit breaker will blow, which will cause all future requests to fail directly. After a specified amount of time, the circuit breaker will see if the resource has recovered and then continue sending requests normally [3].

There are several different implementations of service-mesh, such as Istio [8], Linkerd [44], Consul [45], AWS App Mesh [46], Open Service Mesh [47], and Kuma [48]. All meshes except for AWS App Mesh support running on top of Kubernetes (AWS App Mesh requires that the Kubernetes cluster is running on AWS).

Chapter 3

Approach

In this chapter, the approach used to answer the research questions is explained. This involves the experiment setup, the infrastructure used in order to perform these experiments and the research methodology used.

3.1 Method

The chosen methodology for this thesis in order to answer the research questions is through a method called *action research*. *Action research* is characterized by two stages, a situation assessment phase and a problem intervention phase [49]. The assessment phase can be divided into two parts, *observation*, and *solution*, while the situation assessment phase can be divided into *evaluation* and *reflection* [50]. This process is an iterative one, done until the evaluation is satisfactory. See Figure 3.1 for an illustration of this process.

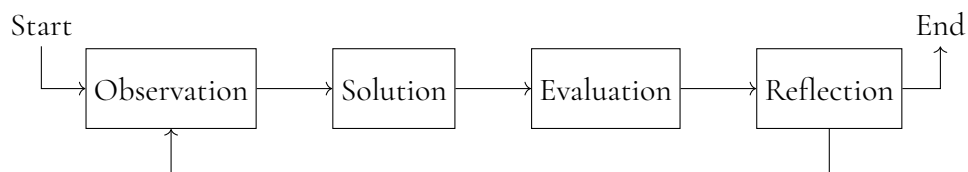


Figure 3.1: The phases in action research.

The research starts by first doing an observation/analysis of the problem that the research aims to solve. Based on that observation/analysis, a set of changes are proposed, which aim to solve the problem. These changes are then implemented, and their effects are evaluated. Lastly, a reflection over how well the solution works is done at the end, which acts as input to the next iteration (or if the research should be concluded) [50].

Since the researcher is both proposing a solution and evaluating it, careful precaution must be taken in order to perform an objective evaluation of the solution. By defining evaluation criteria can for example make the evaluation more objective [50].

In this thesis, **RQ1** was answered by first understanding the configuration possibilities of Istio, and then proposing a solution that could be used for network monitoring. This solution was tried (evaluated) in a small part of the system, in order to understand how it could be used for network monitoring. This process was repeated several times until a solution was found which allowed experimenting in order to answer **RQ2**.

When creating the experiments used to answer **RQ2** and **RQ1**, this methodology was also used. The experiments first started in a small scale, whereby a simple experiment was proposed and then evaluated. The insights learned from each iteration were used to better improve the experiments in order to properly answer the research questions. For example, initially the experiments were performed once, and in later iterations it was decided that the results would be based from an average of ten experiment runs. A more in-depth explanation of the experiments is provided in Section 3.3.

3.2 Infrastructure

Answering the research questions defined in this thesis requires infrastructure to run the experiments on. This infrastructure consists of several nodes connected via a private network created by the authors.

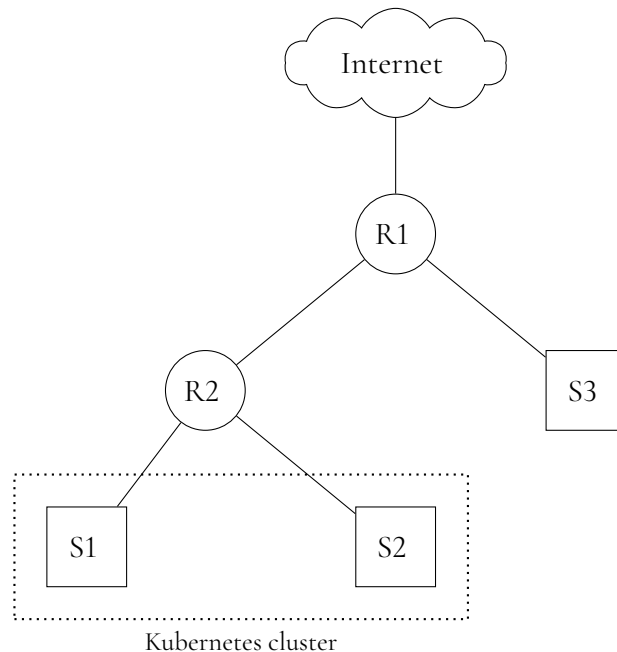


Figure 3.2: Network infrastructure setup

The network infrastructure used in the experiments is shown in Figure 3.2. It consists of virtual servers (depicted as *S1*, *S2* and *S3*) and routers (depicted as *R1* and *R2*). The edges in the graph show the paths between the different machines and the path a network packet has

to take in order to reach its destination. For example, if a packet is sent from $S1$ to $S3$, its route will be $S1 \rightarrow R2 \rightarrow R1 \rightarrow S3$. Similarly, a packet sent from $S3$ to the internet (e.g., Google) will have a path of $S3 \rightarrow R1 \rightarrow \text{Internet}$.

Node	Operating System	RAM	CPU Cores
R1	Ubuntu Bionic (18.04)	512 MB	2
R2	Ubuntu Bionic (18.04)	256 MB	2
S1	Ubuntu Bionic (18.04)	3 GB	2
S2	Ubuntu Bionic (18.04)	22 GB	2
S3	Ubuntu Bionic (18.04)	1 GB	2

Table 3.1: Specifications for the virtual machines shown in Figure 3.2.

All nodes in the setup are running as virtual machines, created and managed via Vagrant [51], which uses VirtualBox [52] (version 6.1.16) as the hypervisor. A configuration file is supplied to Vagrant, written in Vagrant’s own *Domain Specific Language* (DSL). In this file, configuration pertaining to the creation of these virtual machines is specified, such as RAM, CPU, network interfaces, etc. The virtual machine specifications can be found in Table 3.1.

The virtual machines created by Vagrant are running on a machine running Ubuntu 20.04 as operating system, 32Gb of RAM and 8x2.60Ghz *Intel Xeon(R) CPU E5-2697A v4* CPU.

3.2.1 Routers

In Figure 3.2, the nodes are connected directly to each other via the edges. If there is an edge between two nodes, then they can communicate with each other. In order to facilitate communication between two nodes that are not directly connected, there need to be routers that will forward the packets to the correct destination. For example, a packet originating from $S1$ with a destination of $S2$ needs to pass $R2$ in order to reach its destination.

A router is a node that has some knowledge of the correct path a packet should take. For example, a packet originating from $S1$ destined for $S3$ would reach $R2$ first, which would forward it to $R1$, which then forwards it to $S3$.

In these experiments, the nodes acting as the routers are running Quagga [53] (version 1.2.4-1), a software suite used for routing. *Open Shortest Path First* (OSPF) [54] is used in both $R1$ and $R2$ as the routing protocol. *Border Gateway Protocol* (BGP) [55] is used only on $R2$ to facilitate the networking requirements of the Kubernetes cluster. The configuration used can be found in Appendix A.1.

3.2.2 MinIO

The $S3$ node is running an instance of the MinIO [56] (version *RELEASE.2021-03-10T05-11-33Z*) application, which is an object storage implementation. The MinIO server is set up to contain two buckets (folders), each containing a number of files filled with random binary data. The first bucket contains 100 000 files, each 1 kB large. The second bucket contains 1000 files, each being 10 Mb.

MinIO implements the AWS S3 API [57] for object storage [56], which exposes an HTTP API for file operations. For example, the API exposes an endpoint for downloading an object.

3.2.3 Kubernetes Cluster

The Kubernetes cluster used in this thesis consists of nodes *S1* and *S2*, dotted in Figure 3.2. The Kubernetes distribution used in this thesis is k3s [58] (version *v1.19.5+k3s2*), with *S1* acting as the master node.

In order to perform the measurements Prometheus [34] (version 2.21.0) is used. It collects metrics from defined targets and stores them in a database, which can be queried later. The targets define which metrics they expose via HTTP APIs, and Prometheus scraped these metrics at a given interval. In this thesis, Prometheus is configured to scrape every 10th second.

By default, Prometheus is configured to scrape the services running in the Kubernetes cluster. Since the MinIO application is running in *S3*, it is not a part of the cluster. Therefore Prometheus was also configured to scrape the MinIO application using a custom configuration, which can be found in Appendix A.3.

3.2.4 Service Mesh

The service mesh implementation used in this thesis is Istio [8] (version 1.9.0), installed on the Kubernetes cluster. The reason why Istio was chosen is due to the fact that it supports extension to also include virtual machines in the service mesh, i.e. workloads not part of the Kubernetes cluster. It consists of services running in *S1*, *S2* and the MinIO application running in *S3*.

Since Istio is installed in the Kubernetes cluster, its control plane will also reside there. Furthermore, the services in the Kubernetes cluster that will be part of the mesh will also contain an Envoy [59] sidecar, which handles all inter-service communication.

In the virtual machines, which reside outside the Kubernetes cluster, some configuration needs to be done. The Envoy proxy needs to be present there, and all outgoing traffic needs to be routed to it.

The data plane of a service mesh is dependent on the control plane to receive relevant information, such as certificates and access control lists. Therefore, the control plane (running in *istiod* in figure 3.3), must be able to access the data plane (the Envoy proxies shown as purple hexagons in Figure 3.3). For the control plane to access the data plane in the virtual machines, a gateway must be present, which relays the traffic between the Kubernetes cluster and the outside components. The gateway also facilitates inter-service communication from the Kubernetes cluster to a virtual machine and vice versa.

Istio Configuration

The Istio data was configured with custom metrics in order to be used for monitoring on a server level. Istio does not support collecting node names for the requests, but the pod name from where the data is polled is collected by default. However, the goal of this thesis was to find errors on a specific packet route, and therefore the destination of the packets should also be monitored. This was done through customizable metrics in Istio, which allowed for adding

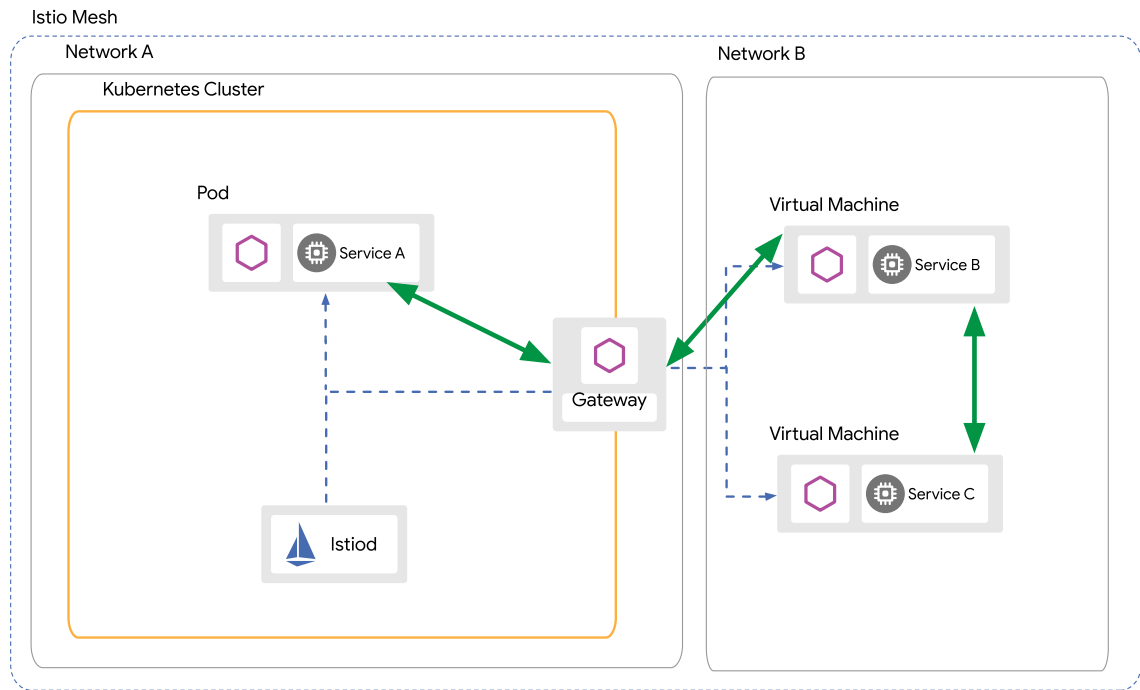


Figure 3.3: Figure showing a multi-network Istio setup. Source: <https://istio.io/v1.9/docs/ops/deployment/vm-architecture/>

metrics for upstream and downstream pod names. The upstream and downstream pod metrics could then be used to see the destination of a packet. When monitoring a request from the destination, the downstream pod name is the origin of the request, and when monitoring from the source, the upstream pod name is the destination. The necessary configuration for the customized metrics can be seen in Appendix A.2.

The pod names are unique to a specific server and can be translated to server names by a translation table created as a recording rule in the Prometheus configuration. The recording rules can be seen in Appendix A.3. The translation table was used to change the pod-names to server names in the Prometheus queries, which could then be used to group by source and destination servers for the desired data.

VirtualService is a feature provided by Istio which allows for traffic control configuration, such as setting a request timeout on a specific route [60]. In the experiments, a timeout was set to 15s for all traffic to the MinIO server to detect packet loss. When a packet has not received a response within 15s, the proxy will consider the packet as lost and create a response with a timeout flag (*UT*) set. It is this flag that is used to detect whether a packet has been lost or not.

3.3 Experiments

The following experiments were conducted in order to answer **RQ2** and **RQ3**. The VMs will be referred to as presented in Figure 3.2 in the following subsections.

Network faults were emulated in the test setup using the built-in traffic control tool in

Linux, *tc* [61] together with the network emulation tool *netem* [62]. *Tc* allows for altering the Linux kernel's network-packet scheduler for a specified interface which can be used to emulate network limitations. *Netem* is an enhancement of *tc* which simplifies certain network limitations such as packet loss and delay. This made it possible to emulate known network faults at specified interfaces on any VM, which are reproducible and controllable.

A Kubernetes job was created to simulate monitorable traffic using Rclone [63] with the *copy* command to copy all data from a specified bucket on the MinIO server. A Kubernetes job functions as a service but with a specific task that will be completed; Kubernetes starts a container running the job and terminates the container upon completion. This allowed the Istio sidecar to be easily deployed together with the job using the automatic sidecar injection. The transferred data was written to RAM in order to mitigate any possible performance impact due to disk write speed or background tasks reading or writing to the disk. The job would be running in the Kubernetes cluster, and the traffic would be routed through *R1* and *R2*. The Kubernetes job specification is provided in Appendix C. Kubernetes was configured to schedule the job on *S2* during the experiments in order to simplify the data collection and resource management in the cluster.

The following experiments in Section 3.3.1 and 3.3.2 were run in ten iterations in order to create an average reading of the data. They were executed by copying both the buckets containing 1000 á 10MB files and the 100 000 á 1kB files.

3.3.1 Network Fault Experiment

The following experiment was created with the goal of answering **RQ2**. It aims to simulate a network fault occurring on a system under load which should be discoverable using the data provided by the experiments.

The test execution precedes according to the following schedule:

1. Apply the Kubernetes Rclone job
The job is applied with the Istio sidecar and copies the specified bucket from MinIO, simulating a data-intensive application.
2. Wait for the job to start transferring and record its start-time
The containers have a boot time, and Rclone performs checks before transferring data which needs to be considered. The start-time is recorded after the checks made by Rclone and when the transferring starts. The start-time is needed to fetch data from Prometheus within the relevant timespan.
3. Wait until 50% of the data has transferred, then apply emulated network fault and record the time when the fault was introduced
The waiting period is used to allow some transfer to happen in order to collect comparable data before and after network fault. The emulated network fault is applied to *R1*, and the time is recorded to use for analysis. The time of the fault introduction is recorded for later use when plotting the graphs.
4. Wait for the job to finish and record its end-time
End-time is recorded once the transfer is done. It is used to fetch data within the relevant timespan from Prometheus.

5. Fetch Istio data from Prometheus and save data to JSON file
The queries used to fetch the relevant Istio data from Prometheus can be seen in Appendix B.
6. Delete job from Kubernetes cluster and remove network fault
Used to automate the tests in order to run multiple iterations.

The values of network faults shown below were chosen based on the testing environment, typical network conditions, and input from the case company. The emulated network faults introduced in the experiment are the following:

1. **Latency**

300 ms latency is applied using the following command. The limit is the maximum packets allowed in the queue before dropping packets. This was set high in order to solely introduce latency and not packet loss.

```
tc qdisc add dev enp0s9 root netem limit 10000000 delay 300ms
```

2. **Bandwidth**

A 15 MB/s network bandwidth limit is applied using the following command:

```
tc qdisc add dev enp0s8 root tbf rate 15mbps burst 3mb limit
100000000
```

burst 3mb is the maximum allowed packet size, and *limit 100 000 000* is the maximum allowed packet count in the queue before dropping packets. These parameters were set high to solely focus on bandwidth and not impact performance in other ways.

3. **Packet loss**

10% packet loss is emulated using the following command:

```
tc qdisc add dev enp0s9 root netem loss 10%
```

Enp0s8 and enp0s9 are network interfaces on *R1*, where enp0s8 is the interface connecting *R1* to *R2* and enp0s9 is the interface connecting *R1* to *S3*. The experiment was automated using an Ansible playbook.

3.3.2 Application Performance Experiment

The following experiment was created in order to answer **RQ3**. The experiment aims to determine the performance impact of using a service mesh like Istio compared to not using a service mesh. It also aims to evaluate the performance impact by applying a proxy like Istio on a data-intensive application.

The test execution precedes according to a similar schedule as previously mentioned in Section 3.3.1:

1. Apply the Kubernetes Rclone job
2. Wait for the job to finish and save the Rclone log file. The Rclone job's logs include a total transfer time which is used to present the results.

3. Delete job from Kubernetes cluster

The Rclone job log reports a total elapsed time which is used for plotting and evaluation. All the tests were executed with and without Istio enabled on both the cluster and the standalone VM. The test was automated using an Ansible playbook.

Chapter 4

Results

This chapter will present the results from the experiments described in the previous chapter and the findings regarding what data Istio provides, which can aid in network monitoring, thus answering **RQ1**, **RQ2** and **RQ3**.

The results in the following sections are gathered by running the experiments and collecting metrics using Prometheus as the experiments run. The metrics are then retrieved from Prometheus and are filtered to show the relevant information. To analyze and visualize the graphs in this report, *Pandas* and *Matplotlib* were used.

4.1 Istio Metrics

By default, Istio exposes a set of metrics, which can be found in its documentation at [64]. These metrics are generated in the Envoy proxies (data plane) since all traffic goes through the proxies. Most of the metrics exposed by Istio are relevant in network monitoring, where the relevant metrics in this thesis are the total number of requests a proxy has processed and request duration, request size, and response size as a distribution. Besides the actual metrics, the exposed metrics also contain additional labels, which can be used to group the data. Some of the relevant labels for monitoring are:

1. **Reporter**
Shows which proxy the metric is taken from. It can be either *source* or *destination*, depending on if the proxy is a client or a server.
2. **Response code**
The resulting HTTP response code from the request.
3. **Response flag**
The resulting response flag for a request reported by the proxy. Shows the reason why a request failed, for example, due to a timeout or if the request was rate-limited.

4. Source app

The name of the application from where the request originates from.

5. Destination app

The name of the application where the request is directed to.

The observability/monitoring features of Istio are mainly focused on monitoring either a single or a set of services. Besides the default metrics, Istio also allows configuring the labels exposed by the Envoy proxies [65]. This opens up the possibility of adding additional labels, which aid in aggregating the metrics by node (required for network monitoring).

This makes it possible to add labels such as *upstream_peer* and *downstream_peer*, which can be used to note the destination and origin of a packet. An upstream connection is the service where the Envoy proxy is connecting to, while a downstream connection is the service where the Envoy is receiving a connection from.

Istio utilizes Prometheus to retrieve the metrics from the proxies at a given interval. As Prometheus scrapes the metrics for each proxy, it can be configured to attach labels based on the metadata of the proxy. One such example is the name of the pod where the proxy is running on.

4.2 Network Fault Discovery

This section aims to answer **RQ2** by showing the results of the experiments captured by the service mesh. The results are divided into subsections based on the three different metrics proposed in the research question.

The results shown in sections 4.2.1-4.2.3 are from the network fault experiment described in Section 3.3.1 and features graphs based on the Prometheus data from the queries in Appendix B. The graph shows the data captured from requests originating from the Rclone job (running on S2) with MinIO (running on S3) as the destination.

The experiments were executed ten times, and the relevant metrics were collected, along with the experiment's start- and end-time and the time the network fault was introduced. These collected metrics were then plotted on a graph, with each graph being centered around the time the network fault was introduced. A red vertical line shows this time in the graphs below. These metrics were then averaged, which are the lines shown in the graphs below. The shaded blue portion of the graphs shows the minimum and maximum values recorded from the ten runs.

4.2.1 Latency

In Figure 4.1, the experiment results are shown when the latency network fault of an added 300 ms delay on outgoing traffic from the Rclone job was introduced. The figures on the left show the experiments where 1000 files á 10MB (large files) were transferred, while the figures on the right show the experiments where 100 000 á 1kB files (small files) were transferred.

When the latency network fault was introduced, the transfer speed (bandwidth) for both file sizes is affected. For the large files (Figure 4.1a), the transfer speed drops to roughly 1/3 of its original speed, while for the small files (Figure 4.1b) it drops to roughly 1/8.

The latency graphs in Figure 4.1c and 4.1d also show a difference before and after the network fault was introduced. The 50th, 90th, and 99th percentile latencies for both file sizes increase consistently after the network fault is introduced. Relative to each other, the small files show a more consistent increase in latency with the added latency of 300 ms. One interesting difference between them is the spike in the beginning in Figure 4.1d, which is not present in Figure 4.1c.

The loss graphs in Figure 4.1e and 4.1f however, are different from each other. In large files, some loss (average 1%) occurs after the fault is introduced, which is consistent with the other large-file graphs. However, this fault lasts only for a small portion of time and goes

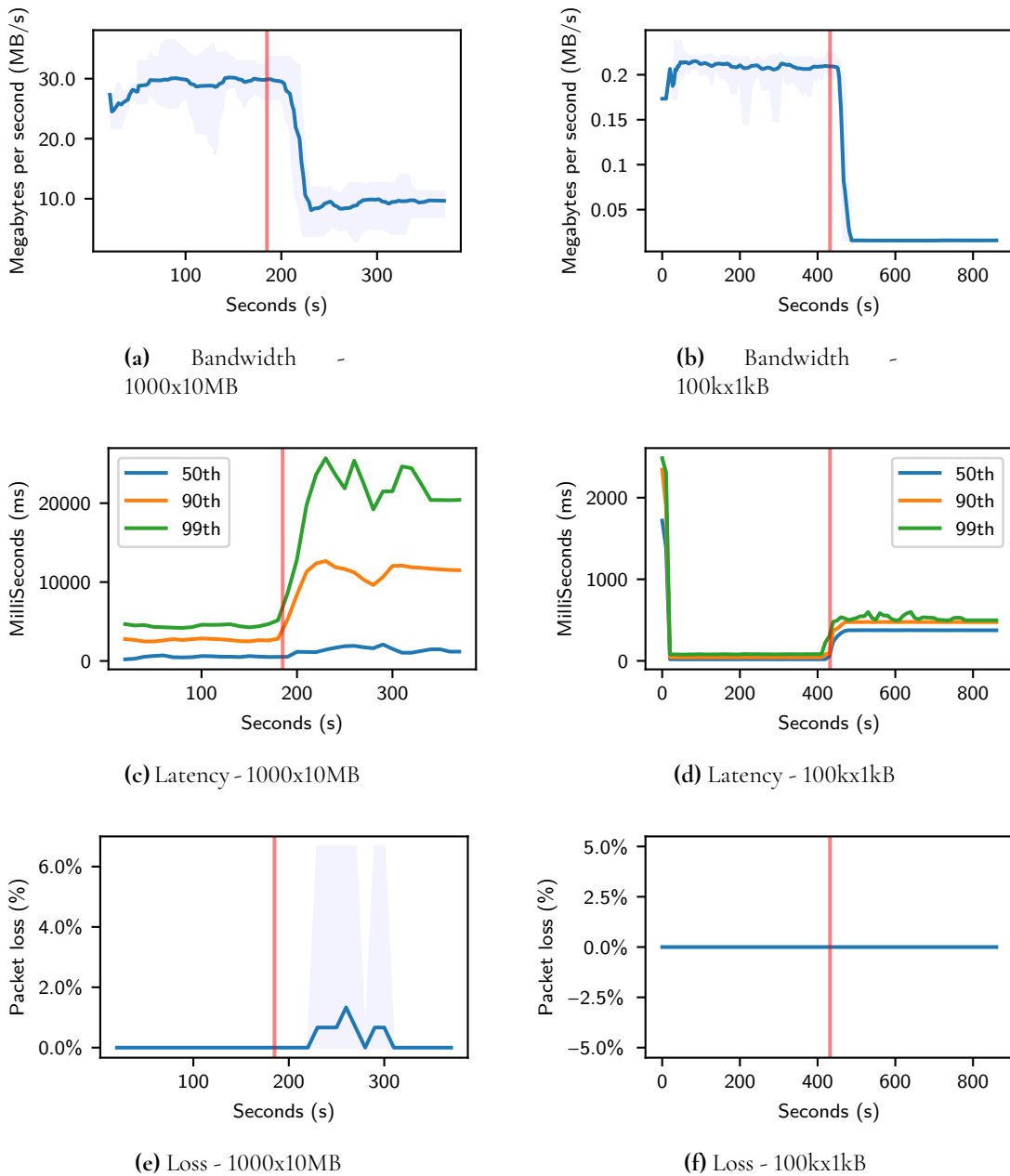


Figure 4.1: Latency of 300ms introduced at the red line.

back to zero towards the end of the experiment. For smaller files, on the other hand, no loss could be detected.

4.2.2 Bandwidth

In this experiment, the network fault introduced was a restriction in bandwidth to 15 MB/s on outgoing traffic from the MinIO server. This means that before fault introduction, files can be transferred as fast as possible (~ 30 MB/s), while after, the files can be transferred with a maximum of 15 MB/s. The experiment results are shown in Figure 4.2, where the left graphs are the transfer of large files, while the graphs on the right are the transfer of small files.

The bandwidth graphs in Figure 4.2a and 4.2b differ quite a lot from each other. The large file transfer in Figure 4.2a starts transferring with an average of ~ 30 MB/s, while after the fault is introduced, it drops to an average of around 15 MB/s. However, the small file transfer in Figure 4.2b maintains a constant average bandwidth of ~ 0.2 MB/s across the entire transfer, even after the bandwidth limit is introduced.

An introduction of bandwidth limit does not seem to affect the packet loss, as shown in Figure 4.2e and 4.2f. The packet loss is 0% in both graphs before and after the limit introduction.

4.2.3 Packet Loss

In this experiment, the network fault that was introduced was a packet loss of 10% on outgoing traffic from the Rclone job. The results of this experiment are shown in Figure 4.3.

The bandwidth graphs in Figure 4.3a and 4.3b are somewhat similar to each other. The large files transfer in Figure 4.3a shows no visible change after the fault was introduced. The small files transfer graph in Figure 4.3b on the other hand, shows a slight decrease in bandwidth from ~ 0.23 MB/s to slightly less than 0.20 MB/s.

The latency graphs in Figure 4.3c and 4.3d show similar results as the bandwidth graphs. For large files, the latency graphs remain unchanged before and after the fault introduction, across all percentiles. However, there is a considerable increase of the 99th percentile graph to around 400 ms for the small file transfer.

The loss graphs in Figure 4.3e and 4.3f show a loss of 0% for both small and large file transfers.

One interesting observation about the large file graphs (Figure 4.3a and 4.3c) is the difference in the beginning of the graphs. The bandwidth is almost zero, while the latency spikes in the beginning.

4.3 Application Performance Impact

This section aims to answer **RQ3** by showing how the service mesh proxy affects a data-intensive application. The graphs and data shown in this section are the results of the experiments described in Section 3.3.2. The tests are run ten times, and the data shown in this section shows the total time it takes Rclone to transfer the files.

First, the results of transferring small files (100 000 á 1kB) will be shown, then the results when transferring the large files (1000 á 10MB) will be shown.

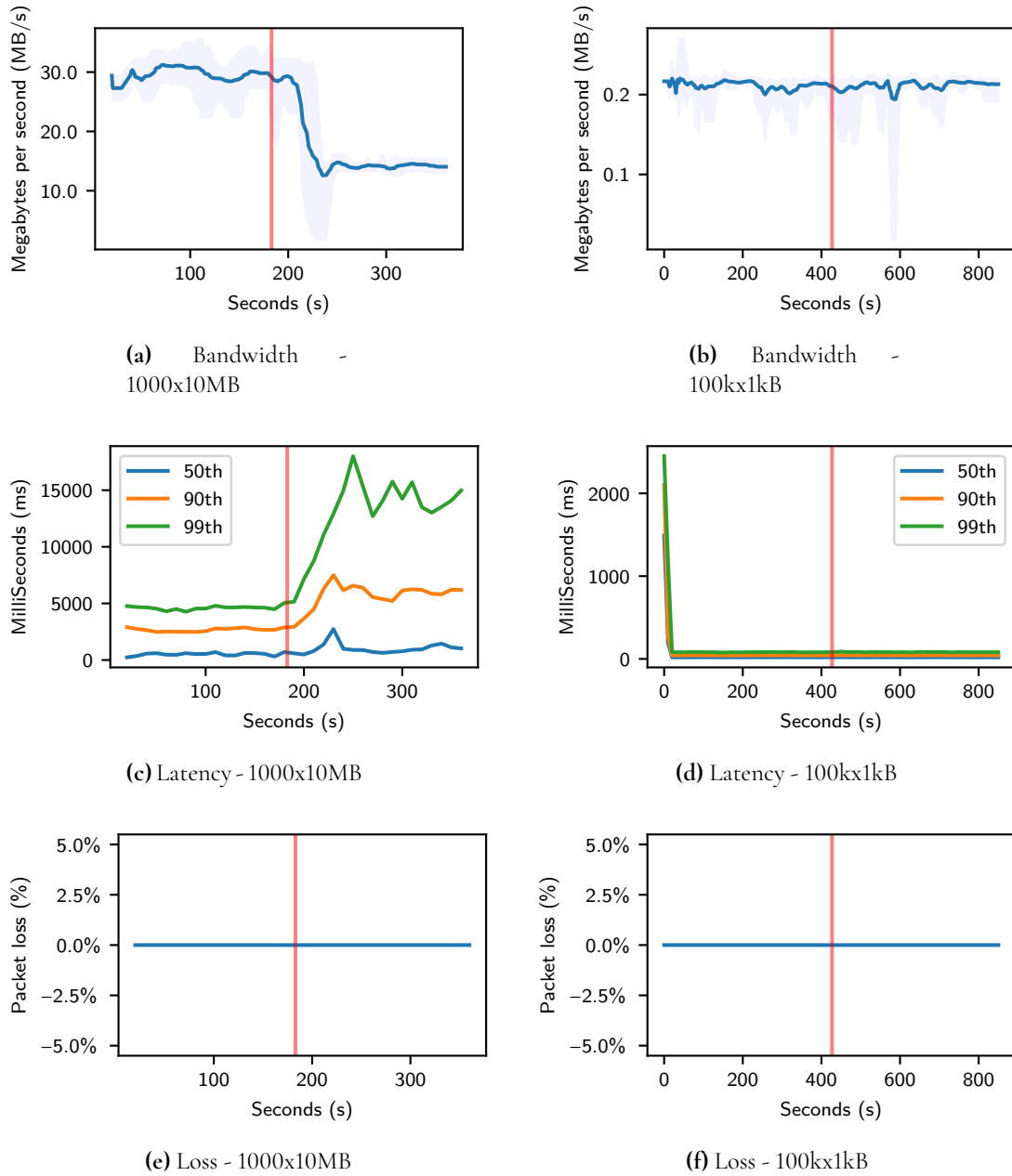


Figure 4.2: Bandwidth limit of 15 MB/s introduced at the red line.

4.3.1 Small Files Transfer

Figure 4.4 shows the median transfer time in seconds for transferring the small files. The left bar plots show the median total transfer time with one parallel transfer, while the bar plots on the right show the median total transfer time with 20 parallel transfers.

Table 4.1 shows the mean, median, 25th percentile, 75th percentile, the minimum, and the maximum time the transfer took based on the ten runs that were done during the experiments. It shows the values for both one transfer and 20 transfers.

As can be seen in Figure 4.4 and Table 4.1, the transfers took longer time on average when

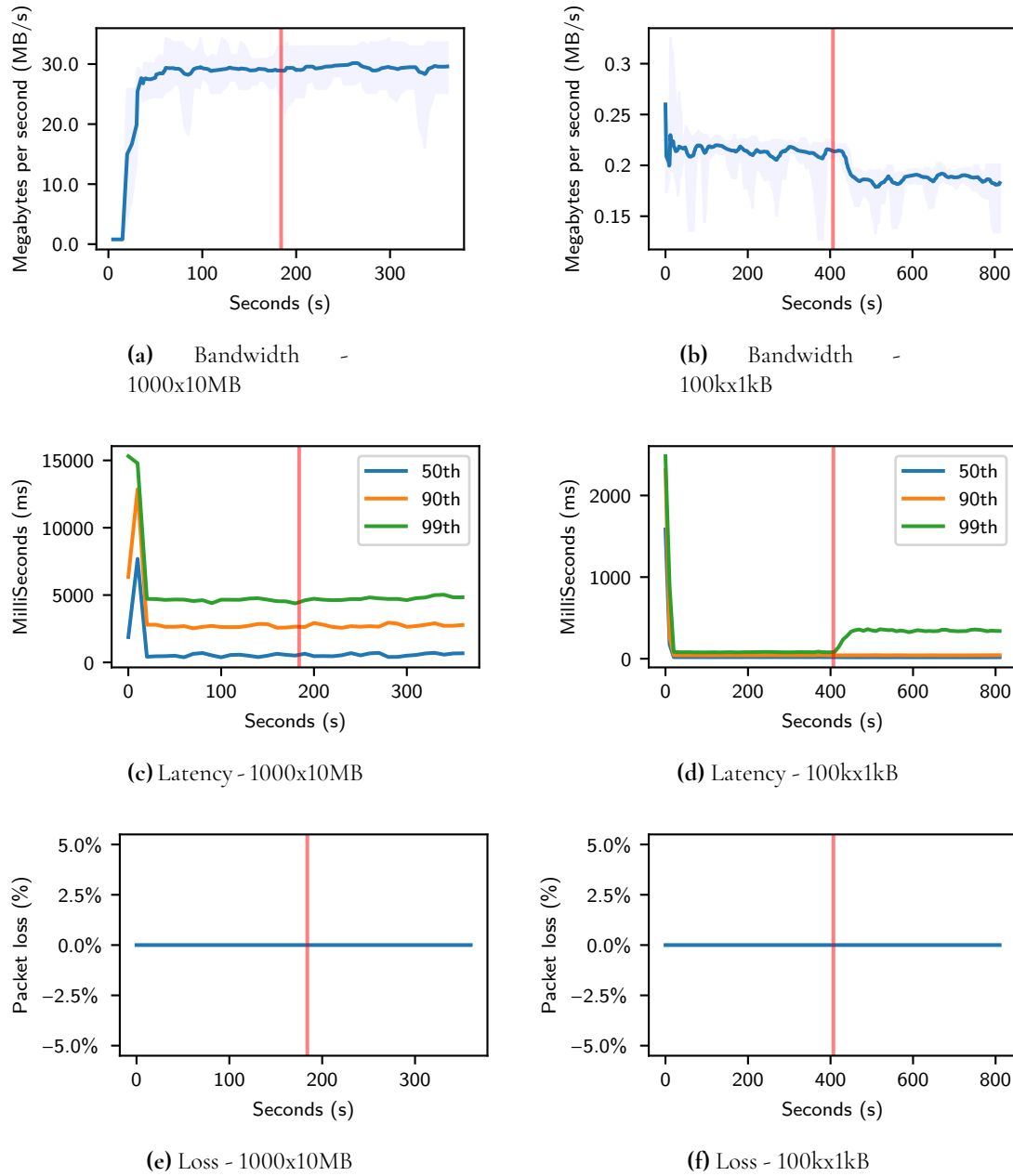


Figure 4.3: Packet loss of 10% introduced at the red line.

Istio was used, compared to without. Using one transfer, the transfer without Istio took on average 25m03s, while using Istio took on average 34m17s. That is a 36% increase in transfer time using Istio. For 20 transfers, the difference is smaller, 17m22s without Istio and 21m44s with Istio (25% increase).

The difference between the number of transfers is also visible in the data. Transferring using one parallel transfer takes on average longer time than using 20 parallel transfers. This difference is present regardless if Istio is being used or not.

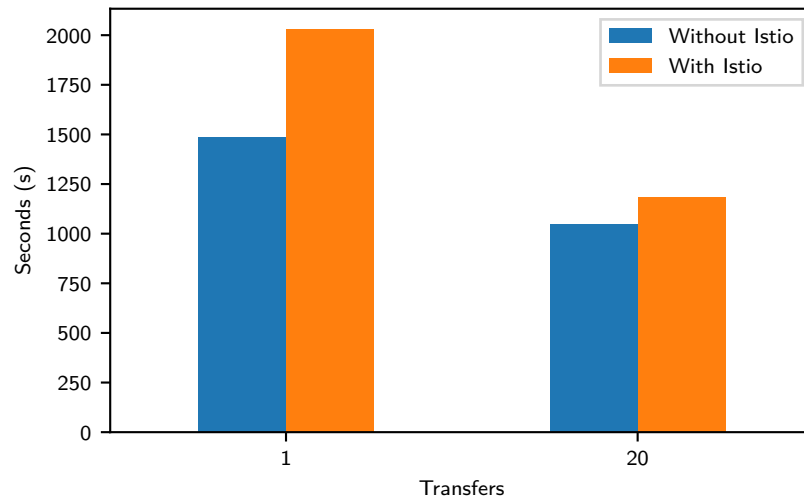


Figure 4.4: Median transfer time for 100k 1kB files from ten runs

Metric	Mean	Median	25th perc.	75th perc.	Min	Max
1 transfer						
<i>Without Istio</i>	25m03s	24m46s	24m36s	25m04s	24m19s	26m29s
<i>With Istio</i>	34m17s (+36.9%)	33m51s (+36.6%)	33m35s	34m02s	33m19s	37m01s
20 transfers						
<i>Without Istio</i>	17m22s	17m27s	17m11s	17m37s	16m40s	17m46s
<i>With Istio</i>	21m44s (+25%)	19m45s (+13%)	19m31s	20m09s	19m15s	39m30s

Table 4.1: The experiments results when transferring 100 000 files á 1kB ten times. Lower is better.

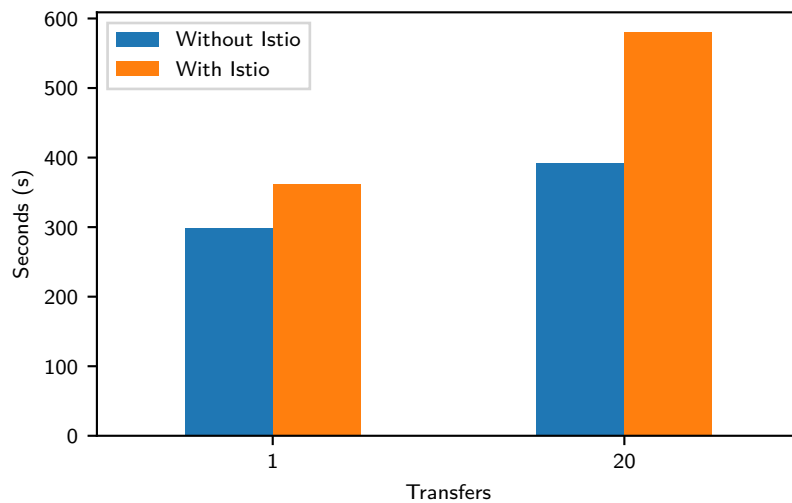


Figure 4.5: Median transfer time for 1 000 10MB files from ten runs

Metric	Mean	Median	25th perc.	75th perc.	Min	Max
1 transfer						
<i>Without Istio</i>	4m57s	4m58s	4m56s	5m00s	4m43s	5m10s
<i>With Istio</i>	6m04s (+24%)	6m01s (+21%)	6m00s	6m03s	6m00s	6m14s
20 transfers						
<i>Without Istio</i>	6m40s	6m32s	6m27s	6m34s	6m12s	8m21s
<i>With Istio</i>	9m36s (+43%)	9m39s (+47%)	9m37s	9m43s	8m46s	10m05s

Table 4.2: The experiments results when transferring 1000 files á 10MB ten times. Lower is better.

4.3.2 Large Files Transfer

Figure 4.5 shows the median transfer time in seconds for transferring the small files. As for small files, the left bar plots show the median transfer time the transfer took with one parallel transfer, and the right bar plots show the time using 20 parallel transfers.

Table 4.2 shows the mean, median, 25th & 75th percentile and the minimum & maximum time the transfers took based on the ten experiment runs. The values are shown for one- and 20 parallel transfers.

Based on Figure 4.5, the transfers for both one and 20 parallel transfers took less time when Istio was not used compared to when it was used. For one transfer, it took on average 4m57s to transfer the files without Istio and 6m04s when Istio was used (24% longer time). For 20 transfers, this difference was larger, 6m40s when Istio was not used and 9m36s when it was used (43% longer time).

Comparing the data for one and 20 parallel transfers, it took on average less time to transfer the data with one parallel transfer. When Istio was not used, this difference is small, while with Istio there is a bigger difference.

Chapter 5

Discussion

In this chapter, the results from the experiments are analyzed, with focus on answering this thesis' research questions. The outcome of this thesis is discussed, together with limitations provided by this solution, and some future work in which our thesis can be used as a foundation.

5.1 Istio Metrics

With the monitoring possibilities presented in Section 4.1, it is theoretically possible to create a solution for monitoring a network using a service mesh like Istio, thus answering **RQ1**. Several default metrics provided by Istio are related to network monitoring, and looking at the default configuration, it seems possible to succeed without any custom configuration. Using the different data provided and grouping by *reporter*, *source app*, *destination app*, etc., it seems like it would be possible to find a solution out of the box. However, the problem lies in the observability features of Istio are based on monitoring a service or a group of services, not the network. These services can be placed on arbitrary nodes, making network monitoring difficult using the default metrics provided by Istio.

The default labels *source app* and *destination app* (see Section 4.1) could be used for network monitoring; however, that requires a one-to-one mapping between a service and the instance that serves it. When there is a one-to-one mapping, the instance will be scheduled on one node, which makes aggregating by node possible. When there is a one-to-many mapping, that approach is no longer useful as instances can be scheduled in arbitrary nodes, making aggregating impossible.

To be able to aggregate by node while maintaining the one-to-many mapping between a service and its instances, Prometheus can be instructed to attach the pod name of the instance it's running on to the collected metrics. Since the pod's name is unique, and Kubernetes stores on which node each pod is running in, a proxy (and subsequently the metrics it provides) can now be tied to a specific node, making node aggregation of metrics possible. By combining

the pod name with the *reporter* label (provided by the proxy) equal to *source*, we can show metrics based on the request's origin.

In order to monitor the network between the nodes, besides the origin of the request, its destination is also needed. Istio, by default, does not provide such a metric. However, it can be added by configuring Istio to include the *upstream_peer.name* and *downstream_peer.name*. These metrics show the destination pod's name and the origin pod's name, respectively. Since the node where a certain pod is running is known, the metrics can be aggregated to show traffic information between all nodes. The configuration can be seen in Appendix A.2.

Even though there is a connection between pods and the nodes they are running on, it is not straightforward to retrieve them directly from Prometheus. To solve this problem, Prometheus was configured to add a *node_name* label to metrics retrieved from Envoy proxies which show the node the pod is running on. This was possible because each pod contains information about what node it is running on (through the *__meta_kubernetes_pod_node_name* label).

Furthermore, translation tables were also created to translate the *upstream_peer.name* and *downstream_peer.name* to their node mapping. This was achieved using recording rules in Prometheus, which create a custom timeseries based on an expression. This timeseries was created by modifying a timeseries from the Kubernetes metrics called *container_memory_rss*, containing both the pod name and the node name. See Appendix A.2 for the full configuration used.

By adding the labels above, it is possible to aggregate all metrics exposed by Istio by the network path between nodes. The relevant default metrics exposed by Istio noted in Section 4.1 were deemed enough to create experiments to answer **RQ2**. Combining the request- and response-size metrics would make it possible to calculate and monitor the bandwidth used in the paths between the nodes over time. By using the request duration metric, the round-trip latency can be calculated. Packet loss can be calculated by the ratio between the failed requests (calculated using the response flag that Envoy attaches to failed requests and the HTTP response code) and the total number of requests.

Using these collected metrics could be a viable solution to answering the questions in this thesis. Latency would be discovered easily using this approach since an increase in latency would affect all pods sharing the same line in the same way; however, bandwidth and loss would not be discovered as accurately. Since bandwidth is shared among all traffic-generators on a line, the full bandwidth capacity would only be monitored by adding all the traffic between pods on the line, which would also monitor a limitation of the bandwidth. Loss is also unpredictable and could manifest itself as only being applied on one of the pods due to its randomness.

5.2 Network Fault Monitoring

The network fault experiments described in Section 3.3.1 yielded results that made it possible to discover some of the errors in **RQ2**. Where in the link the faults were introduced was not possible to detect using a service mesh. To do so, a proxy would have to be added in front of all routers, which could divide each network link monitoring segments into smaller parts. This however requires access to the network infrastructure, which defeats the purpose of this thesis since this thesis assumes no such access.

Rclone sends a request to MinIO to create a checksum for the files to transfer before starting the actual transfer. This takes some time to create, especially for the 100 000 1kB files, and this time is not included in the graphs in Section 4.2. Some of the graphs show a spike in latency or low bandwidth initially, which can be explained as some data being captured from the checksum creation time and is not relevant to the experiment.

Prometheus was configured to have a scrape-interval of 10s during the experiments, which is a very short time compared to the default setting of one minute. This was set low in order to collect more data during the relatively short experiment times to get an accurate reading. In a real implementation, the proposed metrics would be monitored over a longer timespan and would therefore not have much impact on the network monitoring solution.

The traffic generated by Prometheus in order to scrape the services is sent on the same network as the Rclone traffic. Prometheus' traffic is therefore also affected by the network errors introduced, which can affect the resulting metrics. A separate virtual network link could be created between the nodes to handle the Prometheus traffic to remove this variable, but this wouldn't represent a real-life scenario and was therefore deemed irrelevant.

5.2.1 Latency

When an error with a network delay is applied, an increasing latency can be seen in the graphs in Figure 4.1c and 4.1d, showing that a latency error can be detected through the solution proposed in this thesis. However, how effective this solution is, depends on the data transferred and service performance.

The latency measured by Istio is the round-trip duration of a request, i.e., the total time from that a request has reached the Envoy proxy at the origin until a response has returned to the proxy. This includes the time it takes for the destination service to process the request. The desired latency measurement from the aspect of monitoring the network would be the round-trip time for a request without the processing time at the destination service. A solution for this should theoretically be possible to create using Istio. The processing time for a destination service can be accessed by monitoring the request duration milliseconds metric exposed by Istio where *reporter* equals *destination*. With this metric and the previous round-trip metric, it can be seen if a spike in latency is caused by the service or the network. Combining the two measurements by subtracting the processing time measured from the destination from the round-trip latency would yield the desired metrics. However, this was not solved in this thesis; attempts were made, but doing so would require equal groups in the data to subtract. With the upstream and downstream node names being different from the source and destination of the request, subtracting equal groups was not possible. This would need further translation tables in order to match the label, which was deemed as future work for this thesis.

The introduced error of an increased latency of 300 ms should manifest itself as an increased overall latency of 300 ms in Figure 4.1c and 4.1d, for all the percentiles, since it is applied to all requests. The increasing latency is mainly visible in Figure 4.1d when copying the bucket containing 100 000 1kB files. This shows that a network error manifesting itself as an increased latency can be discovered with the configuration used in this thesis. However, in Figure 4.1c showing the same metric for copying the bucket containing 1 000 10MB files, the 50th percentile latency increase is difficult to see. This is due to the fact that latency is measured as the round-trip time for a request, from the time it was sent until a response is

received. The consequence of this is that the latency is dependent on the file size; a larger file takes longer to transfer and therefore yields a higher latency. This results in the relatively small increase of 300 ms, which is difficult to identify.

However, in Figure 4.1c, the latency for the 90th and 99th percentile of the requests increases dramatically. The reason behind this is unknown, but a theory is that the limit of the packet queue in the queuing discipline applied when introducing the delay overflows, and packets get dropped, resulting in a high latency for more than 10% of the packets.

When latency is introduced, the bandwidth also decreases, as can be seen in Figure 4.1a and 4.1b. This is due to the delay occurring between each request to transfer a file. Therefore a larger difference is also seen in Figure 4.2b due to more requests are being sent. The maximum total bandwidth for a network link does not get limited by a latency increase, but the services can utilize less bandwidth due to the extra time when waiting for a response.

5.2.2 Bandwidth

A change in the maximum bandwidth limit was discovered through the recorded data showing bandwidth usage between nodes. However, detecting a network fault in the form of limited bandwidth requires the bandwidth to be used at full capacity in the solution proposed in this thesis. This means that it can be difficult to differentiate when there is an error in the network or when the current bandwidth usage is low.

The results show a clear decrease in bandwidth in Figure 4.2a, down to the specified limit of 15 MB/s shortly after the error is introduced. This shows errors resulting in a limited bandwidth can be detected using the solution proposed in this thesis. This change is noticeable due to the bandwidth being the limit of the transfer speed when transferring the 10MB files. In Figure 4.2b showing the bandwidth when transferring the small files, the change is not visible. This is due to the network bandwidth limit not being fully used, the theory is that the transfer speed is limited by either CPU power, storage device speeds, throughput limitations, or the overhead of opening and closing files in either MinIO or Rclone.

When the bandwidth limit is introduced at the time of transferring the 1 000 10MB files, an increase in latency can also be seen for the 90th percentile and up in Figure 4.2c. This is possibly due to the packet queue for the interface overflowing and dropping packets; the response packets are larger for the 10MB files and take up more space to store in the queue than the 1kB files, which is why it does not show the same result in Figure 4.2d.

5.2.3 Packet Loss

The work conducted in this thesis did not find a way to monitor packet loss through Istio reliably. Istio, by default, does not save information about unfinished requests (requests that have been sent but have not been acknowledged by the receiver). An attempt to solve this problem was using a VirtualService with the purpose of discovering when a response was taking too long. When a response exceeds the timeout in the VirtualService, it creates a response with a timeout response flag set, which can be monitored. However, this workaround did not yield successful results either.

The first approach made to measure loss through Istio was to compare requests with a missing response to all requests. Through several attempts, a conclusion was drawn that there was no way to configure Istio to monitor non-completed requests. The next approach was to

add the VirtualService to introduce a timeout to packets. This timeout results in timed-out packets receiving a response flag of *UT*, which is monitored in the graphs depicting the loss. The approach partially worked; in Figure 4.1e, the latency of the 99th percentile of packets marginally exceeds 15s. At the same instance, spikes showing loss can be seen in Figure 4.1e. This is an undesired behavior but shows the VirtualService generating responses with timeout flags. However, in Figure 4.3e and 4.3f, where loss is should have been detected, the timeout is not reached, and no timeout responses were generated.

The problem seems to lie in that retransmission happens at some point in the request's life-cycle and that Istio logs a retransmitted request as one request. Logs from Rclone don't show any signs of errors which indicates that Rclone is not aware of the loss occurring, and Rclone has a limit of 60s before retransmitting. Several variables may be responsible for retransmission but most probably lies in the TCP protocol, which is handled in the Linux kernel.

In Figure 4.3b, a decreased bandwidth usage can be seen when transferring the 100 000 1kB files after the loss was applied. The same behavior can not be seen for the 1 000 10MB files. The theory of why this happens is that more requests are being sent per second when transferring the small files, which implies that more packets are lost and therefore have a larger impact on the bandwidth. In Figure 4.3d, an increased latency is noted for the 99th percentile of requests when the loss has been applied. This is probably the total latency for the lost packets and further indicates some form of retransmission happening. Theoretically, this should also be seen for the 90th percentile of the requests since 10% of the packets are lost, and there are uncertainties around why this isn't the case.

The loss was applied to outgoing packets from *R1* to *S3*, which in this case, was dropping the requests. And it is believed that this is the reason why an increased latency cannot be seen in Figure 4.3c, since the time for a packet to reach *R1* is so small compared to the total duration of a request together with retransmission has a relatively small impact on the total latency. To confirm this, the experiment was executed again when applying the loss to the outgoing packets from *R1* to *R2*. The results from this experiment can be seen in the figures in Appendix D which shows a loss of the expected value, 10%, when transferring the large files. The latency can also be seen reaching more than 15s that causes the timeout flag to be set, which is why the correct loss is registered. The increased latency difference between Figure 4.3c and Figure D.1a from Appendix D is due to the processing delay before the packets are retransmitted since the packets are lost on the response side.

5.3 Istio Performance Impact

In Section 4.3 the results of the performance experiments were presented, with the goal of answering **RQ3**. The results show a distinct performance impact when Istio is used. In this section, the results of that experiment will be discussed further.

5.3.1 Performance Analysis

When Istio is present, all outgoing traffic from Rclone and MinIO is forced to pass through the Envoy proxy, which routes it correctly. When traffic passes through the proxy, it is processed, and metrics are collected. This process done in the proxy requires some computing

power, which means that it will take some time (added latency) for the proxy to process the traffic. The time it takes for the proxy to process this data has been documented previously [66] by the Istio project maintainers. According to [66], two proxies (one on the client and one on the server-side) add 1.7 ms and 2.69 ms latency for each request to the 90th and 99th percentile, respectively. The fact that the Envoy proxies lead to higher latency explains the increased processing time in our experiments when using Istio as opposed to not using it.

In Figure 4.4 (small file transfer), the impact of having Istio applied is much larger when using one transfer than 20 transfers. When transferring using one parallel transfer, each file needs to be read from the filesystem, processed, then sent through the network. Since one transfer is used, this overhead (I/O) is performed on every initiated file transfer. When using 20 parallel transfers, the I/O overhead is most likely spread out through the 20 transfers, thus minimizing its effect.

On the other hand, when transferring large files (Figure 4.5), Istio's overhead is larger when 20 parallel transfers were used compared to one. This difference likely comes from the file sizes used in the experiments. For the small files, a 1kB file fits into a single packet, while a 10MB file needs to split into multiple smaller packets in order to be transferred over the network. This leads to more packets having to pass through the proxy, which is likely the bottleneck in this case. The bottleneck could, for example, be that the proxy cannot handle that level of packet throughput, leading to a slowdown. This theory is supported by [66], which lists different factors that affect the data plane performance, for example, request rate, request size, and the number of client connections.

An unexpected difference between the results for small and large files is that for small files, in Figure 4.4, a transfer count of 20 results in a shorter transfer time compared to one transfer count. For the large files, in Figure 4.5, this is not the case, as 20 transfers result in a longer transfer time. This difference lies in the different bottlenecks in the two scenarios. For small files, the bottleneck is I/O, i.e., the time it takes MinIO to retrieve the file from the storage and to start transferring it. Using more parallel transfers, this bottleneck becomes less significant since the files can now be read in parallel. For large files, however, the bottleneck is likely the network bandwidth. Using more parallel transfers does not affect the network bandwidth. Instead, more time is spent giving each transfer an equal timeshare on the network, which induces an overhead.

5.3.2 Envoy Throughput Limitations

The performance experiment to evaluate Istio's impact in this thesis is mainly constructed to test the possible delay added for a packet passing through the Envoy proxy. With the simulated network setup used in this thesis, the maximum transfer speed was limited to ~30 MB/s which can be seen in the bandwidth graphs from the network experiment results prior to the applied error. In a real application using non-simulated hardware, the bandwidth can reach speeds over 10Gb/s. There is a possibility that the Envoy proxy can't handle recording metrics at such speeds and therefore has further impact on performance and resource usage. This is also dependant on the CPU resources allocated to the proxy. As discussed in Section 5.3.1, there is a possibility that this limit was reached when transferring the large files with 20 parallel transfers; however, this should be further tested.

5.4 Future Work

Istio has been studied in this thesis as a representation of the service-mesh landscape. Even though the results in this thesis successfully used Istio for network monitoring, other service-mesh solutions could be more usable and produce better results in this context. Therefore studying other service-mesh implementations and comparing them to this thesis' results are left as future work.

A delimitation in this study is focusing on data-intensive applications, modeled by Rclone and MinIO, to generate the network traffic in this thesis. The real world is more complex, with services that have different network patterns and systems that consist of multiple running services. For example, a Kubernetes cluster usually runs multiple services. Therefore a study of the results of this thesis on a real-world scenario with multiple dissimilar services could have a greater benefit and is left for future work.

One thing the results of this thesis showed was that packet loss could not be identified based on the current setup of the experiment. A more in-depth study into this problem and possible solutions could yield more successful results regarding packet loss.

The experiments conducted in this thesis to evaluate the performance impact on the applications when using Istio mainly focused on the possible impact per request sent for the specific implementation in this thesis. One of the areas to further research was presented in Section 5.3.2 and several other factors are listed in the Istio documentation [66] which can affect the performance of the data plane. Some of the factors listed are the number of clients connected, request rate, CPU cores, protocol, etc. How these factors affect the performance impact is not studied and left as future work.

Chapter 6

Conclusion

At the case company, efforts have been made in order to migrate some of the services running on virtual machines to Kubernetes clusters. With the move to Kubernetes, some possible future problems have been identified with regard to network performance. The possible issues identified are related to monitoring network performance from the standpoint of a service owner. Three monitoring metrics have been prioritized in this thesis, bandwidth, latency, and packet loss.

The proposed solution for the described problem was using a service mesh to monitor the network. The service mesh solution Istio was chosen since it supports expanding the service mesh to include virtual machines, thus enabling monitoring services both inside and outside of a Kubernetes cluster.

Istio by default supports monitoring this thesis' prioritized metrics through monitoring request duration (latency), request/response bytes (bandwidth), and total requests which contain a specific response flag (packet loss). However, the main purpose of a service mesh from an observability aspect is to monitor service performance, not network performance.

Through custom configurations of Istio and Prometheus, it was possible to aggregate the metrics exposed by Istio based on a request's origin node and its destination node. By doing this, inter-node communication could be monitored, thus answering **RQ1**.

In order to answer **RQ2**, an experiment was created in order to test whether the metrics mentioned above could be used to identify network faults. This was achieved by introducing known network faults such as bandwidth limitations, increased latency- and loss while transferring data from a node to another in order to evaluate if the error could be identified based on the collected data from Istio.

The results from the experiments in this thesis show that an increased latency and decreased bandwidth can be identified using the data provided by Istio. In this thesis, however, no reliable way to detect an increased packet loss was discovered. Regarding the increased latency detection, the reliability depends on the application used as the measurements are based on the total duration of a request. If the duration of the request is high, a small latency increase due to the network will be less noticeable in the data. Regarding the decrease in

bandwidth, the detection requires that the network bandwidth is fully utilized by the applications. Finally, an increase in packet loss could not be successfully identified. Envoy should theoretically be able to monitor lost packets, and there may be ways to configure it to log and export these metrics manually.

In order to answer **RQ3**, an experiment was created to measure the performance impact on applications when using Istio. The experiment results show that Istio has a distinct negative performance impact of an increased average total transfer time ranging from 24-43% across all of the performed tests.

Although the proposed solution in this thesis did not fully solve the problems at the case company, the research questions were answered, and the implementation used in this thesis can be used in a real-world scenario. The intended use-case of a service mesh like Istio is not related to inter-node network monitoring, however, the observability features of it could possibly be extended to facilitate the use-case in this thesis better.

References

- [1] Robert L. Glass. *Software Engineering: Facts and Fallacies*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321117425.
- [2] Betsy Beyer et al. *The Site Reliability Workbook: Practical Ways to Implement SRE*. 1st. O'Reilly Media, Inc., 2018. ISBN: 1492029505.
- [3] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. 1st. O'Reilly Media, Feb. 2015, p. 280. ISBN: 978-1491950357.
- [4] Fawaz Paraiso et al. "Model-Driven Management of Docker Containers". In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 2016, pp. 718–725. DOI: 10.1109/CLOUD.2016.0100.
- [5] *Kubernetes*. URL: <https://kubernetes.io/> (visited on 03/18/2021).
- [6] Alex Williams et al. *The state of the Kubernetes ecosystem*. en, p. 192.
- [7] Arnon Rotem-Gal-Oz. "Fallacies of Distributed Computing Explained". In: *Doctor Dobbs Journal* (Jan. 2008).
- [8] *Istio*. URL: <https://istio.io/> (visited on 03/11/2021).
- [9] Wubin Li et al. "Service Mesh: Challenges, State of the Art, and Future Research Opportunities". In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2019, pp. 122–1225. DOI: 10.1109/SOSE.2019.00026.
- [10] *Axis Communications AB*. URL: <https://axis.com/> (visited on 03/18/2021).
- [11] Jacyana Suassuna Nunes et al. "Deploying the observability of the SigSaude system using service mesh". In: *2020 20th International Conference on Computational Science and Its Applications (ICCSA)*. 2020, pp. 9–15. DOI: 10.1109/ICCSA50381.2020.00014.
- [12] Meina Song, Qingyang Liu, and Haihong E. "A Mirco-Service Tracing System Based on Istio and Kubernetes". In: *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*. 2019, pp. 613–616. DOI: 10.1109/ICSESS47205.2019.9040783.
- [13] Swen Nijboer. "Traffic analysis of a service mesh". MA thesis. 2021.

- [14] Ronja Mara Jösch. “Managing Microservices with a Service Mesh : An implementation of a service mesh with Kubernetes and Istio”. MA thesis. KTH, School of Electrical Engineering and Computer Science (EECS), 2020, p. 90.
- [15] Lars Larsson et al. “Impact of etcd deployment on Kubernetes, Istio, and application performance”. In: *Software: Practice and Experience* 50.10 (2020), pp. 1986–2007. DOI: 10.1002/spe.2885.
- [16] Ahmed M. El-Shamy et al. “Anomaly Detection and Bottleneck Identification of The Distributed Application in Cloud Data Center using Software–Defined Networking”. In: *Egyptian Informatics Journal* (2021). ISSN: 1110-8665. DOI: 10.1016/j.eij.2021.01.001.
- [17] *Google Cloud for Data Center Professionals: Networking*. URL: <https://cloud.google.com/docs/compare/data-centers/networking> (visited on 03/12/2021).
- [18] Cloud Native Foundation. *CNCF Survey Report 2020*. 2020. URL: https://www.cncf.io/wp-content/uploads/2020/12/CNCF_Survey_Report_2020.pdf (visited on 03/12/2021).
- [19] *Istio / Observability*. URL: <https://istio.io/v1.9/docs/concepts/observability/> (visited on 03/22/2021).
- [20] Fábio Pina et al. “Nonintrusive Monitoring of Microservice-Based Systems”. In: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. 2018, pp. 1–8. DOI: 10.1109/NCA.2018.8548311.
- [21] James Lewis and Martin Fowler. *Microservices - a definition of this new architectural term*. Mar. 2014. URL: <https://martinfowler.com/articles/microservices.html> (visited on 03/01/2021).
- [22] Miika Kalske, Niko Mäitalo, and Tommi Mikkonen. “Challenges When Moving from Monolith to Microservice Architecture”. In: *Current Trends in Web Engineering*. Ed. by Irene Garrigós and Manuel Wimmer. Cham: Springer International Publishing, 2018, pp. 32–47. ISBN: 978-3-319-74433-9.
- [23] Martin Fowler. *Microservice Trade-Offs*. July 2015. URL: <https://martinfowler.com/articles/microservice-trade-offs.html> (visited on 03/02/2021).
- [24] *Hypervisors | IBM*. URL: <https://www.ibm.com/cloud/learn/hypervisors> (visited on 03/22/2021).
- [25] Flexera. *RightScale 2019 State of the Cloud Report*. 2019. URL: <https://resources.flexera.com/web/media/documents/rightscale-2019-state-of-the-cloud-report-from-flexera.pdf> (visited on 03/18/2021).
- [26] *Swarm mode overview | Docker Documentation*. URL: <https://docs.docker.com/engine/swarm/> (visited on 03/18/2021).
- [27] *Nomad by HashiCorp*. URL: <https://www.nomadproject.io/> (visited on 03/18/2021).
- [28] *Apache Mesos*. URL: <http://mesos.apache.org/> (visited on 03/18/2021).
- [29] Betsy Beyer et al. *Site Reliability Engineering: How Google Runs Production Systems*. 1st. O’Reilly Media, Inc., 2016. ISBN: 149192912X.

-
- [30] *Grafana: The open observability platform*. URL: <https://grafana.com/> (visited on 03/03/2021).
 - [31] *Dynatrace*. URL: <https://www.dynatrace.com> (visited on 03/11/2021).
 - [32] Netflix. *Hystrix*. URL: <https://github.com/Netflix/Hystrix> (visited on 03/01/2021).
 - [33] *Spring*. URL: <https://spring.io/> (visited on 03/01/2021).
 - [34] *Prometheus - Monitoring system & time series database*. URL: <https://prometheus.io/> (visited on 03/31/2021).
 - [35] Peter Jausovec. *Fallacies of distributed systems* | Oracle Developers Blog. Nov. 2020. URL: <https://blogs.oracle.com/developers/fallacies-of-distributed-systems> (visited on 03/02/2021).
 - [36] Victor Chircu. *Understanding the 8 fallacies of Distributed Systems - Simple Oriented Architecture*. Feb. 2018. URL: <https://www.simpleorientedarchitecture.com/8-fallacies-of-distributed-systems/> (visited on 03/02/2021).
 - [37] James P. G. Sterbenz. "Protocols for High-Speed Networks: A Brief Retrospective Survey of High-Speed Networking Research". In: *Protocols for High Speed Networks*. Ed. by Georg Carle and Martina Zitterbart. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 243–265. ISBN: 978-3-540-47828-7.
 - [38] Jakob Nielsen. *Nielsen's Law of Internet Bandwidth*. Apr. 1998. URL: <https://www.nngroup.com/articles/law-of-bandwidth/> (visited on 03/02/2021).
 - [39] Sabu M. Thampi. "Introduction to Distributed Systems". In: *CoRR* abs/0911.4395 (2009). arXiv: 0911.4395 [cs.DC].
 - [40] *resilience4j*. URL: <https://github.com/resilience4j/resilience4j> (visited on 03/11/2021).
 - [41] *Sentinel*. URL: <https://github.com/alibaba/Sentinel> (visited on 03/11/2021).
 - [42] *failsafe*. URL: <https://github.com/jhalterman/failsafe> (visited on 03/11/2021).
 - [43] Christian Posta. *Introducing Istio Service Mesh for Microservices: Build and Deploy Resilient, Fault-tolerant Cloud-native Applications*. O'Reilly Media, 2018. ISBN: 9781492052623.
 - [44] *Linkerd*. URL: <https://linkerd.io/> (visited on 03/11/2021).
 - [45] *Consul by Hashicorp*. URL: <https://www.consul.io/> (visited on 03/11/2021).
 - [46] *AWS App Mesh*. URL: <https://aws.amazon.com/app-mesh/> (visited on 03/11/2021).
 - [47] *Open Service Mesh*. URL: <https://openservicemesh.io/> (visited on 03/11/2021).
 - [48] *Kuma*. URL: <https://kuma.io/> (visited on 03/12/2021).
 - [49] Paulo Sérgio Medeiros dos Santos and Guilherme Horta Travassos. "Action Research Can Swing the Balance in Experimental Software Engineering". In: *CoRR* abs/1306.2414 (2013). arXiv: 1306.2414. URL: <http://arxiv.org/abs/1306.2414>.
 - [50] Martin Höst, Björn Regnell, and Per Runeson. *Att genomföra examensarbete*. svenska. Studentlitteratur AB, 2006. ISBN: 91-44-00521-0.
 - [51] *Vagrant*. URL: <https://www.vagrantup.com/> (visited on 03/29/2021).
-

- [52] *Oracle VM Virtualbox*. URL: <https://www.virtualbox.org/> (visited on 03/31/2021).
- [53] *Quagga Routing Suite*. URL: <https://www.quagga.net/> (visited on 03/29/2021).
- [54] *Open Shortest Path First IGP*. URL: <https://datatracker.ietf.org/wg/ospf/about/> (visited on 03/29/2021).
- [55] *RFC 1105 Border Gateway Protocol (BGP)*. URL: <https://tools.ietf.org/html/rfc1105> (visited on 03/29/2021).
- [56] *MinIO | High Performace Kubernetes Native Object Storage*. URL: <https://min.io/> (visited on 03/30/2021).
- [57] *Amazon S3 REST API Introduction*. URL: <https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html> (visited on 03/30/2021).
- [58] *K3S: Lightweight Kubernetes*. URL: <https://k3s.io/> (visited on 03/30/2021).
- [59] *Envoy Proxy - Home*. URL: <https://www.envoyproxy.io/> (visited on 04/01/2021).
- [60] *Istio / Virtual Service*. URL: <https://istio.io/v1.9/docs/reference/config/networking/virtual-service/> (visited on 04/28/2021).
- [61] *tc(8) - Linux manual page*. URL: <https://man7.org/linux/man-pages/man8/tc.8.html> (visited on 03/29/2021).
- [62] *tc-netem(8) - Linux manual page*. URL: <https://man7.org/linux/man-pages/man8/tc-netem.8.html> (visited on 03/29/2021).
- [63] *Rclone*. URL: <https://rclone.org/> (visited on 03/30/2021).
- [64] *Istio / Istio Standard Metrics*. URL: <https://istio.io/v1.9/docs/reference/config/metrics/> (visited on 04/26/2021).
- [65] *Istio / Customizing Istio Metrics*. URL: <https://istio.io/v1.9/docs/tasks/observability/metrics/customize-metrics/> (visited on 04/26/2021).
- [66] *Istio / Performance and Scalability*. URL: <https://istio.io/v1.9/docs/ops/deployment/performance-and-scalability/> (visited on 05/03/2021).

Appendices

Appendix A

Configuration

A.1 Quagga Configuration

A.1.1 OSPFD Configuration

Configuration file for setting up ospfd in Quagga:

Router1

```
1 interface enp0s8
2 interface enp0s9
3 interface lo
4 router ospf
5 default-information originate always
6 passive-interface enp0s9
7 network 172.27.102.0/28 area 0.0.0.0
8 network 172.27.101.32/28 area 0.0.0.0
9 line vty
```

Router2

```
1 interface enp0s8
2 interface enp0s9
3 interface lo
4 router ospf
5 passive-interface enp0s8
6 network 172.27.101.0/27 area 0.0.0.0
7 network 172.27.101.16/27 area 0.0.0.0
8 network 172.27.102.0/28 area 0.0.0.0
```

9 line vty

A.1.2 Zebra Configuration

Configuration file for setting up zebra in Quagga:

Router1

```
1 interface enp0s8
2 ip address 172.27.102.14/28
3 ipv6 nd suppress-ra
4 interface enp0s9
5 ip address 172.27.101.46/28
6 ipv6 nd suppress-ra
7 interface lozebra
8
9 ip forwarding
10 line vty
11 ip route 0.0.0.0/0 172.16.0.2
```

Router2

```
1 interface enp0s8
2 ip address 172.27.101.30/27
3 ipv6 nd suppress-ra
4 interface enp0s9
5 ip address 172.27.102.1/28
6 ipv6 nd suppress-ra
7 interface lozebra
8 ip forwarding
9 line vty
```

A.1.3 BGPD Configuration

Configuration file for setting up bgpd in Quagga:

Router2

```
1 router bgp 64501
2 bgp router-id 172.27.101.30
3 network 172.27.101.1/27
4 neighbor 172.27.101.1 remote-as 64512
5 neighbor 172.27.101.17 remote-as 64512
```

A.2 Istio Operator Configuration

The following code was used in order to apply the custom metrics for Istio. The Istio operator was deployed with Ansible using Helm.

```

1 kind: IstioOperator
2   metadata:
3     name: istio-controlplane
4   spec:
5     values:
6       global:
7         meshID: "{{ mesh }}"
8         multiCluster:
9           clusterName: "{{ cluster }}"
10        network: "{{ CLUSTER_NETWORK }}"
11      telemetry:
12        v2:
13          prometheus:
14            configOverride:
15              inboundSidecar:
16                metrics:
17                  - name: requests_total
18                    dimensions:
19                      upstream_pod_name: upstream_peer.name
20                      downstream_pod_name: downstream_peer.name
21                  - name: request_duration_milliseconds
22                    dimensions:
23                      upstream_pod_name: upstream_peer.name
24                      downstream_pod_name: downstream_peer.name
25                  - name: request_bytes
26                    dimensions:
27                      upstream_pod_name: upstream_peer.name
28                      downstream_pod_name: downstream_peer.name
29                  - name: response_bytes
30                    dimensions:
31                      upstream_pod_name: upstream_peer.name
32                      downstream_pod_name: downstream_peer.name
33              outboundSidecar:
34                metrics:
35                  - name: requests_total
36                    dimensions:
37                      upstream_pod_name: upstream_peer.name
38                      downstream_pod_name: downstream_peer.name
39                  - name: request_duration_milliseconds
40                    dimensions:
41                      upstream_pod_name: upstream_peer.name
42                      downstream_pod_name: downstream_peer.name
43                  - name: request_bytes
44                    dimensions:
45                      upstream_pod_name: upstream_peer.name
46                      downstream_pod_name: downstream_peer.name
47                  - name: response_bytes
48                    dimensions:
49                      upstream_pod_name: upstream_peer.name

```

```
50         downstream_pod_name: downstream_peer.name
51     profile: default
52     # There is a bug in istio where the instructions given in
53     # https://istio.io/latest/docs/setup/install/virtual-machine/#
54     # configure-the-virtual-machine
55     # would fail due to a panic in istioctl. This bug is fixed
56     # upstream in
57     # https://github.com/istio/istio/commit/790
58     # b21dae0c887f5766699231c02f7e319623a5b,
59     # but it is not part of any official release as of today
60     # (2021-02-19). Having the following config skips the panic, as
61     # it
62     # was having an empty proxyMetadata caused the panic.
63 meshConfig:
64     defaultConfig:
65         proxyMetadata:
66             BLA: bla
67         extraStatTags:
68             - source_address
69             - destination_address
70             - upstream_pod_name
71             - downstream_pod_name
72 components:
73     pilot:
74         k8s:
75             resources:
76                 requests:
77                     memory: 256Mi
```

A.3 Prometheus Configuration

Prometheus was deployed with Ansible using helm. The configuration for deploying Prometheus is the following:

```
1 community.kubernetes.helm:
2     name: prometheus
3     chart_ref: prometheus-community/prometheus
4     chart_version: 11.16.2
5     namespace: istio-system
6     create_namespace: true
7     update_repo_cache: yes
8     state: present
9     values:
10         alertmanager:
11             enabled: false
12         pushgateway:
13             enabled: false
14         kubeStateMetrics:
```

```

15         enabled: false
16     nodeExporter:
17         enabled: false
18     serverFiles:
19         recording_rules.yml:
20             groups:
21                 - name: custom_rules
22                   rules:
23                       - record: upstream_pod_name_to_upstream_node_name
24                         expr: (0 * count(label_replace(label_replace(
25                             container_memory_rss, "upstream_pod_name", "$1", "
26                             pod", "(.*)"), "upstream_node_name", "$1", "
27                             instance", "(.*)")) by (upstream_pod_name,
28                             upstream_node_name))
29                       - record: downstream_pod_name_to_downstream_node_name
30                         expr: (0 * count(label_replace(label_replace(
31                             container_memory_rss, "downstream_pod_name", "$1",
32                             "pod", "(.*)"), "downstream_node_name", "$1", "
33                             instance", "(.*)")) by (downstream_pod_name,
34                             downstream_node_name))
35
36     server:
37         podAnnotations:
38             sidecar.istio.io/inject: "false"
39         persistentVolume:
40             enabled: false
41         # Use port 9090 to match Istio documentation
42         service:
43             servicePort: 9090
44             readinessProbeInitialDelay: 0
45             # Speed up scraping a bit from the default
46             global:
47                 scrape_interval: 15s
48     extraScrapeConfigs: |
49         - job_name: vm-logs
50           scrape_interval: 10s
51           honor_labels: true
52           metrics_path: '/stats/prometheus'
53           static_configs:
54             - targets:
55                 - '172.27.101.33:15090'
56               labels:
57                 node_name: 'server13'
58         - job_name: 'envoy-stats'
59           metrics_path: /stats/prometheus
60           kubernetes_sd_configs:
61             - role: pod
62           relabel_configs:
63             - source_labels: [__meta_kubernetes_pod_node_name]
64               action: replace

```

56 `target_label: node_name`

Appendix B

Prometheus Queries

Prometheus queries used to pull the data for the results from the experiments.

B.1 Latency

```
1 //50th Percentile Latency
2 histogram_quantile(
3     0.50,
4     sum(
5         irate(
6             istio_request_duration_milliseconds_bucket{
7                 reporter="source",
8                 destination_workload=~"server13",
9                 destination_workload_namespace=~"vm-namespace",
10                source_canonical_service="clone"
11            }[1m]
12        )
13    ) by (le)
14 ) or histogram_quantile(
15     0.50,
16     sum(
17         irate(
18             istio_request_duration_seconds_bucket{
19                 reporter="source",
20                 destination_app="server13",
21                 node_name="server2"
22             }[1m]
23         )
24     ) by (le) / 1000
```

```
25 )
26
27 //90th Percentile Latency
28 histogram_quantile(
29     0.90,
30     sum(
31         irate(
32             istio_request_duration_milliseconds_bucket{
33                 reporter="source",
34                 destination_workload=~"server13",
35                 destination_workload_namespace=~"vm-namespace",
36                 source_canonical_service="clone"
37             }[1m]
38         )
39     ) by (1e)
40 ) or histogram_quantile(
41     0.90,
42     sum(
43         irate(
44             istio_request_duration_seconds_bucket{
45                 reporter="source",
46                 destination_app="server13",
47                 node_name="server2"
48             }[1m]
49         )
50     ) by (1e) / 1000
51 )
52
53 //99th Percentile Latency
54 histogram_quantile(
55     0.99,
56     sum(
57         irate(
58             istio_request_duration_milliseconds_bucket{
59                 reporter="source",
60                 destination_workload=~"server13",
61                 destination_workload_namespace=~"vm-namespace",
62                 source_canonical_service="clone"
63             }[1m]
64         )
65     ) by (1e)
66 ) or histogram_quantile(
67     0.99,
68     sum(
69         irate(
70             istio_request_duration_seconds_bucket{
71                 reporter="source",
72                 destination_app="server13",
73                 node_name="server2"
```

```
74         }[1m]
75     )
76 ) by (1e) / 1000
77 )
```

B.2 Bandwidth

```
1  sum(
2    (
3      irate(
4        istio_response_bytes_sum{
5          reporter="source",
6          node_name!=""
7        }[1m]
8      )
9      +
10     (on(upstream_pod_name) group_left(upstream_node_name)
        upstream_pod_name_to_upstream_node_name{upstream_pod_name!=""})
11   )
12   or
13   (
14     on (upstream_pod_name) label_replace(
15       irate(
16         istio_response_bytes_sum{
17           reporter="source",
18           node_name!="",
19           upstream_pod_name!="unknown"
20         }[1m]
21       ),
22       "upstream_node_name",
23       "$1**",
24       "upstream_pod_name",
25       "(.*)")
26     )
27   )
28 ) by (node_name, upstream_node_name)
```

B.3 Packet Loss

```
1  sum(
2    irate(
3      istio_requests_total{
4        reporter="source",
5        node_name!="",
6        destination_app="server13",
7        response_flags="UT"
8      }[1m]
9    )
```

```
10 ) by(node_name, destination_app)
11 or (
12     0 / sum(
13         irate(
14             istio_requests_total{
15                 reporter="source",
16                 node_name!="",
17                 destination_app="server13"
18             }[1m]
19         )
20     ) by (node_name, destination_app)
21 )
```

Appendix C

Rclone Kubernetes Job

The Kubernetes job specification for Rclone with Istio applied.

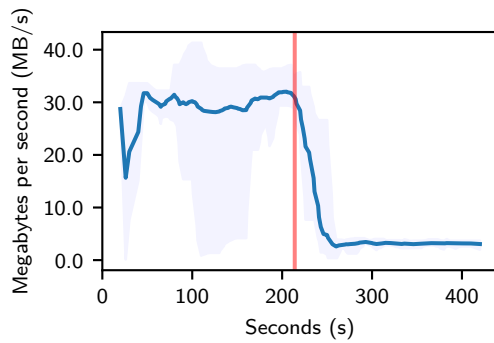
```
1  apiVersion: v1
2  kind: Namespace
3  metadata:
4    name: rclone
5    labels:
6      "istio-injection": "enabled"
7  ---
8
9  apiVersion: v1
10 kind: ConfigMap
11 metadata:
12   name: rclone-config
13   namespace: rclone
14 data:
15   rclone: |
16     [minio]
17     type = s3
18     provider = Minio
19     access_key_id = minioadmin
20     secret_access_key = minioadmin
21     endpoint = http://minio.vm-namespace.svc:9000
22
23 ---
24
25 apiVersion: batch/v1
26 kind: Job
27 metadata:
28   name: clone
```

```
29 namespace: rclone
30 spec:
31   template:
32     spec:
33       tolerations:
34         # only schedule in server2
35         - key: rclone-node
36           operator: Exists
37           effect: NoSchedule
38       containers:
39         - name: clone
40           image: rclone/rclone
41           # The istio-proxy sidecontainer does not exit when rclone is
42             stopped. The commands below force it to quit
43             # when the job is finished.
44           command:
45             - /bin/sh
46             - -c
47             - |
48               until (printf 'GET /healthz/ready HTTP/1.0\r\n\r\n'; sleep 1)
49                 | nc localhost 15020 | grep "HTTP/1.0 200 OK"; do echo \"
50                   Waiting for Sidecar...\"; sleep 3; done;
51               echo \"Sidecar available. Running the command...\";
52               rclone --check-first --dump headers copy minio:{{ bucket }} :
53                 memory: -P;
54               x=$(echo $?); (printf 'POST /quitquitquit HTTP/1.0\r\n\r\n';
55                 sleep 1) | nc localhost 15020 && exit $x
56           volumeMounts:
57             - name: rclone-config
58               mountPath: /config/rclone
59           env:
60             - name: RCLONE_TRANSFERS
61               value: "{{ transfers | default('6') }}"
62       volumes:
63         - name: rclone-config
64           configMap:
65             name: rclone-config
66             items:
67               - key: rclone
68                 path: rclone.conf
69       restartPolicy: Never
```

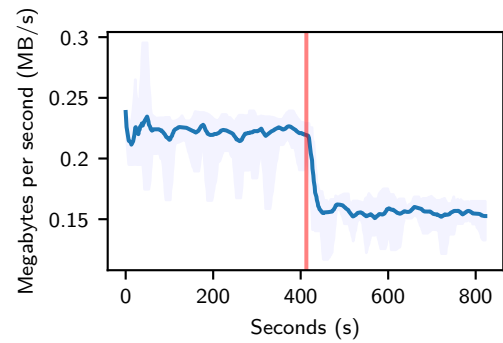
Appendix D

Response Loss Experiment

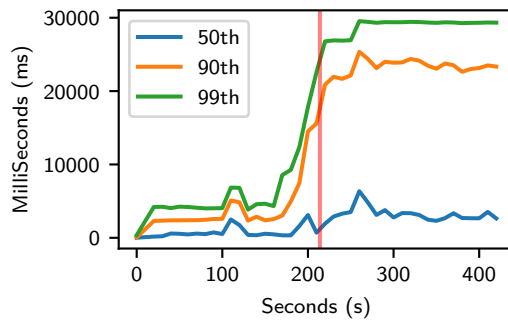
Loss introduced at interface `enp0s8` on *R1* introduced at the red line. This affects the responses from the MinIO server.



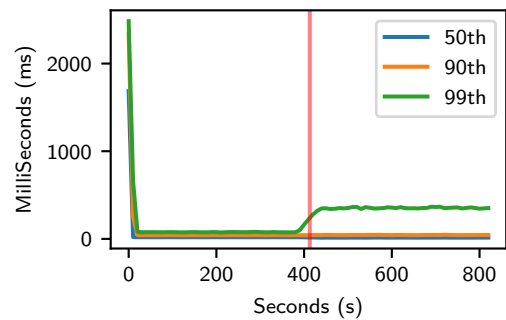
(a) Bandwidth -
1000x1MB



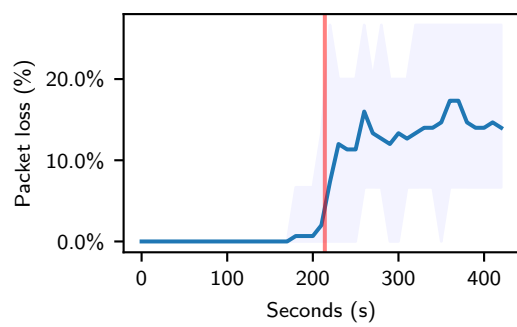
(b) Bandwidth -
100kx1kB



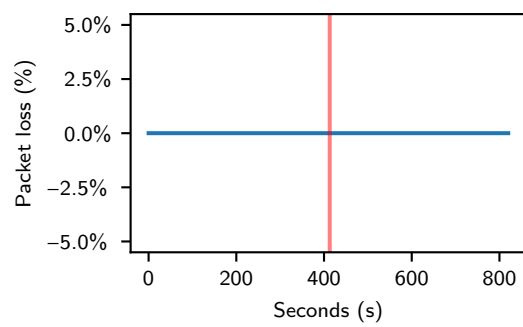
(a) Latency - 1000x1MB



(b) Latency - 100kx1kB



(a) Loss - 1000x1MB



(b) Loss - 100kx1kB