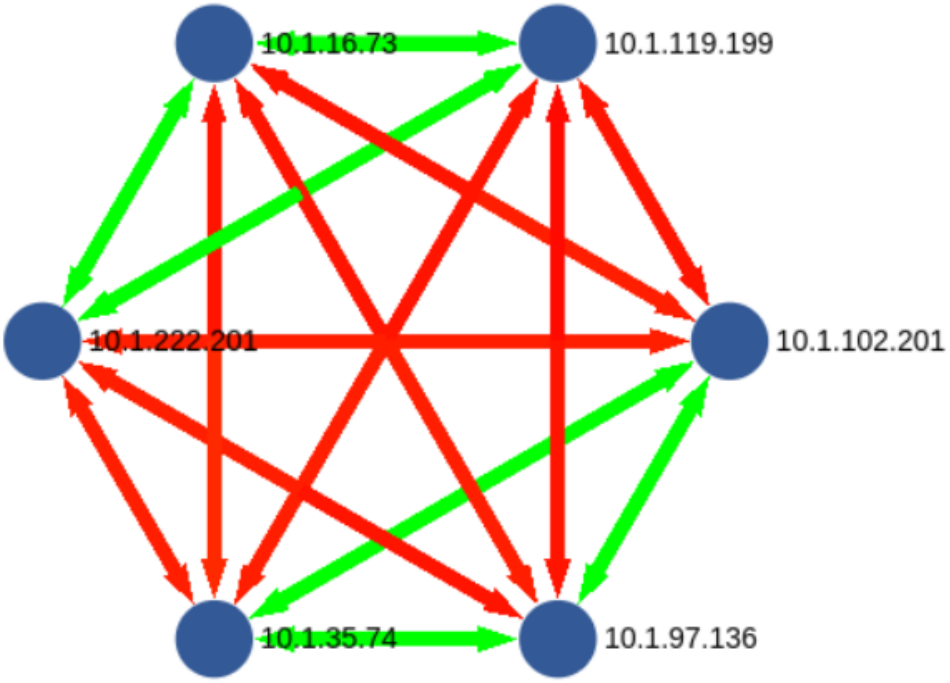
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Detecting Network Partitioning in Cloud Native 5G Mobile Network Applications

Master's thesis in Computer Science and Engineering

Herman Bergström
Oskar Fredriksson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Detecting Network Partitioning in Cloud Native 5G Mobile Network Applications

Herman Bergström
Oskar Fredriksson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Detecting Network Partitioning in Cloud Native 5G Mobile Network Applications

Herman Bergström
Oskar Fredriksson

Supervisor: Romaric Duvignau, Department of Computer Science and Engineering
Advisor: Maysam Mehraban, Ericsson
Examiner: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualization of a complete network partition of 6 nodes in a Kubernetes cluster

Typeset in LaTeX
Gothenburg, Sweden 2022

Detecting Network Partitioning in Cloud Native 5G Mobile Network Applications
Herman Bergström
Oskar Fredriksson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

With the transition of the 5G core network to a cloud native service-based architecture—composed of network functions operating through microservices communicating over the network—there is an increased risk of network failures causing service downtime unrelated to the applications themselves. In particular, cases of partial and simplex network partitionings have been observed in production systems to produce silent failures causing severe symptoms. Thus, diagnosing these failures have proven difficult. As such, the need of monitoring the network between microservices is of particular interest. In this thesis, we devise a distributed monitoring scheme to identify and classify network partitionings in a Kubernetes cluster. We implement and evaluate two approaches of this scheme based on both active and passive monitoring. While both approaches are feasible for our purpose, we conclude that our approach to passive monitoring struggles with classifying simplex partitions due to TCP being a two-way protocol. Similarly, operating the passive mode requires privileges not necessarily suitable for a shared cloud environment. While the active monitoring scheme is able to infer all types of partitions, it will—unlike the passive alternative—increase the overall load on the network. We further present how to make our proof-of-concept implementation scalable when deployed in larger clusters.

Keywords: Network Partitions, Network Monitoring, Distributed Systems, 5G, Cloud Native, Mobile Networks.

Acknowledgements

We would like to extend our gratitude to our supervisor Romaric Duvignau and company advisor Maysam Mehraban for their continued assistance throughout the process. Furthermore, we would like to thank our examiner Ahmed Ali-Eldin Hassan for his valuable feedback, as well as our company manager Daniel Fredriksson for his continued support.

Herman Bergström, Gothenburg, June 2022
Oskar Fredriksson, Gothenburg, June 2022

Contents

List of Figures

List of Tables

1

Introduction

The rise in demand for service availability and data durability has led several organizations to embrace cloud systems. The reason for this is that cloud systems are designed to be highly available, providing on-demand availability of resources in terms of storage and computing despite failure of devices, networks, or even entire data centers [1]. According to IBM [2], 94 % of companies used some form of cloud computing in 2019 and more than half of them are expected to move towards an all-cloud infrastructure by 2021.

One area which aims to benefit from the advantages of cloud-native solutions is the development of 5G. With the promise of offering communication that is highly reliable, low-latency, high speed, and facilitates vast numbers of connected devices [3], the need for a flexible and dynamic infrastructure is crucial for the success of 5G. Therefore, the proposed 5G standard aims to migrate from the previous monolithic core network architecture to a cloud-native Service-Based Architecture (SBA). This architecture allows applications to run in a cloud-native environment using a microservice architecture—a loosely coupled architecture that provides several benefits such as scalability, elasticity and flexibility.

However, the way we test and respond to incidents has fundamentally changed with the introduction of cloud systems. While testing has become an integral part of the deployment pipeline, the absence of failures cannot be guaranteed. Instead, the focus has shifted to partly limiting the blast radius, but also shortening the incident response time—preferably to before the customer even reports the failure—by continuously monitoring applications [4]. An important part of ensuring the correct operation of a distributed system is the underlying network. It therefore stands to reason that the network itself needs to be monitored as well.

1.1 Context

This thesis is a collaboration with Ericsson, who is in the process of transitioning their core network (CN) to a cloud-native infrastructure to meet the rising demands required of 5G systems. With the increasing demands on service availability, 5G aims to offer an increase in throughput and reduction of latency by 10 times when compared to 4G. Similarly, the perceived availability of a 5G network should be 99.999%—around five minutes and 15 seconds of downtime per year [5]. Apart from the general requirements, telecommunications companies also have agreements with the service users known as *Service-Level Agreements* (SLAs).

An SLA is a legal agreement between the service provider and the service user outlining a set of service properties known as *Quality of Service* (QoS) [6]. The agreements focus on particular aspects of a service such as the quality, availability and accountability. Usually, several measurable metrics such as uptime, latency, throughput, and jitter are used as a reference to ensure that each party fulfill the required set of conditions. If a violation of QoS occurs, the party accountable will be liable for penalty fees as per the terms specified in the agreement.

However, as applications move from dedicated hardware to the cloud, the structure of SLAs change. This is due to cloud environments being more complex in nature with multiple stakeholders and an increased difficulty in determining the root cause of service interruptions [7]. One aspect affected by the migration towards the cloud is the communication between different applications and services, since it more often transpires over the network in cloud systems. For applications in a cloud-native 5G environment, disturbances in the network will therefore have a devastating impact on the performance of the applications, especially with the strict requirement on service availability. It is therefore vital for both service provider and user, from a troubleshooting and legal perspective, to monitor the network itself in order to determine if application downtime originated from issues residing on the application or network layer.

1.2 Problem Description

In some environments, application failures indicative of partial network partitioning have been reported [1], [8]. These failures have proven difficult to detect, since their presence are nearly impossible to identify by use of standard up-time monitoring—all the while exhibiting symptoms suggesting a fault in the application.

This project aims to define network failures that can occur in cloud-native telecommunication networks. From this information, a proof-of-concept (PoC) for a tool is created that can be deployed in the cloud to detect the presence of network failures—in particular network partitioning—without prior knowledge of the network topology.

Research Questions

This project strives to answer the following research questions:

1. How feasible and efficient is it to detect network failures in a production Container as a Service platform (CaaS) adapted for 5G applications?

2. How can one introduce alarms that present eventual network partitions as a result of the failures?

1.3 Goals

In order to answer the research questions, we create a containerized application that can be deployed as pods in a Kubernetes cluster. Our application can be run in two modes—*active* and *passive*. Using active probing, the application collects information regarding one-way packet loss and latency. Passive listening enables the application to detect packet loss by the occurrence of TCP retransmissions from any container on the host. All measurement data is collected by a Prometheus server. In addition, a separate application is implemented which visualizes this data and detects occurrences of network partitioning.

1.4 Limitations

One main limitation of the project is that the Kubernetes clusters used is not run in a full production-level 5G environment since such an environment was not available during the time span of the thesis. However, due to the cloud-native nature of the 5G service-based architecture, components should be able to run on commodity hardware in any Kubernetes cluster. Thus, during the development of the proof-of-concept, an already existing staging environment for 5G components is used.

While previous works, such as [9]–[11], use knowledge of the underlying network topology or configuration to localize faults to a physical link, this project does not assume that such in-depth information is available. Instead, our implementation can run on any Kubernetes cluster irrespective of the underlying network architecture. The purpose of this project is to identify and characterize disturbances rather than localizing faults. As such, we are only concerned with pod-to-pod connectivity.

1.5 Thesis Outline

The remainder of the thesis is organized as follows: Chapter 2 introduces some necessary background information, including 5G, cloud systems, theory about and ways to monitor for network failures, as well as previous works. Chapter 3 describes the methods taken to implement the PoC developed in this thesis, while Chapter 4 explains how the PoC is evaluated. The results of this evaluation are then presented

in Chapter 5 and discussed in Chapter 6. Finally, Chapter 7 consists of the overall conclusions of this thesis.

2

Background

This chapter is divided into five sections containing the necessary background knowledge needed to understand the work presented in this thesis. The first section briefly introduces the 5G architecture and a more detailed explanation of the new 5G core. This is followed by an introduction of cloud systems and the different service models available. The third section introduces the different failures that can arise when communication occurs over a network, while the fourth presents different techniques and metrics of interest when monitoring such failures. The fifth and last section presents previous work in the area of end-hosts distributed network monitoring.

2.1 5G Architecture

With the need of dynamic infrastructure, the 5G mobile network architecture embraces cloud-native solutions. As such, the previous monolithic 4G architecture—a tightly coupled architecture where each component works closely together in an integrated manner—will be replaced. However, as it will take a considerable amount of time to implement a fully cloud-native mobile network, the old and new infrastructure will have to co-exist for some time. To start the process of moving towards a fully cloud-native infrastructure, the first point of action is to migrate the core network.

5G Core

The 5G core network architecture (5GC), issued by the third generation partnership project (3GPP), is at the heart of the new 5G specification. It serves as the latest evolution of the 3GPP core network architecture, moving away from the previous Evolved Packet Core (EPC) which served as the core network in the 3G/4G network architecture.

The core network is in charge of establishing reliable and secure connectivity to the network between the *Radio Access Network* (RAN) and the external access network (i.e., the Internet) [12], [13]. It handles a wide variety of mobile network functions such as session and mobility management, authentication and authorization, among others. Compared to previous generations of core networks, services in the 5G core network are handled by *Network Functions* (NF) that are completely software-based

and designed to be deployed in a cloud-native system. This means that the functions themselves are separated from the underlying cloud infrastructure, allowing for faster deployment and more flexibility [13].

The migration of the core network to cloud-native has been made possible through technologies such as *Software-Defined Networking* (SDN) and *Network Function Virtualization* (NFV) [3]. SDN introduces a separation of the network's control logic (control plane) from the physical devices tasked with forwarding users' network packets (user plane) [14]. NFV in turn focuses on the use of virtualization to abstract the NFs (such as those making up the 5G core network) away from expensive dedicated hardware, and onto virtualized software solutions—referred to as Virtualized Network Functions (VNFs)—running on *Commercial-Off-The-Shelf* (COTS) hardware [15]. An added benefit of NFV is that, due to network functions being implemented in software, managing and making changes to a network is often more straightforward. All of these technologies thus prompt the need for a new cloud-native core network standard for 5G—namely the 5G *Service-Based Architecture* (SBA) [12].

Service-Based Architecture

The new SBA making up the 5G core network consists of decoupled NFs offering one or more services to other NFs through Application Programming Interfaces (API). In the SBA, each NF is formed through the combinations of smaller pieces of interconnected services known as microservices that communicate with each other through an inter-process communication model, often via the network. Since the architecture is completely service-based, these microservices can in turn be re-used in other NFs, which in turn reduces the implementation workload and allows for easier redeployment when rolling out new functionality [13].

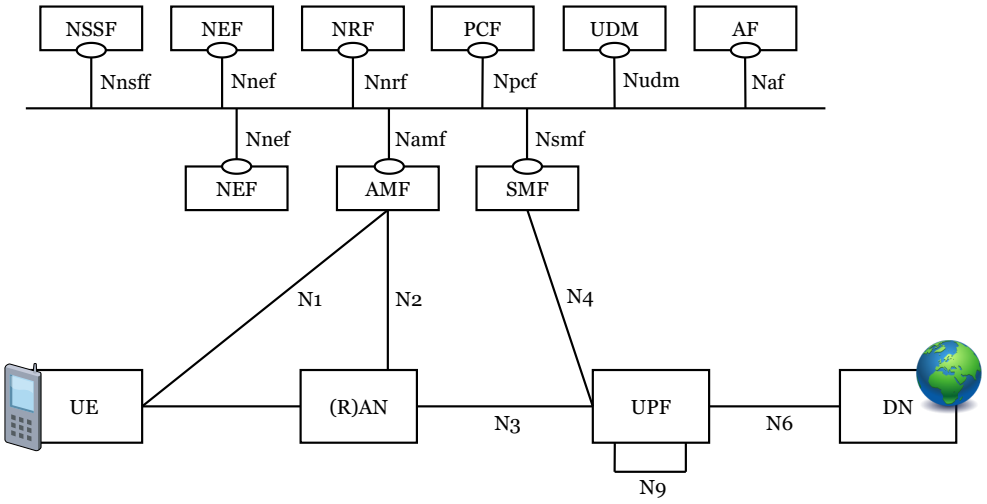


Figure 2.1: 5G Core architecture depicting the service-based control plane and the SBI between the different NFs [12].

An overview of the new 5GC architecture, from an API point of view, can be seen in Figure 2.1. The upper half of the figure shows the new control plane which

implements a service-based architecture consisting of several different NFs, each of which provides a different service. For example, mobility management is handled by the Access and Mobility Management Function (AMF). This NF handles the connection of geographically moving User Equipment (UE) to different base stations in the Radio Access Network (RAN). The API based communication between the different NFs is known as a service-based interface (SBI). An NF can utilize an API call over the SBI in order to invoke a particular service or operation.

2.2 Cloud Systems

Cloud Computing can be considered a fairly new paradigm and involves the on-demand allocation of resources such as infrastructure, storage, databases and various application services to users. This is achieved through *cloud-native computing*—an approach in software development utilizing cloud computing to allow for organizations to build and run applications to meet the different requirements of the users.

The Cloud Native Computing Foundation (CNCF), a Linux Foundation project founded to help advance container technology, defines cloud-native as [16]:

> *Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.*

CNCF oversees multiple cloud technology projects used in production systems today. Some well-known CNCF projects include Kubernetes, Helm, Prometheus, containerd, CoreDNS, LitmusChaos, etc [17].

In order to provide a dynamic environment with capabilities of on-demand resource allocation, the cloud-native architecture is a Service-Oriented Architecture (SOA), which provides resources to users in terms of services. These services can take different forms depending on the degree of management available to the provider and the degree of control the user wants [18]. Figure 2.2 presents four different service models and the respective responsibilities of the service provider and user at the different layers.

Infrastructure as a Service (IaaS)

The IaaS model is the base layer of the different service models and provides users with computing resources such as virtual machines, servers, storage and networking to allow users to run arbitrary software, which can include operating systems and applications [19]. As such, the user is not in control of the underlying infrastructure but still has to configure the different virtual machines, virtual networks, operating systems and runtime environments.

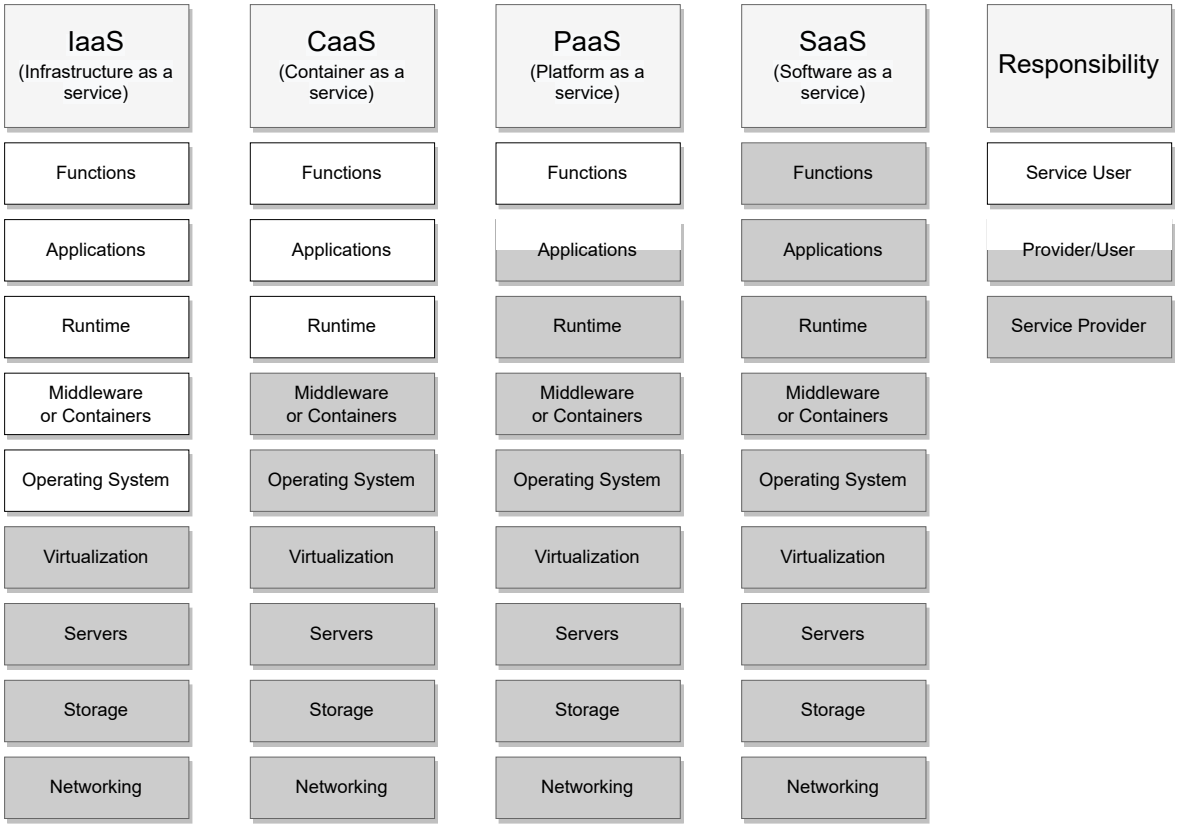


Figure 2.2: Service models and the responsibilities of the service provider/user at the different layers.

Container as a Service (CaaS)

The CaaS model is commonly deployed on top of an existing IaaS model and can be considered as a subset of IaaS, except that instead of using VMs or bare metal servers its primary resource is containers. A container is a package of software that includes all the necessary dependencies, such as code, runtime, libraries, etc., in order to run on any type of host system. A more detailed explanation of how containers work can be seen in Section 2.2.1. The benefit of using CaaS is that container platforms can provide horizontal scaling and migration of services. The platform thus manages the provisioning of physical or virtual machines and the underlying operating system. All the while, the user still retains a high degree of control when it comes to runtimes and other customizations [20].

Platform as a Service (PaaS)

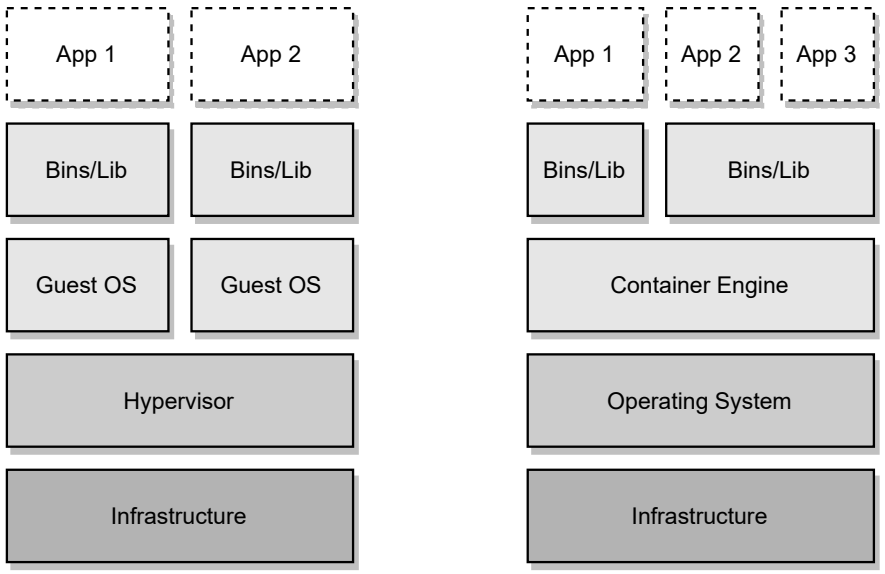
In the PaaS model, a platform has already been configured for the user to deploy applications. With this model, the user does not need to install any operating system, runtime environments or databases as all of this is provided by the service provider. In this type of environment, the user will only focus on building and delivering applications to the platform and can spend less time on configuring the different environments.

Software as a Service (SaaS)

SaaS is the top layer of the different service models, where the provider is responsible for the entire stack, from physical hardware to the application layer. In this model, the user is provided with applications—usually accessible through web pages or program interfaces—and is not responsible for any of the underlying cloud infrastructure [19].

2.2.1 Containerization

When deploying microservices onto commodity hardware, it is possible to use hardware virtualization in the form of virtual machines (VM). However, this would mean that each microservice would have to be packaged inside a monolithic disk image also containing a full guest OS. Such a solution would be characterized by slow startup performance along with greater memory and disk usage [21]—properties undesirable in a microservice architecture.



(a) VM **(b)** Container

Figure 2.3: Example of VM and Container architecture.

The approach containerization takes is also referred to as OS-level virtualization, instead letting containers execute in isolated user spaces on top of the same kernel. By utilizing modern kernel features, such as namespaces and control groups (cgroups) in the Linux kernel, groups of processes can effectively be isolated from each other, and their resource usage can be controlled [21]. A visualization of the difference between the VM and container architecture can be seen in Figure 2.3, where instead of having each disk image containing full guest OS as shown in the VM architecture, the container-based architecture allows applications to share the host OS kernel and sometimes even binaries and libraries. This allows containers to be much more lightweight and have faster startup when compared to VMs.

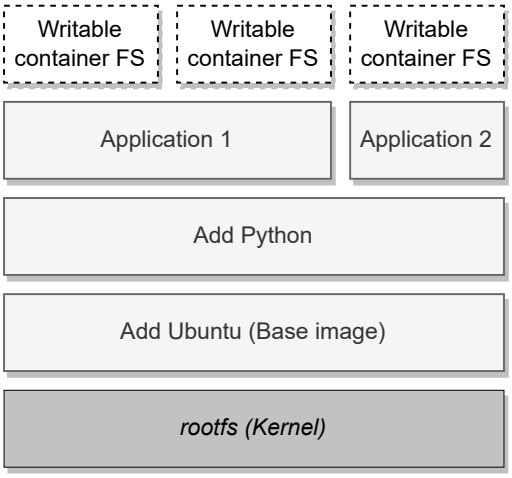


Figure 2.4: Example of file system layers of three distinct containers.

Instead of the monolithic approach to disk images often seen with VMs, container images are usually built in *layers*, with a new image being made by adding a layer on top of another image. Using so-called "union mounts" provided by the kernel, each layer is represented by a read-only file system, with each additional layer describing the difference compared to the parent image. This is also referred to as a copy-on-write file system. The consequence is that images can share underlying layers with other images. The bottom layer is referred to as the base image—often a Linux distribution such as Ubuntu, but without the *rootfs* kernel image as it is provided by the host [21]. In Figure 2.4, we illustrate what a file system may look like with three containers. The first two of the containers are created using the "Application 1" image and the third uses the "Application 2" image. When a container is created, a writable file system is added on top of the image and is part of the container's state. In this case, both applications are written in Python, meaning that their images can be built on top of the same Python runtime image. The Python runtime image is based on an Ubuntu image in this case. When run, the containers may thus all mount the two same read-only file systems based on the Ubuntu and Python layers, while the first two containers also share the "Application 1" file system.

There exist multiple different container systems, one of the most well-known being Docker [22], which is implemented on top of the open-source *containerd* container runtime. containerd follows the Open Container Initiative (OCI) specification. Images following the OCI Image Format (often referred to as "Docker images") make up the vast majority of all container images.

2.2.2 Container Orchestration

By packaging microservices into containers, they can be deployed on a Container as a Service (CaaS) platform. By forming a cluster of multiple hosts, the container orchestration functionality of such platforms can automatically scale microservices horizontally by deploying more containers based on identical images. Such platforms also offer load-balancing support [21]. Networking functionality ensures containers

may communicate with each other, despite the isolation by the container runtime. Although containerization is ideal for microservices, containerized monolithic applications can also see benefits from being managed by a CaaS platform.

One of the most popular options is to build a CaaS platform on top of Kubernetes, an open-source container orchestration system [23]. In Kubernetes, the minimum deployable unit is known as a "pod," which in turn consists of one or more containers run on the same node in a shared context. Kubernetes supports several container runtimes, the most common being containerd. Network traffic between pods is managed by Kubernetes, which assigns a unique IP address to each pod addressable by the rest of the cluster. It is however possible to deploy network policies dictating with whom pods may communicate.

2.2.3 Chaos Engineering

Chaos engineering is an emerging discipline of *Fault Injection Testing* techniques that preemptively exposes a distributed system to disturbances in a controlled fashion to ensure it can withstand turbulent conditions in production [24]. By doing so, it helps build confidence in the ability of distributed systems to withstand realistic disturbances. Similarly, it makes it possible to detect faults in software earlier and reduce the number of unnecessary faults reported by the user [25].

The concept of chaos engineering was introduced by Netflix, which created some of the first chaos engineering tools. One of the most well-known tools is Chaos Monkey, an internal service that randomly selects VMs hosting production services and forcefully terminates them [24]. The introduction of Chaos Monkey has led to the process of developing software services capable of withstanding failures of entire VMs.

With the recent development of cloud-native applications, several chaos engineering tools adapted for container-based deployments have emerged. For Kubernetes, some common tools include kube-monkey[1], Litmus[2], PowerfulSeal[3], ChaosBlade[4], and Chaos Mesh[5].

Each tool offers different ways of injecting chaos into container-based deployments. For example, kube-monkey—an implementation of Netflix's Chaos Monkey for Kubernetes clusters—randomly terminates Kubernetes pods to test if services are resilient to unexpected crashes. Other tools providing additional types of disturbances include Litmus, which offers several chaos experiments on a pod-to-pod or application level, but also on a platform or infrastructure level. The former allows for deleting pods, containers, adding network loss, latency, hogging CPU, etc., whereas

[1] `https://github.com/asobti/kube-monkey`
[2] `https://litmuschaos.io/`
[3] `https://github.com/powerfulseal/powerfulseal`
[4] `https://github.com/chaosblade-io/chaosblade`
[5] `https://chaos-mesh.org/`

the latter injects chaos into the node and can exhaust node memory resources, cause disk corruptions, etc.

2.3 Network Failures

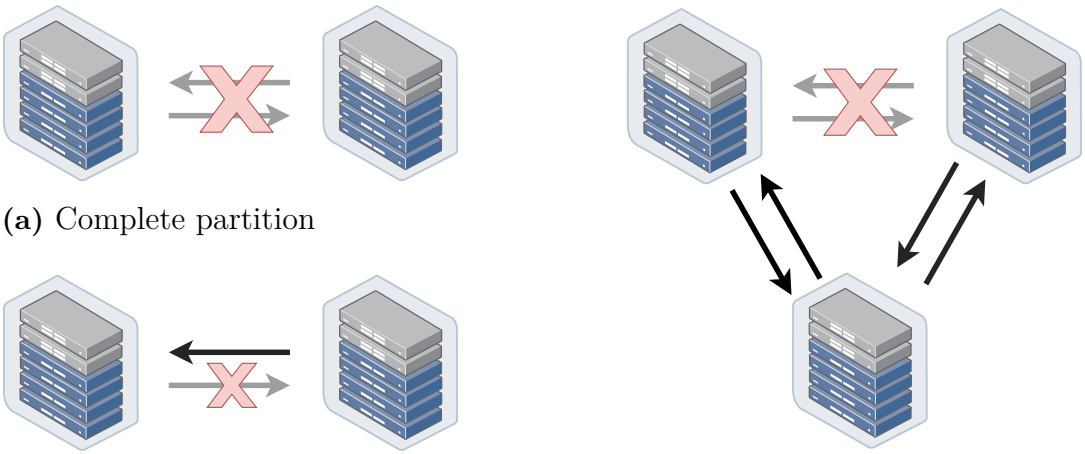
Failures are rampant in large cloud networks. Considering the scale at which some providers operate, however, this should not be a surprise [26]. In a study performed on a large group of production servers by Microsoft engineers, it was found that 8% of them experienced failure during one year (most commonly due to faulty hard drives) [27]. Thus, in order to avoid failures in cloud systems to some degree, fault tolerance measures must be taken [28]. This also applies to the network.

Before introducing the forms network failures may take, it is appropriate to define the terminology that will be used throughout the thesis. We define an *event* to be any exceptional condition in either hardware or software. *Faults* are events that are not caused by another event (also known as the "root cause"). These in turn may manifest as one or more discrepancies from an expected state or condition known as *errors* [28], [29]. We consider *network failures* to be any error that prevents a system from functioning correctly and is thus visible to users [29]—also encompassing *partial network failures* which include degradation and graceful degradation of service. A network failure can be caused by "malfunction or natural or human-caused disasters" [30]. It can therefore be deduced that even heavy congestion would, by this definition, cause a partial network failure. *Symptoms* may be defined as the manifestation of failures that are observable as alarms from monitoring systems [29].

2.3.1 Faults in Practice

Similarly to servers, networking equipment is also expected to fail. Another study in Microsoft data centers [26] points to below 5% of commodity switches failing annually. However, top-of-rack (ToR) switch failures still account for most of the downtime. This is attributed partly to their sheer quantity in large data centers, but interestingly also due to that type of repair being a low priority. Failover techniques, like replication, are expected to maintain service availability. This can also be observed in the fact that—similarly to other operators such as Google [31] and Facebook [10]—redundant ToR switches are not used.

Not all faults are the same. While predictable faults can be modeled statistically, other faults may cause errors harder to predict and localize. Redundancy has been shown to not be entirely effective against failures [26]. We will now present a few types of failures prevalent in the literature. A reoccurring type of failure in literature is silent packet drops. Due to software bugs or faulty hardware, packets are dropped without hardware counters incrementing, causing traditional SNMP monitoring systems to not find any fault [32], [33]. In another instance, a popular chip vendor shipped network cards with a firmware bug causing all ingress packets to be dropped [1], [34]. Sometimes, routing loops may form between a number of switches



(a) Complete partition

(b) Partial partition

(c) Simplex partition

Figure 2.5: The three variants of partitions identified in [1]. Arrows denote directional packet flows.

due to software bugs [32], [34]. It is also not seldom that a fault manifests itself from the control or management plane [35].

2.3.2 Partitioning Failures

While the symptoms of a network failure presented by the network layer may appear "as simple as" packet loss or latency anomalies, the fact that nodes occasionally lose connection with each other plays a central role when designing distributed systems. In particular, distributed systems have to consider partitioning, where one group of nodes loses connection to another group of nodes (see Figure 2.5a). Large-scale cloud providers have emphasized the importance of partition tolerant applications, hinting that it is a common occurrence at their scale, while simultaneously designing distributed systems where partition tolerance is an essential design criterion [34].

While network partitioning and partition tolerance have been explored for decades within literature and can be considered common knowledge, it was only recently that *partial network partitioning* was described in literature [1]. It is a phenomenon where, while packet reachability between nodes form a connected graph it is not a complete graph, as seen in Figure 2.5b. In other words, while two groups of nodes cannot communicate with each other, both can still communicate with a third group of nodes. Along with complete (Figure 2.5a) and partial (Figure 2.5b) partitioning, the authors also identify a third type referred to as *simplex partitioning*, where communication only in one direction is interrupted (Figure 2.5c). Because its effects are not considered when designing some systems, partial/simplex partitioning has been shown to break correctness and cause data loss in popular database and message-passing systems used today in cloud applications. That being said, some systems also include mitigations against partial partitioning [1], [8].

Based on the descriptions given by Alquaraan et al. [1], we formally define the types of partitions as used throughout the thesis. We consider a directed graph $G = (V, E)$, where if there exists an edge $(u, v) \in E$ where $u, v \in V$, packets can be successfully sent from node u to node v. In the general case, a network partitioning has occurred if there exists any two nodes $u, v \in V$ where $(u, v) \notin E$. A complete network partitioning exists iff G is disconnected (i.e., all nodes cannot be part of the same weakly connected component). Furthermore, a partial network partitioning exists if $(u, v) \notin E$ while another directed path from u to v exists. Lastly, a simplex partitioning exists iff for any two nodes $u, v \in V$, $(u, v) \in E$ but $(v, u) \notin E$. We note that by these definitions, multiple types of partitions may exist simultaneously.

While a general solution to partial partitioning is specifically proposed in [8], it is based on routing network traffic through other nodes by deploying OpenFlow-based virtual switches on them. In Figure 2.5b all traffic between the left and right groups would in that case travel through the lower middle group—in essence building another routing layer on top of the physical network. While this possibly could be a suitable solution for certain low-volume traffic, it may lead to scalability issues if networks and nodes are not designed to handle the resulting traffic pattern—in turn prompting new congestion-related failures. Furthermore, it requires software and virtual switches to be deployed on all nodes, which itself could cause faults. To this end, designing applications with partition tolerance in mind would be the ideal solution. Until then, however, there must be a way to correlate failures with potential partitioning events—which is the intended purpose of the proof-of-concept produced in this thesis.

2.4 Network Monitoring

Continuous detection and localization of network faults in data-center settings is not a new topic. Diagnosing a network fault usually involves three steps: (1) identifying that a fault has occurred, (2) localizing/isolating a set of possible faults, and (3) confirming that the actual fault was found [29], [32]. Network monitoring tools may assist in one or more of these steps. In this thesis, we are primarily interested in identifying the occurrence of faults presenting themselves as network failures.

Some authors identify important properties desirable in data-center monitoring systems. In particular, such systems incur low overhead on networks and hosts, are continuously monitoring for and presenting disturbances, and are simple to configure and use [33], [36].

Network tomography is a field of research referring to the act of reconstructing some internal state of a network using a limited number of *monitors* (i.e. nodes collecting metrics). The main sub-field is referred to as *network performance tomography* and primarily entails how to reconstruct per-link metrics (such as latency and packet loss) in a network, given measurements from a limited number of monitors. Fundamentally, this is a matrix inversion problem. For this problem to be solvable, however, the topology and routing paths of the network must be known before-

hand [37, pp. 1–2]. While *network topology tomography* aims to reconstruct the network topology using measurements from monitors—and could theoretically help quantify the performance of individual links when the topology is not known—this method can only recover a logical topology that is equivalent performance-wise to the physical topology [37, pp. 218–220].

Monitoring systems are usually categorized into two groups depending on the way they obtain data: active and passive monitoring [33], [38]–[41]. Their scopes do not completely overlap and could therefore be considered complementary.

2.4.1 Active Monitoring

Active monitoring uses end-to-end measurements or "active probes" to estimate performance characteristics, such as the packet loss and latency, of links in the network [40] (or sometimes even to infer the network topology). It does so by producing and sending its own synthetic packets to endpoints in the network to ascertain network conditions. An example of this is the ubiquitous *ping* utility that traditionally sends "ICMP echo requests," to which target hosts respond with "ICMP echo responses." Active monitoring typically produces results in real-time.

The use of active monitoring can be expensive in terms of the equipment necessary to manage the increased load across the network. Additionally, some resources are also consumed by end hosts actively generating and receiving this traffic [41]. With this in mind, it is common when using an active monitoring approach to design systems that can ascertain as much information about network conditions as possible while applying a small amount of strain to the network. This is however made harder by the fact that a vast majority of active monitoring applications have scalability limitations on large managed networks, due to many systems being centralized and/or having to actively probe every node [40].

In data centers, *equal-cost multi-path* (ECMP) routing is prevalent [9]–[11], [42], [43]. When there exist multiple optimal routes to a destination with equal routing priorities, networking equipment with ECMP enabled load-balance incoming packets among the relevant outgoing interfaces by applying a hash function on part of a packet—most often the packet header. This may however prove problematic for the detection and localization of failures since the hash functions may be proprietary and/or incorporate randomness [11]. In the literature, this has been overcome by either ensuring header fields (on which the hash function operates) in probes are alternated by changing port numbers [36] such that all paths are "probably" covered, or by "steering" probes by encapsulating them in IP-in-IP packets destined for supported switches which decapsulate the probes [33]. Of these methods, the latter requires functionality that may not be available in all environments but provides better precision than the former.

2.4.2 Passive Monitoring

With passive monitoring, the monitoring application itself does not inject new packets onto the network. Instead, such solutions rely on listening to existing network traffic [41]. Thus, while active monitoring schemes can guarantee theoretical full coverage of all paths in a network, passive monitoring instead only interact with paths carrying "real" network traffic. A few of the advantages of this approach are that it does not add additional load to the network, it presents data of actual user experiences as opposed to synthetic packets, and low-rate errors are not being masked by ECMP routing [33].

Contrary to the active solutions with packets purpose-crafted for performing specific measurements, passive solutions rely on generic network traffic. For this reason, measurements may not be as precise. Since capturing and analyzing all network traffic will lead to more data to process, many implementations adapt by instead relying on analyzing accumulated statistics or listening to certain events. However, in some instances, the act of passively monitoring network traffic will produce statistics not accessible by active solutions—most notably the current network throughput.

2.4.3 Metrics

This section presents several different metrics of interest when monitoring the performance of a network. It focuses primarily on metrics relevant to the measurements taken in this thesis. Other examples of metrics not discussed in this section are packet corruption rate and throughput.

Round-Trip Time (RTT)

The Round-Trip Time (RTT) is the duration from a request being sent by a client, until the client receives a response from the target. Measuring the RTT is often an uncomplicated process in the case of active monitoring and can be performed manually with the `ping` utility. Passive solutions may acquire the RTT by listening to existing TCP traffic and are used in, for example, the networking stack of the Linux kernel to estimate a socket's RTT.

One-Way Delay (OWD)

The One-way Delay (OWD) is similar to that of the round-trip time in principle, except we are only interested in the delay from when a sender transmits a packet until the receiver receives it. RFC 7679 [44] describes motivations for measuring the OWD instead of the RTT. One motivation is asymmetric queuing, meaning that the forward and reverse paths experiencing different levels of congestion. Another reason is and asymmetric paths, implying that the forward and reverse paths are not physically identical due to routing differences or asymmetrical link capacities. Thus, while the OWD often is approximated as $RTT/2$, it may be undesirable for some types of measurements.

Accurately measuring the OWD, however, is not straightforward in practice. While

measuring the RTT can be done solely with the clock of the sender (assuming limited clock drift), measuring the OWD requires the clocks of the sender and receiver to be synchronized [44]. Time synchronization is a classical problem in computer science and can either take place via special-purpose hardware such as radio/satellite receivers and specially designed networking equipment or via clock synchronization algorithms that fundamentally rely on RTT measurements. The latter method may not always be appropriate for OWD measurements in the case of asymmetric latencies [45]–[47].

Packet Loss

Packet loss refers to the event where a packet has failed to reach its target destination either due to being dropped or becoming malformed when traversing the network. Packet loss is measured as a percentage of lost packets with respect to the number of packets sent.

One-Way Packet Loss

One-way packet loss is similar to regular packet loss except that we only focus on the loss between a sender and a receiver in one direction, instead of the overall round-trip loss. In RFC 7680 [48], motivations for using one-way packet loss are similar to those of OWD.

With the interest of monitoring different types of network partitioning, the use of round-trip packet loss is insufficient as a simplex partitioning (illustrated in Figure 2.5c) exhibits traits of network loss only affecting one direction. Two-way measurements may not be able to determine in which direction a loss occurred.

2.5 Related Work

In this section, we present previous systems implementing distributed end-host network monitoring. We divide the works into active and passive solutions depending on whether failures are identified by introducing new probe packets or listening to existing network traffic, respectively. A summary of the works are presented in Table 2.1.

2.5.1 Active Monitoring

Pingmesh [9] is an application measuring end-to-end latencies between servers in Microsoft datacenters. Pingmesh agents are deployed on all servers and follow a "pinglist" decided by the Pingmesh controller. The controller considers a hierarchy of graphs. The first level of the hierarchy consists of graphs where vertices represent all servers under the same top-of-rack switch. The second level of graphs, in turn, consists of vertices representing all top-of-rack switches in a data center. The last level is a single graph where each data center is a vertex. The pinglist is constructed such that, for each graph in the hierarchy, the probe traffic between the vertices forms a complete graph.

Table 2.1: Overview of related work.

Name	Type	Probes
Pingmesh [9]	Active	TCP (HTTP)
NetBouncer [43]	Active	UDP/IP-in-IP
NetNORAD [42]	Active	UDP
PTPmesh [36]	Active	UDP (PTP messages)
deTector [33]	Active	UDP
007 [11]	Passive	N/A (TCP/ICMP for localization)
Roy et al. [10]	Passive	N/A
NetPoirot [49]	Passive	N/A
Everflow [32]	Passive	Any

NetBouncer [43] is another application deployed in Microsoft data centers, focusing on localizing links with an elevated packet loss ratio. Probes are encapsulated in IP-in-IP packets destined for a target switch, which in turn decapsulates the packet and "bounces" the probe back to the sender. Due to the datacenters' architecture, only one path exists to an upper-level switch (although this theoretically could be generalized upon by encapsulating packets more than once). A central *controller* constructs a probing plan in such a way that a single faulty link can—with the probing data from individual nodes—be identified by a central *processor*.

NetNORAD [42] is an open-sourced application developed for use in Facebook datacenters, measuring both the packet loss ratio and latency for various QoS classes. A processes referred to as the *reponder* is deployed on all end hosts, while *pinger* processes are only deployed on relatively few to reduce resource consumption. Metrics are aggregated by "proximity tags" in a hierarchical fashion by rack clusters, data center, and region. Localization of faults is thus relatively coarse-grained compared to other solutions and relies on comparison of proximity tags.

PTPmesh [36] explores the idea of using the Precise Time Protocol (PTP)—a clock synchronization protocol—and the open-source implementation PTPd to measure network conditions in cloud data centers. The findings are that, after a five-minute convergence period, the one-way delay (OWD) can be measured by PTP. By introducing an asymmetric load from the slave to the master clock using the `iperf` tool, the OWD in that direction is increased. However, due to how PTP continuously synchronizes clocks (as briefly discussed in Section 2.4.3, this type of synchronization is fundamentally dependent on RTT measurements), the OWD in the opposite direction "appears" to increase as the clock offset changes. This makes the measurements unreliable after some time of heavy asymmetrical congestion in terms of finding the actual OWD. Delay and packet loss measurements are performed and analyzed at three large cloud providers. Despite its name, PTPmesh operates in a master-slave configuration.

The purpose behind deTector [33] is to both detect and localize failures with a minimal number of probes, specifically claiming to do this more efficiently than Pingmesh

and NetNORAD. Using knowledge of the full network topology, a "pinglist" can be generated for each *pinger* by the *controller*. These lists—while minimizing the number of probes—also guarantee identifiability of up to β link failures while each link is covered by at least α distinct probe paths. Pingers send probes to *responders* using IP-in-IP encapsulation to impose a certain probe path. Lastly, data is sent to the *diagnoser*, which it stores and analyzes in order to find links exhibiting packet losses and an estimate of the loss rate.

2.5.2 Passive Monitoring

007 [11] takes a mixed approach, using passive monitoring for identification and active monitoring for localization of faults. It operates by passively monitoring for TCP retransmissions using the event tracing functionality built into Windows. When a retransmission is detected, the affected network path is discovered using a `traceroute` analogue utilizing identical packet headers to ensure ECMP issues are circumvented. Every link in the route is assigned a proportional "blame" value, which are tallied up by an analysis agent to find a faulty link. The advantage of this solution is that no changes in the network or host are needed to deploy this solution while still providing low overhead.

A solution described by Roy et al. [10] uses both knowledge of the network topology of Facebook data centers along with packet marking functionality of core routers to infer the path of a flow. Packet markings are read by end hosts using a custom kernel-level program implemented using eBPF sandboxing technology. By comparing OS-level TCP metrics of a flow with other flows that should theoretically be equivalent performance-wise, end hosts produce a verdict of whether a route is faulty. This also means that if at any moment the performance of all flows is degraded equally, no fault will be detected. This verdict is later sent to a central controller that can filter false positives and produce a final set of faulty network components.

In NetPoirot [49], TCP statistics are collected and run through a machine learning model that attempts to assign blame to either the server application, client application, or network. The focus is thus not restricted to network failures. By using supervised learning attained by the use of fault injection (of for example high I/O, memory or CPU load on the client/server, high disk latency, sporadic packet drops, random connection drops, and high latency), the model is trained offline before being deployed. However, the model has to be re-taught for every new application that it monitors.

Everflow [32] can be hard to categorize as strictly passive, but does nevertheless primarily depend on listening to existing traffic. Furthermore, it does not operate on end-hosts and is thus outside of the scope of this thesis, but deserves a mention due to its similarities with other previous work. Having been designed for and deployed in Microsoft datacenters, a subset of packets is captured, mirrored and encapsulated by supported switches. This subset consists of TCP segments with the SYN, FIN, or RST flags set, every n packet in a flow, all network protocol traffic (e.g., BGP and PFC messages), and packets with an explicit "debug" flag

in the header. The encapsulated packets are forwarded to the processing pipeline's first component—the *re-shuffler*. Consisting of a network switch, the re-shuffler hashes the headers of the encapsulated packet in order to decide which *analyzer* to forward the packet to, ensuring that the same flow is always sent to the same analyzer. Analyzers are a set of distributed servers that reassemble packet traces and aggregate link-level statistics. If a packet trace includes a packet loss or loop, the full trace is stored. To circumvent the limits of passive tracing, the controller may send out "guided probes." For example, because switches typically do not possess timestamping ability, a guided probe can be used to find a link's RTT. Consider a packet instructed to traverse $S_1 \rightarrow S_2 \rightarrow S_1$. We note that because the delay of the path from S_1 and S_2 to the analyzer is unknown, the OWD cannot be measured. Instead, the analyzer measures the time between the two arrivals at S_1.

2.5.3 Our Work

The work done in this thesis consists of two separate solutions—one based on active probing and one on passive listening. Unlike many of the related works presented, our application will not exploit any preexisting knowledge of the network topology. The only other works with this limitation are 007 [11], which relies on passive listening to detect failures, and PTPmesh [36], which currently only operates in a master–slave configuration and—contrary to its name—is not organized in a mesh. Furthermore, none of the systems based on active probing can differentiate between one-way and two-way packet loss, which our implementation based on active probing can. This functionality is needed when detecting simplex partitionings. Finally, none of the other implementations presented in this section have the tooling to identify partitions.

3

Implementation

In this chapter, we will describe the implementation of a proof-of-concept monitoring system that is able to monitor one-way packet loss and identify the three partitioning failures presented in Section 2.3.2 (complete partitions, partial partitions, and simplex partitions). First, a general overview of the system and general design considerations are presented, followed by a more detailed description of its "pinger" and "visualizer" components.

3.1 Overview

The implementation of the proof-of-concept is based on three distinct subsystems able to be containerized, as well as Kubernetes' service discovery. The *pinger* either sends and receives "ping" messages to/from other pingers and/or listens for TCP retransmissions—and records those events. *Prometheus* is an off-the-shelf event monitoring system, pulling the recorded data from all pingers and storing it in Prometheus' time series database. The *visualization* component can in turn pull data from Prometheus which it can visualize, as well as analyze in order to find partitioning events. Kubernetes' service discovery is presented by the Kubernetes API, used for presenting the IP addresses of the active pingers used in the Pinger and Prometheus subsystems to discover pods to ping and scrape data from respectively.

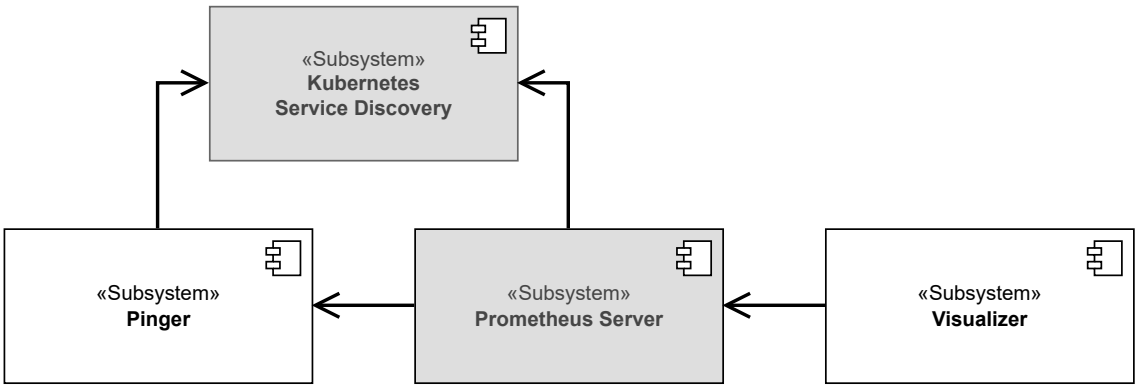


Figure 3.1: Overview of the subsystems. The *pinger* and *visualizer* subsystems are implemented as part of this thesis.

The system components and how they interact with each other can be seen in Figure 3.1.

3.2 Pinger

The "pinger" subsystem consists of two distinct parts—the active pinger and the passive monitor. By contacting the Kubernetes API, other pods also running the pinger can be acquired. From this, each pinger is able to identify targets to ping or monitor for TCP connections from/to.

3.2.1 Design Considerations

There are a few requirements when designing the application. Most of these requirements arise from the fact that the application needs to work in any Kubernetes cluster. In particular,

- the network topology is not known beforehand,
- the configuration or character of network devices is not known beforehand, meaning that approaches utilizing `traceroute`-like approaches such as [9] may not produce a consistently satisfactory result,
- the application should run and be trivially deployable on any Kubernetes cluster.

While active network latency measurements may be improved with the help of frameworks such as DPDK, it would mean traffic in part bypass the kernel's and Kubernetes' networking stack. Since the goal is to monitor failures emanating not only from the network—but the entire CaaS platform—using frameworks that bypass the networking stack would be counterproductive.

Programming Language

In order to decide on the programming language to use for the pinger, the performance of three different programming languages were compared. While not precise, these tests should provide an indication of practicality from a performance standpoint. A small prototype of the final proof-of-concept built upon active probing is implemented in each language, while ensuring each prototype maintains function parity. Specifically, each prototype is a single executable that is able to:

1. use a static number of processes and threads,
2. ping multiple processes simultaneously using a single UDP socket,
3. store a timestamp of the time at which a ping message is sent,
4. respond to ping messages from other identical processes with pong messages, and

Table 3.1: Latency measurements (50th, 90th and 99th percentile) for the prototypes. Tests were performed on the same machine.

Langauge	Median	p90	p99
Python	0.31 ms	0.31 ms	1.06 ms
Java	0.21 ms	0.28 ms	0.33 ms
C++	0.12 ms	0.14 ms	0.39 ms

5. via the standard output either print the latency if a response arrives or indicate a packet loss if a response is not received within a predefined timeout.

The test is conducted using two processes with identical binaries on the same machine. One process sends ping message to the other process every second, to which it responds to with pong messages. The distribution of the resulting latencies from the comparison are presented in Table 3.1.

The greatest performance in the 50th and 90th percentiles was obtained using C++. While a scripting language such as Python might be beneficial due to its simplicity, performance in the 99th percentile indicates that significant jitter is introduced. Thus, C++ is the language chosen for implementing the final PoC.

3.2.2 Active Measurements

Due to the earlier described design constraints, active measurements are done in a full-mesh configuration to ensure full coverage of all network paths.

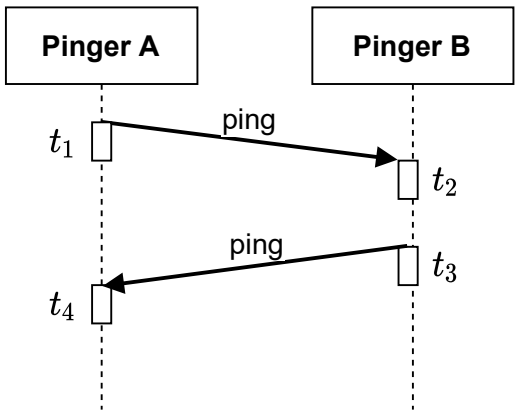


Figure 3.2: Normal operation of two pingers.

An overview of how the different pingers interact with each other can be seen in Figure 3.2. In the figure, one can see how pinger A sends a probe to another pinger B at time t_1, after which pinger B reports that a message is received at time t_2. In this case, pinger B later independently sends a probe at t_3 in the opposite direction to pinger A.

One-way packet loss measurements are performed in such a way that both nodes do not need to be able to both send and receive probes—only that there is at least

one pinger that is able to send probes and another pinger that can receive these probes. In other words, replies are not necessary. As such, we describe the process of sending and receiving probes separately.

Sending Probes

Every node sends UDP datagrams in the form of *pings* to all other nodes each time step. This implies a message complexity of $O(n^2)$, where n is the number of nodes. The period of a time step—how often a probe is sent—can be changed. A longer period will utilize fewer system resources, but also increase the amount of time required to detect low-frequency partial packet loss.

In order to find which nodes to ping, a discovery mechanism is needed—of which the pinger has two. The first relies on occasionally querying the Kubernetes API to find other pods running the pinger. This is needed since Kubernetes is a dynamic environment where pods can be removed, added or even restarted which will alter their current IP address. The second is based on a static list of socket addresses passed as command line arguments, potentially enabling the pinger to be run outside of Kubernetes as a stand-alone application.

The statistic sent to Prometheus regarding probe transmission is

- the number of probes sent for every sender–receiver pair.

Receiving Probes

Based solely on the received pings, every pinger node reports two pieces of statistics per transmitting node to Prometheus, namely:

- the number of received pings, and
- the number of pings that are assumed lost.

While the former statistic of the two is relatively straightforward to measure, the latter relies on detecting gaps in the sequence numbers received. An example of this is illustrated in Figure 3.3, where the message with sequence number 3 is received, even though a message with sequence number 2 is not. Thus, a certain time after message three is received, the receiving pinger can assume that message two is lost. While this could be accomplished with timers, it is difficult to determine a fair timeout since transmission times of probes vary and the pingers' clocks are not synchronized. Instead, we decide that the datagram with sequence number n is considered lost when receiving a datagram with sequence number m, where

$$m \geq n + \left\lfloor \frac{T_{TO}}{T} \right\rfloor + 1.$$

In this case, T signifies the period at which packets are sent and T_{TO} is a lower bound for the timeout.

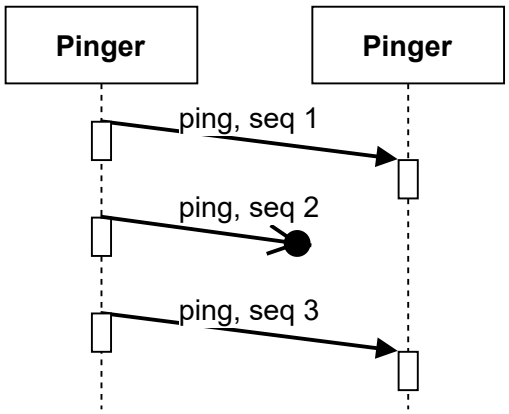


Figure 3.3: Lost ping message. The loss can be inferred by the receiver by comparing the sequence numbers.

Latency Measurement

While network latency measurements are not required in order to detect packet loss, the pinger does present these measurements to Prometheus if available. To enable this, each active probe contains an additional two fields containing high precision timestamps. These fields are (1) the timestamp at which the probe was sent and (2) the time difference between when the probe was sent and when the most recent probe arrived from the recipient, subtracted from the first field in the most recent probe from the recipient. Thus, one-way packet loss measurements need to be enabled bidirectionally (i.e., both from pinger A to B as well as from B to A).

To illustrate a latency measurement, we consider the execution in Figure 3.2 containing four highlighted events, which occur at times t_1 through t_4. First, note that instead of measuring a RTT as $t_4 - t_1$ by assuming $t_3 - t_2$ is sufficiently small (i.e., pinger B responds immediately to probes), we can instead calculate the network transmission delay as $(t_4 - t_1) - (t_3 - t_2)$. At t_1, pinger A sends a probe containing $\left[t_1, 0\right]$. The second field is set to zero because no previous probe from B to A exists. At t_2, pinger B stores t_1 and t_2. Later at time t_3 (occurring exactly one time step from the last transmission of a message from B to A), the times t_1 and t_2 are popped from the temporary store and a probe containing $\left[t_3, t_1 + (t_3 - t_2)\right]$ is sent. When it arrives to pinger A at time t_4, a latency measurement of $t_4 - (t_1 + (t_3 - t_2)) = (t_4 - t_1) - (t_3 - t_2)$ is reported to Prometheus. We note that this method will work even when the clocks of A and B are not synchronized. Also note that at t_3, another measurement is started by pinger B equivalent to that of node A at t_0. Thus, at time t_4, t_3 together with t_4 is temporarily stored by pinger B. Interestingly, due to each UDP datagram being used for up to two different latency measurements, the amortized number of messages per measurement will converge at one message—given that the time steps at which the pingers send messages are identical.

3.2.3 Passive Measurements

For the sake of comparing a solution based on active probing with one based on passive listening, the pinger application is augmented to also passively listen for network traffic.

Instead of sending and receiving UDP probes—as in the case of the active monitoring—socket-level TCP statistics are polled from the Linux kernel's networking stack. This is similar to the method used by Roy et al. [10] to collect statistics. In our case, the statistics are sampled and collected at a predetermined interval using the `ss` utility (which is similar to the `netstat` utility).

Before sending data to Prometheus, socket statistics are grouped together by the sending and receiving pod. In particular, the statistics collected for each sender–receiver tuple are:

- the number of segments sent,
- the number of bytes sent,
- the number of segments received, and
- the number of retransmitted segments.

We note that the number of retransmitted segments may not correspond to the number of lost packets.

Since containerization implies that programs are isolated from each other, the pinger needs additional privileges in order to access the networking namespaces used by other pods on the node. As such, the pinger containers are configured in Kubernetes to run in "privileged mode". However, we note that this option may not be available on managed CaaS platforms since this breaks down the container sandbox.

3.2.4 Software Architecture

The Pinger is comprised of several software components, each of which is in charge of handling certain functionalities within the system. An overview of the parts of the software architecture used for active probing in the form of a UML class diagram can be seen in Figure 3.4. The architecture of the part responsible for passive monitoring is mostly similar.

The core component in the system is the `Healthcheck` class. It is tasked with initializing the socket for communication and to send and receive pings. To accomplish this, it relies on a `PingStore` object to remember the sent and received sequence numbers for each peer, as well as the timestamps needed for latency calculation described in Section 3.2.2. In order to discover the IP address of other pingers, an instance of a class extending the `Discovery` interface is also provided. This can either be a `StaticDiscovery` object providing a static list of socket addresses

defined in the startup arguments, or a `KubernetesDiscovery` object continuously communicating with the Kubernetes API in order to always provide an up-to-date view of the current pods. Finally, the `Reporter` class is responsible for reporting statistics. The base class only prints the statistics to the standard output. This behavior is extended by the `PrometheusReporter` class, which also reports them to a dynamically linked Prometheus client.

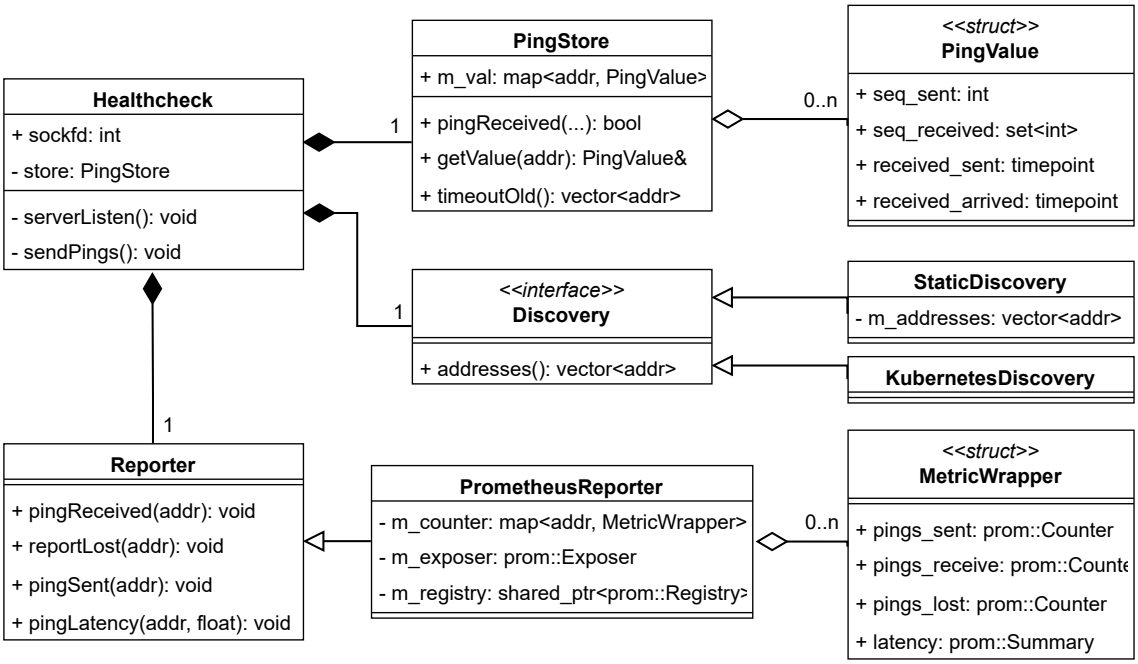


Figure 3.4: UML class diagram of the pinger.

3.3 Visualizer

The visualizer is the application responsible for visualizing the collected data. Additionally, it also analyzes the data in order to find and alert about network partitions. It is a web-based application built in TypeScript using the React[1] library. Data is fetched from Prometheus using its HTTP API.

3.3.1 Visualization

An important part of understanding how a cluster and its applications are affected by a partitioning is to visualize the state of the network, which is accomplished by the use of the Sigma.js[2] graph drawing library. A screenshot of the visualizer can be seen in Figure 3.5, to which we will refer to multiple times in this section.

The visualizer has three different views: the *active view*, the *passive node view*, and the *passive pod view*. The view can be selected in a drop-down menu (Figure 3.5g). In both passive views, the option to filter by Kubernetes namespace appears. In

[1] `https://reactjs.org/`
[2] `https://www.sigmajs.org/`

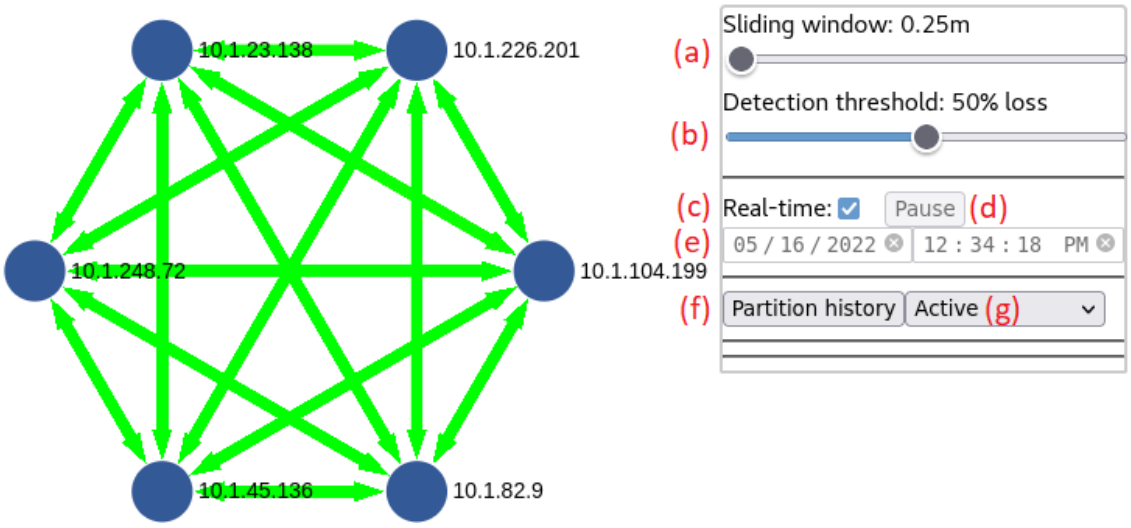


Figure 3.5: Visualization of a "healthy" Kubernetes cluster with six nodes.

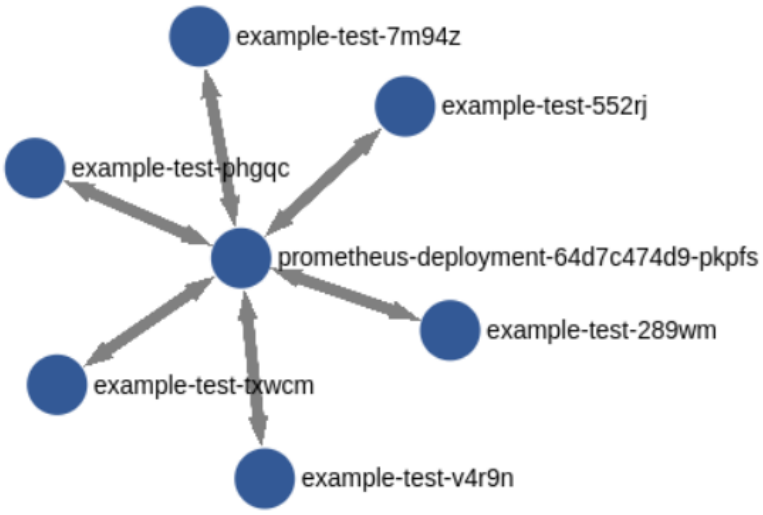


Figure 3.6: Visualization of a Kubernetes namespace using the "passive pod view".

the active and passive node view, each physical node monitored by our application is drawn as a node in the graph placed in a circle. In passive pod view, however, each Kubernetes pod is instead drawn as a node and placed by the ForceAtlas2 algorithm [50]. An example of the passive pod view—filtered by the namespace in which our application is deployed—can be seen in Figure 3.6. Directed edges between graph nodes represent the flow of ping messages or TCP segments.

As the level of packet loss or retransmission increases between the nodes in the graph, the hue of their related edges will transition from green to red. When an edge is red, it is considered "unhealthy." This color transition can be performed smoothly using the HSL (hue, saturation, lightness) representation of the RGB color model

by fading the hue from 120° to 0°. Thus, the color of an edge can be described as

$$HSL = \left(\max\left\{0,\, 120 \cdot \left(1 - \frac{L}{T}\right)\right\},\, 100,\, 50\right)$$

where L is the packet loss rate and T is the detection threshold. In the user interface, the slider called "detection threshold" (Figure 3.5b) decides the proportion of packet loss necessary for an edge to be considered "unhealthy." The screenshot in Figure 3.5 shows a healthy cluster of six pingers. Note that the bidirectional arrows each consist of two independently drawn half-length unidirectional edges. Additionally, two special colorings exist—gray edges representing no probes sent and black edges representing missing data due to unresponsive pingers.

Two parameters can be set in the user interface to adjust the time series used for visualization: the evaluation time of the query (Figure 3.5e) and the duration of the sliding window (Figure 3.5a). Only information that falls in-between the evaluation time and a certain number of minutes back, as indicated by the sliding window selector, is considered when drawing the graph. If the checkbox named "Real-time" (Figure 3.5c) is checked, the evaluation time is continuously updated to reflect the current time.

When clicking on one of the edges in the active view, a modal appears containing the latency measurements between the given pair of nodes, as shown in Figure 3.7. The latency is presented as different percentiles.

3.3.2 Finding Partitions

Since our assumption is that all instances of network partitioning is characterized by the loss of packets sent between nodes, data scraped from our application can be used to infer partitioning events.

Using the criteria for the three types of partitionings we defined in Section 2.3.2, three functions are implemented that each test the criteria of one type of partitioning on a given graph. Since we assume a full-mesh topology, a naive approach will be sufficient. As such, the graphs in question are identical to those used for visualization (e.g. as shown in Figure 3.5), with the difference being that edges considered "unhealthy" (as decided by the "detection threshold" slider) are removed.

Partitionings are presented in two ways—by a message in the sidebar if the currently viewed state of the cluster contains a partitioning, or as an entry in the table of historical partitionings (reachable by a button in Figure 3.5f). Historical partitionings are found by querying Prometheus for discrete times (as defined by Prometheus' global evaluation interval) at which "unhealthy" links are present. By applying the three partition tests at these times, we can find the type of partitionings present. Lastly, to decide the duration of a partitioning event, consecutive time steps where partitionings occur are merged.

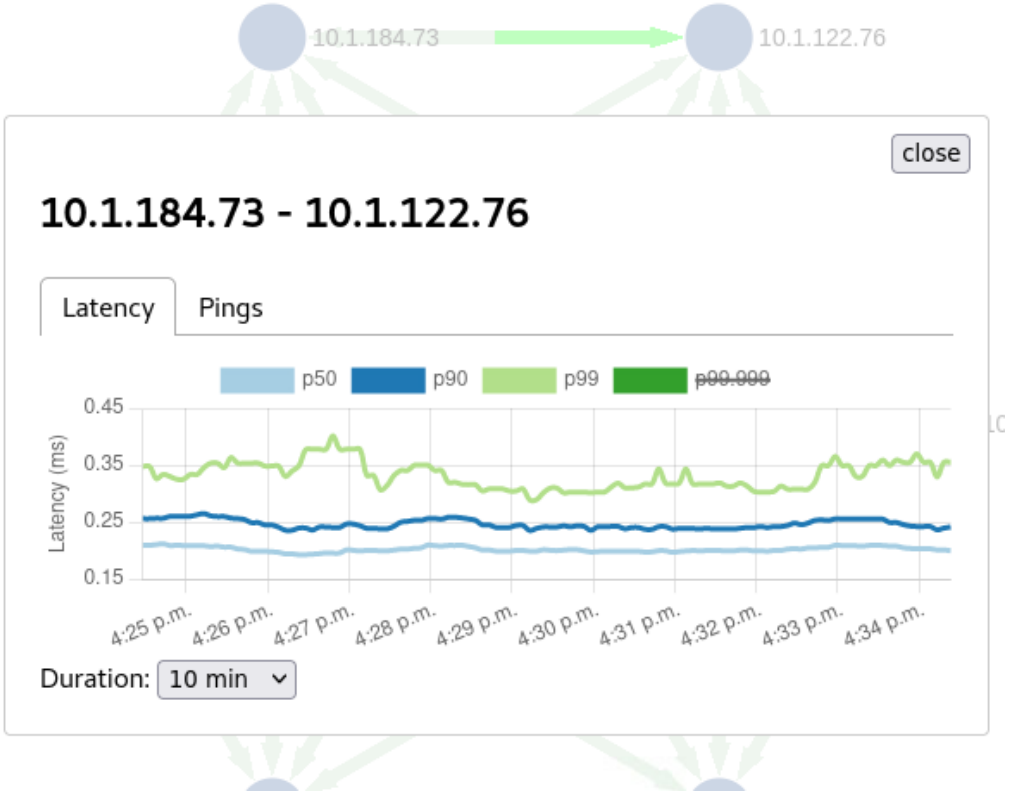


Figure 3.7: Latency between a pair of nodes for a few percentiles as shown by the visualizer.

3.3.3 Data Collection Delay

There are multiple stages that theoretically induce a delay between a 100% packet loss failure taking place and it being reflected in the visualizer. Save for the length of the visualizer's sliding window, these stages are illustrated in Figure 3.8. Thus, we only consider the delay until the first loss/retransmission is received by the visualizer here. In the figure, all the known delays depicted in parentheses can be configured, but currently show default values. We consider program execution delay and network delay to be negligible.

The delay-inducing stages up until the pinger has acknowledged a loss or retransmission differ depending on whether active or passive monitoring is being used. When using active probes, we specifically have to consider the interval between each probe and the timeout period of the probes. Indeed, in the worst case it may take up to the sum of those two durations until the pinger assumes a packet loss has occurred. When passively listening, we remark that the interval between segments as well as the criteria for a retransmission is not in our control, ultimately depending on the containerized applications and TCP stack. Statistics are however being scraped from the kernel by the pinger at a set interval.

When the pinger is made aware of a statistic, it is immediately sent to its bundled Prometheus client. The Prometheus server scrapes the clients' HTTP endpoint

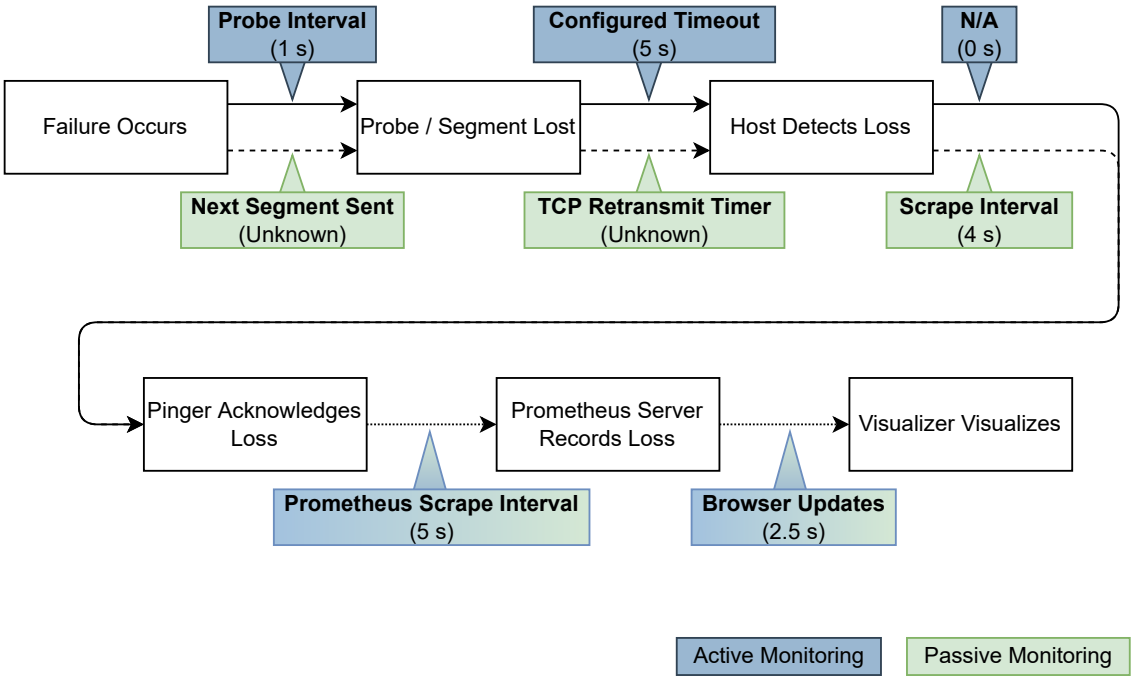


Figure 3.8: Steps which contribute to the delay between a 100% loss event and the first loss being received by the visualizer. Durations in parentheses represent default values.

with a predetermined interval. After this, the visualizer will pull data from the Prometheus server with its own interval. With the default configuration, the worst-case delay until a failure is propagated to the visualizer is 13.5 seconds for active monitoring and—excluding the unknown factors—11.5 seconds for passive monitoring. Assuming faults occur at random times, the average delays are 6.75 and 5.75 seconds, respectively. However, a single loss/retransmission does not warrant the visualizer to assume a link is unhealthy and a partitioning has occurred, meaning the duration of the visualizer's sliding window must be added on top of this.

4

Evaluation

This chapter presents the basis of how we evaluate the performance of our proof-of-concept monitoring system. The first section presents an overview of the chaos testing tool used and how it injects network disturbances into Kubernetes. The following sections cover the different test experiments used to obtain quantitative data. This data is then used to evaluate the performance of the monitoring system.

4.1 Chaos Testing

To ensure that the system can measure metrics such as latency and packet loss while also being able to identify the three types of network partitioning failures presented in Section 2.3.2, we need to be able to simulate disturbances that characterize these types of faults in the network.

Chaos engineering tools (as explained in Section 2.2.3) are usually used to inject faults in order to determine if a system in production and its services are resilient to failures. In this case, we are not so interested in the resilience of a production system and its services, but rather to inject network chaos to ensure that the monitoring application measures the changes in network performance and asserts whether a partition has occurred. For this purpose, we opted to use Litmus to simulate network disturbances for Kubernetes.

With the Litmus toolset deployed in the cluster, it is possible to orchestrate and run experiments. Litmus offers a wide selection of pod-to-pod level experiments such as packet duplication and increased latency. In this case, we only focus on latency and packet loss as well as how these can be used to simulate network partitions. To inject packet loss targeting a specified pod-to-pod link, Litmus uses the `tc` utility (traffic control) to add `netem` (network emulator) rules. The `tc` utility is used to configure the traffic control facilities of the Linux kernel [51], while `netem` consists of the kernel component that provides network emulation functionality. To decide whether to drop a packet affected by random packet loss, a Bernoulli test using a provided probability is performed. Litmus allows for adjustment of parameters such as the probability of random packet loss, the duration of the test experiment, the ramp-up time before the chaos is injected, and which Kubernetes pods to target.

4.2 Control Simulation

The motivation behind this experiment is to find the viability of active and passive monitoring for estimating the frequency of packet loss in a healthy cluster. A comparison is performed using the `ping` utility from the iputils[1] package. Furthermore, a comparison in latency measurements between the active monitoring solution and the `ping` utility is performed (the passive mode is not able to monitor latencies).

Each of the tools under evaluation measures packet loss in a different way—the active solution measuring one-way packet loss, `ping` measuring two-way packet loss, and TCP retransmissions measured by the passive solution depending on the applications using the network and the OS' network stack implementation. Multiple estimators can be used to infer elevated packet loss from passive data. In this case, we focus on two specific ones—namely the number of retransmitted segments as a proportion of transmitted segments as well as the number of retransmitted bytes as a proportion of transmitted bytes.

The experiment is performed twice. First when (1) no additional load is applied to the cluster and subsequently when (2) a synthetic load is applied to the cluster. No deliberate disturbances are added in either case. While no load is applied in the former case, the clusters used still have monitoring tools that generate some traffic. To generate a synthetic load that is characteristic of that demanded by the 5G, an in-house load testing tool injects synthetic 5G signaling traffic.

4.3 Induced Packet Loss

To test how well the application can estimate certain levels of packet loss, we (1) inject a specific percentage of random packet loss using Litmus over a period of time. When running the test, we (2) observe if the number of detected losses correctly converges towards the expected level of loss. In this case, we will be running experiments with both a 25% and 75% random packet loss over one hour.

The experiment will be repeated twice, once with and once without simulated traffic. Measurements will—similarly to the control experiment—be performed by both the active and passive mode of the application developed in this thesis, in addition to statistics gathered from the `ping` utility. From this, we compare the results from the two different monitoring solutions and `ping` in order to determine their level of accuracy when detecting packet loss.

4.4 Network Partition

In this experiment, different types of partitionings are simulated. Using both the active and passive probing mode of our application, we ensure that the visualizer (1) can accurately visualize the connectivity of the cluster and (2) automatically

[1] `https://github.com/iputils/iputils`

correctly identify partitionings. The visualizer is set to a sliding window of 15 seconds, and the state of the cluster is captured 30 seconds after the experiments are applied. The synthetic load from the control simulation (Section 4.2) is applied when testing the passive probing mode.

Loss of network connectivity between two nodes can be envisioned as a 100 % (or at least close thereto) packet loss on the path between them. Because involuntary loss of connectivity induces partitioning, we opt to use the same packet loss experiment outlined in Section 4.3 with some modifications. One of these modifications is to have 100 % packet loss to represent the complete loss in connectivity between pods or nodes. Additionally, loss is selectively applied to a subset of sender–receiver pods in order to simulate specific types of node-to-node level partitionings.

We focus on the three types of partitions described in Section 2.3.2 simulated by using the packet loss experiment—complete partitioning, partial partitioning, and simplex partitioning. The only difference between the different partition experiments is which pods to apply the partitioning to. For complete partitions, the nodes are divided into two groups of equal size—where connectivity within each group forms a clique, but the groups are disconnected. A partial partitioning is created similarly, save for the addition of a third group of nodes that communicates freely with the other two groups. Finally, in the case of simplex partitioning, ingress traffic is blocked for a node.

5

Results

This chapter presents the results of the experiments used to evaluate the overall performance of the monitoring application. The following sections correspond to the tests outlined in Chapter 4 and show the overall outcome of each experiment. Our application is set to send active probes once a second per link and scrape TCP statistics every four seconds. Thereafter, the Prometheus instance will scrape data from each client every five seconds.

5.1 Control Simulation

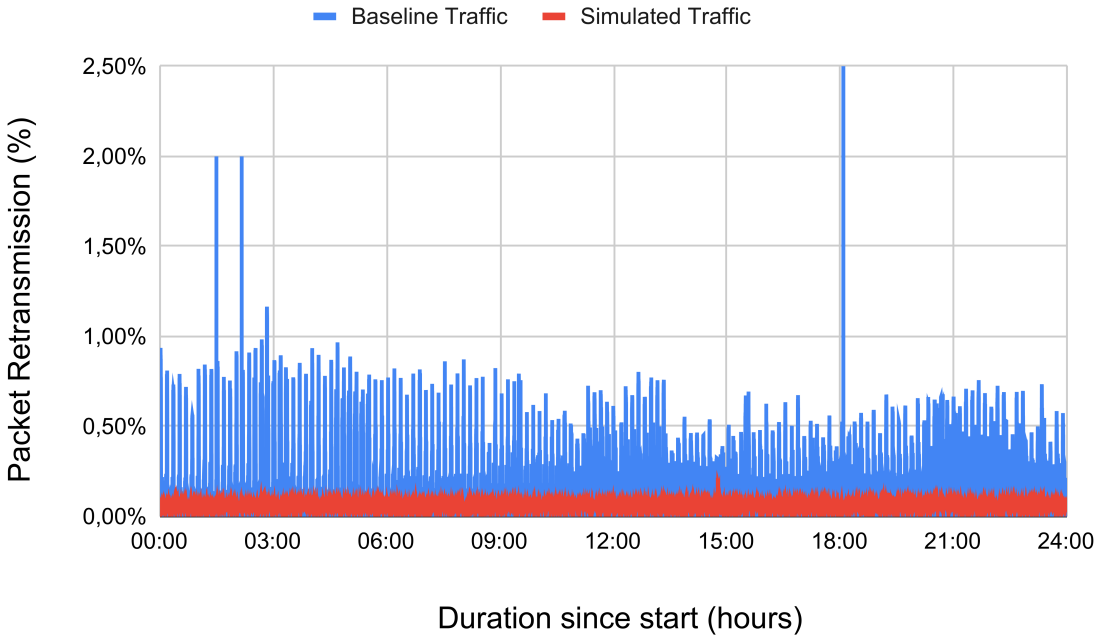
The result of the control simulation experiment, detailed in Section 4.2, is presented in this section. The following subsections cover the packet loss and latency results respectively.

The experiment was performed twice on a cluster with six nodes—once without additional load and once with the synthetic load. The duration of each experiment is 24 hours. In each case, only links originating from one of the six nodes is included (due to data collection and processing constraints arising from the `ping` utility). All graphs show per-link data calculated over a sliding window of one minute, which is then averaged over all links monitored.

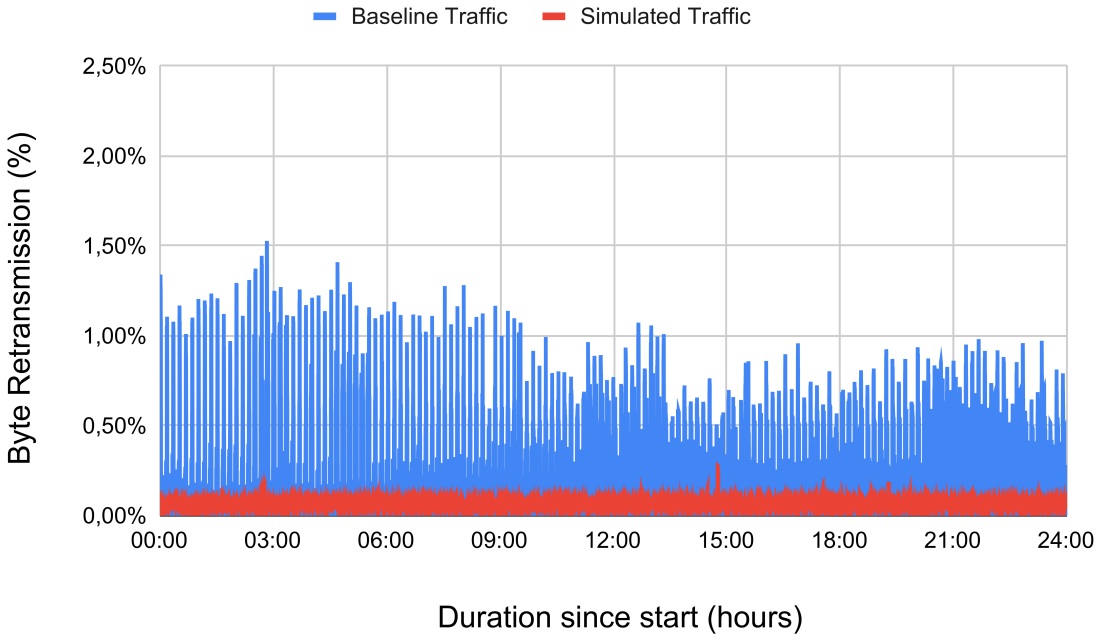
5.1.1 Packet Loss

Packet retransmission rates from passive monitoring, both in terms of the number of segments and aggregate payload sizes, are presented in Figures 5.1a and 5.1b respectively. Our application polls socket statistics every four seconds. As can be seen in the figures, both monitoring the number of segments and the sum of their sizes yield mostly identical results. When no load is applied (referred to as "baseline traffic"), the ratio of traffic retransmitted—while generally being much higher—also fluctuates heavily as compared to the case where simulated traffic is present.

In parallel to the passive monitoring, active monitoring also takes place using the active monitoring mode of our application and `ping`. Both tools send a probe every second to each receiver. The active mode did, however, not record a single case of packet loss on the links under inspection, even when running with simulated



(a) Retransmitted segments as a percentage of transmitted segments.



(b) Retransmitted bytes as a percentage of transmitted bytes.

Figure 5.1: TCP retransmission rates recorded by passive monitoring. The test is performed twice (once with no additional traffic and once with simulated traffic) over a 24-hour period on a cluster of six nodes.

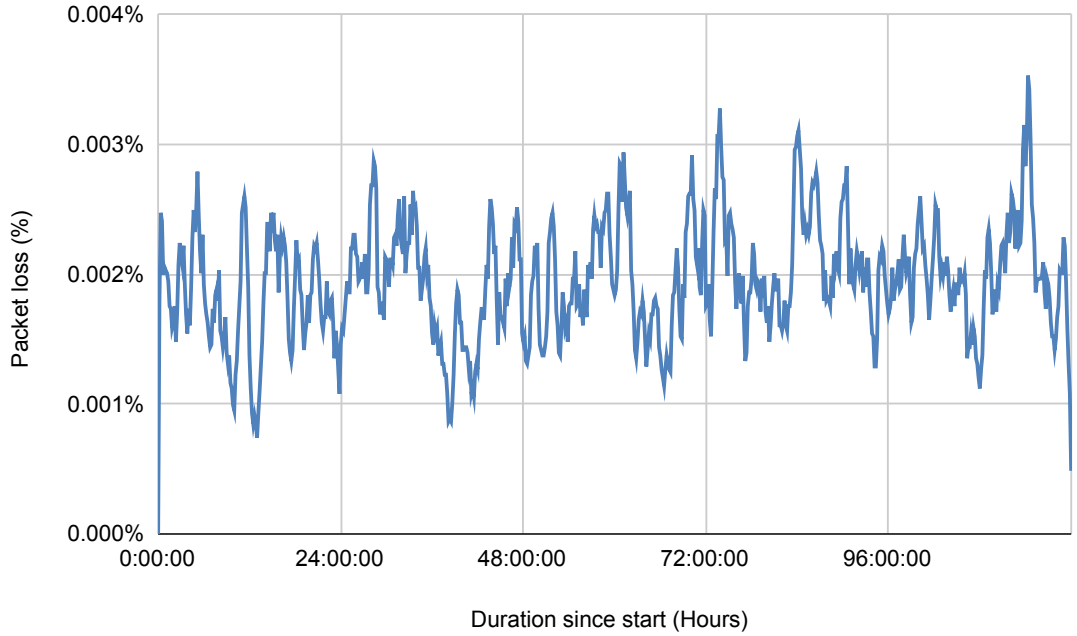


Figure 5.2: Packet loss over five days with a one hour sliding window on a larger cluster.

traffic. In a similar fashion, `ping` did not record any loss when running the baseline traffic. Although, when it comes to the simulated traffic, losses in the single digits were recorded. However, since these findings are not statistically significant and represent a fraction of the total traffic, this data is not illustrated.

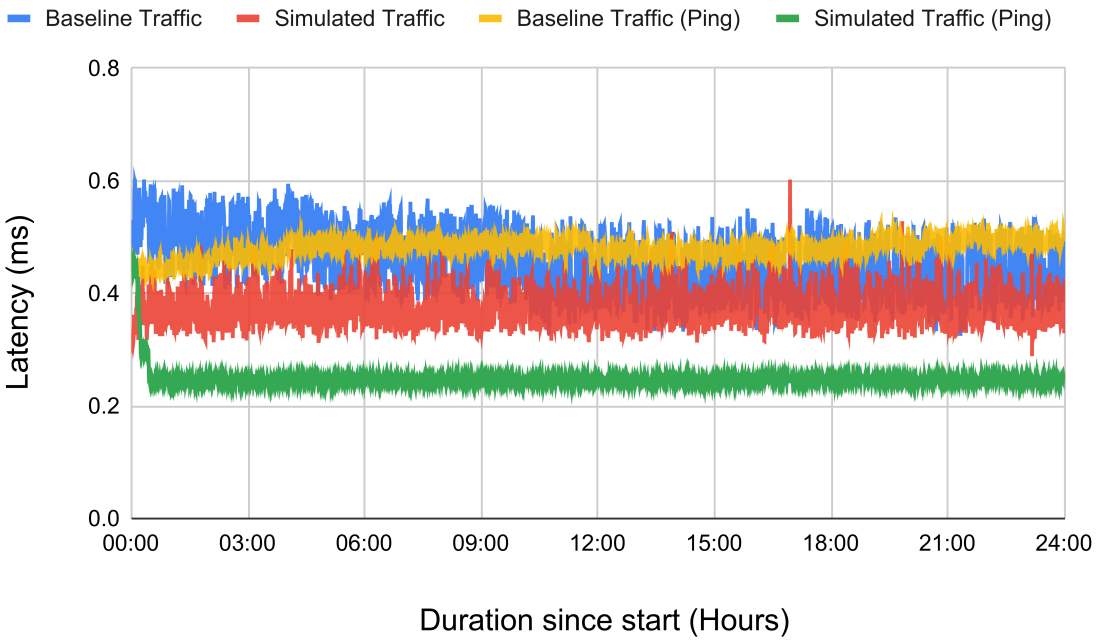
Loss In A Large Cluster

Since the results do not conclude a ratio of packet loss for the active monitoring solution, we perform a follow-up experiment. The experiment is performed using the same methodology, but using a cluster of twelve nodes, a probe periodicity of 100 ms per node–node pair, and a duration of five days. However, we only had the opportunity to test the cluster under simulated load. The result of this experiment is seen in Figure 5.2. In total, 11006 packet losses were detected, resulting in a packet loss of around 0.002%.

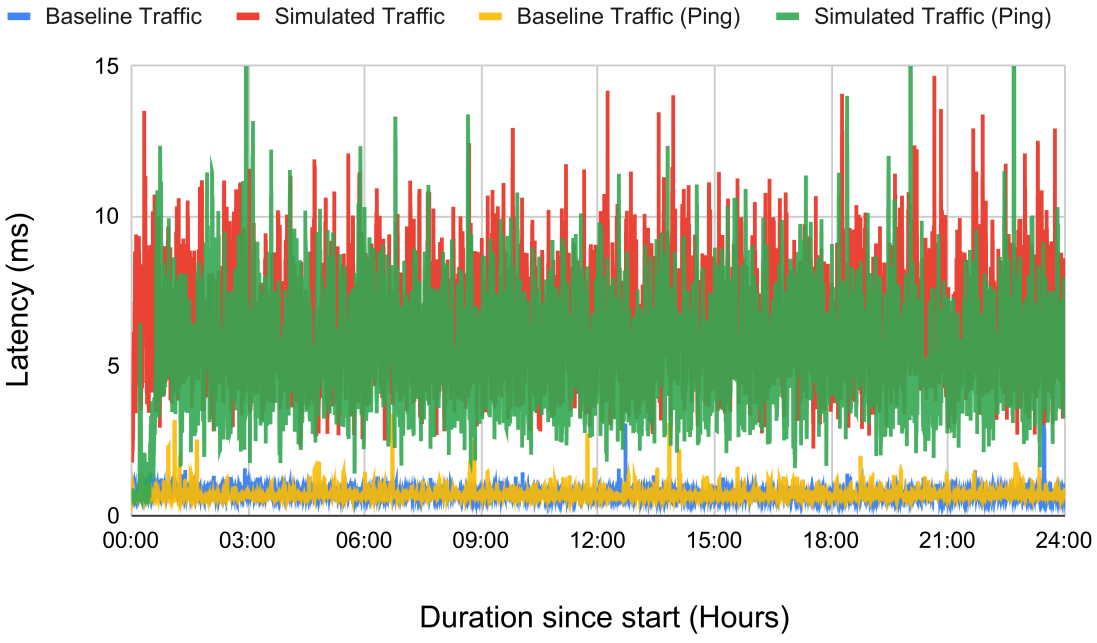
5.1.2 Latency

This section will cover the latency results of the control experiment. The latency for both the baseline and simulated traffic over the course of 24 hours are recorded by `ping` and our active monitoring tool. Latencies in the 50th and 99th percentile are presented. As mentioned in Section 4.2, the latency results do not include any measurements from passive monitoring.

The latency measurements obtained in the 50th percentile are visualized in Figure 5.3a. We observe that using our tool, the baseline latency lies around 0.45–0.5



(a) Average over the *50th percentile* (median) of each link.



(b) Average over the *99th percentile* of each link.

Figure 5.3: Active latency measurements by our application and `ping` over 24 hours. The experiment is performed twice—once with and once without simulated traffic.

ms whereas after injecting a synthetic load, the latency decreases and averages to around 0.35–0.4 ms. This behavior is amplified using `ping`, where the baseline latency of around 0.45—0.5 ms drops down to around 0.25 ms. We also observe that `ping` yields results with less minute to minute jitter.

Latency measurements in the 99th percentile can be seen in Figure 5.3b. Compared to the 50th percentile, the difference in latency between the baseline and simulated traffic is more substantial. Looking at the figure, the baseline latency for both our application and `ping` are within the range of 0.8–1.5 ms (excluding outliers). After introducing traffic, this increases to around an average of 6–8 ms. However, it is quite clear that the latency is volatile after introducing the load, as it often ranges from 3 to 14 ms.

5.2 Induced Packet Loss

This section contains the results of injecting random packet loss in a Kubernetes cluster of six nodes, following the methodology presented in Section 4.3. Using our tool, we record data using both the active and passive monitoring. In addition, we collect the same data using the `ping` utility. For the active monitoring, the statistics are calculated over a five-minute sliding window whereas the passive solution, data is calculated over a one-minute sliding window.

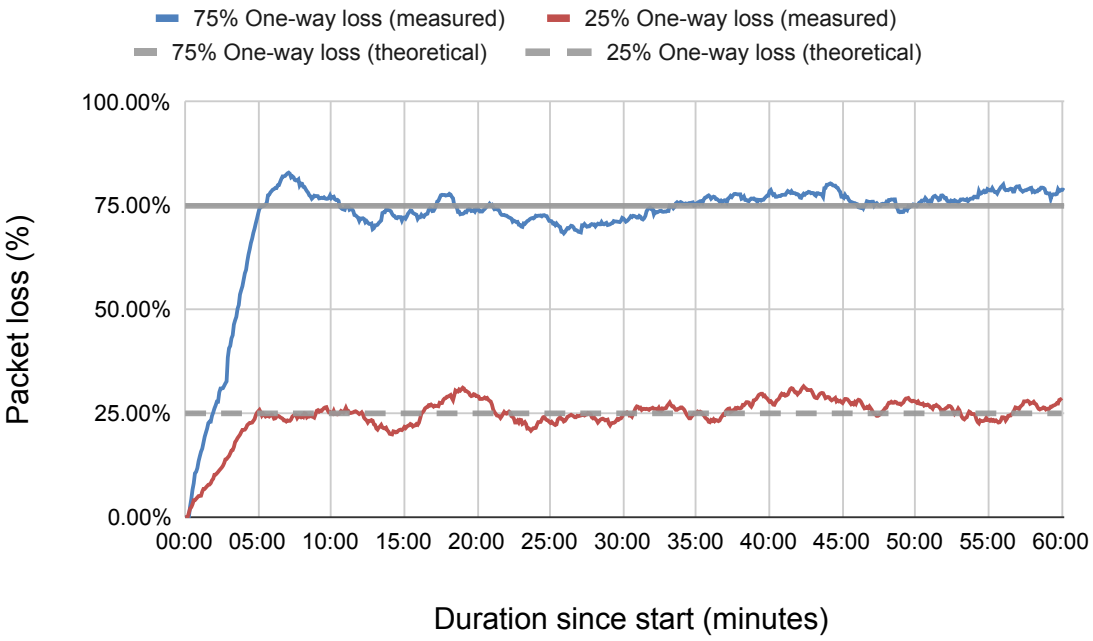
5.2.1 Active Monitoring

Figure 5.4a presents the results of actively monitoring using our application while injecting both 25% and 75% random packet loss on all nodes for an hour. Values are calculated over a five-minute sliding window and a probing interval of one second. When inspecting the results, we see the change in percentage of one-way packet loss for one of the $(n-1)^2$ links. Looking at the results we see that over the course of one hour, the packet loss corresponds to roughly the induced level of one-way loss.

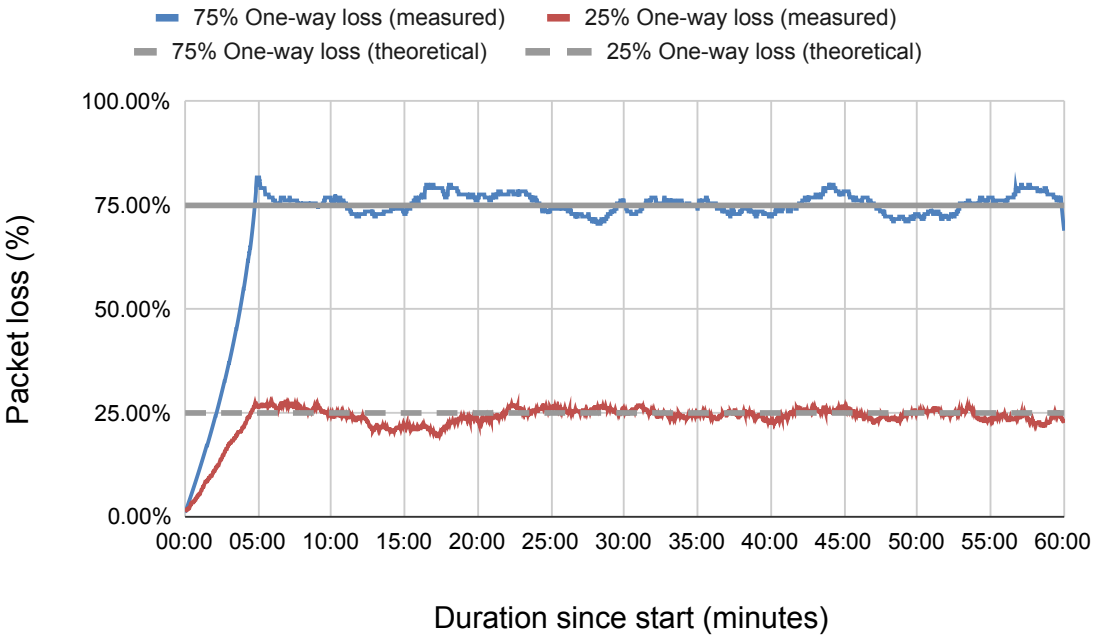
In a similar fashion to our active monitoring tool, `ping` measures the packet loss on the same link with a transmission interval of one second and a sliding window of five minutes. Because `ping` relies on the receiver responding with an "ICMP echo reply" message—which also can get lost—`ping` is only able to record two-way packet loss. With a one-way packet loss of 75%, only $(25\%)^2 = 6.25\%$ of probes make it back and forth whereas for a 25% one-way loss this results in 56.25%. However, if we assume the loss is symmetric, we can use $1-\sqrt{success\ rate}$ to estimate the one-way packet loss. This estimate is shown in Figure 5.4b. The induced one-way losses are also presented in the figures as a reference.

5.2.2 Passive Monitoring

This section covers results from passively monitoring a cluster experiencing elevated packet loss. This experiment is repeated with both a 75% and 25% random packet loss. The data—presented as both a ratio of retransmitted segments and a ratio of



(a) Packet loss actively measured by our application over one link.



(b) Packet loss measured by `ping` over one link.

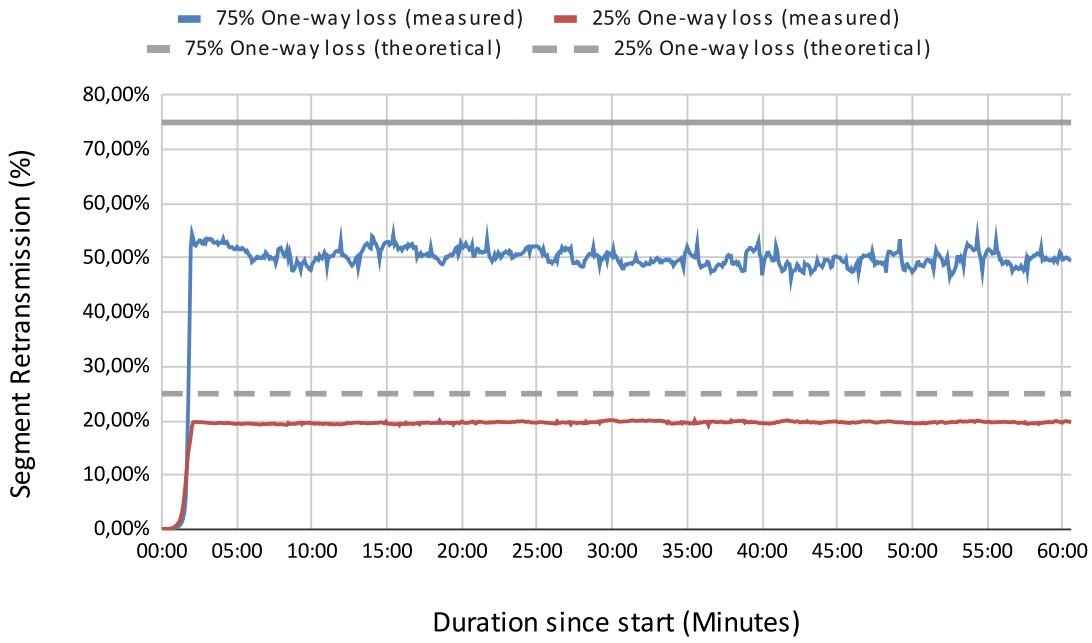
Figure 5.4: Packet loss experienced when injecting a 75% random packet loss over one hour.

retransmitted bytes—is collected over a period of one hour and averaged across all nodes with a one-minute sliding window.

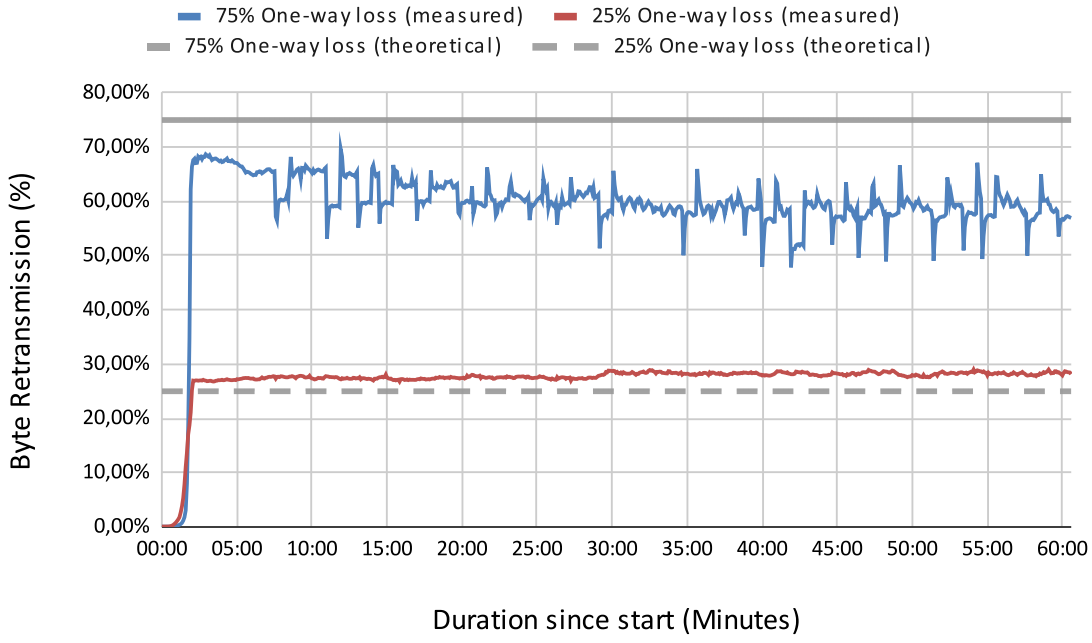
Unlike the last experiment performed using active monitoring, passive monitoring requires traffic to already run between the nodes due to not generating any traffic of its own. Therefore, simulated traffic is introduced. Since most of the traffic transpires over TCP, the pinger can use passive monitoring of TCP statistics to gather data. Specifically, this is done by parsing the TCP socket statistics available in the Linux kernel, from which the rate of retransmissions can be determined.

Looking at the results of the separate packet loss experiments in Figure 5.5, the percentage of packet retransmissions do not directly correspond to the set level of packet loss. The segment retransmission rate, as seen in Figure 5.5a, lies around 20% and 50% for the 25% and 75% experiment repetitions respectively. For the lower levels of simulated loss, the percentage of retransmissions are much closer together. However, as the simulated loss increases, the percentage of retransmissions do not increase to the same extent causing them to be much further away from the percentage of simulated packet loss.

Compared to the packet retransmission rate, the byte retransmission rate in Figure 5.5b is generally at a higher level for both experiment repetitions but still do not directly correlate to the set level of packet loss. One observation to take note of, is that for lower levels of simulated loss, the actual byte retransmissions exceeded the simulated values of loss.



(a) Retransmitted TCP segments as a percentage of all transmitted TCP segments.



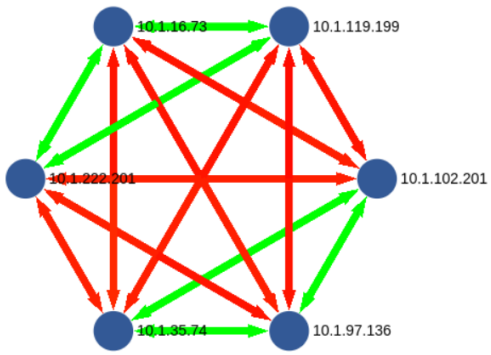
(b) Sum of the size of all retransmitted TCP segments as a percentage of the size of all transmitted TCP segments.

Figure 5.5: Passively collected data of two independent repetitions of the packet loss experiment, performed with a 75% and 25% random packet loss respectively.

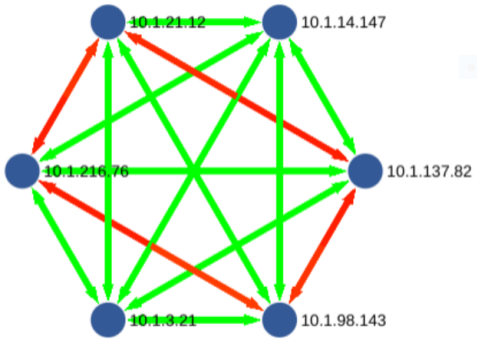
5.3 Network Partition

In order to demonstrate the ability to create, visualize, and categorize network partitions, the methodology described in Section 4.4 is applied to a Kubernetes cluster with six nodes with a pinger deployed on each of them. Screenshots from the visualizer are taken 30 seconds after the experiment has started with the sliding window set to 15 seconds.

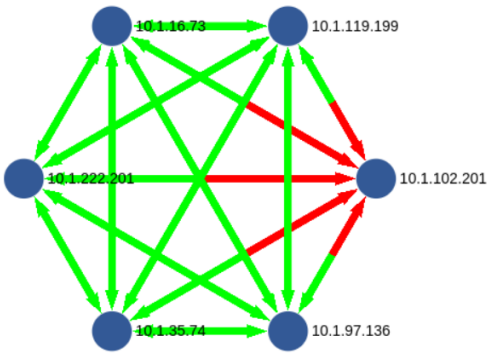
5.3.1 Active Monitoring



(a) Complete partitioning



(b) Partial partitioning



(c) Simplex partitioning

Figure 5.6: Three classes of partitionings recreated in a Kubernetes cluster and detected by the *active monitoring* mode.

In Figure 5.6a, two disconnected strongly connected components—meeting our definition of a complete partitioning—are created. The visualizer correctly identifies this as a complete partition. Similarly, Figure 5.6b shows another cluster where the same setup as in the previous case, except all network restrictions are lifted from one node in each partition—essentially creating a partial partition. The visualizer also correctly identifies this as a partial partitioning. Finally, a simplex partitioning is created by restricting ingress traffic on one of the nodes in an otherwise healthy cluster, as shown in Figure 5.6c. The visualizer correctly identifies this as a simplex partitioning.

close

Historic disturbances

From	Until	Complete	Partial	Simplex
Apr 25, 2022, 4:09 PM	Apr 25, 2022, 4:10 PM	Yes	Yes	Yes
Apr 25, 2022, 4:12 PM	Apr 25, 2022, 4:13 PM	-	Yes	-
Apr 25, 2022, 4:18 PM	Apr 25, 2022, 4:19 PM	-	Yes	Yes

Figure 5.7: Partitioning history after simulating a (1) complete partitioning, (2) partial partitioning, and (3) simplex partitioning with active monitoring.

In Figure 5.7, a screenshot of the partitioning history after applying the three types of partitionings is depicted. We observe that some partitioning events include "false positives" in terms of multiple types of partitionings detected. Watching a playback, all of these false positives are briefly detected in the transient states when packet loss is being rolled out to the pods or the pods are recovering after the packet loss.

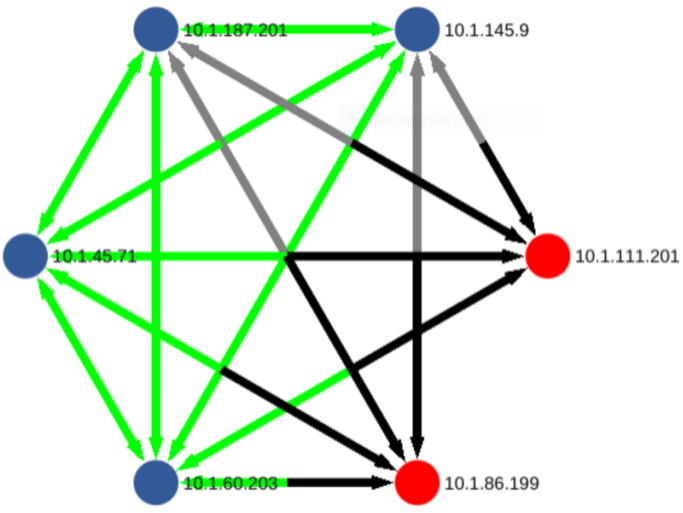


Figure 5.8: Visualizer after creating a partial partition, monitored in active mode, with the Prometheus server as one of the targets.

During the previous partition experiments, the communication between the Prometheus clients and server is maintained. In Figure 5.8, the same partial partitioning experiment as performed earlier (see Figure 5.6b) is applied to the cluster, while ensuring the Prometheus server is also affected by the network partitioning. If the Prometheus server is unable to reach a client, the color of the node's representation in the visualizer changes to red. We would also like to point out that, as earlier explained in Section 3.3.1, ingress edges of unavailable nodes cannot be calculated and appear black while gray edges indicate that no traffic traveled nor was excepted to travel

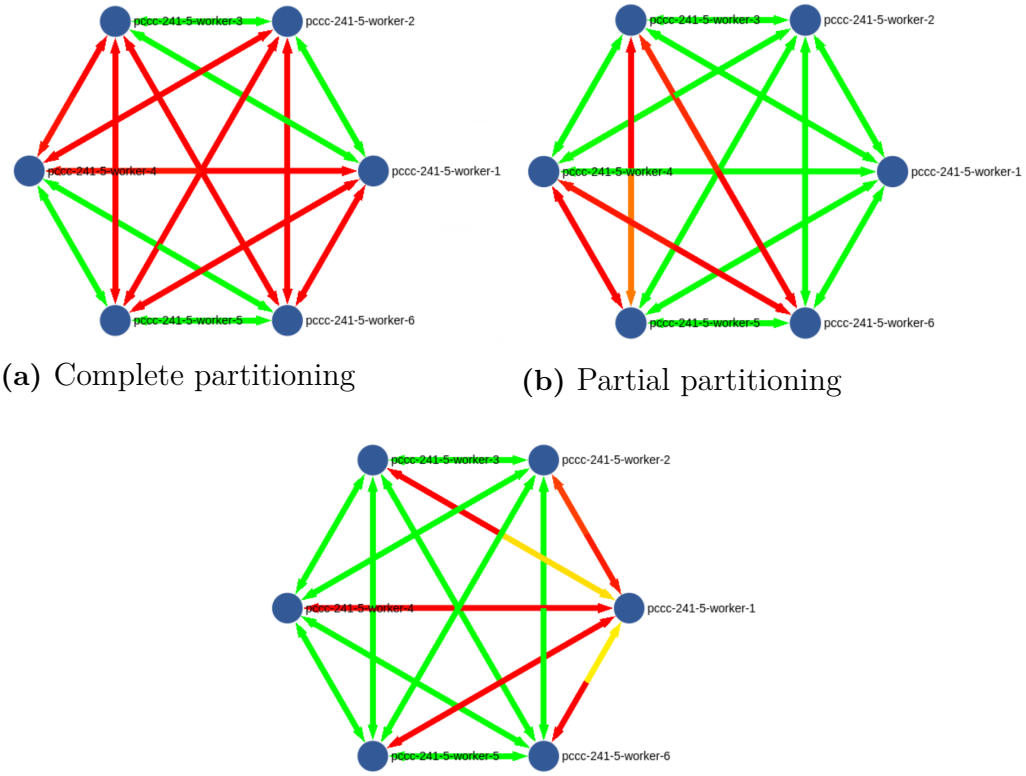
along the edge. We observe that two of four nodes still receive traffic from the two nodes unreachable by the Prometheus server.

5.3.2 Passive Monitoring

Now we introduce the same partitionings as above but monitored passively under a synthetic load. The health of the edge is in this case defined as $1-$*retransmitted segments / sent segments*. As can be seen in Figure 5.9a, the outline of a complete partitioning is present. However, while the number of transmitted segments on affected edges is greatly decreased and the number of retransmissions is increased, the ratio between them is not over 50%, and thus not significant enough to label some affected edges as "unhealthy." Therefore, even though it visually appears as a complete partition, the visualizer identifies this instance as a partial and simplex (but not a complete) partitioning. However, by adjusting the threshold slider it is possible to alter the level of retransmission required to consider links to be unhealthy. By changing the threshold to e.g., 25%, the same complete partition experiment was correctly identified as a complete partition by the visualizer. Similarly, by increasing the threshold, the hue of the links would gradually turn greener as the threshold for a partition would become stricter.

In Figure 5.9b, the partitioning is identified both as a partial and simplex partitioning even though some links are not completely red. Similar to the complete partition, the detection changes depending on the threshold value. When increasing the threshold, the detection is still the same as for 50% except for the slight change in colors of each link. However, when decreasing to 25% the false-positive simplex label disappears, and the partition is only considered to be a partial one.

When it comes to the simplex partitioning visualized in Figure 5.9c, where ingress traffic to the "worker-1" node is blocked, the results greatly differ from the case of active monitoring. Egress traffic appears to experience a higher rate of retransmissions, while the edges between "worker 1" to "worker 3" and "worker 6" appear to not be as affected by the levels of retransmissions resulting in the detection of both simplex and partial partitionings. Unlike the other partition experiments, adjusting the threshold will not alter the visualizer's ability to identify the partition to be simplex. This is especially seen when decreasing the threshold as at some point, the visualizer would consider the cluster to be completely partitioned due to both ingress and egress experiencing retransmissions.



(a) Complete partitioning

(b) Partial partitioning

(c) Simplex partitioning

Figure 5.9: Three classes of partitionings recreated in a Kubernetes cluster and detected by the *passive monitoring* mode.

6

Discussion

This chapter contains an evaluation of the results and suggestions for further improving the proof-of-concept application developed in this thesis. Section 6.1 presents a discussion based on the results of the baseline simulation, whereas Section 6.2 and Section 6.3 discuss the results of the packet loss and network partition experiments respectively. The final section provides a discussion about future work and suggestions on how the proof-of-concept can be improved upon.

6.1 Control Simulation

This section provides a discussion regarding the results of the control simulation experiment presented in Section 5.1. Section 6.1.1 provides a discussion about the findings of the packet loss experiment while Section 6.1.2 introduces a discussion based on the results of the latency experiment.

6.1.1 Packet loss

Neither the active monitoring solution nor `ping` detects a significant level of packet loss over the span of 24 hours—both with and without simulated traffic. The main reason for this is that the number of samples—approximately 432000—is not sufficient to discern a loss ratio. As a comparison, Guo et al. [9] detect a baseline packet loss of around 10^{-5} to 10^{-4} (0.001%–0.01%) in a data center network under normal conditions. While the fact that our active monitoring scheme does not noticeably exaggerate the number of packet losses is a sufficient conclusion for this experiment, we also perform another experiment with a slightly different setup. In this case, where the number of probes exceed half a billion, we detect a packet loss of around 0.002%.

Unlike the active monitoring tools, the passive alternative managed to record a significant number of packet and byte retransmissions. With no simulated traffic, regular spikes in retransmission can be seen in both the segment and byte retransmission rate. This is most likely due to TCP's congestion control interacting with bursts of traffic from background services. Therefore, by basing the passive mode on TCP retransmissions, we always expect some retransmissions to be recorded. With that in mind, it is harder to distinguish between links exhibiting a slightly elevated

rate of packet loss and various applications interacting with congestion control. The fact that the rate of retransmission subsides after traffic is injected into the cluster can also be expected. When simulating many 5G sessions, our belief is that traffic will overall be less dominated by bursts.

As can be seen, there are differences between actively and passively monitoring for packet loss. We expect the active UDP probes to give a more accurate characterization of network links, while passive monitoring yields results based on what applications are deployed on the cluster experience. We therefore believe these two methods to be complementary under normal conditions, with the more favorable method depending on the use-case. We will later discuss how these methods stack up in the case of induced packet loss.

6.1.2 Latency

The latencies in the 50th percentile measured by `ping` and our active monitoring mode without simulated traffic (referred to as "Baseline Traffic") generally coincide with each other. When adding simulated traffic, both our application and `ping` detect an overall decrease in the median latency—made especially evident in the measurements from `ping` where the latency halves. The reason for this latency decrease when increasing traffic is mostly unknown to us, and could be the result of several different factors, such as better cache coherency or the receive queue being processed more often.

For the 99th percentile, both tools manage to record latencies within close proximity of each other. For the baseline experiment, the measured values are quite similar for both tools. However, adding simulated traffic causes a drastic impact on the latency measured by both tools. As depicted in the results, the latencies in the 99th percentile increase drastically with a significant amount of jitter. This is however expected, as it is generally known that tail latencies in a distributed system are disproportionally affected by an increase in load.

Another observation is—similarly to the 50th percentile under load—that the `ping` utility generally estimates the latency to be lower (albeit slightly) than that of our application. The root cause of this is unknown as it can depend on multiple factors. Some examples include ICMP and UDP being layer 3 and 4 protocols respectively, how different protocols traverse the underlying network, CPU process time of our application or even differences in packet sizes for the ICMP and UDP probes. Similar discrepancies were measured by [52] who analyzed the RTT latency between `NetPerf` and `ping` in a Google Cloud environment. They noted, similarly to what we saw in our results, that latency was estimated to be slightly lower for `ping`. However, the size of the discrepancy depended on the transmission interval of the different tools. Additionally, they conclude that even though `ping` measurements were lower, `NetPerf` was still their primary choice of monitoring tool as it is TCP based and therefore exhibits patterns of delay similar to that of actual real-world applications.

6.2 Induced Packet Loss

The induced packet loss experiment, unlike the control experiment, focuses on the monitoring tools' ability to detect a known level of random packet loss. When inspecting the percentage of loss after running both a 25% and 75% random packet loss for one hour, our active monitoring tool was able to accurately measure the actual induced link-loss. However, the values measured over the five-minute sliding window are not consistent throughout the course of the experiment and fluctuate to some degree—which can be expected since the method used for discarding the packets is a Bernoulli process.

The `ping` utility is not able to measure one-way packet loss. However, by assuming that the level of loss is symmetrical (i.e., the same in both directions), it is possible to infer the one-way loss. In this case, the inferred loss is close to the expected induced loss and quite similar to what the active mode measured. However, in environments where disturbances may affect both directions asymmetrically, assuming the one-way loss would not be appropriate. This is especially true in the case of simplex partitions, where the ping utility would assume there is a complete loss in connectivity.

TCP retransmission data were collected using passive monitoring while inducing a 25% and 75% random packet loss. Some general observations are that the rate of retransmissions do not quite converge to the induced packet loss rate. However, there is a clear correlation between an increased packet loss rate and increased retransmission rate, making it viable for detecting more severe disturbances in production environments.

Interestingly, the retransmission rate is less than the rate of induced two-way loss, even though TCP is a two-way protocol. We believe this may be caused by multiple phenomena. Firstly, even if a TCP ACK disappears, a retransmission may not be needed since subsequent ACKs with higher sequence numbers implicitly acknowledges previous bytes. Secondly, ACK segments with an empty data field may drive down the segment retransmission rate. Thirdly, due to TCP being a streaming protocol, a single segment may contain data of multiple segments that are lost—although this should not affect the number of bytes retransmitted. However, even with all of the above reasons combined, we would not see the number of retransmitted bytes fall short of the induced one-way packet loss of 75%. Therefore—lastly—we remark that the application receiving the traffic may close sockets it deems "problematic" before a retransmission can occur.

6.3 Partition Detection

The visualizations created by the visualizer are generally satisfactory. The general impression is that it is easy to see visually which links are problematic.

6.3.1 Prometheus and Partitions

One significant issue with using Prometheus is that if packet loss occurs, there is a chance that our application is partitioned from the Prometheus server, as demonstrated in Figure 5.8. While our system will continue to operate, some data may be lost. This is because Prometheus clients do not retain data in the temporal dimension. Instead, the Prometheus server is responsible for assigning scraped data with a timestamp. This means that while counters containing information such as if a ping is received will still be recorded correctly, it is not possible to determine when during the period between two successful scrapings such an event occurs. We note that because our application is tasked with collecting data regarding communication failures, it can be assumed that following the progress of a failure temporally is beneficial—even if some data would only be available post mortem.

6.3.2 Partition Threshold

The detection threshold and sliding window sliders in the visualizer are directly responsible for determining whether a link between two nodes is "healthy"—an integral part of partition detection. The behavior of the threshold slider when using active monitoring is well understood and—as we have shown in this experiment—offers accurate partition detections. This is not surprising since the measured packet loss rate is directly related to the packet drop rate of the network. However, the same cannot be said in the case of passive monitoring.

The rate of TCP retransmissions—both in terms of the number of segments and number of bytes—is not as easily understood. For example, it may depend on how networking is implemented in a particular application or the operating system. As depicted in the results, outlines of both a complete and partial partitioning are properly depicted. However, the rate of retransmission is not measured as being above 50% between some unconnected nodes, and thus not significant enough to cause all affected links to be considered unhealthy (even though the packet loss is 100%). Thus, a low detection threshold is needed for detecting partitionings, although this potentially may prove to cause more false positives. Furthermore, contrary to our active probes, the passive mode detects retransmissions in both directions during a simplex partitioning. This is expected since acknowledgments cannot be sent, but proves that a more thorough analysis of TCP statistics—possibly reporting the number of segments received to Prometheus—is needed in order to accurately identify simplex partitions.

Apart from the detection threshold slider, the sliding window also affects partition detection. By lowering the sliding window, a lost packet has greater significance on the percentage of packet loss whereas for a greater sliding the percentage will be much lower. If we use a sliding window of one hour and a complete partition occurs during one minute, this will be deemed insignificant by the visualizer as during the course of the hour the percentage of loss was not greater than 50%. As such, it is important to correctly configure the values to make sure that the levels of false-positives are minimized.

In summary, the implementation of the passive mode is not the most suitable one when the end goal is to categorize network partitions or links exhibiting a low packet loss rate, but can nevertheless give insight into traffic patterns of existing applications. Therefore, we find that active monitoring may be the most appropriate when detecting network failures. We would nevertheless like to point out that there exist other passive monitoring schemes that use less simple approaches for detecting packet loss, such as those presented in "Related Work" (Section 2.5.2). However, while the other end-host based passive monitoring schemes [10], [11], [49] can identify partial packet loss, they are not constructed for presenting the specific loss rate of a link.

6.4 Future work

This section presents some of the current limitations of the proof-of-concept application developed in this thesis and based on this, suggests possible paths of improvement which can be pursued in future works.

6.4.1 Scalability

While our current solution easily handles tens of nodes, one downside of our solution is that it is not designed for scalability. In fact, the upper bound for both the number of probes used for active monitoring and the amount of data produced by both the active and passive solution scales with the number of links. If n is the number of nodes, there can exist $O(n^2)$ links if no redundancy is assumed.

While the number of probes can be reduced to zero with passive monitoring, there are other alternatives. While this thesis has worked under the assumption that the network topology is ordered in a full mesh, data center networks are hardly configured this way. By assuming each physical link only needs to be probed once, knowledge of the physical network topology could eliminate superfluous probes. In general, we make the observation that the number of messages necessary to ensure coverage is bounded by the number of links.

Another issue for scalability comes from the analysis of large volumes of data. Currently, all data is collected by a single Prometheus server, preventing horizontal scaling. While a distributed time series database could be utilized, it is also possible to deploy multiple independent Prometheus servers in a tree structure, since Prometheus is designed to be able to scrape other Prometheus servers. However, as the depth of the tree increases, the aggregate metrics near the root must be made more concise by means of aggregation.

6.4.2 Partition Detection

A significant cause for concern is the reproducibility of our results in relation to passive monitoring. Preferably, more experiments should be performed under different load levels and different types of application workloads. It is possible that

the simulated traffic we introduced is especially suitable for passively monitor for network partitions.

As it stands, the current implementation of the partition detection scheme relies fully on the data scraped by Prometheus. However, as explained earlier in Section 6.3.1, there is a risk of data not being collected during a partition. Therefore, it follows that some other scheme—such as a push-based one as opposed to the currently pull-based scheme—would be more appropriate for our use.

On another note, the visualizer does not use any persistent storage and does therefore not retain any data concerning the partitions that have occurred in the past. As such, a large amount of data will need to be processed by the Prometheus server each time the window containing historic partitionings is opened, resulting in significant loading times. Ideally, information regarding past partitionings should be retained.

As a final note, the current partition detection scheme is currently handled by the visualizer application. While we have seen no performance issues in the application (apart from the listing of historical partitions), it is hard to tell what the effect on performance will look like given how the number of links scales quadratically to the number of nodes. As such, it might be of interest to move the partition calculation away from the visualizer. This would allow the visualizer to be a more light-weight application as it would not be required to perform any calculations while simultaneously illustrating the health of each link in the cluster.

6.4.3 Container Privileges

While the active solution has no extra requirements in terms of privileges, the pinger needs to run in "privileged mode" in order to access socket statistics from the Linux network stack. Additionally, in order to capture all traffic between nodes, an escape from the application's container environment—in particular its current Linux network namespace—is necessary. In many circumstances, this will not be allowed. In cloud systems with multiple stakeholders and an assortment of deployed applications, an application with full access to the underlying host may not be permissible—especially if applications are delivered by different stakeholders.

In environments where running pods in "privileged mode" is not a viable solution for passive monitoring, there exist other possible alternate solutions for the passive monitoring scheme. One of them is to deploy the pinger as a side-car container on a pod which has other containers sending traffic. From there, it is possible to measure statistics tied to the specific pod. However, this solution is very tightly integrated with the application and might not be the optimal solution with the purpose of detecting network partitions.

6.4.4 Measurement Improvements

On a more general note, there are some possible improvements that can be made to the measurements collected by the application. In this case we have identified

one possible measurement improvement for the active monitoring scheme and one for the passive as well.

The active monitoring scheme is currently able to monitor one-way packet loss but not one-way delay, as only the RTT is measured. The reason for this, as mentioned in Section 2.4.3, is due to one-way delay being quite difficult to monitor as clocks might not be synchronized. However, it is an interesting metric to monitor since similar to the reason we use one-way packet loss, delay might present itself asymmetrically across the path between two nodes.

When it comes to the passive monitoring scheme, TCP statistics are currently gathered from the Linux kernel at a four second sampling rate. While this is not something we have investigated, the sampling rate might not be the most optimal one, especially in the case of short-lived TCP connections. As such, it would be interesting to do an investigation into which sample rate is most suitable when monitoring TCP statistics.

7

Conclusion

This thesis sets out to explore the feasibility of detecting network failures in a production Container as a Service (CaaS) platform adapted for 5G applications and how to introduce alarms that present eventual network partitions as a result of these failures. In this case, the aim was to be able to detect three different types of partitions: complete, partial and simplex.

In order to achieve the goals stated above, a proof-of-concept monitoring application is developed and deployed in a Kubernetes cluster. The implementation has two modes—active and passive monitoring—both of which provide a different solution to monitor packet loss between nodes in the cluster. These statistics are collected and stored in a Prometheus time-series database and scraped by a front-end application—called the visualizer—to illustrate the health of individual network links in the cluster and to detect whether any partitions have occurred. The delay from a network failure to the visualizer acknowledging it is theoretically bounded by 13.5 seconds for active monitoring and 11.5 seconds from a TCP retransmission for passive monitoring, but is configurable.

To evaluate the performance of the different modes, as well as their ability to monitor for network failures and the network partitionings they induce, we ran multiple experiments and compared the results. Both monitoring solutions are capable of presenting network partitions as a result of these failures. However, while nevertheless able to detect significant failures, the two-way nature of the TCP protocol prevents the passive mode from inferring all types of partitionings. Furthermore, the rate of retransmission detected when using our passive solution is not directly proportional to the packet loss on that link and requires certain container privileges which might not be suitable in a cloud environment. A comparison between `ping` and our active implementation concludes that our tool is as reliable, and—unlike `ping`—is able to identify one-way loss required for detecting simplex partitionings.

In conclusion, we see that it is feasible to detect network failures in a Container as a Service (CaaS) platform adapted for 5G applications. With further refinements to the partition detection methodology and scalability of our solution, we believe that a similar system can successfully be deployed and operated on a larger scale.

Bibliography

[1] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An Analysis of Network-Partitioning Failures in Cloud Systems," presented at the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA: USENIX Association, Oct. 2018, pp. 51–68, ISBN: 978-1-939133-08-3.

[2] IBM. "Cloud computing: A complete guide." (2020), [Online]. Available: `https://www.ibm.com/cloud/learn/cloud-computing-gbl` (visited on 2022-01-17).

[3] Y. Tseng, G. Aravinthan, B. Berde, S. Imadaliz, D. Houatra, and H. Qiu, "Re-Think Monitoring Services for 5G Network: Challenges and Perspectives," in *2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/ 2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, Jun. 2019, pp. 34–39. DOI: `10.1109/CSCloud/EdgeCom.2019.00016`.

[4] J. Gilbert, *Cloud Native Development Patterns and Best Practices: Practical Architectural Patterns for Building Modern, Distributed Cloud-Native Systems*. Birmingham Mumbai: Packt Publishing, 2018, 302 pp., ISBN: 978-1-78847-392-7.

[5] M. Agiwal, A. Roy, and N. Saxena, "Next Generation 5G Wireless Networks: A Comprehensive Survey," *IEEE Communications Surveys Tutorials*, vol. 18, no. 3, pp. 1617–1655, 2016, ISSN: 1553-877X. DOI: `10.1109/COMST.2016.2532458`.

[6] S. Anithakumari and K. Chandrasekaran, "Monitoring and Management of Service Level Agreements in Cloud Computing," in *2015 International Conference on Cloud and Autonomic Computing*, Sep. 2015, pp. 204–207. DOI: `10.1109/ICCAC.2015.28`.

[7] S. Rajan and A. Jairath, "Cloud Computing: The Fifth Generation of Computing," in *2011 International Conference on Communication Systems and Network Technologies*, Jun. 2011, pp. 665–667. DOI: `10.1109/CSNT.2011.143`.

[8] M. Alfatafta, B. Alkhatib, A. Alquraan, and S. Al-Kiswany, "Toward a Generic Fault Tolerance Technique for Partial Network Partitioning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 351–368, ISBN: 978-1-939133-19-9.

[9] C. Guo, L. Yuan, D. Xiang, *et al.*, "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, New York, NY, USA: Association for Computing Machinery, Aug. 17, 2015, pp. 139–152, ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2787496.

[10] A. Roy, R. Das, H. Zeng, J. Bagga, and A. C. Snoeren, "Understanding the Limits of Passive Realtime Datacenter Fault Detection and Localization," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 2001–2014, Oct. 2019, ISSN: 1558-2566. DOI: 10.1109/TNET.2019.2938228.

[11] B. Arzani, S. Ciraci, L. Chamon, *et al.*, "007: Democratically Finding the Cause of Packet Drops," presented at the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), Renton, WA, USA: USENIX Association, Apr. 2018, pp. 419–435, ISBN: 978-1-939133-01-4.

[12] 3GPP. "5G; System architecture for the 5G System (5GS) (3GPP TS 23.501 version 16.6.0 Release 16)." (2019), [Online]. Available: https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.06.00_60/ts_123501v160600p.pdf (visited on 2022-01-25).

[13] Ericsson. "5G Core (5GC) network: Get to the core of 5G." (Dec. 7, 2021), [Online]. Available: https://www.ericsson.com/en/core-network/5g-core (visited on 2022-02-08).

[14] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015, ISSN: 1558-2256. DOI: 10.1109/JPROC.2014.2371999.

[15] D. Cotroneo, L. De Simone, and R. Natella, "NFV-Bench: A Dependability Benchmark for Network Function Virtualization Systems," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 934–948, Dec. 2017, ISSN: 1932-4537. DOI: 10.1109/TNSM.2017.2733042.

[16] "Who we are," Cloud Native Computing Foundation, [Online]. Available: https://www.cncf.io/about/who-we-are/ (visited on 2022-03-10).

[17] "Graduated and incubating projects," Cloud Native Computing Foundation, [Online]. Available: https://www.cncf.io/projects/ (visited on 2022-03-10).

[18] M. Pokharel, Y. Yoon, and J. S. Park, "Cloud Computing in System Architecture," in *2009 International Symposium on Computer Network and Multimedia Technology*, Jan. 2009, pp. 1–5. DOI: 10.1109/CNMT.2009.5374726.

[19] C. Miyachi, "What is "Cloud"? It is time to update the NIST definition?" *IEEE Cloud Computing*, vol. 5, no. 3, pp. 6–11, May 2018, ISSN: 2325-6095. DOI: 10.1109/MCC.2018.032591611.

[20] Atlassian. "Containers as a Service," Atlassian, [Online]. Available: https://www.atlassian.com/microservices/cloud-computing/containers-as-a-service (visited on 2022-04-26).

[21] C. Pahl, "Containerization and the PaaS Cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, May 2015, ISSN: 2325-6095. DOI: 10.1109/MCC.2015.51.

[22] Docker Inc. "Empowering App Development for Developers," [Online]. Available: https://www.docker.com/ (visited on 2022-02-23).

[23] Kubernetes. "Kubernetes," [Online]. Available: https://kubernetes.io/ (visited on 2022-02-23).

[24] A. Basiri, N. Behnam, R. de Rooij, *et al.*, "Chaos Engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, May 2016, ISSN: 1937-4194. DOI: 10.1109/MS.2016.60.

[25] "PRINCIPLES OF CHAOS ENGINEERING - Principles of chaos engineering," [Online]. Available: https://principlesofchaos.org/ (visited on 2022-03-03).

[26] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11, New York, NY, USA: Association for Computing Machinery, Aug. 15, 2011, pp. 350–361, ISBN: 978-1-4503-0797-0. DOI: 10.1145/2018436.2018477.

[27] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, New York, NY, USA: Association for Computing Machinery, Jun. 10, 2010, pp. 193–204, ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807161.

[28] R. Jhawar and V. Piuri, "Fault Tolerance and Resilience in Cloud Computing Environments," in *Cyber Security and IT Infrastructure Protection*, J. R. Vacca, Ed., 1st ed., Boston, MA, USA: Syngress, Jan. 1, 2014, pp. 1–28, ISBN: 978-0-12-416681-3. DOI: 10.1016/B978-0-12-416681-3.00001-X.

[29] M. Steinder and A. S. Sethi, "A survey of fault localization techniques in computer networks," *Science of Computer Programming*, Topics in System Administration, vol. 53, no. 2, pp. 165–194, Nov. 1, 2004, ISSN: 0167-6423. DOI: 10.1016/j.scico.2004.01.010.

[30] Alliance for Telecommunications Industry Solutions. "Network Failure – ATIS Telecom Glossary," ATIS Telecom Glossary, [Online]. Available: https://glossary.atis.org/glossary/network-failure/ (visited on 2022-02-03).

[31] A. Singh, J. Ong, A. Agarwal, *et al.*, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 183–197, Aug. 17, 2015, ISSN: 0146-4833. DOI: 10.1145/2829988.2787508.

[32] Y. Zhu, N. Kang, J. Cao, *et al.*, "Packet-Level Telemetry in Large Datacenter Networks," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15, New York, NY, USA: Association for Computing Machinery, Aug. 17, 2015, pp. 479–491, ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2787483.

[33] Y. Peng, J. Yang, C. Wu, C. Guo, C. Hu, and Z. Li, "deTector: A Topology-aware Monitoring System for Data Center Networks," presented at the 2017 USENIX Annual Technical Conference (USENIX ATC 17), 2017, pp. 55–68, ISBN: 978-1-931971-38-6. DOI: 10.5555/3154690.3154697.

[34] P. Bailis and K. Kingsbury, "The Network is Reliable: An informal survey of real-world communications failures," *Queue*, vol. 12, no. 7, pp. 20–32, Jul. 8, 2014, ISSN: 1542-7730. DOI: 10.1145/2639988.2655736.

[35] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, New York, NY, USA: Association for Computing Machinery, Aug. 22, 2016, pp. 58–72, ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934891.

[36] D. A. Popescu and A. W. Moore, "Measuring Network Conditions in Data Centers Using the Precision Time Protocol," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 3753–3770, Sep. 2021, ISSN: 1932-4537. DOI: 10.1109/TNSM.2021.3081536.

[37] T. He, L. Ma, A. Swami, and D. Towsley, *Network Tomography: Identifiability, Measurement Design, and Network State Inference*, 1st ed. New York, NY: Cambridge University Press, 2021, ISBN: 978-1-108-42148-5.

[38] Y. Qi, C. Fang, H. Liu, *et al.*, "A survey of cloud network fault diagnostic systems and tools," *Frontiers of Information Technology & Electronic Engineering*, vol. 22, no. 8, pp. 1031–1045, Aug. 1, 2021, ISSN: 2095-9230. DOI: 10.1631/FITEE.2000153.

[39] Y. Li, H. Zheng, C. Huang, K. Pei, J. Li, and L. Huang, "Terminator: An Efficient and Light-weight Fault Localization Framework," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Jul. 2020, pp. 580–585. DOI: 10.1109/INFOCOMWKSHPS50562.2020.9163055.

[40] D. Banerjee, V. Madduri, and M. Srivatsa, "A Framework for Distributed Monitoring and Root Cause Analysis for Large IP Networks," in *2009 28th IEEE International Symposium on Reliable Distributed Systems*, Niagara Falls, New York, USA: IEEE, Sep. 2009, pp. 246–255, ISBN: 978-0-7695-3826-6. DOI: 10.1109/SRDS.2009.22.

[41] J. C. Nobre, L. L. Penz, and L. Z. Granville, "Measurement correlation for improving cooperation in measurement federations," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, Lisbon, Portugal: IEEE, May 2017, pp. 584–587, ISBN: 978-3-901882-89-0. DOI: 10.23919/INM.2017.7987335.

[42] P. Lapukhov and A. Adams. "NetNORAD: Troubleshooting networks via end-to-end probing," Engineering at Meta. (Feb. 18, 2016), [Online]. Available: https://engineering.fb.com/2016/02/18/core-data/netnorad-troubleshooting-networks-via-end-to-end-probing/ (visited on 2022-01-19).

[43] C. Tan, Z. Jin, C. Guo, *et al.*, "NetBouncer: Active Device and Link Failure Localization in Data Center Networks," presented at the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), 2019, pp. 599–614, ISBN: 978-1-931971-49-2.

[44] G. Almes, S. Kalidindi, M. J. Zekauskas, and A. Morton, "A One-Way Delay Metric for IP Performance Metrics (IPPM)," Internet Engineering Task Force, Request for Comments RFC 7679, Jan. 2016, 27 pp. DOI: 10.17487/RFC7679.

[45] M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 3rd ed. CreateSpace Independent Publishing Platform, 2017, 582 pp., ISBN: 978-1-5430-5738-6.

[46] V. Shrivastav, K. S. Lee, H. Wang, and H. Weatherspoon, "Globally Synchronized Time via Datacenter Networks," *IEEE/ACM Transactions on Networking*, vol. 27, no. 4, pp. 1401–1416, Aug. 2019, ISSN: 1558-2566. DOI: 10.1109/TNET.2019.2918782.

[47] A. Vakili and J.-C. Gregoire, "Accurate One-Way Delay Estimation: Limitations and Improvements," *IEEE Transactions on Instrumentation and Measurement*, vol. 61, no. 9, pp. 2428–2435, Sep. 2012, ISSN: 1557-9662. DOI: 10.1109/TIM.2012.2190551.

[48] S. Kalidindi, M. J. Zekauskas, A. Morton, and G. Almes, "A One-Way Loss Metric for IP Performance Metrics (IPPM)," Internet Engineering Task Force, Request for Comments RFC 7680, Jan. 2016, 22 pp. DOI: 10.17487/RFC7680.

[49] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred, "Taking the Blame Game out of Data Centers Operations with NetPoirot," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, New York, NY, USA: Association for Computing Machinery, Aug. 22, 2016, pp. 440–453, ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934884.

[50] M. Jacomy, T. Venturini, S. Heymann, and M. Bastian, "ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software," *PLOS ONE*, vol. 9, no. 6, pp. 1–12, Jun. 10, 2014, ISSN: 1932-6203. DOI: 10.1371/journal.pone.0098679.

[51] "Tc(8) - Linux manual page," [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html (visited on 2022-04-19).

[52] D. Phanekham and R. Jones. "Using netperf and ping to measure network latency," Google Cloud Blog. (Jun. 16, 2020), [Online]. Available: https://cloud.google.com/blog/products/networking/using-netperf-and-ping-to-measure-network-latency/ (visited on 2022-05-17).