University of Glasgow | School of Computing Science

Comparison of Service Mesh Technologies

COMPSCI 5082P - MSci Software Engineering with Work Placement - 40 credits

Jose Luis Povedano Poyato (2403203)

March 31, 2023

ABSTRACT

Service Mesh Technologies are microservice infrastructure tools which run alongside the business application abstracting network features from the application business logic. Using service meshes simplifies microservice development by separating network requirements, however, they can lead to an increase in resource usage as well as latency. We propose a new benchmark that simulates microservice applications of different sizes to explore the performance of service meshes as they scale horizontally. We conducted an experiment where we executed the benchmark with three popular service mesh technologies; Istio, Linkerd and Consul. Our results show that Linkerd was the most efficient service mesh, achieving the lowest latency and the least resource utilization. Istio's memory utilization radically increased as the application scaled horizontally or traffic increased and Consul struggled to complete all tests in the benchmark requiring multiple adaptations. Overall, our findings show that the overhead of the mesh will grow alongside the application, highlighting the importance of selecting a fitting mesh to the application's requirements and scale.

1. INTRODUCTION

Microservices have revolutionized how modern applications are developed and deployed. They enable developers to break down large, monolithic applications into smaller, independent services that can be developed, deployed, and scaled independently of one another. However, the microservice architecture brings with it many new challenges, including managing the complexity of the communication between microservices, ensuring their scalability and reliability, and maintaining security.

Service Mesh Technologies (SMTs) are a promising solution to address these issues by providing a dedicated infrastructure to manage service-to-service communication. Service meshes embed themselves into the microservices outside of the main business application, effectively abstracting application development from networking requirements.

In this work, we aim to explore the differences between multiple service mesh solutions focusing on their impact on a microservice system's performance and overhead. Our objectives are to assess the performance and overhead of different service meshes when applied to applications of different scale, perform an in depth comparison of these findings across the different SMTs and use our findings to understand the competencies and limitations of the meshes.

For this purpose, we propose a novel approach that compares the performance of a service mesh as it scales. Previous work has explored the performance of different mesh in small scale demo applications [1, 2, 3]. This paper analyses how Istio, Consul and Linkerd perform when used to mesh a microservice application with increasing numbers of microservices and users. We believe this new approach offers two main advantages, a representation of the mesh performance at different scales and an assessment on the suitability of the service meshes as an application grows.

We begin by introducing microservices, explaining the need for service meshes and highlighting the similarities and differences in the design of Istio, Linkerd and Consul. This is followed by a comprehensive review of the existing approaches to benchmarking SMTs and their limitations. We then describe our scalable solution to benchmark the overhead of Istio, Linkerd and Consul compared to native Kubernetes employing an increasing number of microservices and users. Finally, we present our experiment results and discuss the implications of our findings.

2. BACKGROUND

The following section explains the need for service meshes and examines the architecture of the Istio, Linkerd and Consul service mesh.

2.1 Microservices

Microservices structure software applications as collections of smaller, independent modules that interact with each other to form a larger application. Each module, or microservice, can be deployed, scaled and run independently. They enable loosely coupled development by separating application areas and their smaller size facilitates debugging and maintaining code [4]. Microservices are an increasingly popular approach to software development, in 2021, 85 per cent of respondents from large organizations stated using microservices [5].

However, microservices have added new difficulties to software development. They break down applications into a large repertoire of smaller interdependent services. As these applications grow it becomes more challenging to manage service-to-service communication, ensure security and reliability [6]. Therefore, adopting microservices often requires a significant investment in infrastructure and tooling solutions to support the system perform communication and management tasks such as orchestration, monitoring and load balancing. In 2022, these auxiliary workloads considerably outnumber application workloads (63% vs. 37%) [7].

Many tools have been developed to help tackle some of these issues. Containers are used to package the services so they can run everywhere. Orchestration tools are adopted to automate the coordination, configuration and deployment of the containerised services. However, networking capabilities are often out of scope for orchestration tools, which leads to networking solutions being integrated into the application by the developers. This directly contradicts the separation of concerns principle of microservices.

2.2 Service meshes

Service Mesh Technologies (SMTs) are a rapidly growing approach to manage the networking issues that arise from the microservice architecture. SMTs handle service-to-service communication over a network, abstracting network features from the application's development. Service meshes are becoming an overly popular solution to microservice networking. According to a survey by the Cloud Native Computing Foundation (CNCF) Service Mesh adoption has grown from 27% in 2020 to 47% in 2022 [7] and Ashok et al. even propose considering service meshes a new Network Layer [8].

Most service meshes split the network into two separate layers. The data plane is formed by proxies, attached as sidecars to the services to control network communication between microservices. The control plane establishes the rules and regulations the data plane must follow. The applications within the microservices are often unaware of the external service mesh, helping separate network capabilities from the application.

Service meshes enrich the network simplifying communication across microservices and adding multiple features to the network. The motives leading Service Mesh adoption are the added security (79%) such as mTLS integration, observability (78%) for example metrics and log collection, traffic management (62%) i.e. blue/green deployments and reliability features (56%) such as request retries [9]. On the other hand one of the major challenges encountered by adopters are how the mesh affects performance [9], with the complex effects of the mesh on performance delaying the adoption of service mesh in production environments [10].

This paper examines three of the most widely used service meshes; Linkerd, Istio and Consul.

2.3 Envoy Proxy

Envoy is not a service mesh, however many SMTs like Istio or Consul-k8 depend on it to function. Envoy is an open source sidecar proxy written in C++. It was created by Lyft and currently used by many major companies such as Google, AWS and Microsoft. Envoy provides a range of features such as load balancing, traffic routing, service discovery, authentication and authorization and circuit breaking. Furthermore it can be deployed to run alongside every application, abstracting the network by providing common features in a platform-agnostic manner. Therefore, many service meshes have incorporated it into their system as their proxy to communicate between microservices.

2.4 Linkerd

Linkerd is one of the most popular service meshes available. Its popularity comes from it being one of the most minimalistic approaches to service meshes, notably reducing configuration options compared to other meshes, largely simplifying its usage. Linkerd is a CNCF project since 2017 and was specifically built to be meshed into Kubernetes, with the control plane requiring certain kubernetes components to work.

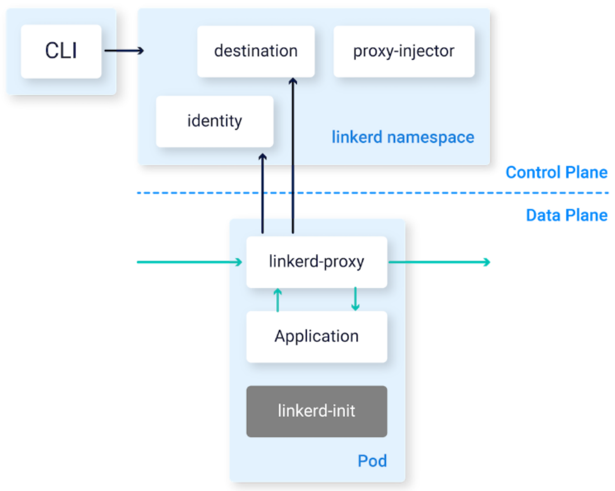

Figure 1: Linkerd Architecture [1]

As of Linkerd 2.12 Linkerd's control plane is composed of the destination service to determine traffic policies, the identity service to provide the proxies with certificates to enable mTLS and the proxy-injector to add a proxy to a pod when deployed. Linkerd's data plane does not rely on Envoy Proxies, as they found them to add too much complexity and instead decided to build their own customised "micro-proxies" using RUST. These proxies are deployed as sidecar containers in each pod.

2.5 Consul

Consul started as a service discovery and configuration tool, with the service mesh being created later on. Consul is completely platform agnostic, being able to run in Virtual Machines and Kubernetes clusters and facilitate communications between multiple clusters or machines. This study focuses on Consul-k8s, their service mesh created for kubernetes, however both Consul and Consul k8 have very similar architectures, Consul-K8s just includes additional features to make operating a Consul Cluster easier[2].

Consul-k8s' injects the *Consul Dataplane* and Envoy proxies as sidecar containers to the pods. The Consul Dataplane manages the Envoy proxies, removing the need for client agents seen in a normal Consul deployment. The Consul control plane then contains one or more consul server agents, which store all state information. The consul cluster elects a server agent to be the leader through a 'consensus' process. The leader processes all queries and transactions, to prevent conflicting updates in clusters containing multiple servers. Consul's control plane can be customised, deploying different consul datacenters in different nodes enabling communications between them, streamlining communication across

[1]https://linkerd.io/2.12/reference/architecture/
[2]https://developer.hashicorp.com/consul/docs/k8s/architecture
[3]https://developer.hashicorp.com/consul/docs/architecture
[4]https://developer.hashicorp.com/consul/docs/connect/dataplane

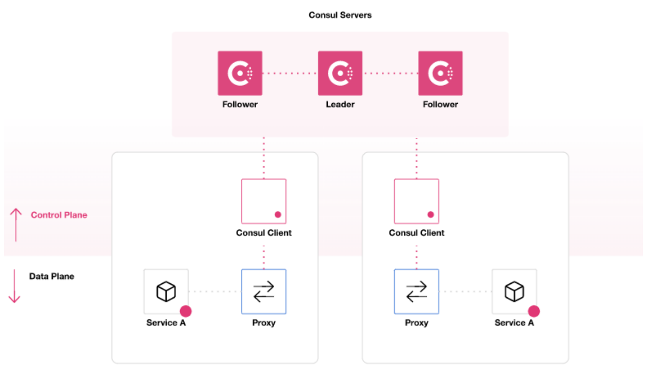

Figure 2: Consul Architecture [3]

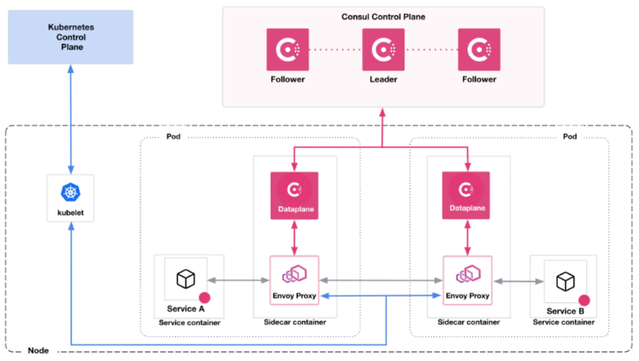

Figure 3: Consul-k8 Architecture [4]

different machines. Consul's main benefit is that many components can be swapped out or integrated with many other technologies offering one of the most customisable meshes, especially for multi platform networking.

2.6 Istio

Istio is a very popular service mesh, mostly because it acts as a jack of all trades. It can be meshed into a kubernetes cluster or a virtual machine, and requires considerably less configuration than Consul to optimise, yet it does not offer as much customisation. The mesh can be configured by using one of its predefined profiles that configures the service mesh automatically. If a more customized configuration is required, the user will need to edit Kubernetes manifests files (e.g., in YAML) to further configure the mesh. Istio is a CNCF project since 2022.

Istio's data plane deploys a set of Envoy proxies as sidecars attached to specified pods. The proxies will mediate and control all network communication between microservices. They also collect and report measurements on all mesh traffic. The control plane employs istiod to provide service discovery, configuration and certificate management. Istiod is actually a combination of multiple individual components Istio relied on before their 1.5 update. Istio's control plane used to resemble Linkerd's, employing the 'Pilot' component for Service discovery, 'Galley' for configuration, 'Citadel' for certificate generation and 'Mixer' for extensibility. However they decided that joining all applications together would better benefit the service mesh [11]. Istiod now configures the sidecar containers by converting high level routing rules, referred to as Custom Resource Definitions (CRDs), into Envoy configurations, and propagating them to the sidecars at runtime.

[5]`https://istio.io/latest/docs/ops/deployment/architecture/`

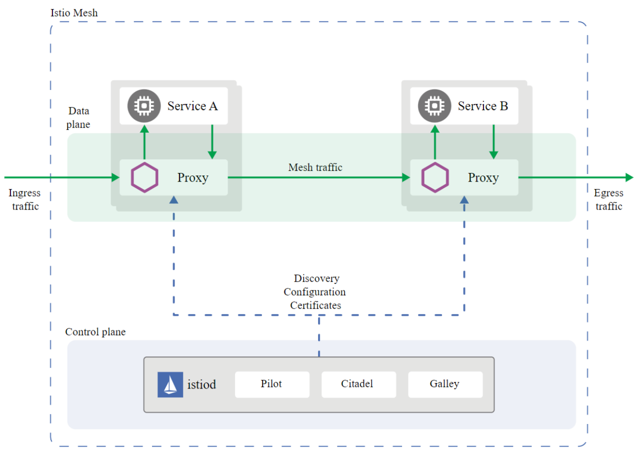

Figure 4: Istio Architecture [5]

Istio's leverage of Envoy originally made it a more feature rich mesh than Linkerd, however Linkerd has undergone accelerated growth adding many of the features that gave Istio an advantage such as Blue/Green Deployments[6] or Distributed Tracing[7] as extensions to the Service Mesh. Istio takes advantage of the additional functionality Envoy has over Linkerd's microproxy to offer these features by default, while Linkerd relies on extensions which integrate other tools, i.e. Linkerd Jaegger[8], or on interfaces to configure this behaviour [12].

3. EXISTING RESEARCH OF SMTS

The following section explores the current research of Service Mesh Technologies to provide context to the experiment proposed in this paper.

3.1 Qualitative Research

An extensive amount of papers have already performed a qualitative study of SMTs. Li et al. provides an early review of their features and challenges [13]. Xie and Govardhan demonstrate how they integrated the istio service mesh into a deep learning application to perform load balancing and task scheduling [14]. Hussain et al. leveraged the Istio service mesh to generate secure APIs [15]. El Malki and Zdun performed a model analysis of established practices from service mesh practitioners to develop a set of recommended service mesh practices to reduce the uncertainty around adopting service meshes [16]. Duque et al. performed a qualitative evaluation of SMTs for mobile edge applications, focusing on traffic management use cases [17]. Some educational resources attempted to provide feature comparisons across different SMTs, yet found this quite hard to do, as most service meshes are rapidly growing, meaning feature lists quickly become outdated. Khatri et al. included a compilation of features, acknowledging the rapid growth of the Service Mesh and how many could change [18].

[6]`https://linkerd.io/2.12/features/traffic-split/`
[7]`https://istio.io/latest/docs/tasks/observability/distributed-tracing/overview/`
[8]`https://linkerd.io/2.12/reference/cli/jaeger/`

Overall, it is understood that a service mesh will abstract the network logic from the application logic and enrich the network with the following fundamental features: Service discovery, load balancing, fault tolerance, traffic monitoring, circuit breaking, and authentication and access control [13].

3.2 Quantitative Research

Other attempts instead focus on the SMTs performance.

Ganguli et al. explored the overhead from the Istio service mesh on an edge application, scaling the number of user requests to see how it affected different architectures and dissecting the service mesh into different areas that could benefit from an in-depth analysis [10]. While Saleh Sedghpour et al. present experimental results on the best way to use SMTs for the circuit breaker functionality in a microservice-based application of a complex topology [19].

The following studies are the most relevant to our experiment, as they performed a cross comparison of the service mesh when applied to an application composed of multiple microservices.

3.2.1 Kinvolk

In 2019, a team at Kinvolk created a benchmark to compare the performance of native Kubernetes, Istio and Linkerd [1]. The benchmark deploys emojivoto[9], an application composed of 3 microservices created by the Linkerd team to demonstrate their service mesh. The benchmark sends multiple requests per second to emojivoto and records the latency per request and resource usage of the cluster.

Kinvolk used a modified version of wrk2[10] as a load generator. Their modified wrk2 sets up multiple simultaneous connections and then sends a consistent load to the server, increasing the submission rate to catch up if the latency of a stream ever falls behind. Kinvolk established a continuous throughput to push the meshes to their limit and compare their worst case scenarios, which they acknowledge may not represent real usage due to scaling capabilities.

They found Linkerd to be more efficient in terms of resource usage and it still achieved a lower latency than Istio. This benchmark has since been reused by the Linkerd team in 2021 to benchmark the newest versions of Istio and Linkerd, which obtained similar results [20].

3.2.2 Elastisys

In 2020, Dahlberg created a benchmark while working at Elastisys to compare the performance of Linkerd and Istio [2]. His benchmark used the TeaStore application, which is composed of seven microservices that simulate a web store. Apache JMeter was used to send concurrent requests to the meshes, to evaluate their performance when handling multiple requests.

Dahlberg's application may be more representative of a real-world scenario than Kinvolk's. TeaStore is composed of more services than emojivoto, including a database and an authentication service. However, the benchmark only tests the application with 40 simultaneous users. This may be representative of smaller internal technologies, but may not accurately reflect larger applications.

Dahlberg found that Linkerd remained the more memory-efficient option, but Istio was faster. He observed that Linkerd struggled more when handling concurrent requests and speculated that this could be due to a resource-limiting feature, as the CPU was not fully utilized, which would explain the added latency when dealing with multiple users.

3.2.3 CloudCover

The CloudCover team builds on Kinvolk's project to benchmark the application for Consul as well as Linkerd and Istio. The team used the helm installation version of Consul, instead of the CLI and created a customised installation defining some Consul configuration values[11]. Their results show that Consul performs at a lower latency than Linkerd and Istio, yet it was the most resource intensive application [3].

The previous benchmarks all followed similar approaches, yet they obtained different results. This is not surprising as they used different applications which will make different use of the meshes and different versions of the meshes which rapidly undergo major changes. Overall, we can observe one common factor throughout these benchmarks. Their applications are dependent on a small number of services. Therefore they may not accurately depict or help visualise the effects the SMTs could have on a larger application, comprised of more microservices and users.

4. METHODOLOGY

The aim of the experiment was to perform an exhaustive comparison of SMTs performance when applied to applications with an increasing number of concurrent users and microservices.

Following the previous benchmarks, we observed that the meshes might perform differently depending on what the application requires from them. To highlight the effects of the mesh, we are interested in an application that focuses on the communication between services, and how scaling the number of services and users affects it.

Therefore, we devised a new benchmark implementation which does not limit the experiment to a fixed number of microservices. Instead, it allows the user to specify how many microservices should communicate before completing the request and simulates an application of that size.

We adopted an experimental approach where we use a single kubernetes deployment to monitor all the different SMTs independently. We used native kubernetes to establish a baseline performance of the application, to provide us with a comparison against the different service meshes.

4.1 Equipment

The experiment was conducted using the OnLogic Helix 600 edge server. It has an Intel Core i7-10700T Comet Lake 4.5 GHz 8-core processor with 32 GB of DDR4 memory, and it runs Ubuntu 20.04. The Kubernetes version used was Kind v0.17.0 and Docker version 20.10.12.

4.2 Cluster Architecture

A kubernetes cluster with one node was created using the Kubernetes in Docker (KinD) distribution. To run an application in Kubernetes it must be deployed to a pod[12]. A kubernetes node allows by default up to 110 pods. Eight

[9]`https://github.com/BuoyantIO/emojivoto`

[10]`https://github.com/kinvolk/wrk2`

[11]`https://github.com/cldcvr/service-mesh-benchmark/blob/dev/scripts/setup-servicemeshes.sh`

[12]`https://kubernetes.io/docs/concepts/workloads/pods/`

pods were used by the kubernetes system to manage the cluster. Three pods are dedicated to collect the resource usage of the node. One pod is used to start the benchmark execution and store the results. Another pod is used to generate requests and record their response rate. 24 pods were used to simulate the microservices. Some of the remaining pods are used for the service meshes control plane.

Originally, the experiment was meant to use 92 pods to simulate the microservices. However, the service meshes struggled injecting so many pods simultaneously in one deployment. The number of pods was reduced to 62 pods so Istio could perform the injection and then was further lowered to 36 pods so Consul could inject the pods as well. The three service meshes would ignore pods where the injection process failed, not transmitting any requests to them and distributing the load across the injected pods. It should be noted that if there were any pods that failed to be injected with Consul, with time they would become more resource heavy, crashing the server overnight. The pods were further reduced to 24 as the monitoring systems consumed too many resources with 36 pods.

4.3 Simulating microservices

We developed a web application using Flask[13], a Python microweb framework. The application works as a counter, it receives a request containing a current value and an objective value. The application increases the current value by one and resubmits the request to a specified url, which would also be hosting the counter application. This process continues until the current value reaches the objective value. By establishing a different objective value on the request, we can simulate applications with a different number of dependent microservices.

To run an application in Kubernetes it must be containerised and deployed to a pod. Deployments can be defined to create multiple replicas of a pod grouped and managed under the same deployment[14]. Deployments simplify scaling applications as only the replica set size needs to be changed to increase the number of pods. The counter application was containerised using Docker[15] and two kubernetes deployments were defined containing 12 pods each running the counter application. Pods can run multiple containers simultaneously, however only one counter container was created per pod. Every replica runs the counter application and is hosted in its own pod.

Two kubernetes services[16] were created to expose the counter deployments to the kubernetes cluster's network. This creates an url per deployment, meaning all pods inside the deployment are pointed at by the same url. When a request is sent to that url, either Kubernetes's or the SMT's load balancer selects which pod receives the request. This allows us to test the service meshes's load balancers.

Two deployments, Counter-A and Counter-B, were specified so Counter-A would send requests to Counter-B and Counter-B would send requests to Counter-A. Originally it was considered to host all 24 pods under the same deployment and have the pods send a request to their own url.

[13]https://flask.palletsprojects.com/en/2.2.x/

[14]https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[15]https://docs.docker.com/

[16]https://kubernetes.io/docs/concepts/services-networking/service/

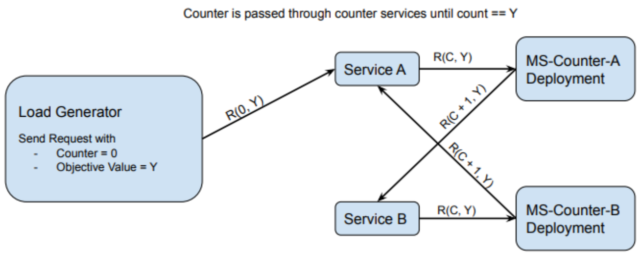

Figure 5: Diagram displays how multiple microservices are simulated by having pods send requests to each other.

However, splitting the counters into two different sets that ping each other seemed like a more realistic approach as normally microservices do not send a request to themselves. Furthermore, pods meshed with consul-k8 would return an empty response when messaging their own service.

4.4 Recording Measurements

To effectively assess the service mesh many measurements must be collected. We are interested in collecting the latency of the different responses, as well as resource utilization per deployment and per service mesh proxy to diagnose any potential bottlenecks or limits.

4.4.1 Generating Requests & Recording Latency

The experiment uses apache ab to simulate a synchronous number of users sending a explicit number of requests. Ab takes a target url, a number of simultaneous users and a total number of requests to send. Then all defined users will concurrently send a request to the specified url, and wait for a response. When a user receives a response it will send a new request, until the total number of requests sent reaches the specified value. While collecting responses, ab stores the percentile of the the responses received and their latency into a file.

A request generator and a benchmark controller are deployed at the beginning of the run. The request generator runs different batches of apache ab requests and sends them to the micro-counter service, collecting the results. It also communicates with the monitoring systems to collect the resource usage per batch. The controller provides the request generator with a collection of users, requests per users and services to simulate. At the end of the run it collects all results for download.

The request generator is deleted and redeployed as an injected pod for every Service Mesh. The benchmark controller remains as an unmeshed pod throughout the different runs, collecting results for the different service meshes and allowing us to remain SSHed into the cluster throughout injection to debug any issues. To ensure that not meshing the benchmark controller does not affect the results a batch was executed with a non injected controller and an injected controller and no noticeable differences in latency or resource usage were found.

4.4.2 Recording Resource Usage

The benchmark uses Prometheus to record the resource usage of the SMTs. Prometheus is an opensource observability system used to collect and monitor different metrics throughout the cluster [21]. Prometheus will consistently scrape the service metrics from exposed endpoints at a specific scraping rate and store them. Developers can then

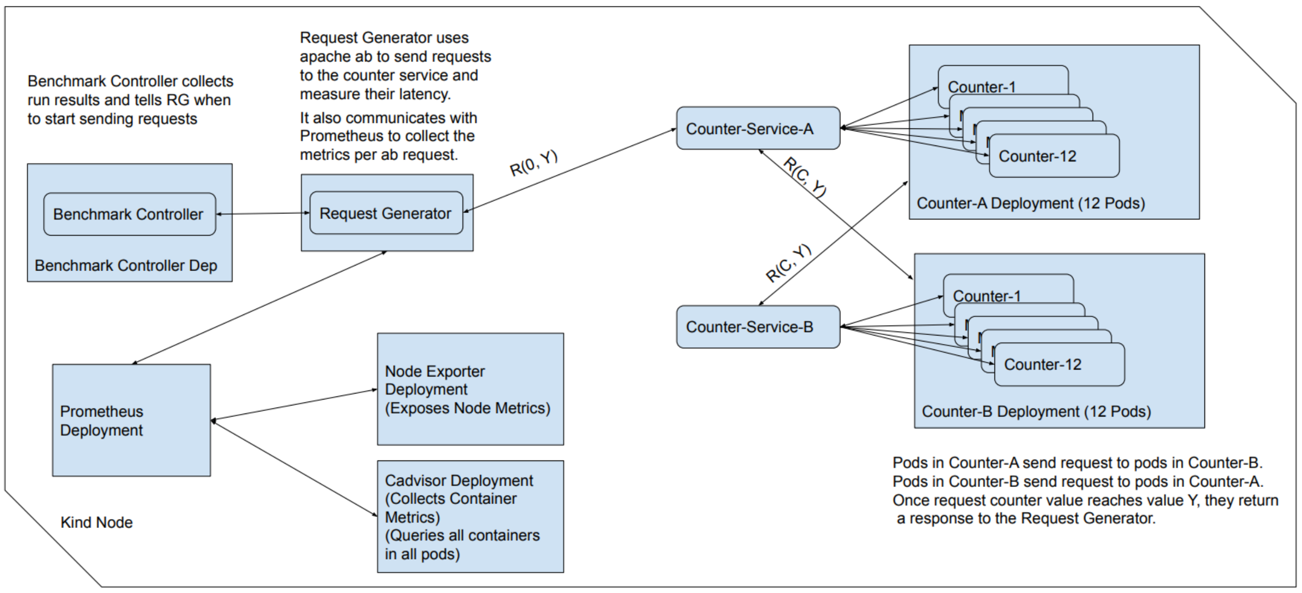
Benchmark Controller collects run results and tells RG when to start sending requests
Request Generator uses apache ab to send requests to the counter service and measure their latency.
It also communicates with Prometheus to collect the metrics per ab request.
Benchmark Controller
Benchmark Controller Dep
Request Generator
R(0, Y)
Counter-Service-A
R(C, Y)
R(C, Y)
Counter-Service-B
Counter-1
Counter-12
Counter-A Deployment (12 Pods)
Counter-1
Counter-12
Counter-B Deployment (12 Pods)
Prometheus Deployment
Node Exporter Deployment (Exposes Node Metrics)
Cadvisor Deployment (Collects Container Metrics) (Queries all containers in all pods)
Kind Node
Pods in Counter-A send request to pods in Counter-B.
Pods in Counter-B send request to pods in Counter-A.
Once request counter value reaches value Y, they return a response to the Request Generator.

Figure 6: Diagram showing the overall experiment architecture

query Prometheus to obtain the collected readings of the application metrics. The metrics Prometheus collects can be exposed through different technologies such as cadvisor, metrics-server or Node Exporter.

Node Exporter[17] is designed to monitor the overall host, collecting metrics for the entire node. Cadvisor[18] is focused on collecting the metrics of running containers inside the cluster. Metrics-server[19] can collect pod metrics and container metrics, but is recommended to be used for scalability purposes not metrics collection. The benchmark uses Node Exporter to collect the metrics of the complete node and cadvisor to collect the metrics of the different containers.

It should be noted that Cadvisor is a very resource intensive application which relies on dynamic scraping to collect metrics. This means the times between its metric collections will vary. Therefore Prometheus may encounter a different number of metric recordings every scrape. To guarantee that enough recordings were being collected to accurately display resource usage the number of pods in the counter deployment were reduced to 24.

4.5 Applying the SMTs

The previous benchmarks used different versions of the meshes with different configuration settings. This could be the reason they obtained different results when compared to each other. To ensure a fair comparison we decided to benchmark the default behaviour of the SMTs with no additional configuration.

We used the latest version of each service mesh; Istio 1.15.3, Linkerd 2.12.3 and Consul-k8s v1.0.2. Each service mesh was installed following the instructions on their website, where applicable the CLI installation was used, not the helm installation. To ensure a fair comparison of the default behaviour of the SMTs, no additional configuration was performed aside from the instructions on the CLI installation page.

When installing Istio, Istio's "*default*" profile was used, as demo has been declared by the Istio team as not compatible with performance tests [22]. Linkerd and Istio were configured to automatically inject everything deployed to the default namespace, however Consul did not support this behaviour and required annotations to be injected.

When executing the benchmark the experiment would first be executed without using a Service Mesh. Then the request generator and the counter deployments would be deleted from the cluster, the corresponding Service Mesh would be applied and the request generator and the counter would be redeployed. The experiment was then executed and all the latency and resource metrics were stored by the benchmark controller. This setup was used for Istio and Linkerd, Consul proved more problematic as unmeshed pods could not communicate with meshed pods by default and using an Ingress controller could affect our results. Therefore the benchmark controller was also meshed with Consul. To guarantee this would not affect the results we also executed the other benchmarks without meshing the controller and found no noticeable difference between their results.

5. EVALUATION

The following section will explore the most noteworthy findings from the experiment. Due to space limitations not all results can be included in the paper, however, a compilation of all results can be found on the experiment's Github `https://github.com/JoseLuisPovedanoPoyato/dissertation`.

The experiment was performed with an increasing number of simultaneous users, microservices and average requests per user (RPU). The number of users ranges from 100 to 800 users, and microservices increase from 20 to 80. Given the scraping limitations of some of the monitoring software, the diagrams included below showcase the usage for 10 RPUs

[17]`https://prometheus.io/docs/guides/node-exporter/`
[18]`https://github.com/google/cadvisor`
[19]`https://github.com/kubernetes-sigs/metrics-server`

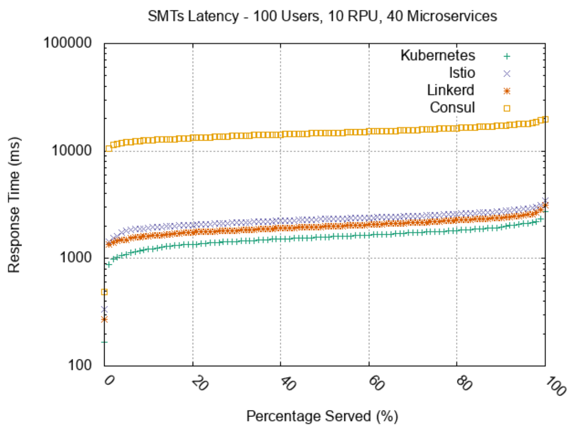

Figure 7: A CDF using logarithmic scale base 10 showing the latency using native Kubernetes, Linkerd, Istio and Consul for an increasing number of simultaneous users.

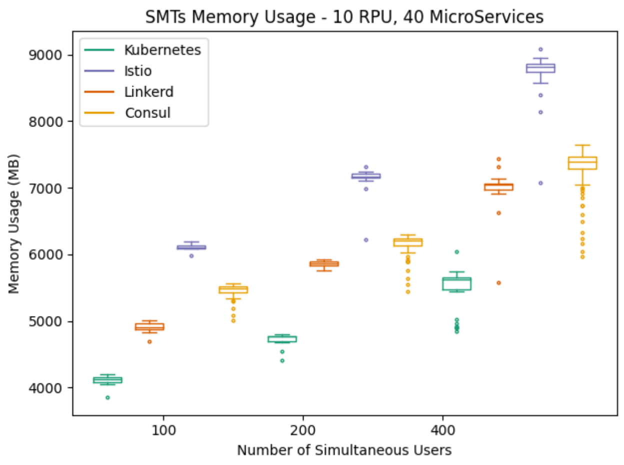

Figure 8: A box plot depicting the memory usage of the three service meshes and native kubernetes for an increasing number of simultaneous users.

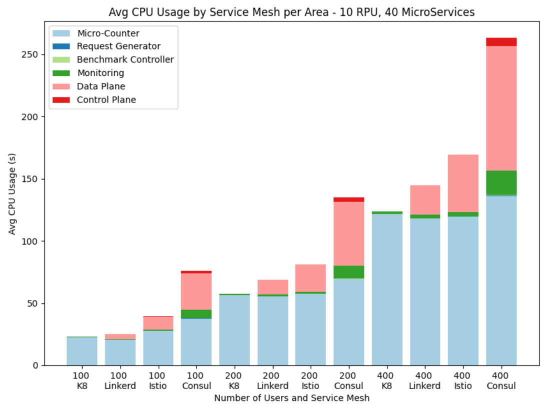

Figure 9: A bar chart depicting the CPU usage by area of the three service meshes and native kubernetes for an increasing number of simultaneous users.

as a larger number of requests per user helps ensure that all areas are scraped sufficiently to collect enough metrics to obtain an accurate reading. Diagrams for the other RPUs, 3 and 5 are still available on the experiment's Github repository.

5.1 Consul Limitations

Consul-k8s out of the box performance had multiple issues. It struggled deploying a deployment of 62 pods, which forced us to reduce the deployment to 32 pods. Its pods were then not capable of sending requests to themselves, returning empty requests when curling their own url. Unlike the other meshes, Consul could not execute the benchmark with more than 400 Users and 40 Microservices. Furthermore, the rate at which the monitoring applications collected metrics was too fast paced for Consul and it crashed the benchmark.

To try to debug some of these issues we contacted Hashicorp, the Consul team. However at the time of writing we have still not received any responses.

A smaller scale version of the benchmark was executed, with reduced scraping rates, less users and less microservices. In this version, Consul's memory usage lied between Istio's and Linkerd's, yet it was far more CPU intensive, specially in the control plane. Consul's latency was considerably larger than the other meshes, to the point we decided to exclude it from the other comparisons.

These findings should not be interpreted as an indication of Consul's inefficacy as a service mesh. CloudCover showed Consul performs very well with some customisation, and a previous benchmark conducted by Hashicorp demonstrated that Consul can perform exceptionally well when properly optimised, updating 10,000 nodes and 172,000+ services in under 1 second [23]. However, it is worth noting that Consul's default performance, appears to be somewhat underwhelming compared to the other service meshes.

5.2 Latency Analysis

Throughout the experiment, excluding Consul, Istio had the highest latency across the different scale number of users. Linkerd performed better than Istio, with up to 18% less latency, while unmeshed Kubernetes had the least latency, reaching 14% less latency than Linkerd.

The difference between Linkerd and Istio increases with the number of microservices, as Linkerd's median values become increasingly lower than Istio's lower quartile latency.

Istio and Linkerd were more predictable in their variance than Kubernetes, which experienced larger outliers. When plotting different scales of users and microservices that amount to the same number of connections for the mesh to handle, i.e. 200 users - 10 services and 100 users - 20 services, Istio and Linkerd had minimal variance, producing mostly overlapping shapes (figure 11). In most cases Istio would experience a slightly greater latency on the final percentiles when increasing the number of concurrent users rather than the number of microservices, suggesting Istio would struggle more handling a larger traffic influx than when scaling services horizontally. Linkerd would sometimes show this behaviour but to a lesser extent, while native Kubernetes was quite unpredictable.

5.3 Memory Analysis

Memory wise, the service meshes followed the same beahviour noted for latency. Native Kubernetes was the least resource intensive, followed by Linkerd then Istio. The difference in memory overhead is considerably more notable than latency,

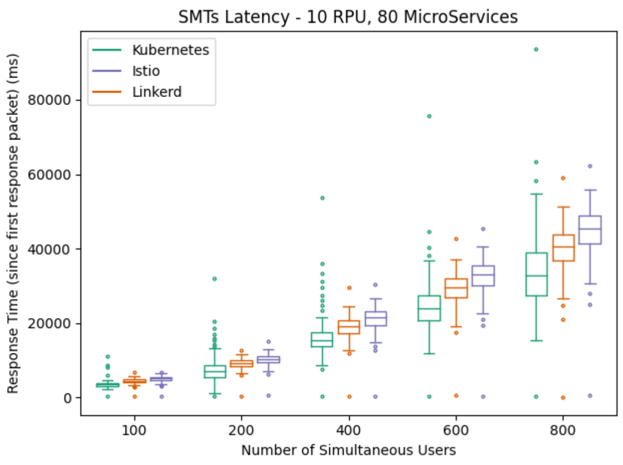

Figure 10: A box-plot showing the latency using native Kubernetes, Linkerd, and Istio for an increasing number of simultaneous users.

with Linkerd requiring 17% to 28% more memory than Kubernetes and Istio requiring 33% to 50% more memory than Linkerd. This difference increases as the number of users and services increase.

The memory distribution breakdown shows that most of Istio's memory consumption originated in the mesh's Data Plane, which eventually overpasses the main application's memory consumption as the number of microservices and users increases (figure 15). When compared to Linkerd's data plane, Istio consumed up to 475% more memory. This difference was not at 80 services but at 40 services.

Istio's data plane employs the Envoy proxy, which seems to be responsible for this considerably larger overhead when compared to Linkerd. However, Consul also employs the Envoy proxy, yet its memory breakdown shows considerably lower values. This suggests that even when using the same technology, different configurations can significantly affect the results.

5.4 CPU Utilization

The CPU usage made by the service meshes follows the same trends as memory and latency, but less pronounced.

Figure 14 displays a breakdown of the CPU utilization of Kubernetes, Istio and Linkerd for 20 and 80 microservices. In both service meshes the data plane is responsible for most of the CPU usage, while the control plane's usage is barely noticeable. Figure 15 compares Istio, Linkerd and Consul's control plane. Consul had the most CPU consuming control plane, consuming approximately 60 times more than Istio. Istio was the next most intensive application, followed by Linkerd. This would mean that istiod requires more resources than linkerd's control plane.

5.5 Discussion

To provide context to the performance analysis below, lets first highlight the qualitative differences between the service meshes. Linkerd is the most minimalistic service mesh, with less configuration and custom designed micro-proxies. It is a service mesh designed for Kubernetes which focuses on ease of use and performance. Istio is designed to run in both Kubernetes and Virtual Machines and can join them under a single control plane. Given its use of Envoy it has a larger

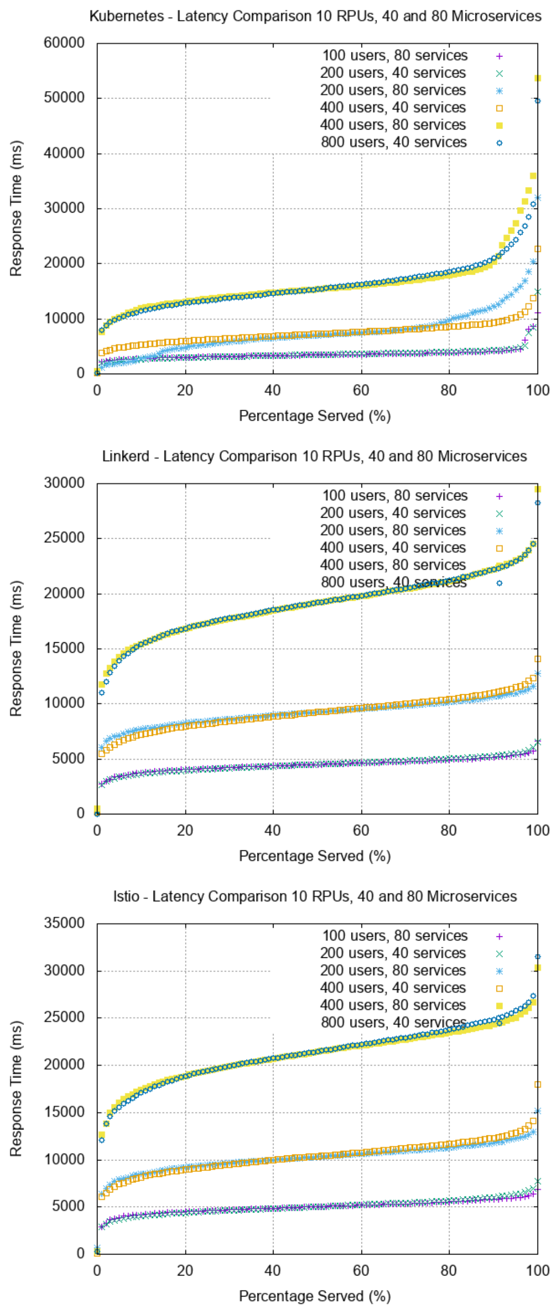

Figure 11: Three CDFs for Kubernetes, Istio and Linkerd for increasing microservices and users paired to display differences when handling differing users and services that amount to the same service calls for the mesh

feature set focused on traffic management, for instance it includes traffic splitting, distributed tracing its own ingress controller. While Linkerd does not include these features by default, the Buoyant team has developed extensions to enable many of these features if desired. Consul on the other hand was a Service Mesh designed for VMs, which eventually was adapted to Kubernetes. It focuses on multi-cluster environments, allowing users to deploy multiple control planes to set up different connected network environments across

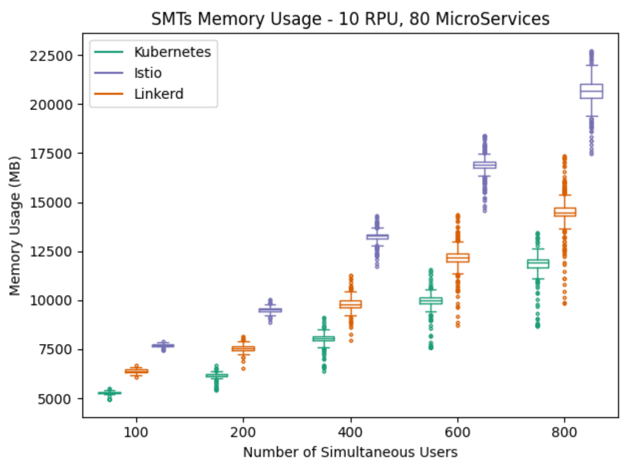

Figure 12: A box-plot showing the memory consumption using native Kubernetes, Linkerd, and Istio for an increasing number of simultaneous users.

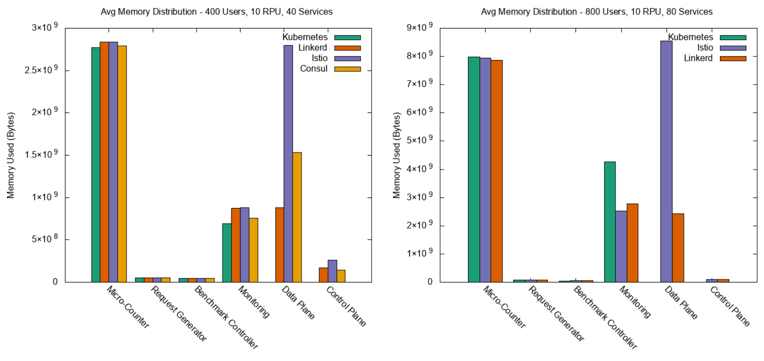

Figure 13: A break down by area showing the memory consumption of K8, Linkerd and Istio and Consul for 400 users - 40 microservices (left) and K8, Linkerd and Istio for 800 users - 80 microservices (right).

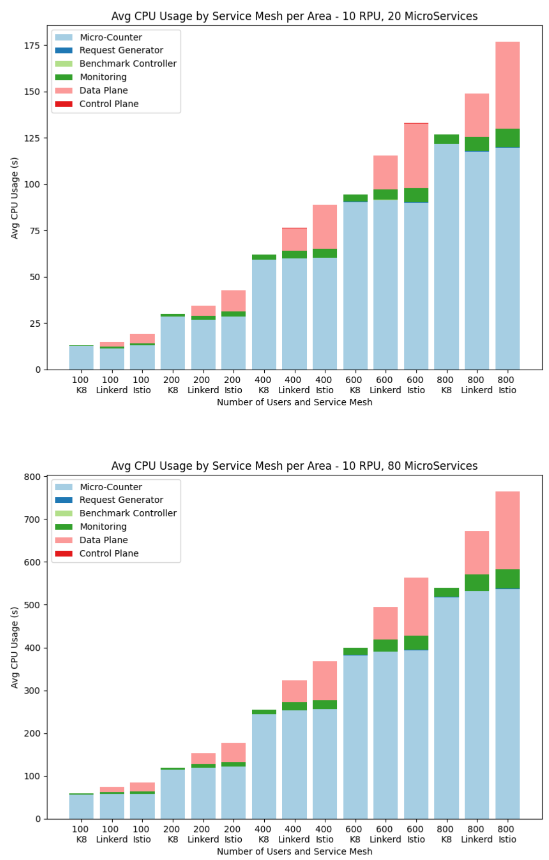

Figure 14: A comparison of CPU utilization for native Kubernetes and each service mesh at the scales of 20 (top) and 80 microservices (bottom). The total value in each case is dissected into the different application sub-processes.

distributed systems.

When analysing the results we can see that all service meshes negatively affected the application's performance. Latency, Memory and CPU Utilization increased. The difference in performance between native kubernetes and the meshes scaled alongside the application, increasing as the incoming traffic or the number of microservices grew. This highlights the importance of choosing a service mesh that fits the requirements and scale of the project.

Overall, Linkerd was the most cost-effective service mesh. The difference between Istio's and Linkerd's performance can be attributed mainly to Envoy against Linkerd's microproxy. When designing the proxy, Buoyant believed that the proxy as the foundation of the data plane should be as simple as possible to facilitate operational simplicity and help guarantee security [24]. The proxy gives Linkerd the overall advantage when it comes to resource usage and latency. The Envoy proxy on the other hand is equipped to handle more complex tasks, which Buoyant incorporates into Linkerd through extensions or tools built on top of the mesh, to contain the additional resource consumption to a minimum.

Consul-k8 default configuration struggled with the experiment, achieving considerably higher levels of latency. Despite using Envoy like Istio, Consul's memory usage was considerably lower, yet the mesh exhibited considerably larger CPU usage values. This large difference in performance when compared to Istio is very interesting as both rely on the same proxies, highlighting the importance of proper configuration which suits the hardware equipment and application requirements. The reasons for this large difference are unclear, some hypothesis could be that Consul is more dependent on its Control Plane, slowing down the overall application or that Hashicorp's default configuration of Envoy relies more in its CPU rather than using more memory which could lead to a bottleneck. However, we have not managed to prove any of these theories.

Ultimately, Linkerd proved to be very resource efficient and minimalistic, therefore it will be ideal for networks that do not require complicated configurations of the mesh. Istio was not as efficient as Linkerd, however it might offer more options in situations where a more customised solution is required. Consul is more complicated to set up, based on its results will require a larger effort to be customised and did not work anywhere near as effectively as Istio or Linkerd out of the box. However, other benchmarks have demonstrated that when properly optimised it can be a very powerful mesh and may be better equipped to handle overly large complex networks and multi-platform solutions [23].

An interesting note is how broad the meshes are further

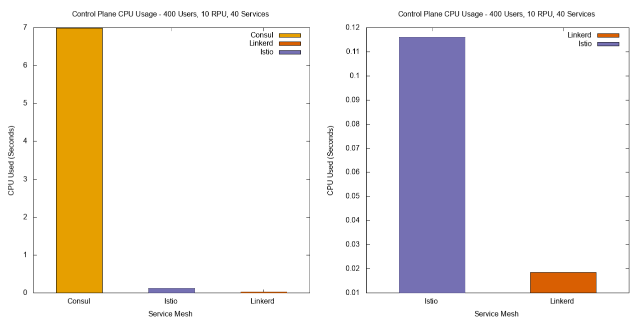

Figure 15: A comparison of CPU utilization of Istio, Linkerd and Consul's Control Plane (left). Scaled down to Istio and Linkerd for visibility (right)

affects their overhead. Consul is platform agnostic and performed considerably worse than the service meshes customised specifically for Kubernetes. Istio can be deployed on VMs as well as kubernetes, and relies on the Envoy proxies for its data plane which are capable of running in many different machines. While Linkerd is a specialised solution for Kubernetes. This suggests that the optimum solution would still be a network curated specifically for an application, with service meshes enabling users to achieve production level networks with a significant reduction of effort at the sake of some performance.

Ultimately, we believe the experiment was a success. We met our objectives, we successfully depicted the performance and overhead of different service meshes as an application scales. We presented an in-depth comparison of the results detailing the different meshes performance and we used our findings combined with other research to highlight different competencies and limitations of the meshes in their default implementation.

5.6 Limitations

This research aims to depict a general understanding of the differences between well known service meshes and their impact on overhead and performance. However, there are some limiting factors that must be taken into consideration when interpreting our results.

The study deploys a fixed amount of microservices that send a variable number of requests between each other before returning a response. This allows us to test applications with different numbers of dependent services dynamically, without having to do major deployment changes. However, the study is not actually composed of a setup that big. Therefore our results are an approximation to the results of traditional production applications of that scale.

The monitoring systems were not meshed into the service meshes to try to maintain scraping performance as consistent as possible. However, some service meshes allow you to integrate the monitoring systems into the mesh to facilitate metric collections. This behaviour is not explored in this study and we encourage future pieces of work to explore how well monitoring solutions are integrated into different meshes.

Ultimately, the study evaluated the service meshes in their default configuration. Although this approach provides some insight into the meshes' operational capabilities, particularly those that have undergone minimal customization, it does not accurately depict the complete potential of the mesh.

6. CONCLUSIONS

Service meshes separate network logic from the business application logic, and enrich it with additional features. This paper proposed a new benchmark to explore how service meshes affect performance as they scale horizontally, by creating an application that redirects requests to simulate applications with different numbers of dependent services.

The study compared the Istio, Linkerd and Consul service meshes, focusing on how increasing the number of simultaneous users and microservices affects the meshes' performance, comparing their latency, CPU and Memory utilization against a native kubernetes baseline.

We found that all the service meshes increase the resource overhead and lead to a higher latency. This effect scaled alongside the application, highlighting the importance of choosing a fitting service mesh to the applications requirements and scale. Overall, Linkerd was the fastest and most resource efficient mesh, mostly because its data plane used a simple custom designed micro-proxy instead of a more complex solution. Istio's data plane relied on the Envoy proxy and thus was not as efficient as Linkerd, however it is capable of offering more complex features without incorporating additional tools. Consul struggled with the experiment and its 'out of the box' performance was very underwhelming compared to Istio and Linkerd. However, other works have demonstrated that when properly optimised it can be a very powerful mesh.

7. FUTURE WORK

This experiment subjected the meshes to the stress of large microservice networks to explore how different SMTs scale horizontally. Although Istio, Linkerd, and Consul were extensively compared, several other service meshes were not included. Additionally, due to performance reasons, the number of simulated microservices was restricted to 80. Therefore, the benchmark could be improved by including more service meshes, like Kuma or Tetrate, and refactoring the service simulation to be less resource intensive and scale to a larger number.

Further research outside of scaling microservices could explore the performance of different meshes in realistic applications. This experiment used an application that simulated microservices to explore horizontal escalation. However, benchmarking intricate microservice architectures that can take more advantage of the SMTs additional functionality, for instance the DeathStarBench[20], could lead to a comprehensive comparison of different service mesh features.

Finally, a current problem in the microservice architecture is how observable the services are. To tackle this, multiple monitoring solutions such as Prometheus or Node Exporter have been developed. Future research could entail exploring how different service mesh technologies integrate themselves with popular monitoring solutions. These findings could assess the usability of the service mesh when integrated with other tools.

Acknowledgments. I would like to express my gratitude to Dr Yehia Elkhatib, my project supervisor, for his support and insight throughout the project. Additionally, I would like to thank Dr Gethin Norman for looking out for the well-being of computer science since first year, and Derek

[20]`https://github.com/delimitrou/DeathStarBench`

Somerville for his ongoing support to placement students. Lastly, I would like to thank my family, I learned a lot from them.

8. REFERENCES

[1] Kinvolk, "Performance Benchmark Analysis of Istio and Linkerd." `https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd/`, May 2019.

[2] L. L. Erik Dahlberg, "Benchmarking Istio 1.5.0 and Linkerd 2.7.1 (Master Thesis)." `https://elastisys.com/benchmarking-istio-linkerd-erik-dahlberg-master-thesis/`, July 2020.

[3] "CloudCover | Benchmarking Istio, Consul, and Linkerd." `https://cldcvr.com/news-and-media/blog/benchmarking-istio-consul-and-linkerd/`.

[4] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering* (M. Mazzara and B. Meyer, eds.), pp. 195–216, Cham: Springer International Publishing, 2017.

[5] "Microservices use in organizations worldwide 2021."

[6] Y. Sun, S. Nanda, and T. Jaeger, "Security-as-a-Service for Microservices-Based Cloud Applications," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 50–57, Nov. 2015.

[7] "CNCF Annual Survey 2022." `https://www.cncf.io/reports/cncf-annual-survey-2022/`, Jan. 2023.

[8] S. Ashok, P. B. Godfrey, and R. Mittal, "Leveraging Service Meshes as a New Network Layer," in *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, HotNets '21, (New York, NY, USA), pp. 229–236, Association for Computing Machinery, Nov. 2021.

[9] "Service meshes are on the rise – but greater understanding and experience are required." `https://www.cncf.io/blog/2022/05/17/service-meshes-are-on-the-rise-but-greater-understanding-and-experience-are-required/`, May 2022.

[10] M. Ganguli, S. Ranganath, S. Ravisundar, A. Layek, D. Ilangovan, and E. Verplanke, "Challenges and Opportunities in Performance Benchmarking of Service Mesh for the Edge," in *2021 IEEE International Conference on Edge Computing (EDGE)*, pp. 78–85, Sept. 2021. ISSN: 2767-9918.

[11] C. Box (Google), "Introducing istiod: simplifying the control plane." `https://istio.io/latest/blog/2020/istiod/`.

[12] "Linkerd and SMI." https://linkerd.io/2019/05/24/linkerd-and-smi/.

[13] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service Mesh: Challenges, State of the Art, and Future Research Opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 122–1225, Apr. 2019. ISSN: 2642-6587.

[14] X. XIE and S. S. Govardhan, "A Service Mesh-Based Load Balancing and Task Scheduling System for Deep Learning Applications," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pp. 843–849, May 2020.

[15] F. Hussain, W. Li, B. Noye, S. Sharieh, and A. Ferworn, "Intelligent Service Mesh Framework for API Security and Management," in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pp. 0735–0742, Oct. 2019. ISSN: 2644-3163.

[16] A. El Malki and U. Zdun, "Guiding Architectural Decision Making on Service Mesh Based Microservice Architectures," in *Software Architecture* (T. Bures, L. Duchien, and P. Inverardi, eds.), Lecture Notes in Computer Science, (Cham), pp. 3–19, Springer International Publishing, 2019.

[17] A. O. Duque, C. Klein, J. Feng, X. Cai, B. Skubic, and E. Elmroth, "A Qualitative Evaluation of Service Mesh-based Traffic Management for Mobile Edge Cloud," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 210–219, May 2022.

[18] A. Khatri and V. Khatri, *Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul*. Packt Publishing Ltd, Mar. 2020. Google-Books-ID: Mg3aDwAAQBAJ.

[19] M. R. S. Sedghpour, C. Klein, and J. Tordsson, "Service mesh circuit breaker: From panic button to performance management tool," in *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, HAOC '21, (New York, NY, USA), pp. 4–10, Association for Computing Machinery, Apr. 2021.

[20] "Benchmarking Linkerd and Istio: 2021 Redux." `https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/`.

[21] Prometheus, "Overview | Prometheus."

[22] "Istio Configuration Profiles." `https://istio.io/latest/docs/setup/additional-setup/config-profiles/`.

[23] A. Mishra and P. McCarron, "HashiCorp Consul Global Scale Benchmark." `https://www.hashicorp.com/blog/hashicorp-consul-global-scale-benchmark`.

[24] "Why Linkerd doesn't use Envoy." https://linkerd.io/2020/12/03/why-linkerd-doesnt-use-envoy/.