Proceedings of 14th IOE Graduate Conference
Peer Reviewed
Year: 2023 Month: December Volume: 14
ISSN: 2350-8914 (Online), 2350-8906 (Print)

Optimizing Microservice Communications: A Hybrid Service Mesh Approach Merging Linkerd and Istio with Maglev Hashing

Rabin Raj Gautam [a], Daya Sagar Baral [b], Ganesh Gautam [c]

[a, b] *Department of Electronics & Computer Engineering, Pulchowk Campus, IOE, Tribhuvan University, Nepal*
[a] gautamrabinraj@gmail.com, [b] dsbaral@pcampus.edu.np

Abstract
As the tech world continues its shift from monolithic software systems to a collection of smaller microservices, new challenges emerge. One of the main challenges is overseeing and ensuring the safety of the interactions between these microservices. This is where the concept of a service mesh shines, playing a critical role in helping everyday applications communicate securely and efficiently. In our study, we closely examined two leading options in this realm: Linkerd and Istio. Our findings indicated that, in many areas, Linkerd came out on top. Armed with this knowledge, we ventured to design a new, hybrid service mesh, integrating the strengths of both Linkerd and Istio. A standout feature of our design is the inclusion of Maglev Hashing, which not only propels its performance beyond that of Linkerd but also ensures lower computer power and memory demands. This research offers valuable insights and points toward a more efficient future for managing the ever-increasing microservices in day-to-day applications.

Keywords
Microservices, Service mesh, Linkerd, Istio, Maglev Hashing

1. Introduction

Kubernetes, a widely embraced open-source platform, streamlines the orchestration and administration of containerized workloads and services. Its robust features and community support make it a go-to solution for efficiently managing distributed applications in diverse environments [1, 2]. Originally developed by Google, it became open-source in 2014. With the rise of container deployment, Kubernetes emerged as a lightweight virtualization system, offering tools for container management. It ensures uninterrupted operations by automatically restarting containers and provides features like load balancing and automatic scaling. To meet enterprise needs, companies like Google, Amazon, and Microsoft offer fully managed Kubernetes solutions that are widely adopted by third-party companies to build modern cloud architectures. These solutions facilitate the deployment of millions of containers daily, and notable companies such as Netflix heavily rely on Kubernetes for managing their services composed of numerous microservices. However, the rapid adoption of Kubernetes also raises security concerns [2, 3, 4, 5].

The rapid advancements in cloud computing, container technology, and microservice architecture [6] have brought about new security challenges in cloud computing platforms. Security and privacy concerns have become increasingly common in cloud services. With Kubernetes being the dominant container cloud technology, ensuring its security is of utmost importance [7].

In broad terms, the shift from servers and virtual machines to cloud-native environments [8], utilizing containers and Kubernetes, did not eradicate the threats or consequences of attacks. It also did not absolve cloud providers of security responsibilities. Instead, it introduces a range of new and unique security challenges [9]. Kubernetes relies on containers to deploy applications, and these containers are constructed from pre-existing images. If an attacker manages to insert harmful code into a container image, they have the potential to compromise not only the application running inside that container but also the Kubelet, which enables communication between different nodes. Additionally, the containers themselves executing on the nodes can pose a significant risk, especially when considering the vulnerability of the cluster network as a whole. These factors combined can make the Kubernetes cluster susceptible to attacks.

A secure Kubernetes cluster requires a comprehensive approach that addresses multiple levels of security. As depicted in figure:1, the breach of a Kubernetes cluster can be categorized into four key aspects: infrastructure, Kubernetes, containers, and applications [10]. Each aspect demands specific attention and measures to ensure the overall security of the cluster. Infrastructure security forms the foundation of the security pyramid, representing level four. This level encompasses various aspects such as network security, storage security, securing the host operating system, and managing access to the hosts.

Moving up to level three, it focuses on the computing platform components, with Kubernetes offering configurable policies like Identity and Access Management (IAM), Role-Based Access Control (RBAC), Security Context Control (SCC), and Pod Security Policies (PSP) [11]. These measures help secure the deployments in the upper levels of the pyramid.

Level two operates within the core of the running pods or containers [12, 13]. It ensures that data securely flows between containers and implements safeguards to protect the integrity and confidentiality of the data.

Finally, level one, the top level, is centered around application security. This level provides functionalities that are directly tied to the applications, such as authentication and

authorization mechanisms. It addresses the security needs specific to the applications themselves. By addressing each of these levels and their respective security concerns, a comprehensive security approach can be established to protect a Kubernetes cluster effectively.

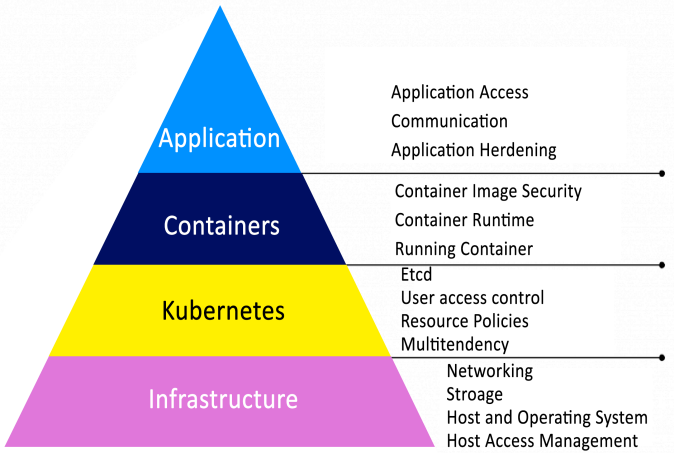

Figure 1: The many dimensions of Kubernetes security

According to figure:1, security concerns at the pod or container level primarily arise from the risk of unauthorized or untrustworthy third parties eavesdropping on data. There are two main types of container-related data that are susceptible to such eavesdropping.

The first type involves user requests transmitted over HTTP from external sources to applications within a Kubernetes cluster. These requests can contain sensitive information, such as user credentials or confidential data, which need to be protected from interception or unauthorized access.

The second type of data pertains to the communication between application pods within a Kubernetes cluster using transport protocols. This data flow, occurring internally within the cluster, must also be safeguarded to prevent unauthorized parties from intercepting or tampering with the information

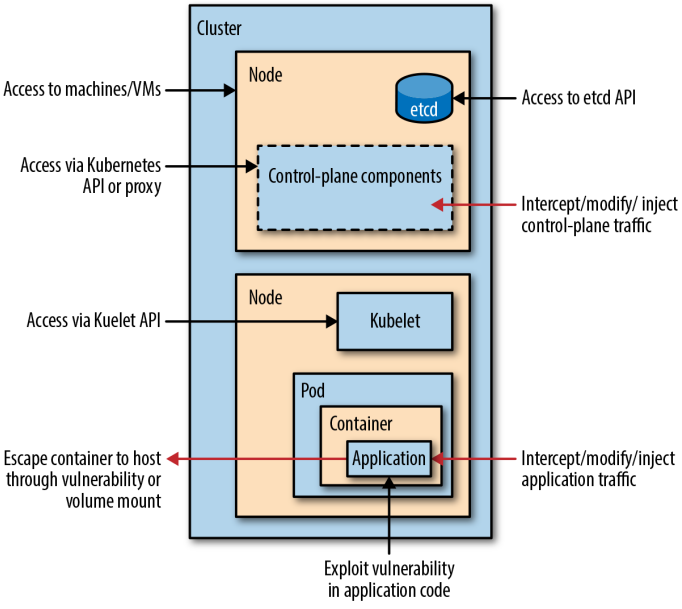

Figure 2: Illustrates several attack vectors.

being transmitted. Attackers have the ability to initiate various attacks on servers or machines within a Kubernetes environment by generating artificial traffic load. This includes flooding the target's service with network packets, which is a malicious act known as a DoS or Distributed Denial of Service (DDoS) attack.

Securing these two types of container-related data is crucial to maintain the confidentiality and integrity of the communication within a Kubernetes cluster. In addition to the mentioned attacks, there are various other attack vectors that exist which is shown in figure:2. In a cloud environment, both user-facing (North-South) and inter-service (East-West) traffic exist. Protecting the confidentiality of East-West traffic is crucial to prevent attacks and monitor suspicious activities. Our research aims to implement Zero Trust for securing microservices' East-West traffic in a containerized environment. To achieve this, we address the following sub-questions: how to regulate the flow of East-West traffic and how to ensure confidentiality during data transit.

2. Problem Statement

A survey from StackRox shows that 94% of organizations have encountered serious security problems in the Kubernetes container environment [14]. According to the article by StackRox [14], Tesla experienced a breach in their cloud infrastructure in March 2018 due to insecurely configured Kubernetes clusters. This allowed attackers to gain unauthorized access to Tesla's internal systems and unlawfully acquire sensitive data, including customer information and valuable trade secrets. Similarly, in June 2020, hackers successfully infiltrated a K8s toolkit and utilized it to distribute malware specifically designed for cryptocurrency mining across multiple clusters. The malware effectively utilized the resources of the compromised clusters to mine cryptocurrency for the benefit of the attackers. Furthermore, in December 2021, several companies became victims of data theft as a result of exploiting a vulnerability within the Kubernetes API. This allowed the attackers to gain unauthorized access to the companies. Kubernetes clusters, leading to the theft of sensitive data, including customer information and financial records. Furthermore, a survey conducted by the CNCF (see figure 3) shows that security and networking are the top challenges for Kubernetes users. The main focus of this paper is to introduce an encryption-as-a-service architecture that specifically addresses the security concerns at level two in figure:1. There are several reasons for emphasizing this level. Firstly, security at the application level is typically handled by default within the application itself, while security at the platform and infrastructure levels is usually well-provided and continuously upgraded within a Kubernetes cluster. Secondly, the security at the container level, particularly the data flow between containers using IPC communication, is often lacking adequate protection. Therefore, there is a need for a straightforward, adaptable, and efficient solution to encrypt this data flow.

Furthermore, In today's microservices-driven ecosystem, the prominence of service meshes like Linkerd and Istio has surged due to their facilitative role in service communication,

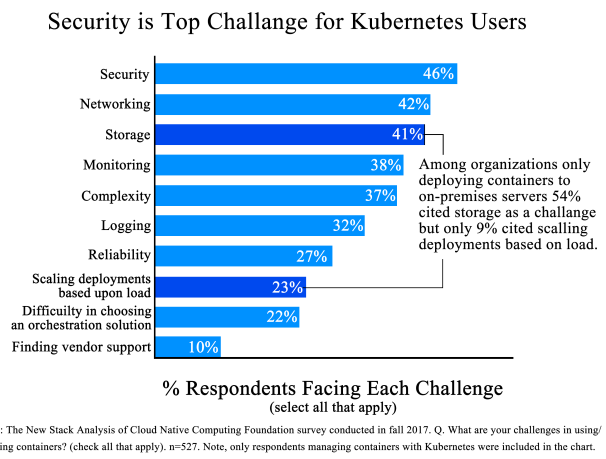

Figure 3: Kubernetes top challenges (source: thenewstack)

security, and observability [15]. Both, while popular, have shown specific advantages and limitations, especially in terms of memory and CPU usage, latency, and performance overhead. This prompts a crucial inquiry: Is it possible to create a hybrid service mesh that integrates the best of both Linkerd and Istio, leading to an optimized solution for microservices communication challenges? This thesis seeks to construct such a hybrid model, juxtaposing its performance against the individual capabilities of Linkerd and Istio. The overarching goal is to determine if a combined approach can overcome the inherent constraints of the existing service meshes, offering a more streamlined, efficient, and scalable framework for contemporary microservices infrastructures. Furthermore, the existing envoy proxy [16] uses Round Robin as a load balancing, which consumes more computing resources.

3. Literature Review

Traditionally, researchers have suggested using a service mesh to encrypt data flow between microservices on Kubernetes [17]. The service mesh acts as a special layer that handles communication between services. It functions similar to how a container separates the application from the operating system. By abstracting the communication process, the service mesh facilitates the management of keys, certificates, and TLS configuration for continuous encryption. Additionally, it enables policy-based authentication, allowing secure encrypted communication (service-to-service) and user authentication.

Another proposal by Sarada Prasad et al. introduces Yugala, a lightweight decentralized encrypted cloud storage architecture that utilizes blockchain technology. Yugala aims to maintain file confidentiality, eliminate centralized data deduplication, and enhance file integrity [18].

Anton Vedeshin et al. present a secure and reliable infrastructure and architecture solution. Their approach incorporates the limitations of the computing process into the defense strategy, making distributed file storage and transmission highly secure. The key idea is to replace asymmetric or public key encryption functions with cryptographic hash functions that ensure non-key, conflict, second pre-image, and antigenic image properties [19].

Luigi Coppolino et al. propose a method called Virtual Security Zone (VISE) that combines Intel SGX and homomorphic encryption. VISE relocates the execution of sensitive homomorphic encryption primitives (like encryption) to the cloud, within the remotely verified SGX secure area. It then utilizes the secure area's available memory resources to process the sensitive data outside the secure area. This approach safeguards the data in use from privileged attackers in the cloud [20].

Furthermore, in recent academic explorations from 2018, the role and benefits of service mesh in various contexts have been elucidated. The study by [21] outlined the inherent advantages of implementing service mesh, while [22] touched upon its significance in microservices governance. Furthermore, [23] examined its potential in enhancing cloud applications, particularly for the Internet of Things (IoT). The research presented by [24] shed light on its applicability in the architectural frameworks for edge environments. Additionally, [25] underscored the relevance of service mesh in the control mechanisms of MAPE (Monitor, Analyze, Plan, Execute) loops.

4. Methodology

4.1 Hybrid Envoy Building Blocks

In the development of our hybrid service mesh model, we integrated elements from both the Linkerd and Istio methods. Additionally, we incorporated Maglev Hashing, a technique developed by Google, to ensure a balanced distribution of user requests. This prevents any segment of our system from becoming overwhelmed. The integration of this method is anticipated to optimize our system's performance by reducing computational efforts and improving response times. Ultimately, our objective is to enhance the overall system efficiency. Following figure 4 shows the traffic request flow:

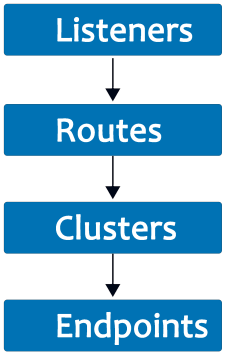

Figure 4: Building Blocks of Hybrid Envoy

It starts with the listeners. Envoy exposes listeners that named network locations, either an IP address and a port or a Socket path. Envoy receives connections and requests through listeners. Consider the following Envoy configuration:

```
static_resources:
  listeners:
  - name: listener_0
```

```
address:
  socket_address:
    address: 0.0.0.0
    port_value: 10000
filter_chains: [{}]
```

In the provided Envoy configuration, a listener termed "listener_0" is specified with an address of 0.0.0.0 and a port of 10000. This configuration implies that Envoy actively awaits incoming requests on the address 0.0.0.0:10000.

The "filter_chains" [26] field remains unpopulated, signifying that no supplementary actions are mandated post packet reception. To transition to the subsequent component, i.e., routes, it is essential to instantiate one or multiple network filter chains ("filter_chains") as shown in figure 5

Envoy classifies filters into three primary categories: listener filters, network filters, and HTTP filters. Listener filters are activated immediately upon packet reception, predominantly interacting with the packet headers. Notable instances of listener filters encompass the proxy listener filter, responsible for extracting the PROXY protocol header, and the TLS inspector listener filter, which discerns if the incoming traffic is TLS-encoded and subsequently retrieves data from the TLS handshake if applicable.

Every incoming request via a listener may traverse several filters. Moreover, configurations can be tailored to opt for diverse filter chains contingent on specific properties of the incoming request or connection.

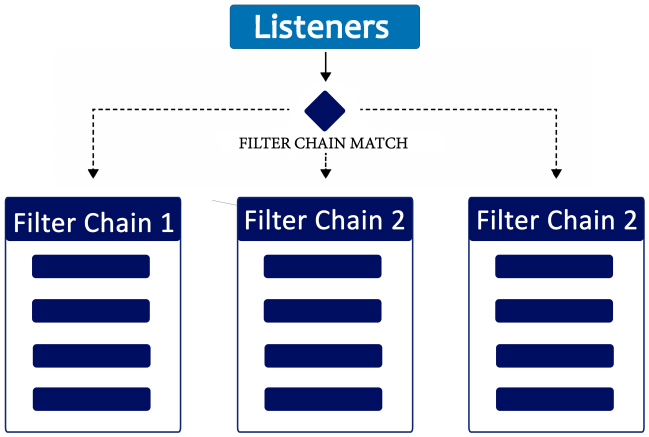

Figure 5: Filter Chains

In Envoy's framework, there exists a distinctive, intrinsic network filter termed the HTTP Connection Manager filter, often abbreviated as HCM. The HCM filter (see figure 6) has the proficiency to transmute raw byte data into messages at the HTTP level. Its capabilities encompass access logging, request ID generation, header manipulation, route table management, and statistics aggregation.

While multiple network filters can be delineated for each listener (with the HCM being one such filter), our system additionally facilitates the specification of multiple HTTP-tier filters within the purview of the HCM filter. These HTTP filters can be articulated under the designated field, "http_filters".

In the sequence of HTTP filters, the terminal filter should

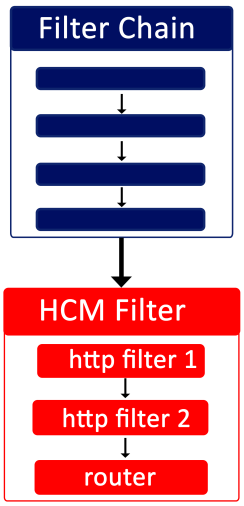

Figure 6: HCM filter

invariably be the router filter, identified by 'envoy.filters.HTTP.router'. This filter shoulders the responsibility of executing routing operations, leading us to the subsequent foundational component: routes.

The route configuration is encapsulated within the HCM filter, specifically under the 'route_config' field. Using this configuration, we can evaluate incoming requests based on metadata aspects such as URI, headers, etc. Subsequent to this evaluation, we can delineate the traffic direction.

A salient element within the routing configuration is termed a 'virtual host'. Every virtual host is endowed with a distinctive name, which is primarily utilized for statistics generation and not for the routing process. Furthermore, each virtual host comprises an array of domains to which it routes. Let's consider the following route configuration and the set of domains:

```
route_config:
  name: my_route_config
  virtual_hosts:
  - name: ioe_host
    domains: ["ioe.io"]
    routes:
    ...
  - name: test_hosts
    domains: ["test.ioe.io", "qa.ioe.io"]
    routes:
    ...
```

When an incoming request is directed towards 'ioe.io'—that is, if the Host/Authority header is assigned one of its values—the routes outlined within the 'ioe_hosts' virtual host are then evaluated and processed.

Conversely, should the Host/Authority header indicate either 'test.ioe.io' or 'qa.ioe.io', the routes encompassed by the 'test_hosts' virtual host are initiated. With such a configuration, it's feasible to employ a solitary listener (0.0.0.0:10000) to manage an array of primary domains. If we specify multiple domains in the array, the search order is the following:

1. Exact domain names (e.g. `ioe.io`)
2. Suffix domain wildcards (e.g. `*.ioe.io`)
3. Prefix domain wildcards (e.g. `ioe.*`)
4. Special wildcard matching any domain (`*`)

After Envoy matches the domain, it's time to process the routes field within the selected virtual host. This is where we specify how to match a request and what to do next with the request (e.g., redirect, forward, rewrite, send a direct response, etc.).

Let's look at an example:

```
static_resources:
  listeners:
  - name: listener_0
    address:
      socket_address:
        address: 0.0.0.0
        port_value: 10000
    filter_chains:
    - filters:
      - name: envoy.filters.network.
      http_connection_manager
        typed_config:
          "@type": type.googleapis.com/
          envoy.extensions.filters.network.
          http_connection_manager.v3.
          HttpConnectionManager
          stat_prefix: hello_world_service
          http_filters:
          - name: envoy.filters.http.router
            typed_config:
              "@type": type.googleapis.com/
              envoy.extensions.filters.http.
              router.v3.Router
          route_config:
            name: my_first_route
            virtual_hosts:
            - name: direct_response_service
              domains: ["*"]
              routes:
              - match:
                  prefix: "/"
                direct_response:
                  status: 200
                  body:
                    inline_string: "yay"
```

The initial segment of the configuration aligns with our previous observations. We have incorporated the HCM filter, designated the statistics prefix as 'hello_world_service', introduced a singular HTTP filter (router), and outlined the route configuration. While the capability to send a direct response from the configuration is often advantageous, typically there exists a collection of endpoints or hosts to which we direct the traffic. In Envoy, this is realized by specifying clusters.

Clusters encompass a collective of analogous upstream hosts that are receptive to the incoming traffic. Such clusters can be constituted by a compilation of hosts or IP addresses to which our services are actively responsive.

For example, let's say our hello world service is listening on 127.0.0.0:8000. Then, we can create a cluster with a single endpoint like this:

```
clusters:
- name: hello_world_service
  load_assignment:
    cluster_name: hello_world_service
    endpoints:
    - lb_endpoints:
      - lb_policy: maglev
        maglev_lb_config:
            table_size: 69997
      - endpoint:
          address:
            socket_address:
              address: 127.0.0.1
              port_value: 8000
```

Clusters are delineated parallelly with listeners through the 'clusters' field. Within the route configuration and during statistics emission, the specified cluster name is employed. This name must maintain uniqueness among all clusters. Within the 'load_assignment' field, we can stipulate the roster of endpoints for load balancing in addition to the configuration of the load balancing policy..

Maglev is Google's network load balancer. It is a large distributed software system that runs on commodity Linux servers [27].

function POPULATE
Initialize a set S
foreach $i < N$ **do**
  next[i] $\leftarrow 0$
end
foreach $j < M$ **do**
  entry[j] $\leftarrow -1$
end
$n \leftarrow 0$
while $n < M$ **do**
  foreach $i < N$ **do**
    if $n \geq M$ **then**
      return
    end
    $c \leftarrow$ permutation[i][next[i]]
    while S *contains* c **do**
      next[i] $\leftarrow$ next[i] + 1
      $c \leftarrow$ permutation[i][next[i]]
    end
    entry[c] $\leftarrow i$
    S insert c
    next[i] $\leftarrow$ next[i] + 1
    $n \leftarrow n + 1$
  end
end
end

Algorithm 1: Optimized Populate for Maglev Hashing

5. Comparing the Performance and Resource Consumption of Linkerd and Istio

5.1 Methodology and Configuration for Experiments

In our experiments, we tested Linkerd and Istio using the Kinvolk benchmark suite. We used the latest stable versions of both projects: Linkerd (with its default setup) and Istio (configured minimally). The benchmark tests were done on a Kubernetes cluster using the Lokomotive Kubernetes distribution.

To ensure fair testing, we made sure the latency remained consistent throughout the tests. Cloud platforms can have varying performance, especially concerning networks, so maintaining consistency is important for accurate comparisons.

Our setup consists of the cluster havin 6 worker nodes, each with s3.xlarge.x86 configuration (Intel Xeon 4214 with 24 cores @ 2.2GHz and 192GB RAM) for running the benchmark app. Additionally, there was a load generator node and a K8s master node with the same configuration.

We tested both service meshes under three different loads: 20 requests per second (RPS), 200 RPS, and 2,000 RPS. For each load, we ran multiple tests, each lasting 10 minutes, for Linkerd, Istio, and a case without any service mesh. We reinstalled all benchmark and mesh resources before each run. For the 20 and 200 RPS tests, we did 8 runs and excluded the 3 runs with the highest baseline latency. For the 2,000 RPS test, due to time, we did 7 runs and manually removed the run with the highest latency for both Istio and Linkerd. (We have all the detailed data if needed.)

It is imperative to understand that the service mesh is tested by the Kinvolk framework in a distinct manner, and we maintained its original configuration without any alterations. Furthermore, the results we present incorporate impacts from both the service mesh and the test configuration. Consequently, while these metrics are not unequivocal, they are significant for contrasting different options within the same environment.

5.2 Result, Discussion and Analysis

The graphs below display the outcomes of our conducted experiments. Each data point within these graphs represents the average value obtained from five separate runs. The error bars accompanying each point indicate the range of one standard deviation from the mean. The bars themselves are indicative of Baseline, Linkerd, and Istio. In a performance comparison between Linkerd and Istio, two prominent service mesh tools, Linkerd demonstrated a clear edge in several key metrics. Remarkably, Linkerd consumed only 1/6th of the memory that Istio did under the same conditions. Furthermore, when evaluating CPU usage, Linkerd utilized just 55% of the resources Istio required. In terms of responsiveness, Linkerd proved to be more efficient, adding only 1/3rd of the median latency that Istio introduced. These findings suggest that Linkerd, in these scenarios, was more resource-efficient and provided faster response times compared to Istio.

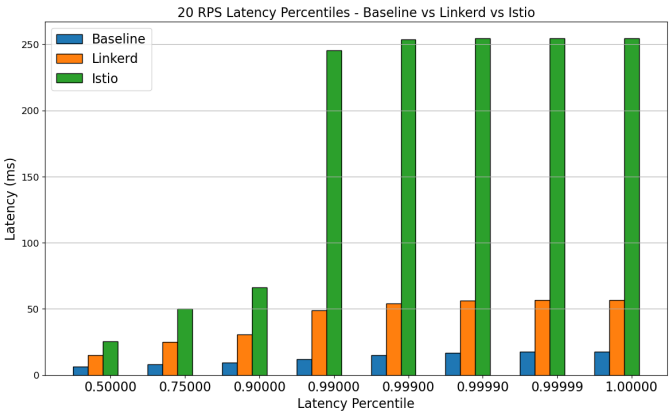

Figure 7: Latency at 20 RPS

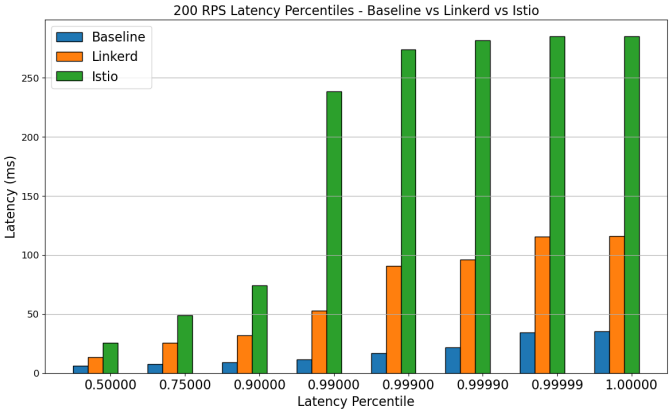

Figure 8: Latency at 200 RPS

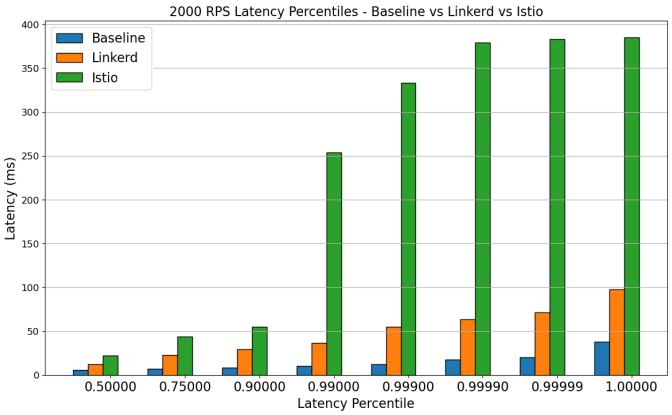

Figure 9: Latency at 2000 RPS

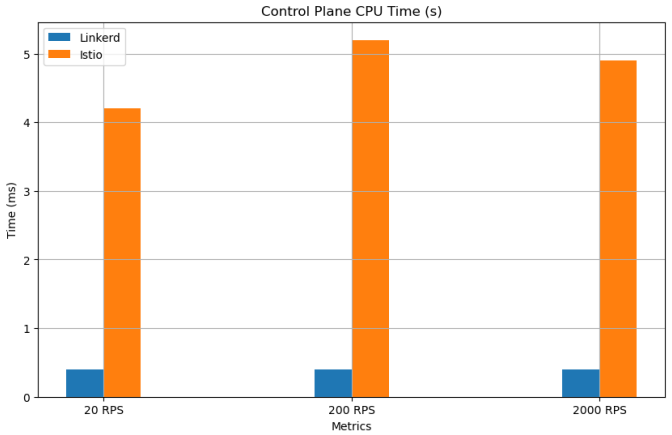

Figure 10: Control Plane CPU Time (s)

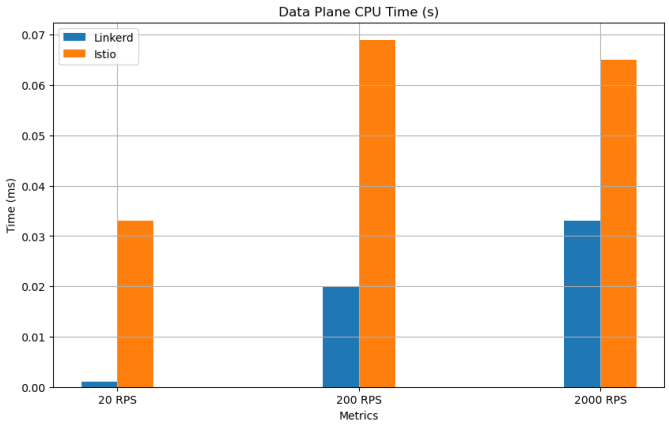

Figure 11: Data Plane CPU Time (s)

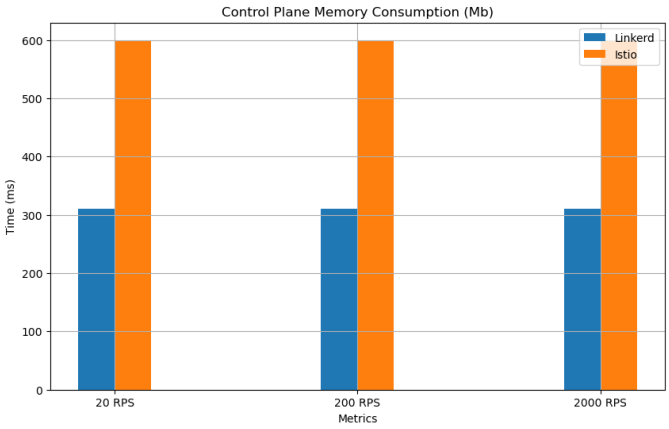

Figure 12: Control Plane Memory Consumption (Mb)

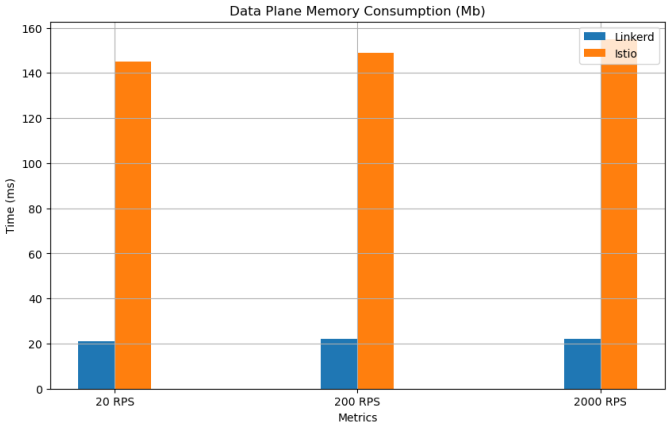

Figure 13: Data Plane Memory Consumption (Mb)

6. Conclusion, Recommendation and Future Research Direction

In our comparison of Linkerd and Istio, facilitated through the Kinvolk benchmark on a Kubernetes cluster, Linkerd showcased better performance metrics in terms of memory consumption, CPU usage, and latency. It's pertinent to mention that Istio leverages the Envoy proxy, which we have adapted to offer encryption-as-a-service. Based on these modifications, it's anticipated that a hybrid service mesh combining the strengths of both will result in CPU usage that is even lower than both standalone Linkerd and Istio. Hence, organizations aiming for resource efficiency and faster response times should consider the potential of this hybrid mesh. Nevertheless, considering the specificities of our test environment and configurations, we recommend organizations to undertake pilot tests aligned with their unique requirements before a full-fledged implementation.The prime focus of upcoming studies should be on refining the hybrid service mesh to achieve even lower latency while maintaining or enhancing other performance metrics.

References

[1] Redis. Redis documentation. `https://redis.io/docs/about/`, Accessed 2023.

[2] Cloud Native Computing Foundation. Kubernetes documentation. `https://kubernetes.io/docs`, 2022. Visited on 04/06/2022.

[3] Google. What is kubernetes. `https://cloud.google.com/learn/what-is-kubernetes`, 2022.

[4] Shubham Rasal. Why you should use kubernetes? `https://faun.pub/why-you-should-use-kubernetes`, Jan 2021.

[5] Jaehyun Nam et al. Bastion: A security enforcement network stack for container networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 81–95. USENIX Association, July 2021.

[6] Sam Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2021.

[7] Farzad Sabahi. Cloud computing security threats and responses. In *2021 IEEE 3rd International Conference on Communication Software and Networks*, pages 245–249. IEEE, 2021.

[8] Sam Newman. *Kubernetes: up and running: dive into the future of infrastructure*. O'Reilly Media, Inc., 2020.

[9] Tetiana Yarygina and Anya Helene Bagge. Overcoming security challenges in microservice architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, 2018.

[10] Nadin Habbal. Enhancing availability of microservice architecture: A case study on kubernetes security configurations. 2020.

[11] Jonathan Baier and Jesse White. *Getting Started with Kubernetes: Extend your containerization strategy by orchestrating and managing large-scale container deployments*. Packt Publishing Ltd, 2018.

[12] Xing Gao and et al. Containerleaks: Emerging security threats of information leakages in container clouds. In *2019 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019.

[13] Victor Marmol, Rohit Jnagal, and Tim Hockin. Networking in containers and container clusters. In *Proceedings of netdev 0.1, February*, 2015.

[14] StackRox. Kubernetes-native Security, 2020.

[15] Brendan Burns. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*. O'Reilly Media, 2018.

[16] Envoy Proxy Authors. Envoy proxy documentation. https://www.envoyproxy.io/docs, 2023. Available online: https://www.envoyproxy.io/docs.

[17] Wubin Li et al. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019.

[18] Sarada Prasad Gochhayat et al. Yugala: Blockchain based encrypted cloud storage for iot data. In *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2019.

[19] Anton Vedeshin et al. A secure data infrastructure for personal manufacturing based on a novel key-less, byte-less encryption method. *IEEE Access*, 8:40039–40056, 2019.

[20] Luigi Coppolino et al. Vise: Combining intel sgx and homomorphic encryption for cloud industrial control systems. *IEEE Transactions on Computers*, 2020.

[21] O. Sheikh, S. Dikaleh, D. Mistry, D. Pape, and C. Felix. Modernize digital applications with microservices management using the istio service mesh. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, CASCON '18, pages 359–360. IBM Corp., 2018.

[22] K. Indrasiri and P. Siriwardena. *Microservices Governance*, pages 151–166. Springer, 2018.

[23] A. Edmonds, C. Woods, A.J. Ferrer, J.F. Ribera, and T.M. Bohnert. Blip: Jit and footloose on the edge. *CoRR*, abs/1806.00039, 2018.

[24] H.-L. Truong, L. Gao, and M. Hammerer. Service architectures and dynamic solutions for interoperability of iot, network functions and cloud resources. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, page 2. ACM, 2018.

[25] N.C. Mendonça, D. Garlan, B. Schmerl, and J. Camara. Generality vs. reusability in architecture-based self-adaptation: The case for self-adaptive microservices. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, ECSA '18, pages 18:1–18:6. ACM, 2018.

[26] Listener configuration (proto). https://www.envoyproxy.io/docs/envoy/latest/api-v3/config/listener/v3/listener.proto, November 2023. envoy 1.29.0-dev-246dcd documentation.

[27] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 523–535, USA, 2016. USENIX Association.