ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA MAGISTRALE

in

Infrastructures for cloud computing and big data M

A performance analysis of mesh models for cloud-based workflows

Tesi di Laurea di:

Francesco Maria Cultrera

Relatore:

Chiar.mo Prof. Ing. Antonio Corradi

Correlatore:

Dott. Ing. Andrea Sabbioni

Anno Accademico 2021-2022

Sessione I

ABSTRACT

Nell'ambito della loro trasformazione digitale, molte organizzazioni stanno adottando nuove tecnologie per supportare lo sviluppo, l'implementazione e la gestione delle proprie architetture basate su microservizi negli ambienti cloud e tra i fornitori di cloud. In questo scenario, le service ed event mesh stanno emergendo come livelli infrastrutturali dinamici e configurabili che facilitano interazioni complesse e la gestione di applicazioni basate su microservizi e servizi cloud. L'obiettivo di questo lavoro è quello di analizzare soluzioni mesh open-source (istio, Linkerd, Apache EventMesh) dal punto di vista delle prestazioni, quando usate per gestire la comunicazione tra applicazioni a workflow basate su microservizi all'interno dell'ambiente cloud. A questo scopo è stato realizzato un sistema per eseguire il dislocamento di ognuno dei componenti all'interno di un cluster singolo e in un ambiente multi-cluster. La raccolta delle metriche e la loro sintesi è stata realizzata con un sistema personalizzato, compatibile con il formato dei dati di Prometheus. I test ci hanno permesso di valutare le prestazioni di ogni componente insieme alla sua efficacia. In generale, mentre si è potuta accertare la maturità delle implementazioni di service mesh testate, la soluzione di event mesh da noi usata è apparsa come una tecnologia ancora non matura, a causa di numerosi problemi di funzionamento.

As part of their digital transformation, many organizations are adopting new technologies to support the development, deployment, and management of their microservice-based architectures across cloud environments and across cloud providers. In this scenario, service and event meshes are emerging as dynamic and configurable infrastructure layers that facilitate complex interactions and the management of microservice-based applications and cloud services. The aim of this work is to analyze open-source mesh solutions (istio, Linkerd, Apache EventMesh) from the performance point of view, when used to manage the communication of microservice-based workflow applications in the cloud environment. For this purpose, a system was created to perform the deployment of each component within a single

cluster and in a multi-cluster scenario. The collection of metrics and their synthesis was carried out with a customized system, compatible with the Prometheus data format. The tests allowed us to evaluate the performance of each component along with its effectiveness. In general, while it was possible to ascertain the maturity of the tested service mesh implementations, the event mesh solution we used appeared to be a not yet mature technology, due to numerous operational problems.

SUMMARY

INTRODUCTION

In recent years, many organizations realized that keeping private infrastructures is a major expense and they started to outsource their infrastructure, leaving other players to take charge of the hardware maintainance. With the advent of cloud, the evolution of distributed architectures took a step further with companies providing elastic, pay-per-use services accessible from remote locations, but with the appearance of being internal to the enterprises. The evolution of architectures came together with the one of software applications that were designed first as big monoliths, running either as a single process or as a small number of processes spread across a handful of servers. These monoliths, characterized by slow release cycles, were updated relatively infrequently: at the end of every release cycle, developers packaged up the whole system and handed it over to the operations team, who then deployed and monitored it. In case of hardware failures, the operations team manually migrated it to the remaining healthy servers. Today, these big monolithic legacy applications are being broken down into smaller, independently running components, termed microservices. Since microservices are decoupled from each other, they can be developed, deployed, updated, and scaled individually, making possible to change components quickly and as often as necessary to keep up with today rapidly changing business requirements.

In this scenario, where many technologies arise to support the development of microservices and more specifically, the advent of containers has led to the development of technologies able to manage the entire lifecycle of business applications, in particular, Docker and Kubernetes: the first provided a way to package an application, together with the libraries it needs, by defining a boundary with the machine on which it runs; Kubernetes made it possible to create services with automation to help with autoscaling and management.

However, there are still many challenges involved in developing, deploying, and managing microservices-based applications and even though an orchestrator like Kubernetes does the heavy lifting, the growth in complexity of business applications made service and event meshes emerge. At the same time, with the explosion of big

data, many enterprises are investing in their next generation data infrastructures, based on a paradigm that draws from modern distributed architectures called data mesh.

Service mesh is a term used to describe a decentralized application networking infrastructure that allows applications to be secure, resilient, observable, and controllable. It defines an architecture made up of a data plane that uses application-layer proxies to manage networking traffic on behalf of an application and a control plane to manage proxies. This architecture allows to build important application-networking capabilities outside of the application without relying on a particular programming language or framework.

Event mesh is an architectural layer that dynamically routes events from one microservice to another irrespective of deployment location. The dissemination of massive amounts of data across a highly distributed infrastructure challenges enterprises on how to move this data, in an efficient, scalable, and economical way, across infrastructure that is not just geographically dispersed, but also exists in separate and heterogeneous clusters. The event mesh solves this problem by using a set of brokers in charge of distributing events across any environment. In fact, this technology does not necessarily require Kubernetes, but it can work seamlessly with it, enabling event-driven microservices to scale.

Data mesh, finally, is a new approach to thinking about data based on a distributed architecture for data management: the idea is to make data more accessible and available to business users by directly connecting data owners, data producers, and data consumers. Data mesh aims to improve business outcomes of data-centric solutions as well as drive adoption of modern data architectures. This approach to data allows companies to improve decision-making, help detect fraud or alert the business to changes in supply chain conditions. To create high-value data products, companies must address culture and mindset shifts and commit to a more cross-functional approach to business domain modeling.

Given, then, the importance and the recent rise of meshes, with this work we want to carry out a theoretical study of meshes, to then proceed to experiments that involve

service and event mesh technologies, selected for being open-source (isio, Linkerd, and Apache EventMesh), in order to test their performance when used to manage the communication of microservice-based workflow applications in the cloud environment, considering both the case of single cluster and multi-cluster.

Below is an overview of the thesis structure.

Chapter 1 discusses about concepts regarding cloud computing, which is the basis of our work, and some of the main topics covered in this dissertation. From Chapter 2 the meshes are discussed with first dealing with the data mesh, and then continuing with service and event mesh. In Chapter 3 different implementations of mesh technologies are analyzed, explaining the architecture and main characteristics of each. In Chapter 4 an overview of the project carried out for the thesis work is presented. Chapter 5 illustrates the implementation of each element of our design used to carry out the tests. Finally, Chapter 6 illustrates the experimental results obtained.

1 CLOUD COMPUTING

During the last few decades, the concept of outsourcing resources and services and the notion of utility computing merged in preparing a unique technological environment termed *cloud computing* that impacts organizations of any size, creating new opportunities by inspiring businesses not only to fulfill existing goals, but also to set new objectives based on the extent to which cloud-driven innovation can further help optimize business operations [1]. The wide adoption of the cloud has led to the development of *new technologies* and *paradigms*, such as microservices, orchestrator, and meshes, to cope with a growing system complexity. As mentioned in the introduction, the analysis of these technologies is the specific object of this work. Since we will focus on their application on cloud, we want to start from the analysis of the fundamental concepts of cloud computing.

1.1 ADVENT OF CLOUD

From the early days of computing, there has been the need, in many situations, for a great computational power. *Distributed systems* allowed developers to get results from computational demanding jobs and many companies adopted their own hardware, sometimes dividing it across different departments, to enhance business processes. Then the growth of the Web expanded many businesses, requiring more demand in hardware and technological competences [2], [3].

With time passing, the cost of keeping the infrastructure private became a major expense and new solutions were adopted. Third-party entities started providing ways to outsource the infrastructure, taking charge of the hardware maintenance. This organization also allowed companies to concentrate server equipment in a single physical location, a *Data Center*.

From this situation, the industry gradually moved to a more organized model which provides elastic and pay-per-use services from a remote location, but with the appearance of being internal to the company. This new approach has been called *cloud computing*.

It is possible to identify a set of characteristics common to the majority of cloud environments that enable the remote provisioning of scalable and measured IT resources in an effective manner [4]:

- ***On-demand resource usage***: a cloud consumer can use cloud resources on-demand usually with a pay-per-use subscription model. Generally, cloud consumers do not know the exact location of the provided resources, but they may be able to specify the region or zone like country, state, or even Data Center.
- ***Ubiquitous access***: cloud platforms are widely available, sometimes with world-wide coverage, allowing customers to access them from remote through a web interface.
- ***Multitenancy***: a cloud provider can serve different consumers (tenants) whereby each is isolated from the other.
- ***Elasticity***: cloud services can transparently scale by provisioning and releasing resources, as required in response to changing runtime conditions. In many cases, the demand for resources changes throughout the year, month or even the day. Elasticity allows customers to change the allocation of resources dynamically, adding facilities to deal with service peak times and removing them when they are no more needed, paying just for the facilities they use.
- ***Measured usage***: this characteristic represents the ability of a cloud platform to keep track of the usage of the IT resources. This allows the cloud provider to charge consumers depending on how many resources were used for a specific timeframe. In that sense, measured usage is closely related to the on-demand characteristic. From the consumer point of view, measured usage encompasses monitoring of the cloud resources allowing for comprehensive usage reports.
- ***Resiliency***: cloud services provide robust systems through the distribution of reduntant copies of the resources across physical locations. In case of one resource becoming deficient, another redundant copy can take it over. Thanks

to the resiliency of cloud-based IT resources, cloud consumers can increase both reliability and availability of their applications.

- ***Flexibility***: to accommodate an incremental business growth a startup can begin by leasing minimal computing, storage, and communication facilities. Then, it can adapt the requirements by adding more resources.

1.1.1 VIRTUALIZATION

The flexibility of cloud in the allocation of resources has been made possible thanks to *virtualization*. A cloud provider does not need to allocate groups of physical servers, but it can just create a set of software-defined virtualized servers [5].

This allowed for a rapid creation and removal of new virtualized istances at any time, without changing or rebooting physical machines. Furthemore, many virtualized resources can be *consolidated* in a few physical servers, running concurrently, each one completely isolated from others not only on the computational side, but also on the data one. For that reason, the cloud provider can choose where to create new virtual machines to get the best benefit in terms of resource usage and business cost. This guarantees also to avoid placing too many virtualized servers on the same physical machine balancing the load across all physical servers in the Data Center.

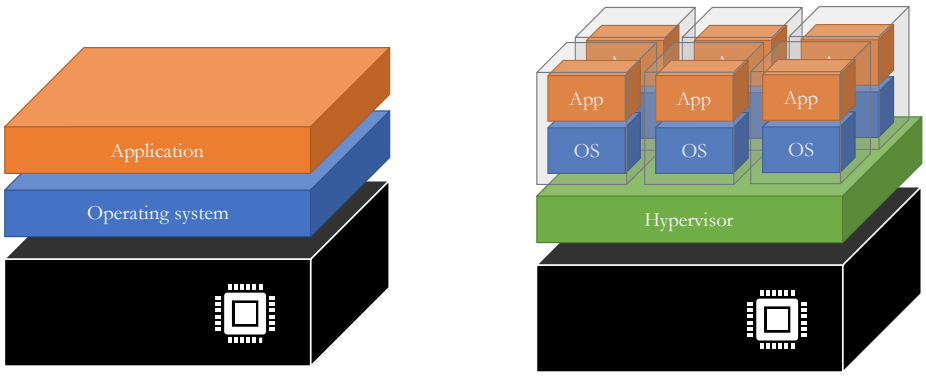


Figure 1-1. Differences between a traditional architecture (left) and a virtual architecture (right).

From a customer point of view, a virtualized server appears like a physical one, running applications that communicate through the internet. Cloud providers offer

options to easly deploy applications in a cloud environment, including the possibility to handle increasing flows of requests arriving from the internet by creating more application copies quickly. But they also provide the possibility to create test environments in which developers can create new software before staging it in production systems.

Virtualization gives also more security and replication defines generally more robust systems that, by tolerating faults and surviving to crashes, can always give an answer to the customer.

1.2 MODELS

Before the cloud computing emerged, private clusters and grid computing made use of many parallel machines to solve high demanding jobs, while *utility computing* and *Software as a Service* (SaaS) provided services managed and delivered from remote by one or more providers through a *pay-per-use subscription model* [6].

The concept of cloud computing has gradually evolved from these models and the way resources are owned and organized defines two different kinds of *models* in which cloud offerings can be divided.

1.2.1 CLOUD DEPLOYMENT MODELS

A cloud *deployment model* represents a specific type of environment, primarly distinguished by ownership, size, and access [1], [7].

1.2.1.1 PUBLIC CLOUD

From the early 2000s, when Amazon started its own cloud service, many different companies arose in the cloud industry. Those companies have been offering a *public cloud* accessible through a pay-per-use or a subscription model.

The cloud provider oversees managing the cloud infrastructure and all the IT resources and it makes the service accessible from the internet by any user who paid for it. Many providers offer a worldwide presence of their services.

1.2.1.2 COMMUNITY CLOUD

Similar to the public, a *community cloud* limits its access to a specific community of consumers. It may be owned by the community itself or by a third-party vendor that provisions a public cloud with limited access. Members of the community generally share similar security, privacy, performance and compliance requirements [8].

1.2.1.3 PRIVATE CLOUD

A company can also decide not to use a public cloud solution, but to use that technology to build a *private cloud* within the domain of an intranet owned by the organization. The company must maintain the infrastructure and limit the access to owning clients and their partners. By giving local users a flexible and agile private infrastructure to run service workloads within their administrative domains, a private cloud is supposed to deliver more efficient and convenient cloud services, but it may impact the cloud standardization, while retaining greater customization and organizational control.

1.2.1.4 HYBRID, MULTI AND DISTRIBUTED CLOUD

Many companies are adopting a *hybrid solution* in which multiple cloud models are taken into consideration. It provides orchestration, management, and application portability across different platforms, resulting in a single, unified, and flexible environment. For example, private clouds can implement a local infrastructure with computing capacity from an external public cloud: the organization can keep production applications on-site while conducting all its testing in the cloud or enabling on-demand scaling as needed.

The rapid growth of the hybrid cloud led to new challenges on how to take full advantage of the public cloud capabilities without the need to actually deploy applications on public cloud, on how to introduce those capabilities on-premises and on the edge, or how to use all these technologies on other public cloud services.

Some of these challenges were addressed by the *multi-cloud*, an approach that enables cloud solutions to not belong to just one stack, such as the Azure public cloud with Azure Stack on-premises, but applications, data, and cloud services can be used from

different cloud vendors, such as Microsoft Azure and Google Cloud Platform [9]. The multi-cloud allows companies to reach great flexibility when building business services, preventing performance problems, limited options, or unnecessary costs resulting from the dependence from a single provider. Usually, this kind of solutions rely on open-source, cloud-native technologies because in many cases, they are supported by the majority of the public cloud providers.

There are different reasons why an organization would choose a multi-cloud solution [9], [10]:

- ***Redundancy***: it is possible to rely on multiple cloud providers to keep the business running in case of a failure or to take advantage of the right technology for the specific business requirements.
- ***Latency minimization***: applications highly sensitive to latency benefit from local cloud providers that are near to the organization. In some situations, business processes can be accelerated thanks to the combination of edge and public cloud computing.
- ***Enhanced scalability***: applications can take full advantage of the cloud power by quickly scaling on multiple cloud provider solutions.
- ***Legality and governance***: in some cases, there is the need to use different cloud providers to comply with local data regulations, which require certain types of data to reside in specific geographies.

In most of the cases, the management of a multi-cloud solution can be very difficult. For example, changing access roles or security constraints can require accessing each individual operations dashboard of the various cloud providers. The *distributed cloud* solution solves operational and management inconsistencies through a central control plane and it can also help by replicating the capabilities of a cloud service provider on infrastructures located outside of the public offering. In that way, it allows to extend the public cloud to on-premises, private cloud, or even edge environments. Many providers are offering that service and some major names are: IBM Cloud Satellite, AWS Outposts, Google Anthos, and Microsoft Azure Stack; each of these has its own

strength points, but all of them are sharing the same idea of extending the public cloud capabilities to customer environments while having a single control pane [11].

1.2.1.5 CLOUD DEPLOYMENT MODEL COMPARISON

Private clouds leverage existing IT infrastructure and personnel within the company, balancing workloads more efficiently within the same intranet. A private cloud can provide high customization and enforce data privacy and security policies more effectively.

On the other hand, public cloud solutions promote standardization, offer application flexibility, and preserve capital investments by avoiding expenses due to the mantainance of IT hardware, software, and personnel [12].

Hybrid clouds operate in the middle, offering the benefits of both public and private cloud and taking advantage of existing architecture in a Data Center. The hybrid approach allows applications and components to interoperate across boundaries, between cloud instances, and even between architectures driving cost savings and supporting fast-moving digital business transformation.

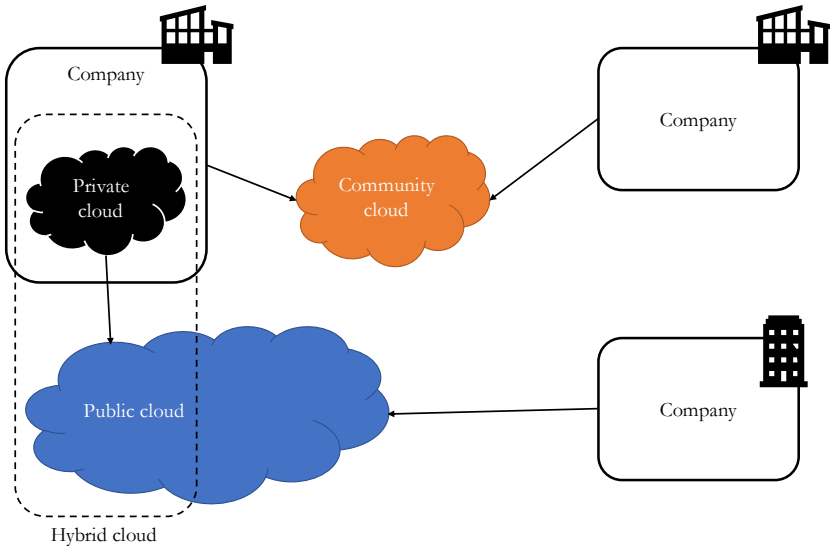


Figure 1-2. Cloud computing deployment models.

1.2.1.6 CLOUD DELIVERY MODELS

While some organizations started virtualizing their own computing machines to lower the IT costs, companies engaged in the public cloud industry started delivering different levels of virtual environments: *infrastructure*, *platform*, and *software as services* [12]. These cloud delivery models represent a specific, pre-packaged combination of resources that allow public cloud customers to upgrade their IT efficiency significantly. These models are available in a pay-as-you-go subscription and they are often offered based on Service Level Agreements (SLA)[1], addressed in terms of service availability, performance, data protection, and security.

Three common cloud delivery models have become widely established and formalized:

- ***Infrastructure as a Service (IaaS)***: the cloud provider offers the network, the storage, and the virtual machines on demand. In that case, customers are responsible for all the layers on top of the virtual machine together with the configuration of the operating system and its updates. The purpose of that solution is to give to the customers the highest control and responsibility level over the configuration and the usage of the virtualized resources.
- ***Platform as a Service (PaaS)***: this model represents a pre-defined environment usually comprised of already deployed and configured IT resources. The cloud provider oversees mantaining the operating system and provides the middleware solution like database, enterprise messaging, and runtime containers. In that model, the customers are focused on the application features and on the automation of their distribution.
- ***Software as a Service (SaaS)***: in this model the cloud provider manages the hardware and the software offering it as a final product or a generic utility. It is used to make a reusable cloud service widely available commercially to a range of cloud customers.

[1] An SLA is a documented agreement between a service provider and a customer that identifies both the services required and the expected level of service.

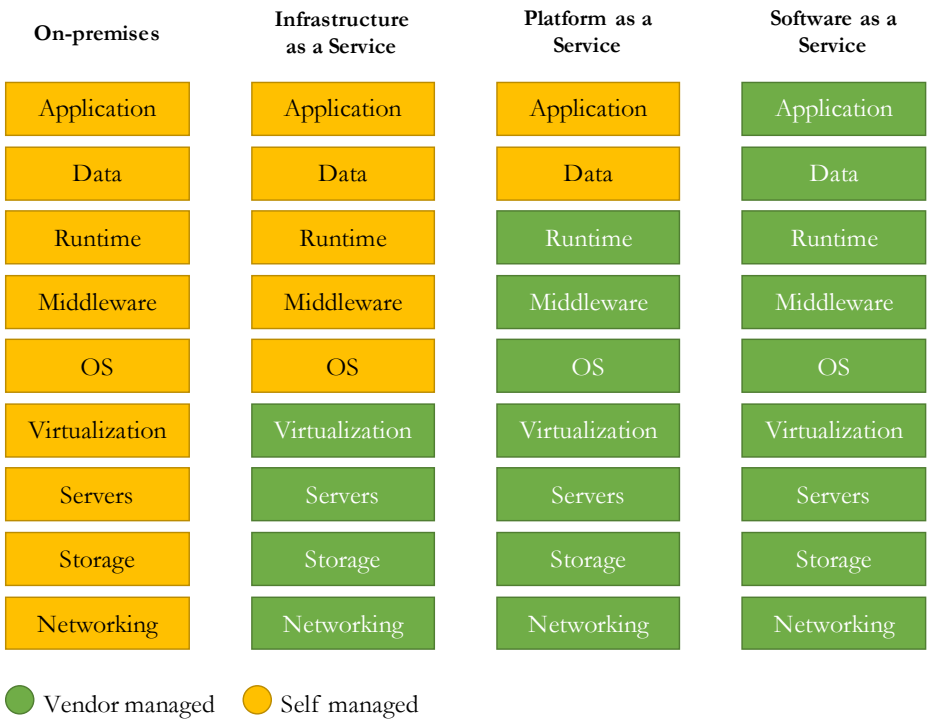


Figure 1-3. Cloud delivery models.

These models are the basis on top of which companies can deploy their own business processes. The adoption of an automatable, scalable, elastic and highly resilient cloud platform makes possible to innovate the way applications are provided by supporting a higher velocity product lifecycle that moves a software product through a development, test, stage and production environment without making huge investments in on-premises infrastructures [13].

1.3 DevOps

In the context previously mentioned, it is important to talk about *DevOps* that is a set of principles and practises that encourage the participation of both the development and operations teams in the entire software development lifecycle, software maintenance, and operations [14].

In the past, the development phase was separated from the one responsible for operating the software. Developers were focused on developing new features then passing the software to the operating staff, who ran and maintained it in production.

There was a little overlap between the two figures because the two departments had quite different goals [2], [15]. With the advent of cloud, the DevOps movement took its origin because it was more evident that things traditionally separated are now interconnected and interdependent [14]. The team responsible for the development must understand how their software relates to the rest of the system, and people responsible for the operations have to understand how the software works or fails.

The DevOps movement brings the two teams to collaborate, to share understanding and responsibility for systems reliability and software correctness, and to improve the scalability [16]. The companies that want to embrace this new approach can use several processes and tools to help automating the software delivery, improving speed and agility, reducing release times through *continuous integration and delivery* (CI/CD) pipelines, and *monitoring* the applications running in production.

The adoption of this new perspective in the organization requires a deep cultural transformation for businesses, but companies engaging it are more agile in the marketplace, while improving the quality of their products.

1.4 MICROSERVICES

The development of an application often involves best practices and architecture patterns to take full advantage of a specific platform. In the case of cloud computing, the application development moved in many cases from the classic monolithic approach in which all the features are integrated in a single package, to a new one in which the application is divided into different *microservices*, loosely coupled between them, simplifying the development and the testing phases and enhancing the modularity, fault tolerance and more generally, the quality [17], [18].

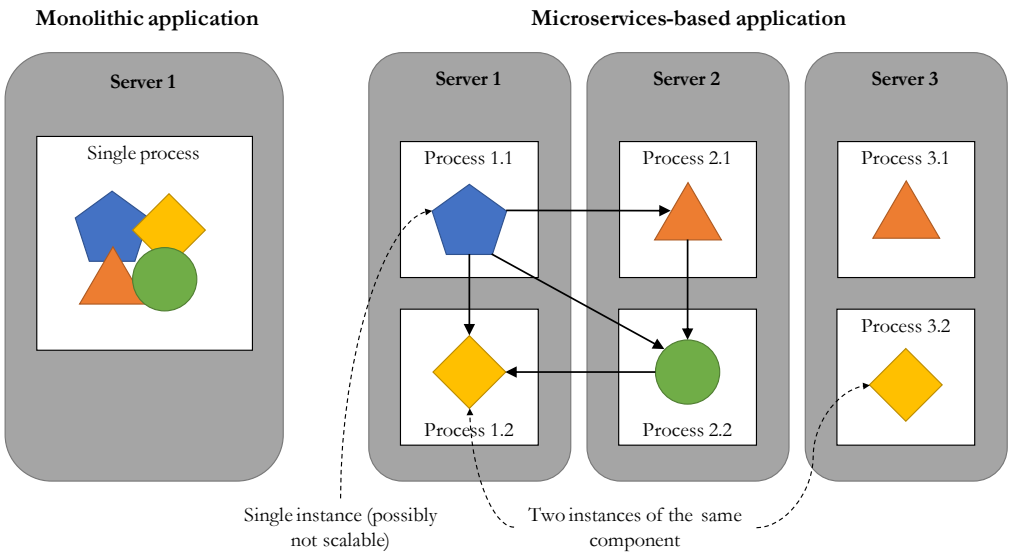


Figure 1-4. Differences between a typical monolithic and a microservice-based applications with different deployments.

Micro is not intended in the sense of the size of the application, but in its *purpose*. Every microservice implements a single feature of the whole business application with the greatest quality through well defined contracts [19]. This simple concept is the basis for the properties and the design of the microservice architecture.

1.4.1 PROPERTIES

Within the context of implementing a funcionaltity with quality, microservices exhibit some other properties and behaviors that characterize themselves from previous incarnations of service-oriented approaches [13], [20]:

- *Autonomous and isolated.*
- *Elastic, resilient and reactive.*
- *Message-oriented and programmable.*
- *Configurable.*
- *Automatable.*

More in detail, microservices are capable of existing *autonomously* with *loosely coupled dependencies* on other services. Each one can respond, react, or develop

independently of the whole and each of the microservices is designed, developed, tested, and released according to that.

They can be reused into different scenarios providing *scalability*, *fault tolerance*, and *high availability*. For that reason, they should be able to scale independently from the other microservices according to the required usage; the failure due to a single service should not extend to other services and it must guarantee a recovery in short times; at the end, they must guarantee the right performance for their usage scenario [21].

Microservices are also *message-oriented* and *programmable* because they must define clearly and completely the way through which they communicate to reach their common goal. For that reason, API interfaces are defined and they realize a set of endpoint visible on the network, and contracts over the data that define the structure of the exchanged messages.

Furthermore, microservices must be *reusable* and must be able to address the needs of each system. This is the reason why each one must provide a means by which it can be appropriately molded to the usage scenario.

Finally, complex projects could cause the proliferation of independent microservices. This situation requires a fully *automated control* over the software development cycle.

1.4.2 BENEFITS AND CHALLENGES

Microservices are providing great benefits to application development, but when systems become complex, they can be difficult to manage. Before adopting a microservices-based architecture is important to be able to compare benefits and challenges.

1.4.2.1 BENEFITS

By combining the concepts of information hiding and domain-driven design with the power of distributed systems, microservices can help deliver significant gains over other forms of distributed architectures [20], [22]:

- ***Clear module boundaries***: each service responds to the principle of "single responsibility" and it has simple and well-defined interfaces, thus reinforcing the modular structure and making the software more robust in time.
- ***Independent distribution***: it becomes possible to modify and update a single service without involving the entire application. This allows to distribute bug fixes and new features faster, while remaining competitive in the market.
- ***Independent development***: developers have a better awareness of responsibilities and control, and this leads to greater innovation, with a consequent improvement in release rates.
- ***Error isolation***: each malfunction remains confined within the area of a single service, without involving the entire application. However, resilience must be ensured by developers through adequate precautions.
- ***Granular scaling***: services can scale as needed independently. Furthermore, the higher density of components, which can be instantiated on each machine, allows to take advantage of the most of available resources.
- ***Heterogeneity of technologies***: it becomes possible not only to use the technology suitable for a specific problem, but also to experiment and introduce technically heterogeneous software easier. Furthermore, the managing libraries and dependencies becomes much easier.

1.4.2.2 CHALLENGES

Introducing microservices in an enterprise system can bring a host of complexity. Some of the most important points are [20], [23]:

- ***Complexity and skills***: the design of a microservices application, seen in its entirety, is more complex than other solutions. It is therefore necessary to have coordinated and particularly skilled development teams to be able to develop and administer it.
- ***Lack of governance***: the decentralized approach to application development, combined with the great freedom in the use of technologies, can lead to excessive fragmentation of languages, libraries and frameworks used, making the entire project difficult to manage.

- ***Development and testing***: in addition to the tests that are traditionally carried out on an application, it is necessary to specifically test microservices and their interactions. This can be complex especially when development is very fast.
- ***Network congestion and latency***: building a distributed project with an excessive number of services can lead to an increase in network communications. This problem, if not properly managed, causes a problematic increase in latencies.
- ***Data integrity***: given that persistence is managed directly by each microservice, it is necessary to maintain consistency between different components.

1.5 CONTAINERIZATION

With the industry adoption of microservices, some new technologies were created to deal with the growing complexity of applications. One of the most important is the *containerization* of microservices that is a technology that allows to run an application in isolation from the rest of the system, including all the necessary resources: code, runtime, system tools and libraries, and configuration files [24]. Assuming the need to create an application, each developer uses his own machine to deal with a portion of code. Often the developed application makes use of libraries and configuration files that may vary slightly from one development environment to another. Furthermore, in order to maintain a certain uniformity in the project, there is the need to emulate the company test and production environments, including the related software (e.g., database) [25] which is not always easy to install and configure, with a consequent waste of time and resources.

Containers are used in this context, making an application easily *portable* from one environment to another, without having to change the source code or correct errors in the configuration as the software moves from one phase of production to another [26].

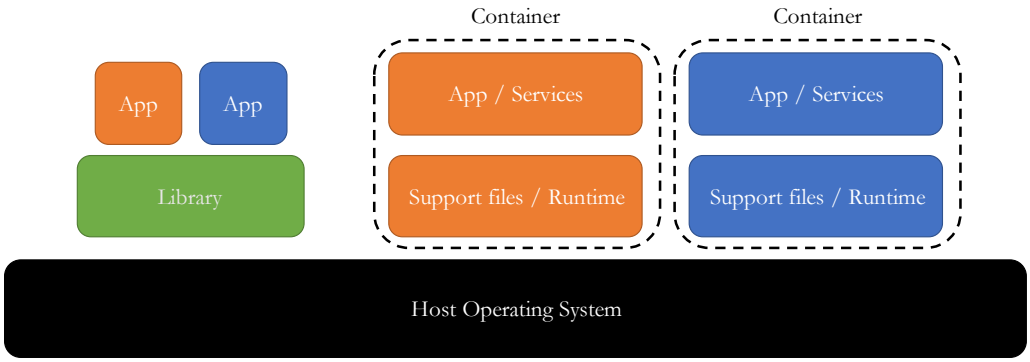


Figure 1-5. Differences between a native application (left) and a containerized one (right).

Containers fit into the DevOps development cycle, helping to reduce conflicts and clearly separating areas of responsibility: developers can focus on the source code of applications, while information technology (IT) people can focus on infrastructure [27]. It is a very flexible technology, applicable in many contexts in which *portability*, *configurability*, and *isolation* are necessary, without however depending on the infrastructure used. Containers can thus be deployed on-premises, in the cloud or, where necessary, in a hybrid solution.

1.5.1 DIFFERENCES BETWEEN VIRTUAL MACHINES AND CONTAINERS

Before containers were born, technologies were already available on the market that allowed the creation of virtual machines [3]. Today the containers are widely used [28]; however, the two techniques coexist, each with their own advantages.

Traditional virtualization allows to run a complete operating system inside a virtual machine. This allows to have multiple instances of systems, even different from each other, operating at the same time on the same physical machine. This is made possible by a hypervisor that provides optimal isolation between physical and virtual machines [29].

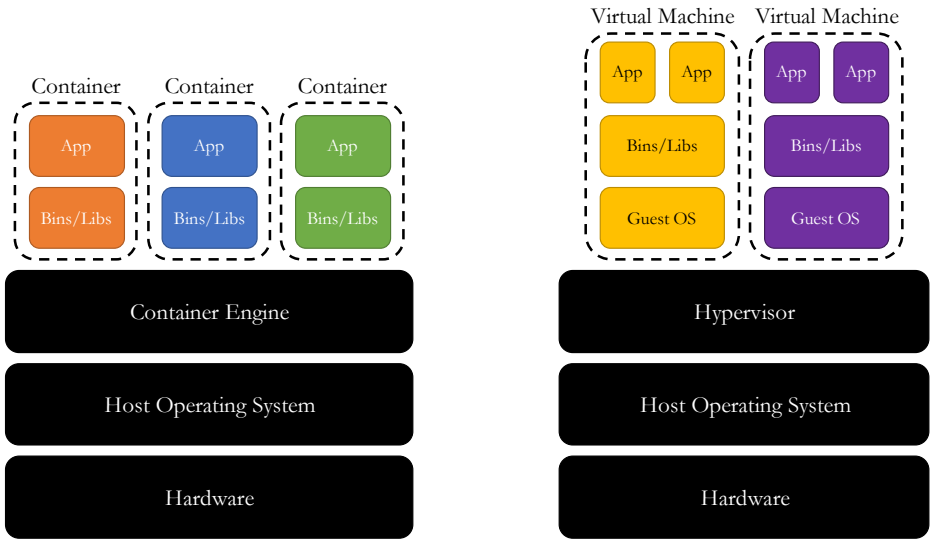


Figure 1-6. Comparison between containers (left) and virtual machines Type-2 (right).

Virtual machines, although technologically complete, present some problems. First, booting a full operating system usually takes a long time. Second, each virtual machine requires a fixed amount of memory to work and has a significant overhead in CPU usage. Furthermore, I/O communications are inefficient: in the case of a Type-2 virtualization, an application makes calls to the guest kernel which in turn calls what it believes to be the hardware; however, this call is intercepted by the hypervisor, passed to the host kernel and finally to the hardware. The answer will necessarily have to retrace the same route in the opposite direction. Although Type-1 virtualization technologies exist, which allow the hypervisor to rely directly on the hardware, it is always necessary to traverse the entire stack, resulting in performance degradation in terms of both overhead and latency. Furthermore, the resources are not allocated in a granular way as the inactive ones remain associated with a single virtual machine and cannot be reused. Finally, maintaining virtual machines is difficult: since the size of the software to be managed is considerable, it becomes more complex to keep the system updated, also considering the downtime necessary for the update. Otherwise, containers allow to reduce overhead to a minimum as a single kernel manages logically separate instances of applications (process containers) or operating systems (machine containers) [30]. The I/O communications go directly through the host kernel and the

performance is very similar to that of native applications. The isolation is weaker than virtual machines, but still present: containers can, in fact, share some resources. Additionally, container startup times are much faster as there is no need to boot the operating system. Finally, resource allocation is very flexible because it is managed by the host kernel.

1.6 ORCHESTRATION

With the increase in complexity of their systems, organizations needed to automate the configuration, coordination, and management of computer systems and software [31]. In this scenario, an *orchestrator* is a system that can deploy and manage applications, dinamically responding to changes, such as adapting to the current business demand [32].

In the cloud field, when running very complex microservices-based applications (hundreds or even thousands of entities) in a cluster environment, the *container orchestration* is a system that automates the deployment, management, scaling, and netwoking of containers. Orchestrators can be used in any environment like private or public clouds, making easier to manage complex business situations [33].

More in detail, one of the first problems this kind of software addresses is *resource scheduling*. In fact, it is possible that the services require to be performed on machines that meet particular requirements, but not all of them are adequate [34]. Moreover, very often, it is necessary to carry out *load balancing* which consists in making the most of the cluster resources, without having containers that work too much and others that are inactive. Applications must be able to *scale* as needed. It is therefore necessary to launch new containers in the case of intense traffic, in order to serve all requests; when they are no longer needed, they must be eliminated automatically. It is very important to be able to handle peak customer demands. Finally, it should be possible to easily *monitor* and *control* in a simple way all the containers.

Generally, container orchestration tools are configured through user-defined *guidelines, labels*, or *metadata*. Once the host is assigned, the orchestration tool

automates and manages the services throughout the lifecycle based on the rules defined.

In general, orchestrator tools provide the following services:

- *Containers configuration and scheduling*.
- *Containers provisioning and deployment*.
- *Resource allocation*.
- *Scaling of containers to balance requests*.
- *Service discovery*.
- *Containers monitoring*.
- *Traffic routing*.

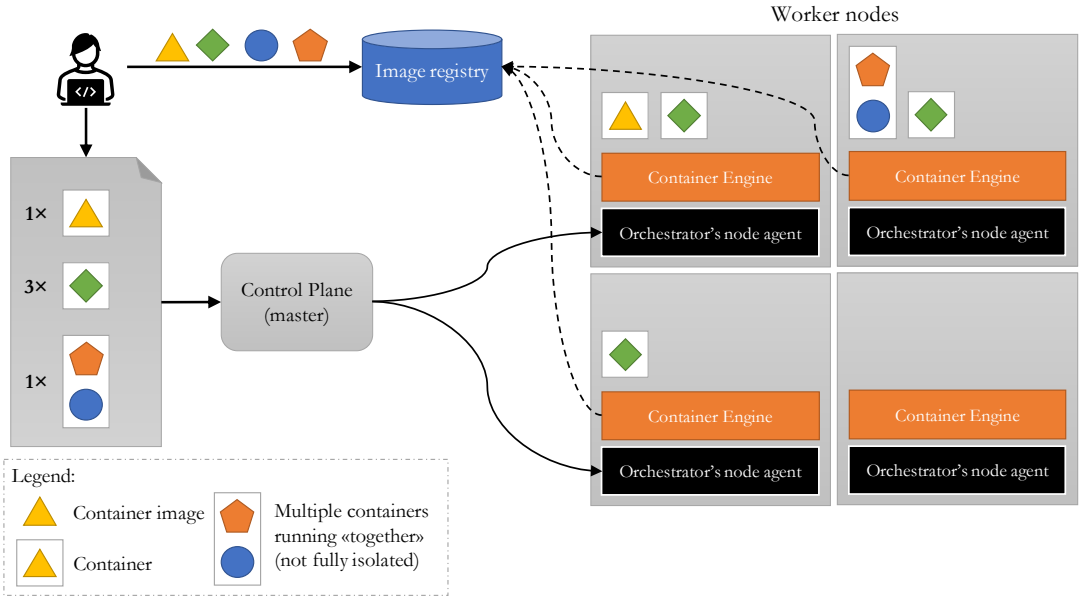


Figure 1-7. A basic overview of an orchestrator architecture and an application running on top of it.

Due to their importance in managing microservice-based applications, many different orchestrator tools were implemented. One of the most famous is Kubernetes [32].

1.6.1 KUBERNETES

Originally developed by Google, *Kubernetes* is an open-source orchestrator that allows to coordinate containerized applications across clusters of machines. It is a platform

designed to manage the entire lifecycle of containerized applications and services through methods that provide reliability, scalability and high availability [35], [36].

The core concepts on which Kubernetes has been based are:

- *Immutability*.
- *Declarative configuration*.
- *Online self-healing systems*.

Containerization tools like Docker[2] and orchestration tools like Kubernetes are based on the concept of infrastructure *immutability*. The current state of a system is not represented by a set of updates and changes applied incrementally, but by a single and unchangeable entity. In this way, any updates or changes are made through new container images that replace the previous ones.

For the *configuration* of an immutable system, it is not convenient to use the traditional imperative approach, which envisages exercising control over the system through the direct execution of a series of instructions, but it is encouraged instead the use of descriptions of the present objects called *declarative configuration objects*. These define the state of the system and have some advantages: being in written form, versioning and code revision can be applied; the effects of the declarative configuration can be understood before being executed, drastically reducing the number of possible errors in the system configuration; finally, it is possible to rollback easily by restarting the system previously stated.

The configuration mentioned above does not only describe the initial state, but it is also used to keep the current system state *consistent* with the provided configuration. This requires an orchestration system, such as Kubernetes, to monitor the state of the entire cluster so that it can react to failures or problems that could destabilize the system (*online self-healing*).

Orchestrators, such as Kubernetes, led to many benefits in terms of velocity, scaling, efficiency, and infrastructure abstraction, but they focus largely on scheduling,

[2] Docker is a popular platform used to create and manage containerized applications.

discovery, and health, primarly at a low infrastructure level, usually leaving microservices with unmet service-level needs. For this reason, new specialized technologies and paradigms were defined, sometimes relying on container orchestrators, to deal with service-to-service communications and big data growth.

2 MESH TECHNOLOGIES AND PARADIGMS

In this chapter we want to focus on the *mesh* that is a recent concept about distributed architectures that regulates complex topologies based on connections and constraints among all nodes and components of business applications. Recent solutions that started from this concept are technologies and paradigms that are modernizing the development and management of business applications and they are termed as *service*, *event*, and *data mesh*.

Service and *event mesh* are configurable infrastructure layers that facilitate complex interactions and management of microservices-based applications and cloud services. More specifically, the first is focused on synchronous interactions, while the second is dedicated to the distribution of events. On the other hand, a *data mesh*, from whose analysis we will start, is a paradigm that evolved from traditional data platforms, introducing organizational and process changes that companies need to manage data as a tangible capital asset of the business.

2.1 FROM TRADITIONAL DATA PLATFORMS TO DATA MESH

Capturing and analyzing the right information is crucial and companies are also using data as an asset both to improve customer interactions and to increase efficiency. Data can be, for example, the basis for personalization, dynamic pricing, market expansion, product innovation, or supply chain optimization [37], [38].

2.1.1 LIMITS OF TRADITIONAL DATA PLATFORM ARCHITECTURES

The goal of becoming a data-driven organization has many challenges and traditional architectural paradigms are no more efficient in many contexts. To cope with information complexity and business needs, there were three different generations of data platform architectures before data mesh was defined [39]:

- The ***first generation*** was composed mainly by proprietary enterprise *data warehouse*[3] and *business intelligence* (BI)[4] platforms mantained by a specialized team. Those systems were expensive and they left companies with large amounts of technical debt in terms of numerous of unmaintainable ETL (Extract, Transform, Load) jobs, tables, and reports.
- The ***second generation*** shifted to the concept of *Data Lake*, a complex enterprise data platform architecture chatacterized by being flat, centralized, monolithic, and domain agnostic. This means that it stores in a central position large and varied sets of unstructured, semi-structured, or structured data collected from different sources across the organization [40].
- The ***third generation*** is similar to the previous one with the addition of *streaming* data, cloud, machine learning, and other technologies.

Usually, a Data Lake sits in what is called an *analythical data plane* which is a temporal and aggregated view of the facts of the business over time. It depends on the *operational data*, organized in a different plane, that sits in databases behind business capabilities served with microservices, has a transactional nature, keeps the current state and serves the needs of the applications running the business. In this scenario, the two planes are different but yet integrated through Extract, Transform, Load (ETL) jobs producing many difficulties in connecting those two planes with an ever-growing complexity of the data pipelines.

[3] A data warehouse is a system designed for data analytics, which involves reading large amounts of data to understand relationships and trends across the data.
[4] Business Intelligence refers to a set of technologies that enable data preparation, data mining, data management, and data visualization. Business intelligence tools and processes allow end users to identify actionable information from raw data, facilitating data-driven decision-making within organizations across various industries.

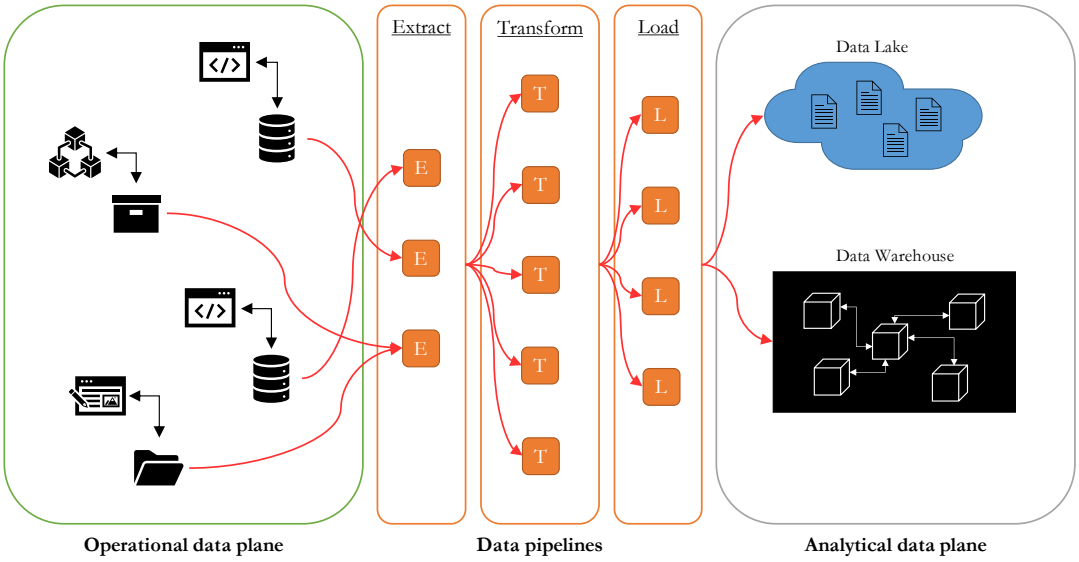


Figure 2-1. Schema of the division between the operational and the analytical data planes connected by complex ETL pipelines.

2.1.2 THE GREAT DIVIDE OF DATA

The separation between the analytical and the operational data planes on one side and the ETL pipeline organization on the other are not the only problem originated from the systems previously mentioned. In fact, it must be considered that they also have other limitations:

- ***Centralized and monolithic architecture***: the data platform contains all the information that logically belong to different domains. This organization may work for small organizations, but it is not suitable for enterprises with large numbers of data sources and diverse data consumers.
- ***Siloed and hyper-specialized ownership***: the organizational units who build and own the data platform consist of hyper-specialized data engineers separated from the teams providing source data and the consumer units retrieving the processed data. While the latter groups are domain-oriented, the former team must be domain-agnostic.

In order to address the previous limitations of information platforms, a change in thinking about data, its locality, and ownership is required to successfully implement a data mesh architecture.

2.1.3 DATA MESH

The *data mesh* is a new approach in sourcing, managing, and accessing data for analytical purposes at scale, involving both teams and technical architectures organization [41]. It breaks the traditional monolithic Data Lake paradigm into several independent subsystems or domains, each of those managed by a dedicated team. This allows to move from a centralized ownership of data, collected into monolithic data platforms, to a decentralized model of data products whose ownership is near the business domains where data is produced. In addition, a data mesh removes the gap between where the data originates and where it gets used in its analytical form, deleting all the pipelines between the operational and the analytical data planes. At the same time, it attempts to connect the two planes through an inverted model and topology based on domains and not technology stack, with a focus on the analythical plane [42]. In other words, it shifts from technology solutions that treated data as a bybroduct of pipelines, governed in a centralized operational model, to solutions in which data and the code that mantains it, are treated as an autonomous unit, inside a federated organizational model with computational policies embedded in the nodes on the mesh.

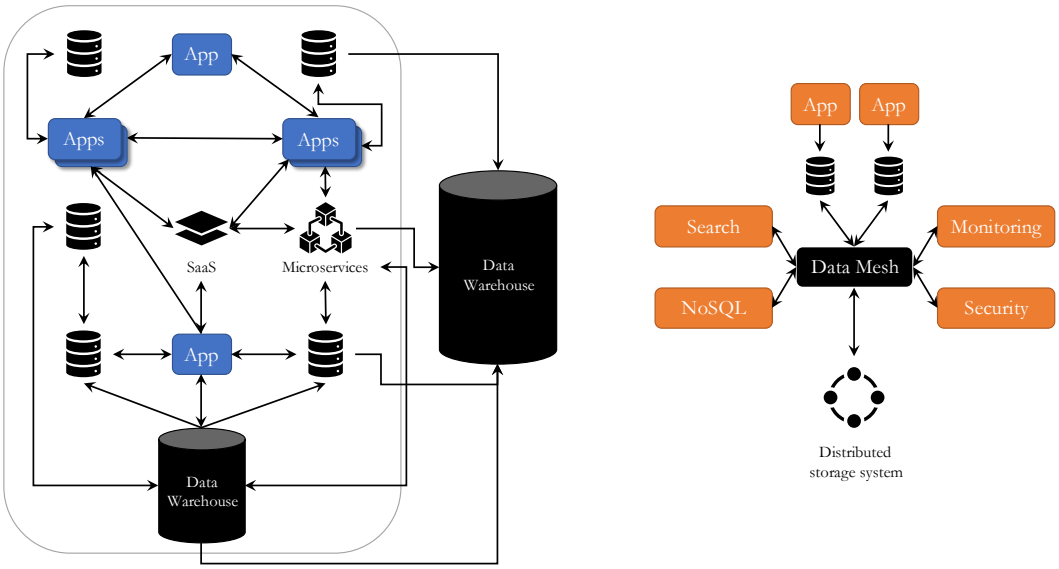


Figure 2-2. Comparison between a centralized data ownership on the left in which the responsibility for data becomes the domain of the data warehouse (DWH) team and a decentralized one on the right in which ownership of a data asset is given to the "local" team that is most familiar with it.

A Data mesh is based on four principles [43]:

- *Decentralized architecture and ownership organized into domains*.
- *Data as products*.
- *Self-serve data infrastructure as a platform*.
- *Federated management of computing resources*.

The data mesh has a *decentralized architecture* in which each *domain* is an independently deployable cluster of related microservices which communicate with users or other domains through modular interfaces. To promote the decomposition of the data ecosystem and the ownership of each component, there is the need to model an architecture that organizes the analytical data by domains. In this architecture, the domain interface to the rest of the organization not only includes the operational capabilities, but also access to the analytical data that the domain serves. The decomposition follows the separation between different business domains that organizations already provide and it makes possible to localize the impact of continuous change and evolution to the domain bounded context.

The data is treated *as a product* like any other, complete with a data product owner, consumer consultations, release cycles, and quality and service-level agreements. From an architectural point of view, the data mesh introduces the concept of data product as its architectural *quantum* which is the smallest unit of the architecture that can be indeplendently deployed with high functional cohesion, including all the structural elements for its own functioning. These elements are mainly three:

- ***Code***: for data pipelines responsible for consuming, transforming and serving upstream data; for APIs that provide access to data; to enforce traits such as access control policies, compliance, provenance, etc.
- ***Data and metadata***: depending on the nature of the domain data and its consumption models can be served as events, batch files, relational tables, graphs, etc., while maintaining the same semantic. Usually, data comes associated to metadata.
- ***Infrastructure***: it enables building, deploying, and running the data product code, as well as storage and access to big data and metadata.

A key point of data mesh is a *self-serve data infrastructure as a platform* that empowers consumers to independently search, discover, and consume data products. Data product owners are provided standardized tools for populating and publishing their data product. A self-serve data platform supports the provisioning of the underlying infrastructure required to run the components of a data product and the mesh of products. It should provide an interface that abstracts the provisioning complexities to support data product developers workflow.

The last principle on which a data mesh is based is *federated management*. It is embodied by a cross-organization team that provides global standards for the formats, modes, and requirements of publishing and using data products. This team must maintain the delicate balance between centralized standards for compatibility and decentralized autonomy for true domain ownership.

The four principles described above are collectively necessary and sufficient. Each of them complements the others, addressing challenges that may arise from each principle

took alone. For example, decentralized domain-oriented ownership of data can result in data siloing within domains, and this can be addressed by the data as a product principle that demands domains have an organizational responsibility to share their data with product-like qualities inside and outside of their domain [42].

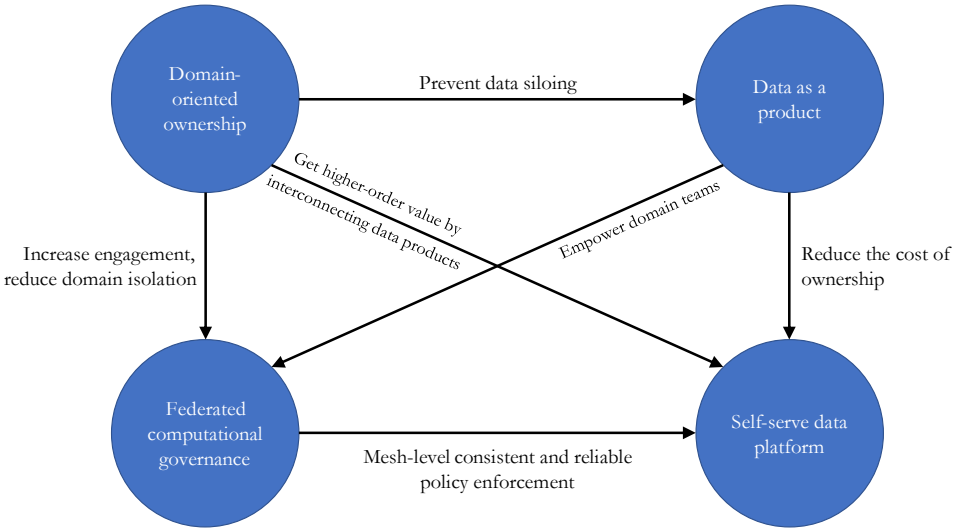


Figure 2-3. Representation of the interplay of the four data mesh principles. Each arrow starts from the principle that may present a challenge and it reaches the principle that addresses it.

2.1.4 CONCLUSION

Traditional data architectures are becoming difficult to manage and use with the increase of big data. Companies can rely on monolithic architectures, keeping all the data just in a single place, but to stay competitive they need to create new products, services, and solutions, or enhance existing ones, for their customers. This requires a human and technical organization that lets an enterprise to spend time looking for business opportunities, rather than for data in the Data Lake, analyzing the market and chasing new customers. On the other side, a data mesh can be an effective way of building enterprise data platforms as distributed architectures that make data accessible and available to business users by directly connecting data owners, producers, and consumers. Approaching data mesh can be a huge undertaking for companies, but it means building a solid foundation that supports the evolution of the data-driven business.

2.2 MICROSERVICES MANAGEMENT USING SERVICE MESH

Turning now to the analysis of the service mesh, in order to better define it, it is necessary to observe that with the advent of microservices, many of the problems for developing agile applications in a distributed environment were addressed by splitting the business logic of applicatives into different entities. Although this approach allowed organizations to embrace agile principles, enhancing their production velocity, efficiency, and by decoupling the different parts of their applications, achieving better quality of service, it became important to document and control how different parts of an application share data.

2.2.1 CHALLENGES WITH MICROSERVICES

Microservices communicate significantly over the network to achieve the business logic of the application they realize. This dependency adds many potential hazards that grow with the number of connections microservices business applications depends on. The fast development cycles lead to new challenges in managing an increasing deployment complexity [44], [45]:

- ***Observing interactions*** between services in a large and distributed environment with many loosely coupled components can be difficult.
- ***Traffic management*** at each service endpoint becomes more important to enable specialized routing such as for A/B testing[5] or canary deployments[6] without impacting clients within the system.
- ***Securing communications*** by encrypting the data flows between decoupled services with different binary processes and written in different languages is more complicated.
- ***Network unreliability*** can be experienced in a complex scenario where dealing with timeouts, high-latency routes, and communication malfunctions can lead to cascading failures that are difficult to correctly address.

[5] A/B testing refers to a randomized experimentation process wherein two or more versions of a variable (web page, page element, etc.) are shown to different segments of website visitors at the same time to determine which version leaves the maximum impact and drives business metrics.
[6] A canary deployment is a strategy that reduces the risk of releasing new software that could impact the workload by allowing to release the software gradually to a small subset of users.

Many companies in the web market tried to resolve the problems mentioned above directly in the services code, developing *specialized frameworks and libraries*. This approach was not ideal, because it required an agreement on the same solution approach between all the microservices involved in a communication to assure consistency. These frameworks that companies created were very language and platform specific and in some cases, made it difficult to bring in new application services written in programming languages that did not have support from these resilience frameworks. Whenever these frameworks were updated, the applications also needed to be updated to stay in lock step. Finally, adding extraneous logic to the application code to provide a solution to the problems mentioned above is extremely error prone.

2.2.2 SERVICE MESH

A *service mesh* is a different approach to address challenges in a complex business scenario. It is a programmable *infrastructural layer* that provides a uniform way to connect, secure, manage, and monitor microservices. It does not establish connectivity between microservices, but instead it has policies and controls that are applied on top of an existing network to govern interactions between microservices. Service meshes are language agnostic and can be used in existing environments usually without applications code changes [46], [47].

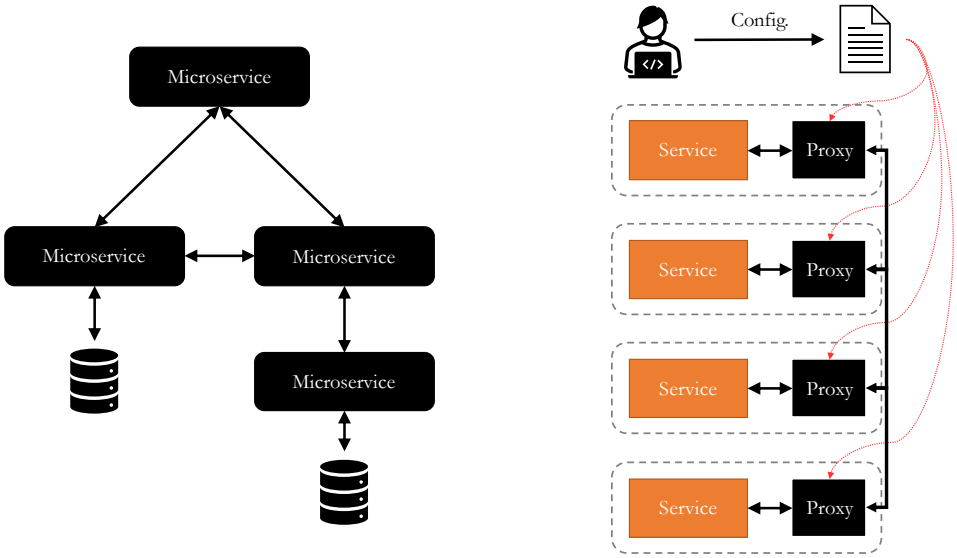


Figure 2-4. Comparison between a traditional microservices-based architecture (left) and a service mesh (right). Interactions between components are mediated by proxies specifically configured.

A service mesh enhances a microservices-based infrastructure by providing the following chatacteristics [48]:

- ***Observability***: service meshes are deployed in a transparent way, giving visibility into and control over the traffic without requiring changes to the application code.
- ***Traffic control***: service meshes provide granular, declarative control over network traffic. That enables advanced features that enhance overall resiliency such as circuit-breaking[7], latency-aware load balancing, eventually consistent service discovery, retries, timeouts, and deadlines.
- ***Security***: service meshes enforce security, policy, and compliance requirements. Most solutions provide a certificate authority (CA) to secure service-to-service communication.

The value service meshes provide is independent from the number of services running and many organizations are adopting this technology in their business environments.

[7] The circuit breaking is a pattern that provides stability while the system recovers from a failure and minimizes the impact on performance.

Many of these companies use service meshes to modernize their business keeping their existing applications (monoliths or microservices) in the actual environment. To serve that purpose some service mesh implementations work in diverse enviroments, from containers to virtual machines and bare-metal hosts. In any case, all the systems share a common architectural composition.

2.2.3 General architecture

Although there are a few variants, each service mesh architecture is composed of two structural components [49]:

- The *data plane* manages the real communication between services.
- The *control plane* manages the configuration and policies of the data plane communications.

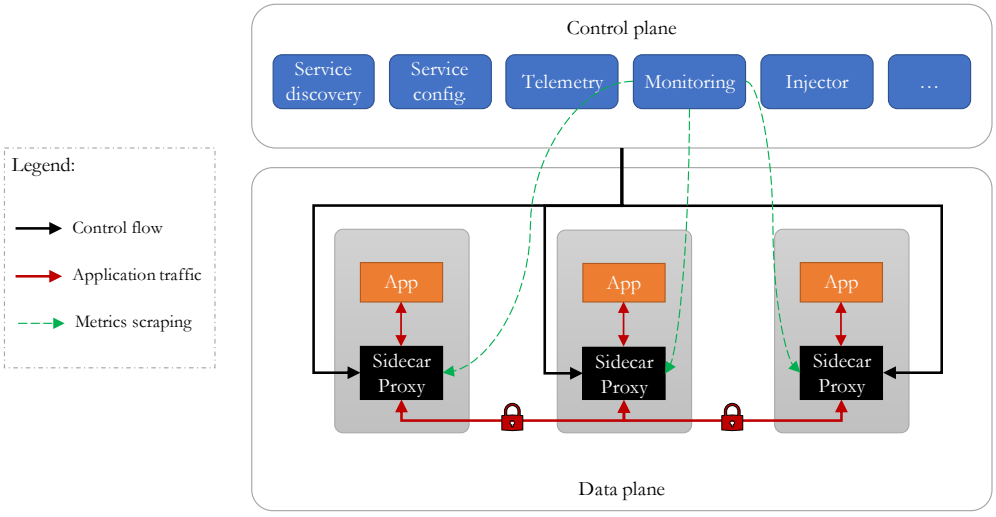


Figure 2-5. General architecture scheme of a service mesh.

2.2.3.1 Data plane

The data plane (sometimes called *proxying layer*) is composed by a set of *proxies* that are responsible to mediate and control the communication between services. When used with containers and orchestrators, the proxy is inserted in the container as a sidecar, acting transparently, so that applications, while communicating, are unaware

of the data plane existence. The proxy intercepts every packet in the request coming from or arriving to the application. By having communication routed between proxies, each of those serve as a key control point to perform complicated tasks. Each proxy is responsible for routing, health checking, load balancing, authentication, authorization, and generation of observable signals. The proxies used in many service mesh implementations can understand a variety of communication protocols, including application-level data.

The data plane is not only responsible for intra-service communication, but also for inbound (ingress) and outbound (egress) service mesh traffic. Wether the traffic is entering the mesh (ingress) or leaving it (egress), the application service traffic is directed first to the proxy for handling prior to sending, or not sending, along to the application [50]. Since all the communication complexity is handled by proxy sidecars, applications are freed from the network logic. In a real business scenario with high complexity and many components running, the management of proxies can be difficult. In that situation a control plane acts as a single point of visibility and control over the data plane.

2.2.3.2 CONTROL PLANE

In general, control planes provide policy and configuration for service meshes, turning isolated and stateless proxies in a complete service mesh. The role of a control plane is to operate *out-of-band*, without touching any network packet. It is responsible for injecting proxy sidecars before new applications start, informing proxies of the presence of services, updating the mesh topology, enforcing traffic and authorization policies, and gathering metrics. It must be noted that generally proxies cache the state of the mesh, but they are considered stateless.

2.2.4 ORCHESTRATORS VERSUS SERVICE MESHES

As already mentioned, an orchestrator allows microservices-based applications to be executed and distributed on a large scale. On the other hand, service meshes are demanded to service-to-service communication management.

Container orchestrators like Kubernetes have different mechanisms for dealing with network management. In a Kubernetes native cluster, a *kube-proxy* component is deployed on each node and it mediates the communication between the different pods. It also communicates with the Kubernetes API Server and gets information about the services in the cluster. Because each node in a Kubernetes cluster runs many pods, adding a sidecar on a per-pod basis can increase the response latency than a single kube-proxy, due to more hops when the sidecar intercepts the traffic. Each kube-proxy has global settings and for this reason, they cannot control each service at a granular level [50], [51]. Kube-proxy does not offer advanced features to control network traffic. It implements load balancing across multiple pod instances of a service, but it does not provide, for example, traffic division by percantage to different applications or applying canary and blue-green releases.

On the other hand, in a service mesh, sidecar proxies allow for a more fine-grained traffic management because they work on a per-pod basis.

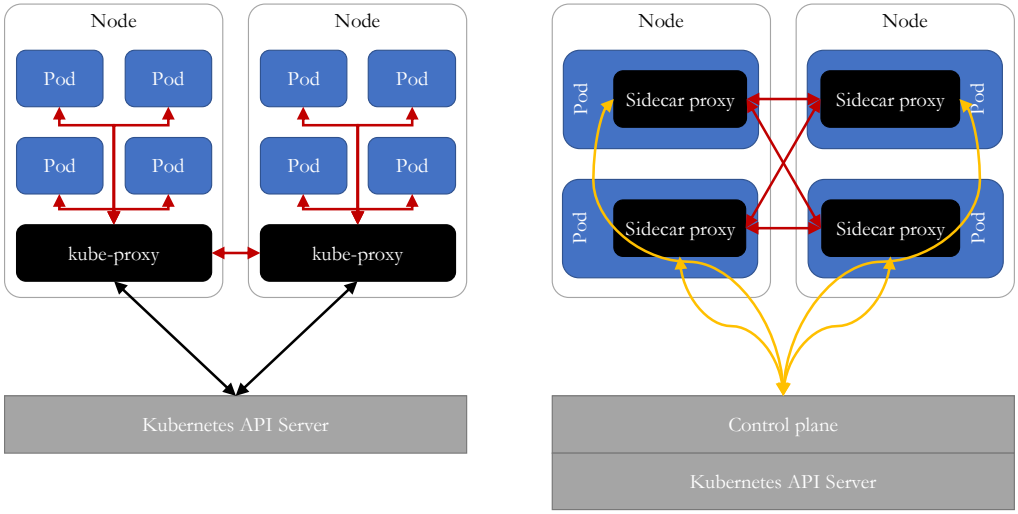


Figure 2-6. Kubernetes general architecture versus a service mesh deployment.

Many service mesh solutions run seamlessly with orchestrators like Kubernetes giving the advanced tools to build, deploy, and manage, microservices-based architectures. In particular, service meshes are able to monitor what services are communicating with

each other, if those communications are secure, and they can control the service-to-service communication in Kubernetes clusters so that applications can run securely and resiliently [52].

2.3 Entering the world of event-based architectures

The growth of many companies set new challenges for maintaining and scaling enterprise systems. In many cases, as data, connection count, and performance requirements increase, these problems do not involve only technical topics, but also teams organization. Particularly, the dissemination of massive amount of data across a highly distributed infrastructure becomes difficult to manage for a large enterprise.

Accessing data on another teams domain is always more difficult than accessing it locally, because this requires crossing both implementation and business communication boundaries. Often, this situation can create a strict technical dependency between teams, requiring them to work in synchronicity whenever a change is made.

In many real situations, data is copied across the enterprise system and the inability to correctly disseminate data throughout a company is very problematic. The larger the data set and the more complex its sourcing, the more likely a copy will be out of sync with the original. Often, systems expect each other to have perfect, up-to-date copies, particularly when interacting about the same data.

A weak or non-existent data communication structure can have a severe consequence for the company service quality and for that reason, new architectural solutions, such as those based on events, were defined.

2.3.1 Event-driven communication structures

An *event-driven architecture* is a software paradigm that offers an alternative to the traditional communication structures: it serves dynamic business needs decoupling the production and ownership of data from access to it. In particular, this kind of architectures can be useful when there is the need for handling big data sets at scale and with real time constraints. Event-based communications do not replace request-

response ones, but instead they define a completely different way of interacting based on *events*.

An *event* is used to share information and it can be defined as a change, action, or observation in a system or domain that produces a notification. In most cases, the application generating the event notification does not expect any response, and it lets the consuming application decide what to do with that information. Even if the applications generating the event notification expect a response, they expect it only indirectly [24], [53].

In an event-based system, all shareable data is published to a set of *event streams* that continuously describe what has happened in the organization, from simple occurrences to complex, stateful records. It is important to notice that events are not just signals indicating some information, such as the readiness of a certain element in the system, but *they are the data*: events act both as data storage and as a means of asynchronous communication between services [54].

2.3.2 EVENT-DRIVEN MICROSERVICES

Many organizations are migrating from the monolithic approach to microservice-based architectures to reach better agility, scalability, and reusability. In a real-world scenario this means overcoming challenges associated with distributed processing, ecosystem integration and service harmony.

Joining the two approaches, the *event-driven* and the *microservice-based*, gives the best of the two worlds, sharing many of the same microservice architecture principles, while the interaction between services is focused on events [55]. Unlike in synchronous communication patterns, event-driven architectures define indirect communication, i.e., it is not possible to directly send messages to consumers and get an acknowledgment about successful consumption. In most cases, there is an intermediate system, such a message broker, able to consume, store, and deliver events to consumers in a reliable way.

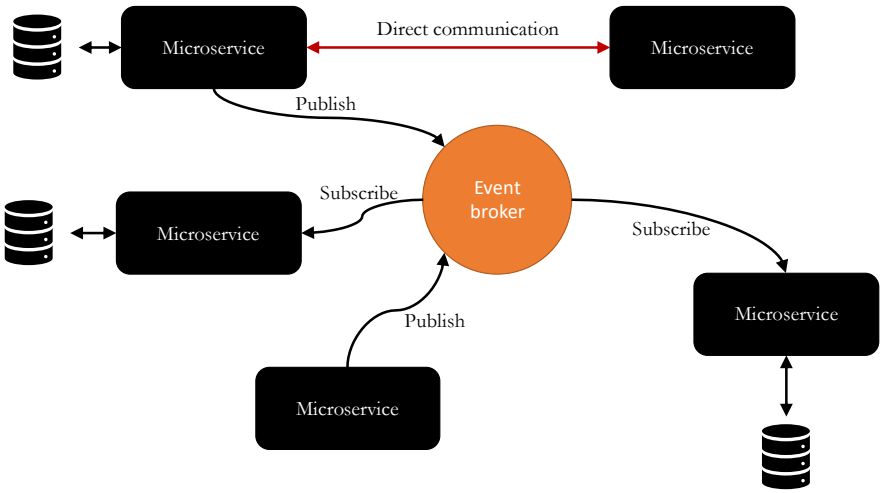


Figure 2-7. Scheme of an event-based microservice architecture.

The adoption of asynchronous event-driven microservice architectures can give many benefits [54]:

- ***Granularity***: services map neatly to bounded contexts and can be easily rewritten when business requirements change.
- ***Scalability***: individual services can be scaled up and down as needed.
- ***Technological flexibility***: services use the most appropriate languages and technologies. This also allows for easy prototyping using pioneering technology.
- ***Business requirement flexibility***: ownership of granular microservices is easy to reorganize. There are fewer cross-team dependencies compared to large services, and the organization can react more quickly to changes in business requirements that would otherwise be impeded by barriers to data access.
- ***Loosely coupling***: event-driven microservices are coupled on domain data and not on a specific implementation API. Data schemas can be used to greatly improve how data changes are managed.
- ***Continuous delivery support***: it is easy to ship a small, modular microservice, and roll it back if needed.

- ***High testability***: microservices tend to have fewer dependencies than large monoliths, making it easier to mock out the required testing endpoints and ensure proper code coverage.

2.3.3 EVENT MESH

The *event mesh* was born from event-driven architectures and it is a dynamic infrastructure layer for delivering events among decoupled applications, supporting complex and widely distributed topologies, from on-premises to multi-cloud deployments. An event mesh intelligently handles information routing between brokers allowing the cluster of brokers to appear as a single virtual entity, decoupling producers and consumers from a logical, geografic, and administrative or platform domain point-of-view.

The usage of traditional event-based microservice architectures can have its own challenges because it focuses mainly on an application level. In contrast, an *event mesh* works at the infrastructure level, allowing event-based network communication between applications, cloud services and devices [56], [57]. Event meshes are characterized by being environment-agnostic: they are designed to deliver events across disparate platforms, from cloud to IoT devices, no matter where applications are deployed and without the need for configuration of event routing.

Parts of the event-based systems, such as *brokers*, are integrated in the architecture and managed independently from the applications. This allows services written in different languages to produce and consume events independently of the specific event-based system implementation. The architectural organization makes an event mesh support a loosely coupled integration between legacy applications, databases and devices, and the latest microservice-based and cloud-native applications.

An event mesh is usually characterized by the following capabilities [56], [58]:

- Support for *various messaging services*.
- *Fault tolerance* for reliable message delivery, including automated recovery from network failures and fallback destinations for undeliverable messages.

- Support for *multi-protocol bridges* between disparate events, applications, and messaging platforms.
- Support for both *on-premises* and *multi-cloud* deployment.
- Support for *multicast* (all subscribers receive a copy of each message) or *anycast* (one subscriber receives a copy of each produced message) addresses.
- *Secure connections* and transmission of event messages.

In summary, an event mesh gives application developers and architects a foundation on which to build and deploy distributed event-driven applications, wherever they need to be built and deployed, dinamically and all in real-time.

2.3.4 COMPARISON BETWEEN SERVICE AND EVENT MESH

Both meshes implement at the infrastructure level advanced features to handle application components communications. The two solutions are complementary, providing different communication options.

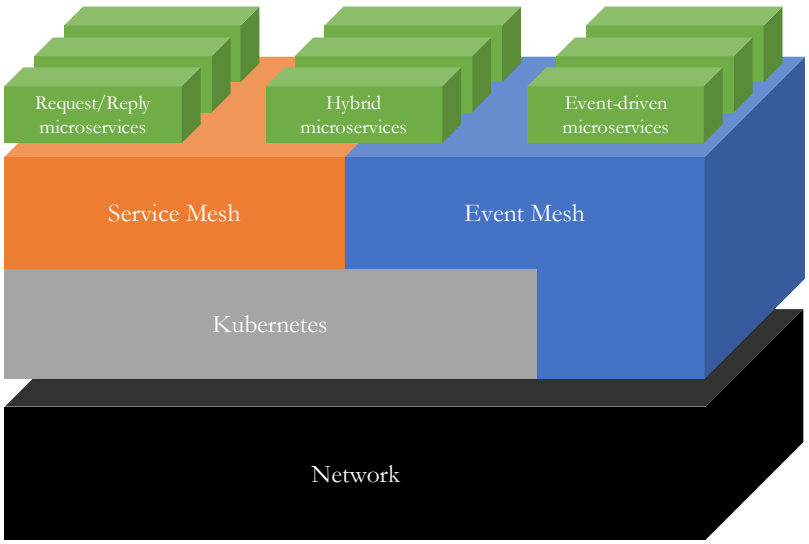


Figure 2-8. Comparison between service mesh and event mesh architectures.

Service meshes connect components in a synchronous way, supporting sometimes multiple cloud environments and legacy applications. They are subjected to some important restrictions [59]:

- No support for asynchronous event or streaming processing.
- Most traffic shaping and network services only apply to synchronous request-reply message exchange patterns and HTTP transport.
- Limited to connection-oriented routing and the targeting of transport connections, not the routing of actual data.
- Service mesh is currently mostly available on Kubernetes clusters.

Many microservice deployments can work well within the constraints of a service mesh, but companies are shifting to a more modern approach that requires also to introduce *event-driven applications* to achieve better performance, decoupling, and real-time response beyond the single Kubernetes cluster. In that context, an event mesh can help by connecting different type of applications, wherever they have been deployed, through asynchronous and event-based messaging.

Service meshes and event mesh can cohexist, providing developers the flexibility to choose which option best fits the application they are designing. From an infrastructure point of view, both can operate in the cloud and with the support of Kubernetes, but the event mesh can also connect nodes that are cloud-native without Kubernetes, on-premises, and even bare-metal. This expands the deployment options, giving enterprises the possibility to embrace hybrid solutions.

However, adding a new tool can often increase *complexity* and *costs*. For example, service meshes require injecting a sidecar alongside each component and this requires an additional number of resources while an event mesh requires a set of replicated brokers that need to keep messages even when a failure happen. For this reason, while for a small number of microservices, without explicit security requirements, or complex deployments, adding meshes can make a company incur in some exceeding costs and management complexity, in the case of enterprises, that have highly distributed modern applications, often with high scaling requirements, benefits outweigh the costs of operating microservices. This explains why many implementations of these technologies were born from enterprises who were publicly using microservices in production for many years, such as Google and Twitter [60], [61].

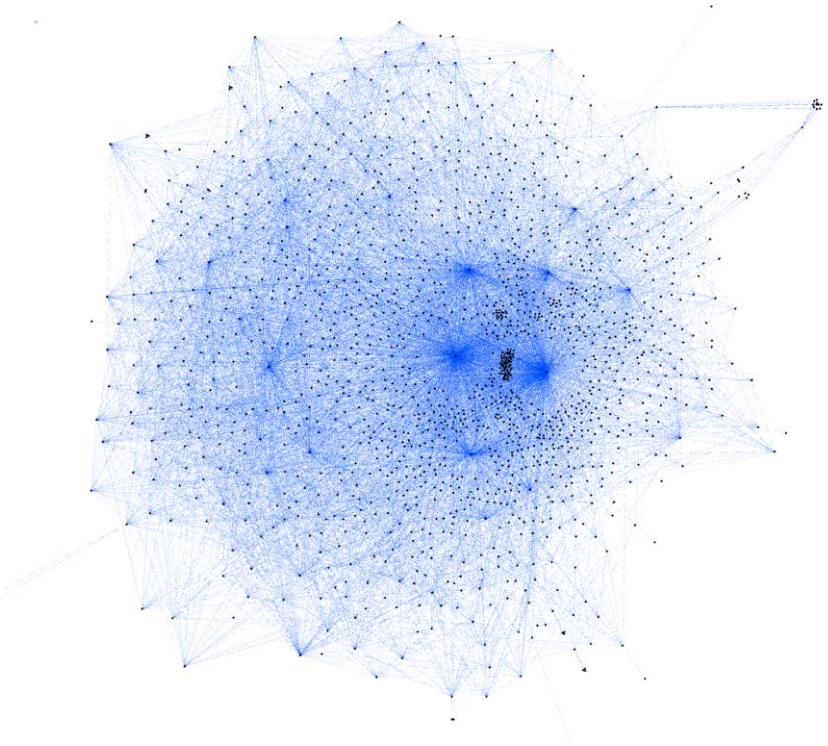


Figure 2-9. This graph represents a real complex microservice-based application from Monzo as of 2019 comprising 1500 components. Black points are microservices, blue lines are enforced network rules allowing traffic.

In the next chapter we will analyze some of these implementations, starting from service mesh and then concluding with event mesh.

3 SERVICE AND EVENT MESH IMPLEMENTATIONS

During the past few years, the service mesh ecosystem grew with many different implementations, some proprietary and other open source. The fact that there are several options on the market validates the interest on service mesh and it shows that the community has not already selected a de facto standard. While most of the solutions share the same reference architecture mentioned in the previous chapter, there are variations in approach and project structure that need to be taken into consideration when selecting a service mesh for a specific situation.

The implementations that we will analyze are Istio, Linkerd, Consul, NGINX Service Mesh, and Open Service Mesh. These were chosen because they appear as the most interesting for our work: either because they are particularly popular, or because they are open-source, or because they have an interesting architectural feature such as an alternative interaction system.

Later, we will take into consideration an event mesh implementation from the Apache Incubator project that actually is the only completely open-source in that scenario.

3.1 ISTIO

Originally founded by Google, IBM, and Lyft, Istio is a completly *open-source service mesh* implementation, licensed under the Apache License 2.0, and part of the CNCF Cloud Native Landscape.

Istio has grown to include contributions from companies beyond the original founders, such as VMware and Cisco and it is used by many organizations to design their commercial products to *protect*, *connect* and *monitor* large distributed applications that are deployed in hybrid or multi-cloud environments. For all these reasons it is one of the most important players in the service mesh scenario [46].

Istio itself has been built on top of other open-source projects, such as Envoy, Kubernetes, Jaeger, and Prometheus.

3.1.1 ARCHITECTURE

Istio logically subdivides into a control plane and a data plane:

- the ***control plane*** manages and configures proxies to *route* traffic, it applies *policies*, and it acquires *telemetry* data too. Up to Istio 1.4 it was divided into five independently deployable microservices, but from version 1.5 the control plane has been unified into a single entity called *istiod* [62].
- the ***data plane*** is composed of a set of intelligent proxies deployed as *sidecars*. These proxies mediate and control all network communication between microservices and they also collect and report telemetry on all mesh traffic.

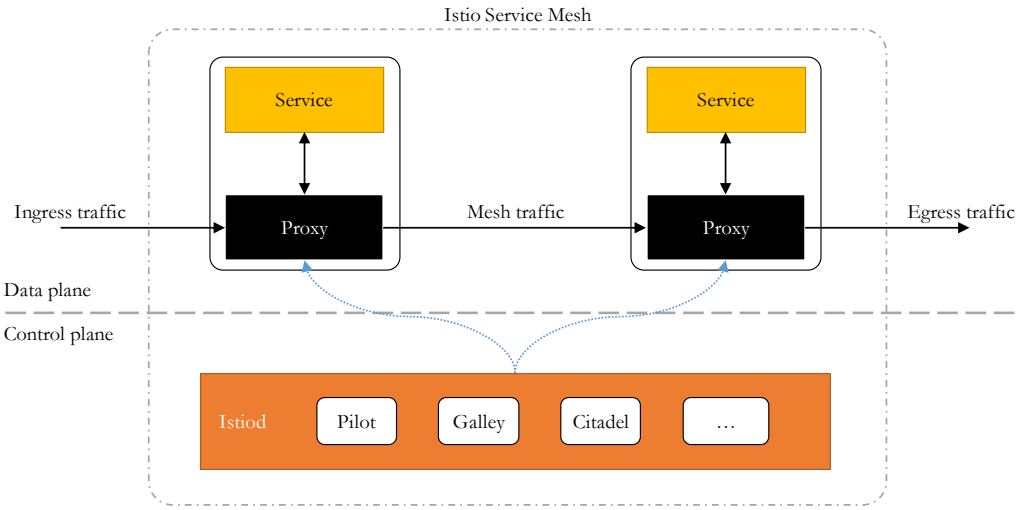


Figure 3-1. Scheme of istio architecture.

3.1.2 CONTROL PLANE

As already mentioned, istiod packages all the control plane components in a single binary and it is built on the same code and API contracts as the separate components, keeping the same features: service discovery, configuration and certificate management. Because of that, looking at the separate components can help in understanding the different areas covered by the control plane.

3.1.2.1 PILOT

Pilot is used to *manage traffic* and control inter-service traffic flows and API calls. It acts as a central controller of the service mesh and it is in charge of communicating with the sidecars by using the Envoy APIs. All the high-level routing rules that control traffic behaviour are converted into Envoy-specific configurations and propagated to the sidecars at runtime. Pilot is also responsible for resiliency features such as circuit breakers and retry logic.

Pilot abstracts the *service discovery* mechanisms into a *canonical representation* (standard format) independent of the underlying platform and it generates a specific configuration based on the canonical representation that all the sidecars conformant to the Envoy API can consume.

Platform-specific adapters in Pilot are responsible for populating this canonical model appropriately. For example, the Kubernetes adapter implements the necessary controllers to watch the Kubernetes API server for changes to the pod registration information, ingress resources, and third-party resources that store traffic management rules.

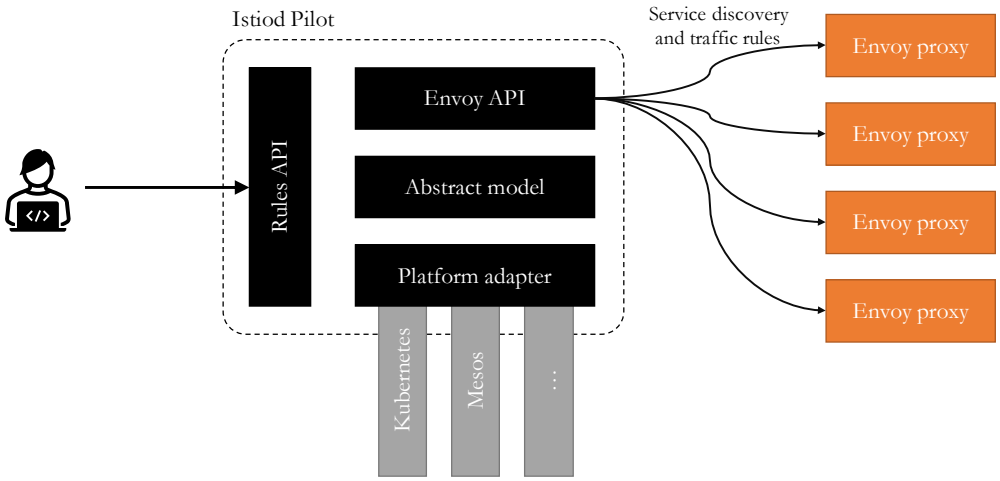


Figure 3-2. Pilot architectural scheme with platform-specific adapters.

3.1.2.2 *MIXER*

Mixer was a central component in istio architecture, mainly responsible for collecting *telemetry* data from the sidecar proxies as-well-as other control plane services.

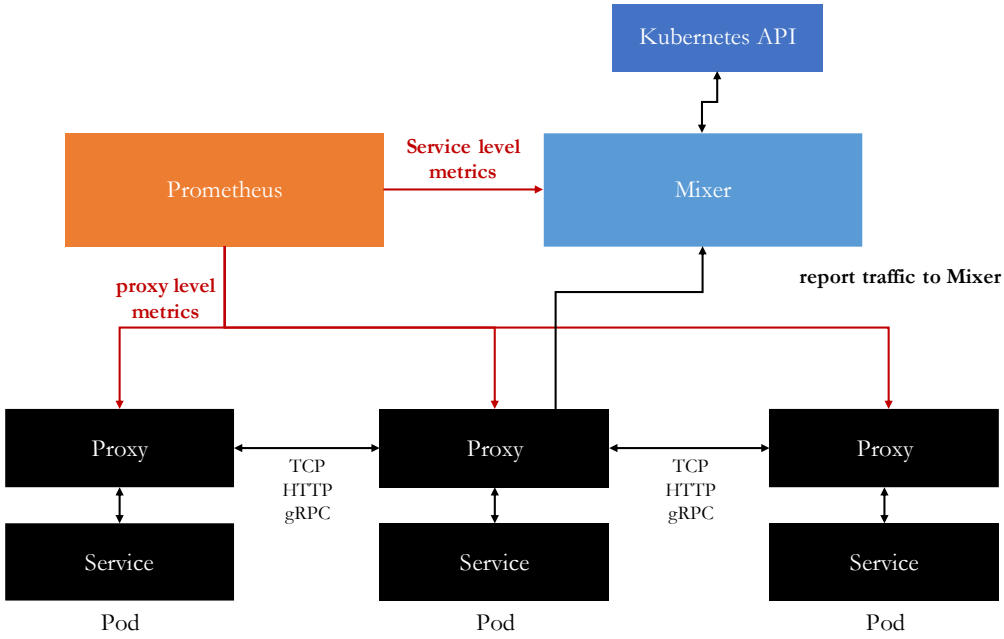


Figure 3-3. Scheme of istio with the Mixer component.

In the architecture shown in Figure 3-3, each Envoy sidecar calls Mixer after each request to report telemetry information. The proxies are sending data about the source and destination side of the request and most importantly the unique ID of the source and destination workloads used essentially as a unique Pod ID in a Kubernetes environment. After that operation, it is a Mixer responsibility to enrich the reported traffic information with metadata from Kubernetes and to expose the metrics on a specific endpoint for Prometheus to scrape [63].

Although the Envoy sidecars buffer the outgoing telemetry requests, that architecture generated significant resource consumptions in larger environments because an active connection was necessary between every proxy and Mixer. That obviously caused higher CPU and memory consumption in the proxies, and subsequently caused higher

latencies as well. To reduce the resource consumption, the telemetry feature was re-factored with a focus on the data plane.

3.1.2.3 ISTIO NEW TELEMETRY SYSTEM

The second version of istio telemetry system (Telemetry V2), thanks to an implementation of an Envoy extension, removed the Mixer central component, requiring the proxies themselves to directly access the Kubernetes metadata to enrich metric information.

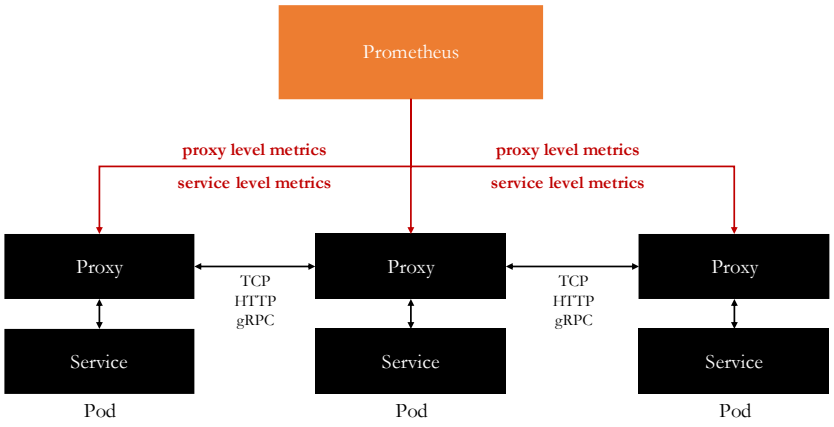


Figure 3-4. Scheme of istio without the istio component.

This was hard to do in older versions because developing an extension for Envoy required a monolithic development process: the binary needed to be deployed, needed rolling updates, etc. and the overall process was difficult to manage[8].

To enhance the development process, istio implemented on Google V8 engine a WebAssembly (WASM) runtime for Envoy through which developers can write their custom code, compile it to WebAssembly plugins, and configure Envoy to execute it. These plugins can hold arbitrary logic, so they are useful for all kinds of message integrations and mutations.

[8] In addition, the extensions needed to be written in C++.

In-proxy service-level metrics in Telemetry V2 are provided by two custom plugins:

- ***metadata-exchange***: how to make client/server metadata about the two sides of a connection available in the proxies.
- ***stats***: it records incoming and outgoing traffic metrics into the Envoy statistics subsystem and makes them available to monitoring systems.

Envoy proxies collect metrics automatically from the traffic they handle: the WebAssembly-based plugin system allows Envoy to process traffic data that flows through the proxy and then it can prepare metrics in Prometheus format to be emitted. Then Prometheus, that is a software that can be used to monitor the service mesh deployment, uses Kubernetes built-in service discovery to directly scrape the Envoy proxies and collect the results. Finally, the metrics can be visualized by using tools such as Grafana and Kiali.

3.1.2.4 GALLEY

Another component in the istio control plane architecture is *Galley*. It is used to validate and process the istio API configurations made by users. It insulates the other istio components from the details of obtaining data from the underlying platform, such as Kubernetes.

3.1.2.5 INJECTOR AND CITADEL

The last relevant istio control plane elements are Injector and Citadel.

The first was responsible for auto-injecting the data plane proxies in the application containers and setting up bootstrap, while the second was used to enforce strong service-to-service and end-user authentication with built-in identity and credential management.

3.1.2.6 CONTROL PLANE UNIFICATION

From its beginning, the entire istio project was developed following the microservices design approach. As already shown, the control plane was divided into different services, like a modern cloud-native application.

As istio adoption increased, the development team decided to shift to a monolithic approach. By receiving customers feedback, they noticed that in most istio installations, all the control plane components were delivered and managed as a single entity, mainly because operated by a single team or individual. The istio operators were paying for the greater operational complexity without gaining much of the benefits typically associated with microservices: independent rollout, scale, and security isolation.

The solution to this problem was to consolidate the control plane into a monolith called *istiod*, comprising Pilot, Citadel, Galley, and Injector. As already mentioned, the Mixer component was removed in a concurrent project aimed in which the Envoy proxies became directly responsible of the telemetry reporting to the monitoring back end. This change was due to similar reason to that of istiod in that a design pattern, identified as necessary at massive operational scale, was not justified against the maintenance and performance overheads for typical users [64], [65].

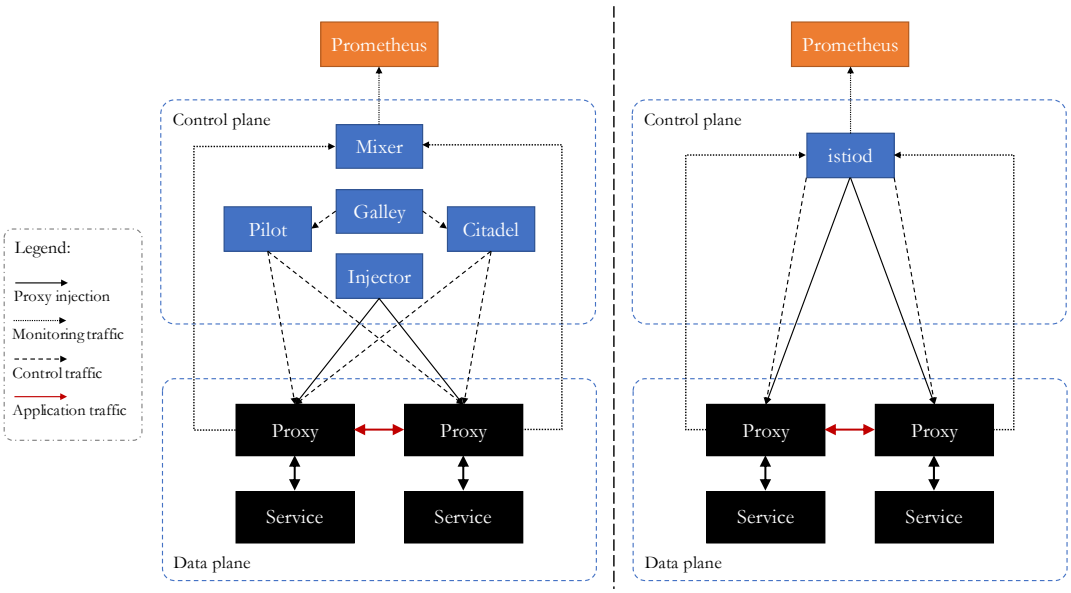


Figure 3-5. The architectural evolution of istio control plane: (a) microservices-based and (b) monolith.

Compared with the microservices-based approach, this new monolithic control plane offers several benefits [66], [67]:

- ***Simplified installation***: it is simpler to deploy a single binary than multiple components with their associated configurations.
- ***Easier configuration***: many of the configuration options that istio had before were related to the orchestration of the components.
- ***Simple debugging***: having fewer services means less cross-service environmental debugging.
- ***More scalable***: the control plane monolith becomes relatively easy to scale because it is made of just one component.
- ***Easier version management***: the time to start, upgrade, and remove are reduced because the monolith does not require to deal with dependencies and start-up orders.

The costs of istio control plane are mainly dominated by serving the Envoy dynamic resource discovery (xDS) APIs, which we will describe later, that program the data plane [66]. Every other feature has a marginal cost. This means there is very little value to having those features in separately scalable microservices.

It should be noted that internally istio still maintains the logical separation between some of its original control plane components and that each capability is exposed as a discrete API. This is required when there is the need to be retrocompatible, but also it can be particularly useful when making complex deployments, such as multicluster ones, where istiod can be deployed as a single-purpose service such as “injection”, “certification provider”, or “validation”.

3.1.3 DATA PLANE

The second element of the istio architecture, the data plane, has *Envoy* as the main component. Envoy is a proxy for the container it comes up beside and because it is injected as a sidecar, there is no need to redesign the microservices-based application running on Kubernetes [68]. Developers can write the code the way they used to, without having to worry about the operational and security aspects.

Istio also uses Envoy to run gateways [46] and this allows to manage inbound and outbound traffic from a mesh by defining a specific proxy at the edge of the mesh

capable of managing HTTP/TCP connections. Each gateway communicates with the control plane like any other proxy in the data plane does.

3.1.3.1 ENVOY ARCHITECTURE

The Envoy architecture is made of a *Listener* followed by a *filter chain*. The first component allows Envoy to listen to network traffic at a configured address, so that it is possible to enable a flow of traffic through the proxy. Each Listener defines a filter chain: built-in or custom filters can be composed and arranged in such a way to configure Envoy to compute different operations like translating protocol messages or generating statistics [69].

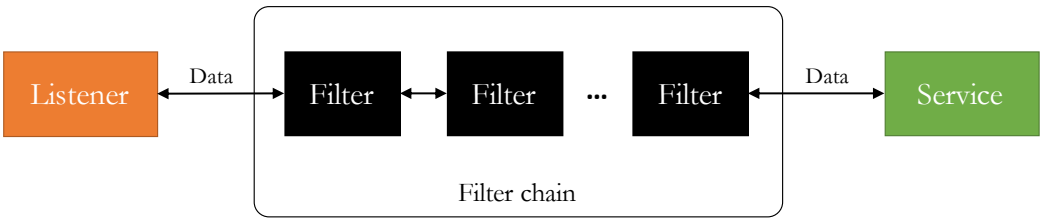


Figure 3-6. General overview of the Envoy filter chain architecture.

Envoy provides three types of filters that form a hierarchical filter chain:

- ***Listener Filters*** access raw data and manipulate metadata of L4 connections during the initial (pre)connection phase. For example, the TLS Inspector Filter identifies if a connection is TLS encrypted and parses the TLS metadata associated with the connection.
- ***Network Filters*** access and manipulate raw data on L4 connections i.e., TCP packets. For example, the TCP Proxy Filter routes client connection data to upstream hosts and it also generates connection statistics.
- ***HTTP Filters*** operate at L7 and are optionally created by a final Network filter i.e., the HTTP Connection Manager. These filters access and manipulate HTTP requests and responses. For example, the gRPC-JSON Transcoder Filter exposes a REST API for a gRPC backend and translates requests and responses into corresponding formats.

Once a routing decision has been made by a filter stage (L3/L4 connection or a new HTTP stream), these filters ask the *cluster manager* for connection to an upstream service. This component handles all the complexity of knowing which hosts are available and healthy.

3.1.3.2 SIDECAR INJECTION PROCESS

One important feature introduced firstly by istio was the ability to automatically inject a sidecar proxy.

Before the initialization of a container, istio prepares the environment for sending the traffic to the proxy. For this specific task an *init container* called *istio-init* is used to setup the iptables rules of the container. In that way all the inbound and outbound traffic goes through the sidecar proxy.

An init container is different than an app container in the following ways:

- It runs before an app container is started and it always runs to completion.
- If there are many init containers, each should complete with success before the next container is started.

3.1.4 CONTROL PLANE AND DATA PLANE COMMUNICATION

After having described the two planes of the istio architecture, we want to delve into the communication between those.

Envoy is architected such that it is possible to make *static* and *dynamic* proxy configurations. The first case can be useful in simple scenarios with a limited number of entities in the service mesh. But the more the complexity, the less static configuration helps: configurations must be provided by hand and proxies must be restarted to accept them. In the second case, each Envoy proxy discovers dynamic resources at runtime, without the need to restart, by watching a path in the local filesystem or by querying one or more management servers. These discovery services and their corresponding APIs are known as *x Discovery Service* (xDS) [70], [71].

The resources can be requested by proxies through three different kind of subscriptions:

- ***filesystem watching***: it monitors files in a specific path.
- ***gRPC streaming subscription***: each xDS API can be individually configured to point to the cluster of the corresponding upstream management server.
- ***REST-JSON polling subscription***: a single xDS API can perform synchronous polling of REST endpoints.

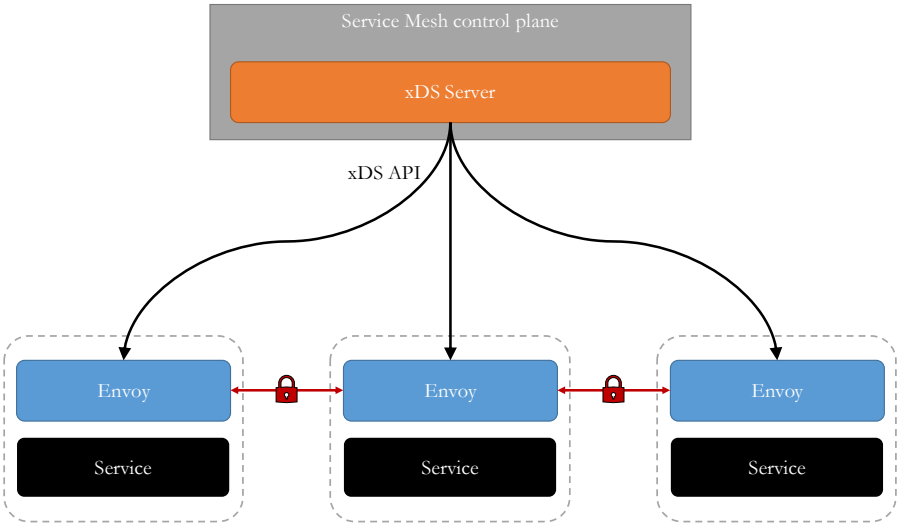


Figure 3-7. General overview of a service mesh. Black lines indicate the xDS communication.

Istiod acts as a central xDS management server and it uses gRPC streaming subscriptions to configure the data plane. Inside each pod resides a component that is called *istio-agent* or *pilot-agent*. It is still part of the control plane, but it runs on a *per-Pod* basis and it helps istiod to discover endpoints, TLS secrets and other cluster information.

3.1.5 ADVANCED FEATURES

Istio is often used toghether with Kubernetes, on which it relies to realize some features. For example, it does not provide any service discovery, but it keeps an internal *service registry* containing the set of service endpoints running in the mesh.

Pilot adapters automatically add to the registry the services discovered from underlying platforms, such as Kubernetes, through its APIs.

Although one of the main scenarios in which istio is usually deployed is in a single Kubernetes cluster, it can work also in different and more advanced situations:

- It supports not only containers, but virtual machines too.
- The mesh can be confined in a single cluster or distributed across multiple clusters.
- The services can be located in a single fully connected network or they can be deployed across multiple networks by using gateways.
- To ensure high availability it is possible to use multiple control planes instead of a single one.
- The clusters can be connected in a single multi-cluster service mesh or they can be federated into a multi-mesh deployment.
- The service mesh can also support multi and hybrid cloud deployments.
- Locality allows to deploy istio to create in multi-zonal services.

3.2 LINKERD 2

Linkerd is an open-source service mesh solution specifically designed for Kubernetes with a focus on providing a lightweight implementation with a minimalist design [46].

It is licensed under the Apache License 2.0 and it is hosted by the CNCF, with major contributions from Buoyant, which founded the project.

First based on Twitter Finagle, Linkerd was written in Scala and designed to be deployed on a per-host basis, but criticisms of its comparatively large memory footprint led to the development of Conduit [72], a lightweight service mesh specifically designed for Kubernetes, written in Rust and Go. This project was since folded into Linkerd, which relaunched as Linkerd 2.0 [50].

The previous 1.x version is still accessible, but it is in maintainance and no new features will be added. For this reason, when Linkerd will be mentioned during the dissertation, we will implicitly refer to version 2.0.

3.2.1 ARCHITECTURE

Like many other service meshes, the Linkerd architecture is logically divided in a control plane and in a data plane [73]:

- the ***control plane*** aggregates telemetry data, provides user-facing APIs, manages and configures proxies. It is composed by a set of services that run in a dedicated Kubernetes namespace (*linkerd* by default).
- the ***data plane*** consists of transparent proxies deployed as sidecars that mediate and control all network communication between microservices and that send also telemetry data to the control plane.

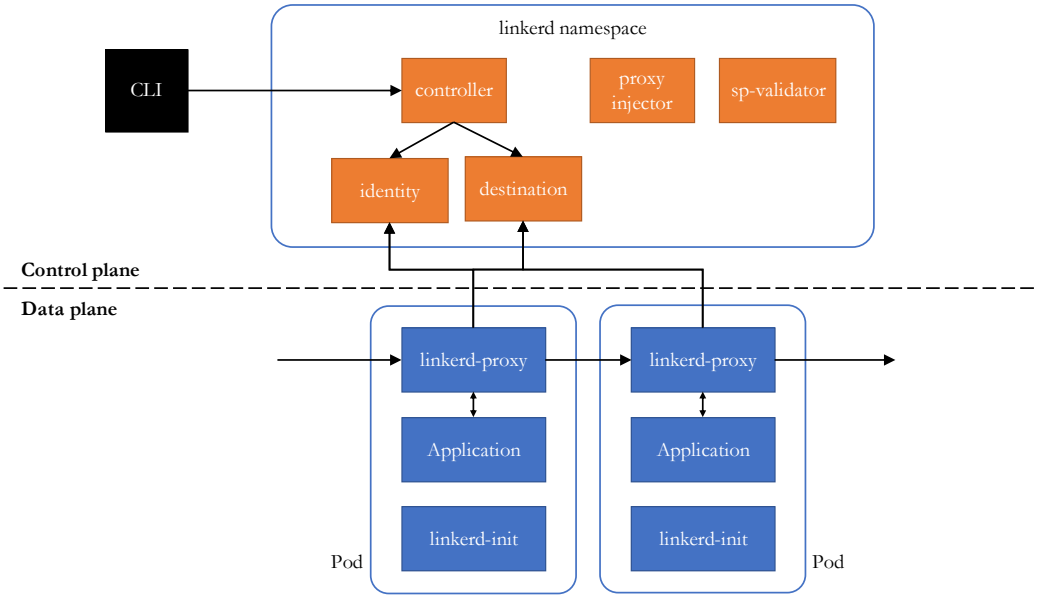


Figure 3-8. Linkerd general architecture.

3.2.2 CONTROL PLANE

Linkerd control plane is not made of a single entity, but it is splitted into a multiplicity of operational components based on their functional boundaries (destination, api, web, etc.). These components run on the same data plane as the application does, allowing to use the same tooling to inspect the behavior.

The first component is the *command-line interface* (CLI) that typically runs outside of the cluster, helping users accomplish various tasks by interacting directly with the control plane.

Other notable components are the following:

- The ***Controller*** oversees mediating the communication between CLI and the Linkerd internal controller components by exposing an API interface.
- The ***Destination*** is mainly responsible for service discovery and it mediates the communication between the data plane and the information Kubernetes provides. Linkerd Destination looks up the IP address in the Kubernetes API and then, if the IP corresponds to a Service, Linkerd will load balance across the endpoints of that Service and will apply any policy from that Service Profile. Otherwise, if the IP address corresponds to a Pod, Linkerd will not perform any load balancing or apply any Service Profiles.
- The ***Proxy Injector*** relies on a Kubernetes admission webhook and it is responsible, if enabled, to automatically add the data plane proxy to pods.
- The ***Identity*** component acts as a TLS Certificate Authority that accepts CSRs from proxies and returns signed certificates.
- The ***Service Profile Validator*** validates new service profiles before they are saved.

In addition to the built-in components, Linkerd provides some other features as extensions. One of those is *viz* that includes by default a full on-cluster metrics stack, comprising CLI tools, a web dashboard, a Prometheus instance, and a pre-configured Grafana dashboard. The metrics stack may require significant cluster resources. Prometheus, in particular, consumes resources as a function of traffic volume within the cluster.

3.2.3 DATA PLANE

The proxy on which the data plane relies, *Linkerd2-proxy*, is specifically designed for the second version of the Linkerd service mesh and because it is tightly coupled with the control plane, it cannot be used by other projects [74].

With respect to other proxies, such as Envoy, NGINX Plus and other solutions that are flexible enough to provide many different features (ingress, egress, proxy sidecar, API gateway etc.), Linkerd proxy is designed just for service meshes and for that reason it has a simpler architecture. Less complexity means not only a *more maintainable code*, but also it allows to optimize resource consumption: the aggregate CPU and memory consumed by the data plane by many proxies in a production system is a critical cost of a service mesh; so simplyfing the proxy architecture means *better resource usage*. Moreover, keeping the complexity at the minimum allows to define a smaller attack surface, so *better security* overall.

Linkerd2-proxy is designed also to work with *zero-config* [75]. That means that there is no user-facing YAML because it is configured automatically through environment variables set at injection time and then modified when needed by the Linkerd control plane at runtime. This simplifies a lot the service mesh setup, requiring no tuning nor tweaking thanks also to features like protocol detection and Kubernetes native service discovery.

It is important to highlight that Linkerd2-proxy is written in Rust to achieve high performance and memory safety, but, unlike Envoy, it is not possible to develop extensions for that proxy. However, by installing, for example, the *viz* control plane extension, it is possible to access a wide variety of features for metrics analysis: so each proxy is capable to gather metrics automatically from the traffic they handle and subsequently, the Prometheus instance will collect and will store all Linkerd metrics by scraping proxy */metrics* endpoint.

Finally, Linkerd shares with istio the same out-of-process architecture design: the proxy is language agnostic and it runs alongside the application requiring to be injected at the container creation. For this reason, the data plane comprises also the *linkerd-init* container that, similarly to the istio-init container, is added to each meshed pod before any other container starts using iptables to route all the inbound and outbound TCP traffic through the pod.

3.3 CONSUL CONNECT

Consul Connect is an open-source, distributed and highly available system with a unique architecture that allows to use each feature individually, such as the service discovery, or together to build a complete service mesh solution [46].

The project was founded by HashiCorp and it is licensed under the Mozilla Public License 2.0. HashiCorp provides also a commercial version called Consul Enterprise.

3.3.1 ARCHITECTURE OVERVIEW

A distinctive feature of Consul Connect is that it works *per-node* instead of per-pod. Each node in the service mesh need to run an *agent* that is responsible for health checking the services on the node as well as the node itself [76]. The agent is not required for discovering other services or to store data because this information is kept by *servers* which are responsible for storing and replicating data. The servers maintain a catalog, which, formed by aggregating information submitted by the agents, maintains the high-level view of the cluster, including which services are available, which nodes run those services, health information and more.

Consul has been developed to support complex deployment scenarios and for this reason it is organized into *Data Centers*, each running a cluster of Consul servers. When a cross-data center service discovery or configuration request is made, the local Consul servers forward the request to the remote Data Center and return the result [77].

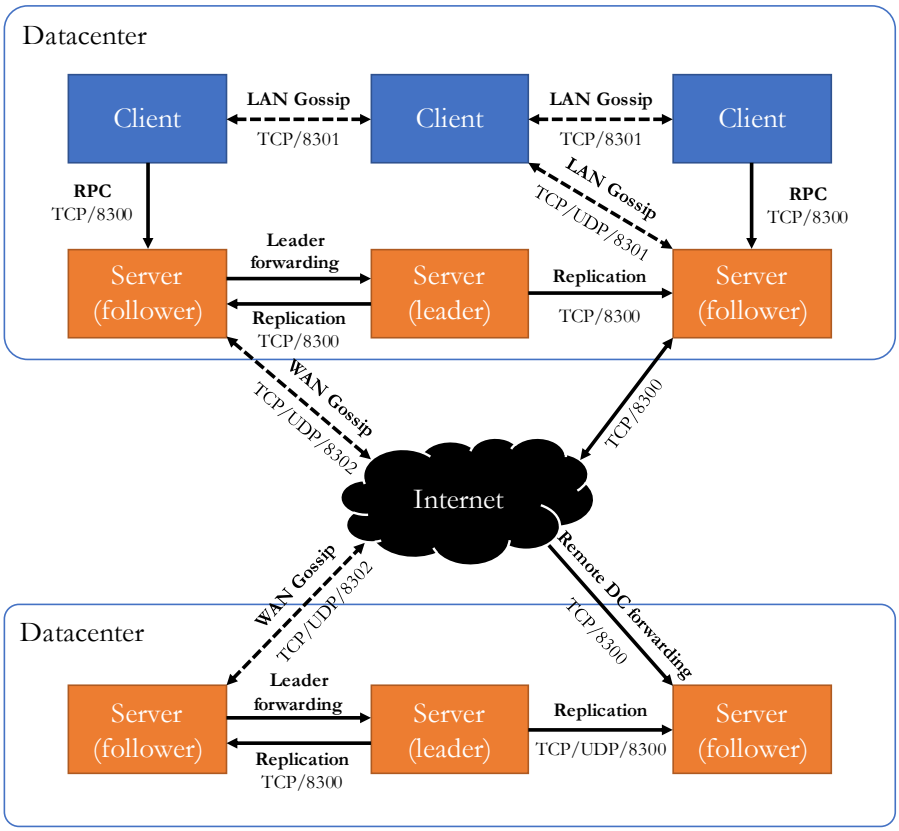


Figure 3-9. Consul Connect general architecture showing multiple data centers involved in the service mesh.

Consul Connect has a different architecture than istio and Linkerd, but it is possible to recognize the control and data planes:

- the ***control plane*** provides mainly four features: service discovery, secure communication, resource configuration and network segmentation. It is composed by agents which can act as *clients* or *servers*.
- the ***data plane*** uses proxies to control the communication that happens between services.

3.3.2 DATA AND CONTROL PLANES

With respect to the data plane, while Consul ships with a simple built-in proxy, it cannot be used for production deployments because it does not support many Consul Connect features and it is not in active development. Instead, Consul data plane relies mainly on the same third-party proxy istio uses which is *Envoy* [78], [79]. Since Consul

uses this component in the same way that other service mesh implementations use it, and having already covered it previously, we continue our discussion by addressing Consul control plane analysis.

At its core Consul control plane is composed by *agents* that, deployed on each node of the cluster, maintain membership configuration, register services, run checks etc.

Each agent can run in two modes:

- ***client*** node makes up the majority of the cluster, and it is very lightweight as it interfaces with the server node for most operations and maintains very little state of its own;
- ***server*** node has the extra burden of participating in the consensus quorum[9], storing cluster state, and handling queries.

The control plane is therefore made up of a single binary, without multiple components that need to be deployed and configured. Each client keeps a local cache that is constantly updated by the server. This reduces the need for any external communication and allows for quick and effective changes to be made at the edge. Moreover, because there are no centralized planes that could cause bottlenecks and adversely affect performance, APIs respond quicker.

To better understand the control plane architecture, it is appropriate to take a look at the agents lifecycle.

When an agent is first started, it has no information about others and it must discover its peers by *joining* the cluster. After the node joins, this event is gossiped to the entire cluster. If the agent is a server, existing servers will begin replicating to the new node.

In the case of a network failure, some nodes may become unreachable and for that reason they are marked as failed and the catalog is updated accordingly. Because it is impossible to distinguish between a network failure or a node crash, both cases are

[9] The higher burden on the server nodes means that usually they should be run on dedicated instances. So, they are more resource intensive than a client node.

treated in the same way. Once the network recovers or a crashed agent restarts the cluster will repair itself and unmark a node as failed.

When a node wants to *leave*, it must specify its intention and the cluster marks that node as having *left*. Unlike the failed case, all the services provided by a node are immediately deregistered. If the agent was a server, replication to it will stop.

To prevent an accumulation of dead nodes, the ones in either failed or left states, Consul automatically removes them out of the catalog with a process termed *reaping*.

The balance between the number of *clients* and *servers* is not static, but it can change over time to keep balance between availability and performance, as consensus gets progressively slower as more machines are added. To be noted that there is no limit to the number of clients and they can easly scale.

3.3.2.1 TELEMETRY

The Consul agent collects and periodically aggregates various runtime metrics about the performance of different libraries and subsystems. These data, if properly configured, are streamed to a *statsite* or *statsd* server which is responsible for collecting all telemetry information, then flushing it to Graphite or any other metric store [80].

3.3.3 INGRESS-EGRESS

Consul uses *gateways* to provide connectivity into, out of and between service meshes. Different modes are provided by different kind of gateways [81]:

- ***mesh gateways*** enable service-to-service traffic between Consul data centers. Those data centers can reside in different clouds or runtime environments where general interconnectivity between all services in all datacenters is not feasible.
- ***ingress gateways*** accept potentially unauthenticated traffic from outside to services in the mesh.
- ***terminating gateways*** route traffic from services in the Consul service mesh to external services that do not have sidecar proxies or are not integrated natively.

These may be services running on legacy infrastructure or managed cloud services running on infrastructure not in direct control. Terminating gateways effectively act as egress proxies that can represent one or more services, terminating mTLS connections, enforcing Consul intentions, and forwarding requests to the appropriate destination.

3.3.4 ORCHESTRATORS AND CONSUL

With regards to orchestrators, Consul Connect is well integrated with *HashiCorp Nomad*[10], providing features like secure service-to-service communication between Nomad jobs and task groups, while taking advantage of some Nomad characteristics such as the ability to use the dynamic port feature. Consul can, however, also run directly on *Kubernetes* providing a way to automate, secure, and observe the connections between pods and clusters, at the same time enhancing the scalability and resiliency of the microservices platform.

One of the most important features Consul provides is *consistency* when securely connecting nodes within Kubernetes clusters with external services. This is possible as long as they can communicate to the Consul server nodes via the network, responsible for the cross-data center communication, as already explained. Within the Kubernetes environment, Consul clients can run as pods on every node and expose the Consul API to running pods, enabling many Consul tools to work on Kubernetes since a local agent is available. This will also register each Kubernetes node with the Consul *catalog* for full visibility into the infrastructure.

The Consul catalog can be synchronized with the Kubernetes service registry so that, on one hand, pods inside a Kubernetes cluster can connect to external services using the native Kubernetes service discovery, on the other hand, external services can connect to registered Kubernetes services using the Consul service discovery.

[10] Nomad is a flexible scheduler and workload orchestrator that enables an organization to easily deploy and manage any containerized or legacy application using a single, unified workflow. Nomad can run a diverse workload of Docker, non-containerized, microservice, and batch applications.

The synchronization between Consul and Kubernetes can occur in three ways [82]:

- ***Bidirectional sync*** allows Kubernetes services to be available to Consul agents and services registered in Consul, but external to the Kubernetes cluster, can be available as first-class Kubernetes services.
- ***Kubernetes services synced to the Consul catalog*** enables the first to be accessed by any node that is part of the Consul cluster, including other distinct Kubernetes clusters.
- ***Syncing Consul services to Kubernetes service*** enables non-Kubernetes services, external to the cluster, to be accessed in a native Kubernetes way, through *kube-dns* or environment variables. This makes it very easy to automate external service discovery, including hosted services like databases.

3.4 NGINX SERVICE MESH

NGINX Service Mesh (NSM) is a complete service mesh solution for QoS and resilience in general, thanks to the highly proven technologies on which it builds. The project is owned and developed by F5 and it is not open source, but some of the technologies it uses are, such as the Kubernetes Ingress Controller and NATS.

3.4.1 ARCHITECTURE OVERVIEW

Like the other service mesh solutions that we have already analyzed, NGINX Service Mesh is divided in two layers [83]:

- The ***control plane*** is responsible to enforce a desired state across managed applications.
- The ***data plane*** follows the sidecar pattern so that it replicates with the applications workload.

The service mesh can be controlled using the *NSM CLI*, an utility that communicates with the control plane through the Kubernetes API server.

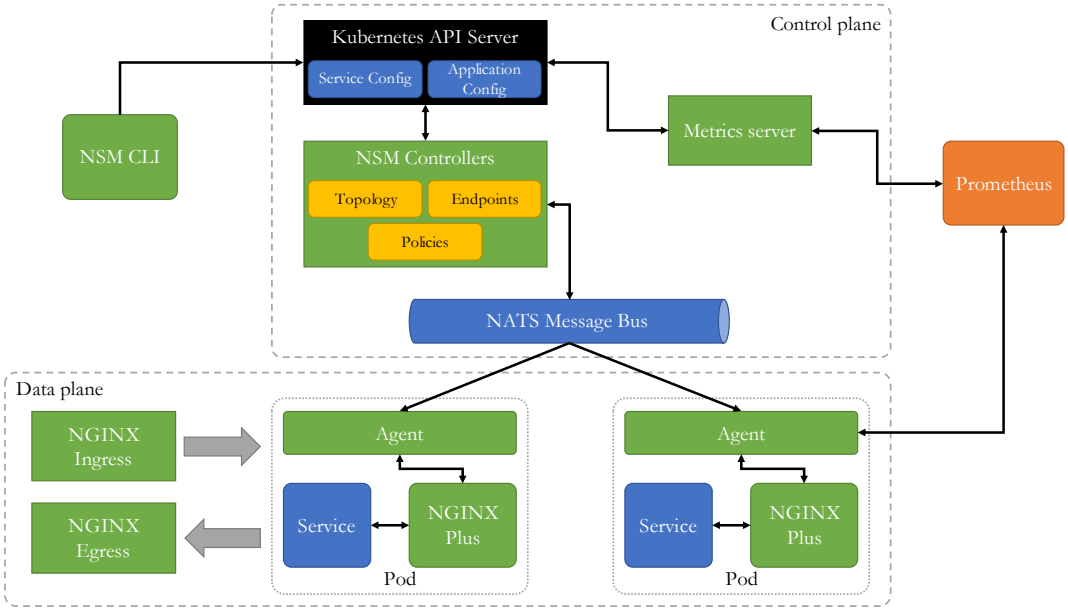


Figure 3-10. NGINX Service Mesh general architecture.

3.4.2 CONTROL PLANE

The NSM control plane is composed by the following elements [84]:

- NSM Controller;
- NATS;
- NSM metrics server.

NSM Controllers, the core of the control plane, oversee setting the desired service mesh configuration. They watch a set of native Kubernetes resources (Services, Endpoints and Pods), a collection of custom resources defined by the Service Mesh Interface (SMI) specification, and individual resources specific to NGINX Service Mesh. When a new event occurs in Kubernetes that NSM Controllers are looking for (e.g., a new application or traffic policy has been created), then the control plane builds an internal configuration based on this data and it sends it to the sidecars. The communication between the control plane and the data plane is transparently managed by *NATS*, an advanced technology for messaging in large, cloud-native, distributed systems.

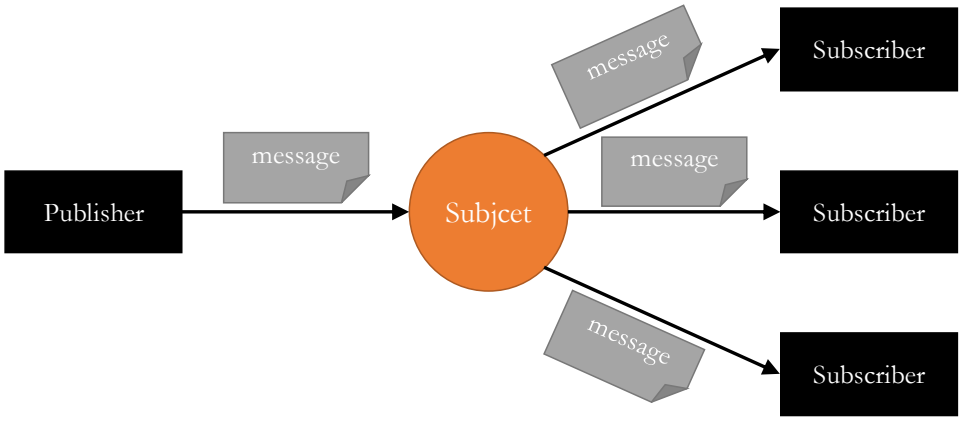


Figure 3-11. NATS publish subscribe model.

NATS follows the publish-subscribe model, and it manages addressing and discovery not based on hostnames and ports, but on *subjects*, sometimes called topics, that are categories for messages [85], [86]. In this way, the communication does not depend on network location and this provides an abstraction layer between the application or service and the underlying physical network. Differently from the other service mesh solutions, in which the control plane communicates through 1:1 API calls, NATS allows NSM to perform M:N communications, reducing the control plane development complexity and the consumption of resources at runtime, enhancing the scalability of the whole system. Other advantages of using NATS are that it is language agnostic and it can run on multiple platforms, such as inside Kubernetes clusters, virtual machines, or bare metal, enabling hybrid deployments.

The NSM control plane comprises also the *metrics server* that provides quick access to the metrics data in a standard format by extending the Kubernetes APIs. NGINX Service Mesh automatically deploys a Prometheus instance that directly scrapes data plane sidecars and the control plane metrics server to collect all information about the service mesh cluster[11].

[11] Metrics can be accessed by directly querying Prometheus, the NSM metrics server, or viewing the built-in Grafana dashboard.

3.4.3 DATA PLANE

NGINX data plane follows the sidecar pattern like other service mesh solutions, but it consists of two components[12]: an *agent* and a *NGINX Plus* instance.

The *agent* accepts the NGINX Service Mesh control plane configuration via NATS and uses this data to configure the NGINX Plus instance.

NGINX Plus is a commercial version of the famous NGINX web server. Here it is used mainly with the purpose of being a proxy, managing the communications between services, but it can be used also in other ways, such as a load balancer and API gateway. In particular, it is possible to deploy NGINX Plus as an Ingress or Egress Controller to provide production-grade control over north-south traffic[13] with a single configuration. Because of its integration within the NSM solution, the Ingress Controller can communicate without a sidecar injected with the workloads, reducing latency and complexity to the Kubernetes environment.

3.5 OPEN SERVICE MESH

Open Service Mesh (OSM) is an open source, lightweight, and extensible cloud native service mesh, initially developed by Microsoft. It leverages out-of-the-box observability features for highly dynamic microservice environments and it configures through SMI open standard specification [87], [88].

3.5.1 ARCHITECTURE OVERVIEW

Open Service Mesh relies on Kubernetes and, like the projects we already analyzed, it is divided in two parts:

- The ***control plane*** manages and configures proxies to route traffic.
- The ***data plane*** is responsible for mediating the communication between services.

[12] NSM uses *init containers* to inject sidecars and they work in the same way as the projects we already described.
[13] North-South is the traffic that goes in and out of the cluster, while East-West resides in it.

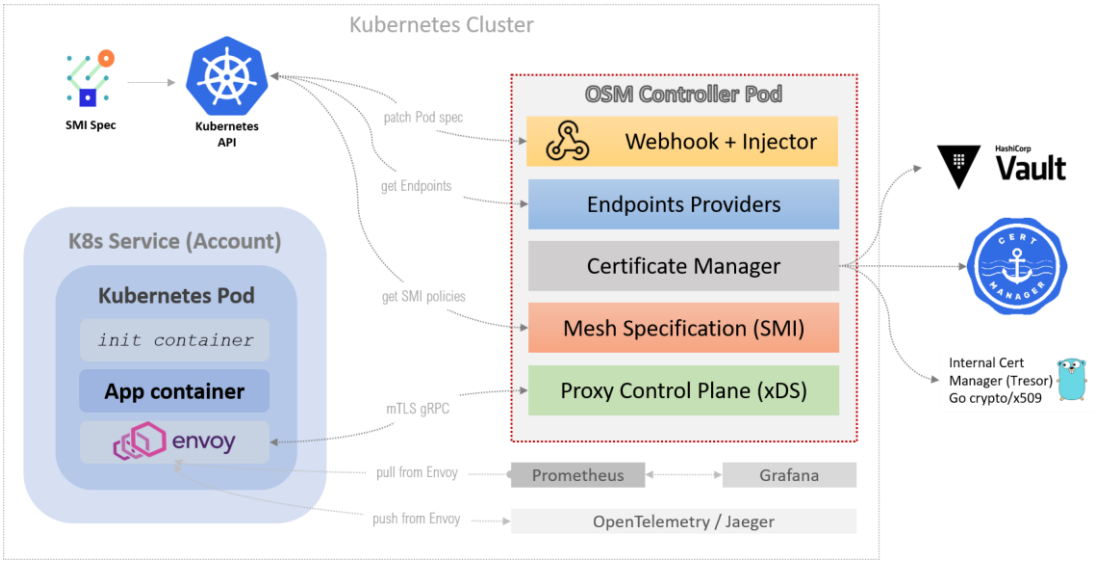


Figure 3-12. Open Service Mesh general architecture with the main components and their interactions.

3.5.2 CONTROL PLANE AND DATA PLANE

The OSM *data plane* uses by default Envoy for proxy sidecars, but it accepts also other reverse-proxy solutions with service mesh capabilities. The data plane receives configuration updates, through mTLS gRPC calls, from the control plane, which implements the Envoy xDS API.

The OSM *control plane* is made of the following components:

- Proxy Control Plane;
- Endpoints Providers;
- Certificate Manager;
- Mesh Specification;
- Mesh catalog.

The *Proxy Control Plane* is responsible for the communication with the proxies in the data plane. It oversees enforcing the configuration updates and it implements the interfaces required by the specific reverse proxy chosen. Because the Proxy Control Plane has a central role in the service mesh when it comes to traffic policy enforcement and connectivity management between services, it is designed to be stateless and highly available.

The *Endpoints Providers* abstract the controller workings from the underlying compute platforms (Kubernetes clusters, on-premises machines, or cloud-providers VMs) participating in the service mesh.

The *Mesh catalog* collects inputs from all other components and dispatches configuration to the proxy control plane. It combines the outputs of all other components into a structure, which can then be transformed into proxy configuration and dispatched to all listening proxies via the Proxy Control Plane.

The *Certificate Manager* provides each service participating in the service mesh a TLS certificate that is used to establish and encrypt connections between services using mTLS.

The *Mesh Specification* is a wrapper around the existing SMI Spec components. It provides simple methods to retrieve SMI Spec resources, abstracting away cluster and storage specifications.

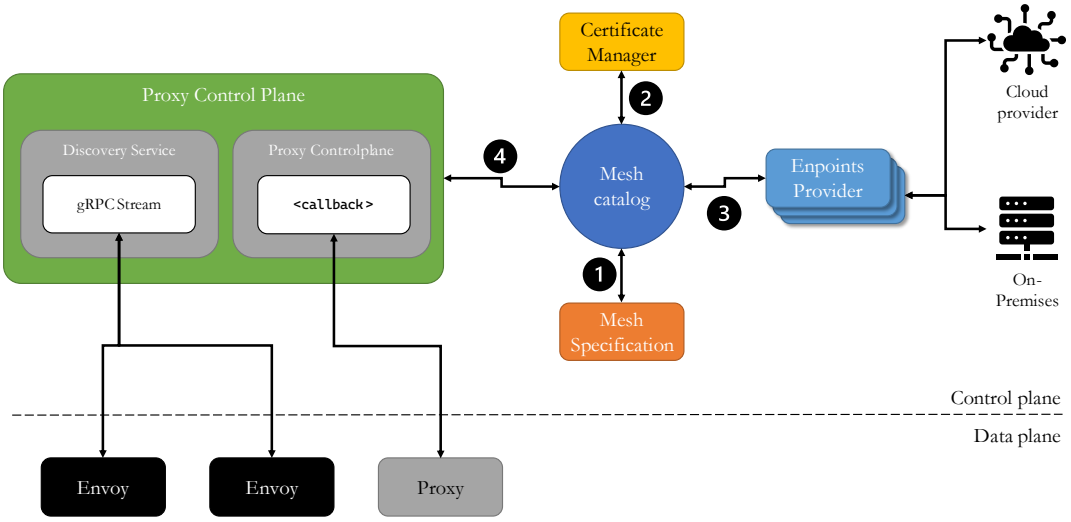


Figure 3-13. OSM control plane architecture with the scheme of its interaction with the data plane.

The Figure 3-13 shows not only the OSM architecture, but also how the control plane components interact to communicate with the data plane:

1. The Mesh Catalog communicates with the mesh specification module (4) to detect when a Kubernetes service is created, changed, or deleted via SMI Specification.
2. Then it reaches out to the Certificate Manager and requests a new TLS certificate for the newly discovered service.
3. It retrieves the IP addresses of the mesh workloads by observing the compute platforms via the Endpoints Providers.
4. Combines the outputs of points 1, 2, and 3 above into a data structure, which is then passed to the Proxy Control Plane (1), serialized and sent to all relevant connected proxies.

3.5.2.1 TELEMETRY

Open Service Mesh comes with the observability stack by default, including Prometheus for metrics collection, Grafana for metrics visualization, Jaeger for tracing, and Fluent Bit for log forwarding to a user-defined endpoints.

Each sidecar proxy collects information for incoming and outgoing traffic, errors, and response timing for requests. Periodically, Prometheus directly gathers and stores consistent traffic metrics and statistics for all applications running in the mesh by scraping over the proxies. Then, the collected metrics can be viewed with Grafana, which uses Prometheus as backend timeseries database.

3.6 COMPARISON OF SERVICE MESHES

Finished the analysis of the single service meshes, we want now to summarize in a table the characteristics of each of the implementations, in order to offer an overall picture that can immediately show the differences and similarities between the different technologies.

	Istio	***Linkerd***	***Consul***	***NGINX***	***OSM***
Legal					
Founder(s)	Google, IBM, Lyft	Buoyant	HashiCorp	F5	Microsoft
Open source	✓	✓	✓ - Community version only	✕	✓
License	Apache License 2.0	Apache License 2.0	Mozilla Public License 2.0	Proprietary	Apache License 2.0
Hosted by	OUC	CNCF	HashiCorp	F5	CNCF
Architecture					
Control Plane	Monolithic - Single binary	Multiple components	Agent - Single binary	Multiple components	Multiple components
Data Plane	Custom Envoy proxy	*Ad-hoc* micro-proxy	Envoy	Agent + NGINX Plus	Envoy
Deployment	Per-pod	Per-pod	Per-node	Per-pod	Per-pod
DP extensibility	✓ - multiple languages, but WASM compatible	✕	✕	✓	✓ - multiple languages, but WASM compatible
Ingress-egress	✓ - like any other proxy, HTTP/TCP	✕	✓	✓	✓ - Third-party
Service discovery	Third-party (default Kubernetes)	Third-party (default Kubernetes)	✓	✓ - by NATS	Third-party (default Kubernetes)
Service catalog	Kept internal, platform-agnostic	NA	✓	✓	Kept internal, platform-agnostic
CP main language	Go	Go	Go	NA	Go
DP main language	C++	Rust	C++	NA	C++
Control plane – data plane communication					
Static configuration	✓	✓ - automatic	✓	✓	✓
Dynamic configuration	✓	✓ - automatic	~	✓	✓
Interaction system	xDS API through gRPC protocol, central istiod calls proxies	NA	xDS API through gRPC protocol	pub-sub through NATS	xDS API through gRPC protocol, CP Proxy calls proxies

	Istio	*Linkerd*	*Consul*	*NGINX*	*OSM*
Observability					
Prometheus compatible	✓	✓	✓	✓	✓
Works with Grafana	✓	✓	✓	✓	✓
Works with Kiali	✓	✓ - through SMI	✕	NA	✕
Works with Jaeger	✓	✓	✕	✓	✓
Monitoring	Envoy prepares data, Prometheus directly scrapes proxies	Linkerd2-proxy prepares data, Prometheus directly scrapes proxies	Client aggregate data, then it streams to the metrics server	NGINX Plus prepares data, Prometheus directly scrapes proxies	Envoy prepares data, Prometheus directly scrapes proxies
Supported protocols					
TCP	✓	✓	✓	✓	✓
UDP	✓	✕	✕	✓	✕
HTTP/1	✓	✓	✓	✓	✓
HTTP/2	✓	✓	✓	✓	✓
HTTP/3	✕ - but experimental in Envoy	✕	✕ - but experimental in Envoy	✕	✕
gRPC	✓	✓	✓	✓	✓
QUIC	✕ - but experimental in Envoy	✕	✕ - but experimental in Envoy	✕	✕
Traffic management					
Blue/Green Deployments	✓	✓	✓	✓	✓
Circuit Breaking	✓	✕	~[14]	✓	✓
Fault Injection	✓	✓	✓	✓	✓
Rate Limiting	✓	✕	✓	✓	NA
Complex scenario deployment					
Multi-cluster support	✓	✓	✓	NA	✕
Multiple network support	✓	✓	✓	NA	NA

[14] Circuit-breaker for Envoy is currently supported, but only Consul 1.10 (currently in beta) supports custom configurations beyond the default one.

	Istio	*Linkerd*	*Consul*	*NGINX*	*OSM*
Multiple control planes	✓	✓	✓	NA	✕
Multiple mesh support	✓	✕	✓	NA	✕
Multiple cloud support	✓	✕	✓	✓	NA
Hybrid cloud support	✓	✓	✓	✓	NA
Multiple zone support	✓	✓	✓	✓	NA
External CP deployment	✓	✕	✓	✕	✕
Supported workloads	Kubernetes & VM	Kubernetes only	Kubernetes, VM, bare-metal	Kubernetes, VM, bare-metal	Kubernetes
Supported Hardware architectures					
x86	✓	✓	✓	✓	✓
amd64	✓	✓	✓	✓	✓
arm	⚠	✓	✓	NA	✕
arm64	⚠	✓	✓	NA	✕
power	NA	✕	✕	✕	✕
Security					
Traffic encryption	mTLS	mTLS	mTLS	mTLS	mTLS
Certificate Management	✓	✓	✓	✓	✓
Installation					
Deployment	via `istioctl`, Operator and Helm	via Helm and `linkerd` command	via Helm	via Helm or `nginx-meshctl`	via Helm or OSM CLI
Namespace(s)	Multiple	Single	Multiple	Multiple	Multiple
Observability plane installed by default	✕	✕	✓	✓	✓

Table 3-1. Qualitative summary of service meshes. ✓*: present;* ✕*: missing;* ⚠*: experimental;* ⁓*: partial;* NA*: data not available.*

As it is evident, *istio*, that is widely adopted, is an open source, mature, and feature-rich service mesh implementation, capable of offering many features for the more advanced business scenarios. In fact, it has support for all the major communication protocols, for all the traffic management options, and it supports complex deployments, such as multi-cluster or multi-cloud.

Linkerd has as its main strength a simple architecture which certainly has important advantages in terms of speed, safety, and simpler configuration, but which can reveal at the same time to be a limit itself. In fact, it does not allow extensions to the data plane, but only at the control plane level; furthermore, as stated in the documentation [89], for simplicity reasons Linkerd does not provide an ingress controller, but it can use third-party solutions such as Ambassador, Kong, or NGINX. Finally, it must be considered that having been recently refactored, some advanced features are missing. In fact, actually its proxy implementation does not support the UDP protocol and some traffic management options, such as circuit breaking and rate limiting, while it supports, as a control plane extension, only a multi-cluster option which unlocks also the hybrid-cloud possibility [90].

The *Consul* service mesh solution makes no assumptions about the underlying network and uses a pure software approach with a focus on simplicity and broad compatibility. It works in many different complex scenarios, such as multi-cluster or multi-cloud, with a different architecture approach with respect to the ones previously analyzed, that deploys agents per-node instead of per-pod, and it is organized into Data Centers allowing services inside a Kubernetes cluster to communicate with services located outside, on virtual machines or bare-metal.

NGINX Service Mesh is the only completely proprietary solution analyzed, based on NGINX Plus as an alternative to Envoy proxy, with an interesting feature related to the service discovery that is built using NATS, allowing a publish-subscribe interaction between the control and the data planes.

Finally, *Open Service Mesh*, that we have chosen to analyze because it is the latest technology in the service mesh panorama, is also the most limited. In fact, while it supports many of the key features, it still does not support many of the more advanced business scenarios, such as multi-cloud.

3.7 APACHE EVENTMESH

Turning now to the analysis of an event mesh solution, as we have already written in the introduction of this chapter, we will examine the Apache EventMesh project which

is an open-source implementation of a dynamic event-driven application runtime that distributes events across different environments, decoupling the application and the back-end middleware layer. Born at WeBank, a private chinese neobank, with help from Oppo, Huawei, and the OpenMessaging community, the objective of this project is to build an infrastructure layer that can support a wide range of use cases, from complex multi-cloud deployments to on-premises, using diverse technology stacks [91].

The nature of Apache EventMesh allows to implement the data mesh paradigm offering the possibility to build a completely decentralized architecture, with multiple clusters available on different environments, and allowing customers to independently consume data as events, that can be described in a standard format, such as CloudEvent, simplifying the consumption of messages. To help in separating the organization domains, events are organized into *topics* that are managed by the underlying back-end messaging engines, such as Apache RocketMQ, with guarantees of delivery.

Apache EventMesh is an open-source project, licensed under the Apache License 2.0, with an active community of developers. Actually, it is in early stages, with many features that need to be implemented yet, but the central components are completed and they can be sufficient to build an event mesh, locally or in the cloud. We want now to take a look to its architecture and to the messaging engine it actually relies on.

3.7.1 ARCHITECTURE

Apache EventMesh has a simple architecture made of the following primary components [92]:

- Runtime;
- Connector API;
- SDK.

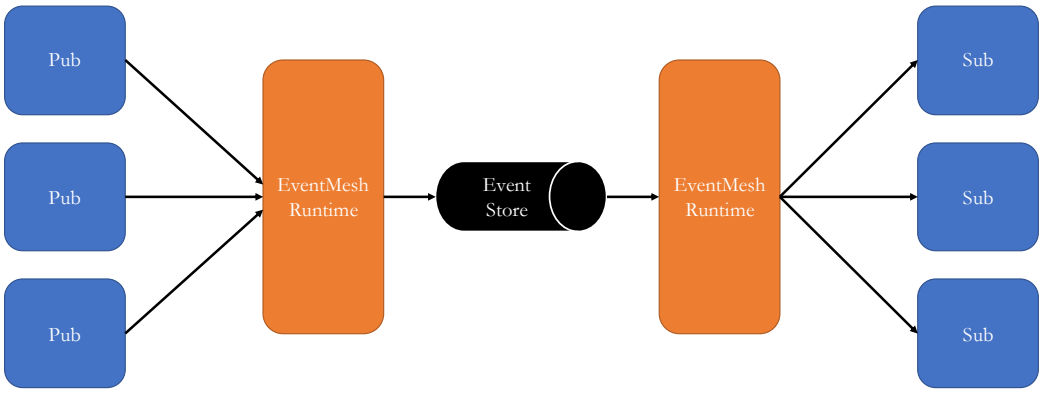


Figure 3-14. Apache EventMesh general architecture.

At the core of Apache EventMesh there is the *Runtime* component which is the middleware that manages the communication between producers and consumers through the use of a messaging engine, such as Apache RocketMQ. This component operates as a standalone entity and it can be deployed as a microservice in the cloud or it can run on a local machine. The Runtime server receives messages that the applications prepare by using the SDK component and then it initiates the communication that, depending on the protocol, can generally be asynchronous, broadcast, or even synchronous [93].

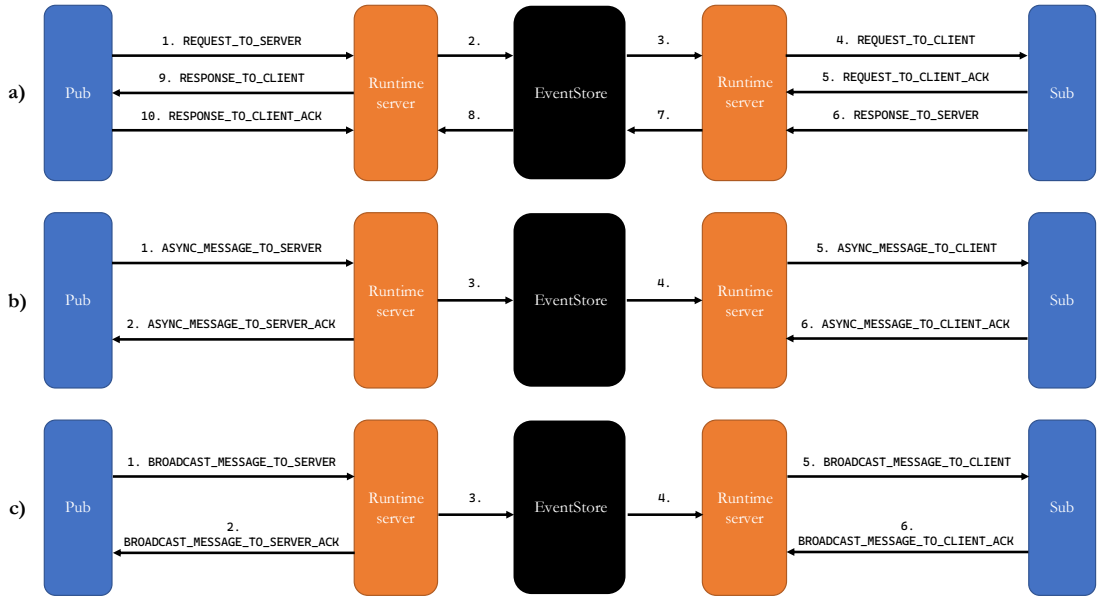


Figure 3-15. Interaction schemes for TCP protocol: synchronous message (a), asynchronous message (b), broadcast to multiple subscribers (c). Publishers and subscrbers communicate, through the SDK, with the Runtime server which forwards and receives messages to and from the EventStore (Apache RocketMQ).

Applications that want to participate in the communication must create a client through the *SDK* that allows to prepare messages and to forward them to the Runtime server, which can be located on the same machine or it can be in a different node. The SDK supports TCP, HTTP, and gRPC protocols and it can handle both native EventMesh messages or CloudEvents. This last option is a specification for describing event data in a standard format, providing interoperability across services, platforms, and systems.

One of the strong points of Apache EventMesh is that it implements just the event mesh logic, leaving the event storing and exchanging details to the underlying platforms that are pluggable through the *connector API*, an api layer based on the Service Provider Interface (SPI) mechanism, which allows to automatically find and load concrete implementation classes of the extended interfaces at runtime [94]. This allows Apache EventMesh to maintain a simple architecture, adding new and advanced features through plugins. Actually, the only back-end messaging engine supported is Apache RocketMQ that we now want to analyze.

3.7.2 APACHE ROCKETMQ

Apache RocketMQ is a distributed messaging and streaming platform, which the developers claim guarantees low latency and reliability. It has been used in production for many years and it has been chosen by EventMesh to be the default back-end messaging engine.

Apache RocketMQ consists of the following components [95]:

- Name server;
- Broker;
- Producer;
- Consumer.

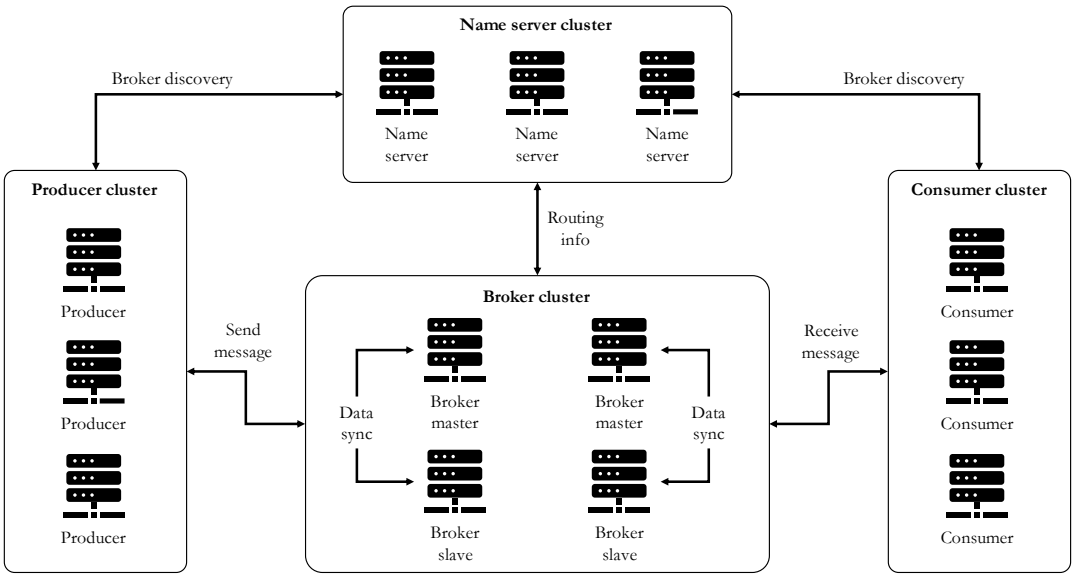


Figure 3-16. Apache RocketMQ general architecture with each component replicated.

Producers and *consumers* are the actors sending and receiving event messages that RocketMQ organizes into *topics* [96]. The latter have very loose relationship with producers and consumers: a topic can have zero, one, or multiple producers that send messages and it may also be subscribed by zero, one, or multiple consumer groups.

The *name server* is the routing information provider that allows lightweight service discovery inside the RocketMQ cluster, coordinating each component (producers, consumers, and brokers) of the distributed system. It manages brokers, accepting their registration when new brokers are introduced, removing them when they are crashed or deleted, and syncing routing information between them. In addition, the name server indicates to producers and consumers which broker they can communicate with in order to allow publishing and consuming events [97].

Brokers are servers mainly responsible for message storing and delivery and for these reasons it is very important to achieve high availability and strong fault tolerance guarantees. With respect to the first need, brokers can be organized into *master-slave groups*, in which the slave is a read-only server that does not accept messages from the publishers. Each group can be replicated to enhance high availability. With respect to fault tolerance, each broker can be configured to save event data on the disk, reducing the performance by a little. Otherwise, to maximize brokers performance, they can be configured to avoid storing events on disk, but in case of failure messages are lost [98].

Each broker server is composed of several sub-modules:

- ***Remoting module*** is the entry of the broker and it handles the requests from clients.
- ***Client manager***, as the name suggests, manages the producers and consumers, maintaining, for example, topic subscription of consumers.
- ***Store service*** provides simple API to store or query message in physical disk.
- ***High Availability service*** provides data sync feature between the master and the slave brokers.
- ***Index service*** builds an index for messages by specified key and provides quick message delivery.

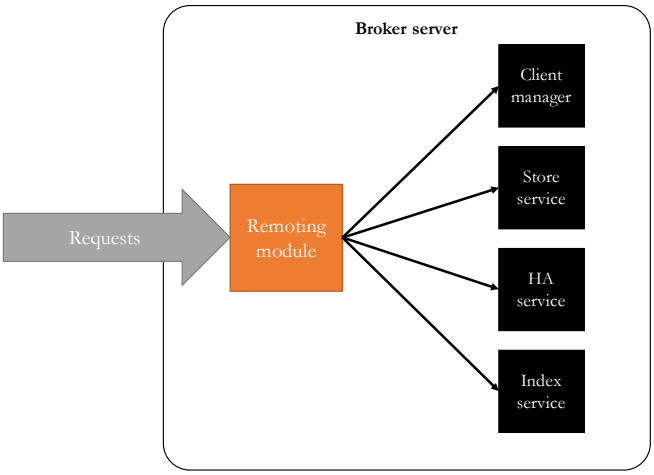


Figure 3-17. Inner working of an Apache RocketMQ broker server.

3.7.3 CONCLUSION

In conclusion, we can say that, in the event mesh panorama, that is recently emerging[15], Apache EventMesh is the only *open source* modern solution that supports standard messages through CloudEvent format, a pluggable messaging engine, with Apache RocketMQ as the default back-end [91]. This last feature allows EventMesh to keep its own architecture simple, concentrating on realizing the mesh logic. Finally, Apache EventMesh, although in early development stages, it can be already used in different scenarios such as inside a Kubernetes cluster as we will do in our project and describe in subsequent chapters.

[15] Actually, the main competitor is Solace PubSub+ that can create an event mesh. However, it is not open source and it has limited access.

4 PROJECT OVERVIEW

The project that has been decided to carry out has the purpose of testing the performance of a workflow microservices-based application in a cloud environment through the use of different mesh technologies.

The project focuses, as we will explain in detail in this and in the next chapters, on the infrastructure layer, looking at the differences between diverse service meshes and how they compare in performance with an event mesh implementation. The applicative part consists of a series of echo applications that simulate a generic workflow system.

4.1 OBJECTIVES

Microservices based architectures, as we already discussed about, offer many benefits with respect to traditional monolithic application development, because they allow to achieve a huge degree of *flexibility* in choosing technology, handling robustness and scaling, organizing teams, and more. Precisely this flexibility is, in part, why many organizations, especially larger ones, are adopting microservices, thus demonstrating how these can be effective [22].

However, microservices bring with them a significant degree of *complexity* that needs to be correctly managed. In fact, implementing a large system as a collection of many microservices, can simplify the development of the whole application, but the operations can be more complicated: orchestrating a large number of entities into a coherent and complex system cannot be expected to be an easy task.

Think for example of an online store application divided into a store front-end, an order management service, a payment processing service, and a shipment management service, in which these parts are made of multiple microservices; these must be highly available and must handle high workloads, scaling up when necessary and finally must communicate reliably and efficently, sometimes across different environments: from on-premises to cloud.

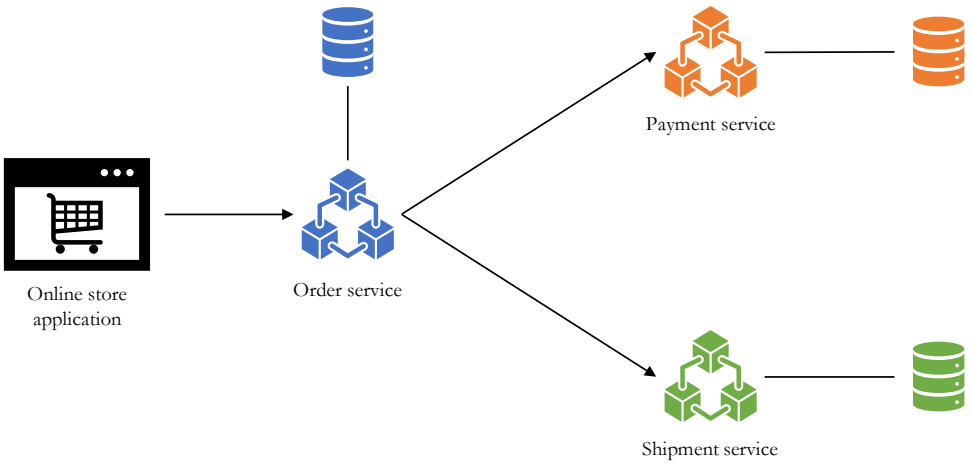


Figure 4-1. An example of a store application with the different services.

So even for microservices seems to be valid what Frederick P. Brooks wrote in the article "No Silver Bullet" [99]:

> *There is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement in productivity, in reliability, in simplicity.*

According to Brooks, the reason for this phenomenon is the presence of *essential complexity* in software systems. In fact, he points out that, while in any codebase there is always some *accidental complexity*, the one related to our own implementation choices, most of the complexity in designing and implementing software systems is not accidental, but, instead, it is related to the very essence of the complexity of modeling the problem domain itself so that if we try to simplify a system beyond its essential complexity, we would take away from its core model, and it would no longer be the same system.

Regarding the large distributed systems, such as microservices-based architectures, we can therefore affirm that it is impossible to eliminate the essential complexity. However, it must be considered that it is possible to *shift* the complexity from one area

of the system to another and this can be particularly beneficial when it is possible to *automate* the part that we are making "harder".

In other words, we can simplify the code by breaking it up into multiple microservices, but it might seem like we have not gained so much as we complicate the operational part. In reality, the increased complexity of operations matters much less if they can be automated: shifting complexity from the area we cannot, yet, automate, design and code, into the area we have the possibility to automate, the operations, constitutes a substantial advantage. For this reason, it was pointed out that a microservices architecture can be *materially simpler* than its alternatives when implementing complex systems [18].

In this scenario, many technologies emerged in the past years to deal with the complexity of operations, such as Docker and Kubernetes, but also service and event mesh. In our work we decided, precisely because of their relevance also in the perspective we have talked about, to focus mainly on the infrastructural components, in particular, testing service and event meshes, to see how the different system parts interact when using synchronous and asynchronous communications and with which performance.

4.2 TECHNOLOGIES CHOSEN

Passing now to the analysis of the technologies used in our project, it must first be noted that a complex microservices-based system can in principle profit greatly from using common protocols and data formats, or from choosing in general a certain homogeneity in the technologies. However, as pointed out by Carneiro and Schmelmer in "Microservices From Day One" about polyglot services[16], "the majority of services in a microservice environment expose many areas that together make up a multidimensional vector for potential improvements" [20]; therefore, apply the uniformity of programming languages, data stores, or other technology components

[16] Polyglot services allow developers to select the programming language of their choice in order to capture additional functionality and efficiency not available in a single language. Numerous companies evolved to support a polyglot microservice architecture, like Google, eBay, Twitter, and Amazon, prooving that this approach enhances developer creativity and out-of-the-box problem solving [100], [101].

across all the services might not be the best choice. There are many reasons why to select a technology with respect to another: different technologies can offer different performances, security features, protocol support, deployment options, ease of configurability and maintanability, and it is important to study which could be the right technology from which we can take advantage for our specific application purpose, before we move to it.

For this reason, we selected, for our technological choices, the following guiding principles that we will illustrate in this paragraph:

- Reproducibility;
- Cost and license;
- Resource usage.

At first, it is necessary to provide the ability to *reproduce* the software tests, allowing anyone, before choosing the technology, to reproduce them in the same environment, with the same tools and conditions, to verify the results. It is also important to take into account *costs and licenses*: an open-source tool allows us to carry out tests using a technology that is openly available and very often widely adopted in the industry; moreover, this type of license generally allows the distribution, visualization, and sharing of the source code, also to the advantage of reproducibility. Finally, the applications created must work well even on relatively powerful machines and must consume the *least possible amount of resources*, to be cheaper and ensure better exploitation of the cluster. Furthermore, in the specific case of containers, whose images require to be archived in a registry, it is advantageous that these are of a smaller size both because in this way it is easier to upload, download, create and archive them, and because the optimization of these factors also leads to a reduction in costs.

In consideration of these guiding principles, we have decided to use in the range of service mesh solutions Istio and Linkerd. The first is a complete, open-source, and quite popular implementation; the second is equally open-source, stable and provides a different data plane from the one that istio and the other solutions we have analyzed in the previous chapters offer.

On the other hand, in the field of event mesh, we have chosen to use Apache EventMesh, an open-source technology part of a foundation of great importance such as the Apache Foundation, which, being still in incubation, we wanted to test to verify its actual potential.

4.3 EVOLUTION OF THE PROJECT STRUCTURE

At this point, before moving on to the description of the project that we will carry out in the next chapter, we believe it is important to briefly illustrate the most significant passages through which the final structure of our work was articulated.

We want to start from the studies we conducted to learn how service meshes work and how to implement a simple testing environment. Being focused on istio and Linkerd, we decided at first to run benchmarks to load-test each different technology using *Fortio*[17], an open-source tool from the istio community with a simple *client-server* architecture and specifically designed to run benchmarks. This tool packages also a *report server* which allowed us to generate graphs from statistics made persistent by the client.

[17] Fortio was chosen with respect to other tools, like locust.io, because it has complete protocols support (HTTP, HTTP/2, GRPC, and TLS) without requiring complex configurations. Locust has experimental GRPC support and statistics are less complete with respect to the ones created by Fortio, but Locust could be more suitable in a real scenario in which a business application is deployed. In fact, it allows to create ad-hoc tests for a specific application with finer control than Fortio.

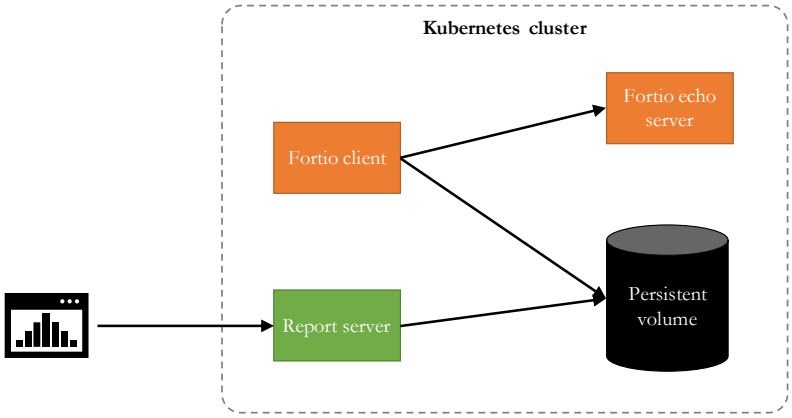


Figure 4-2. Fortio architecture with browser accessing the reports generated by the relative server and the volume used to persist benchmark data.

The Fortio application was containerized using Docker and its distribution was supported by a Kubernetes infrastructure: at first it was sufficient conducting tests on minikube to verify the correct functioning and configuration, then we moved to cloud solutions to enhance our test environment and proceed with more advanced developments. In particular, we were interested in an topic that is increasingly important in cloud infrastructures that is the *observability*[18], that refers to a characteristic of a system that is measured by the level to which it is possible to understand and reason about a system internal state just by looking at its external signals and characteristics [103]. In that perspective, the statistics just provided by Fortio were no more sufficient and we decided to adopt more advanced solutions that could give us the possibility to collect and visualize not only the application statistics, but also any information that Kubernetes could expose: Prometheus and Grafana.

Subsequently, after carrying out the tests for the service meshes in the cloud environment, we focused on the study of Apache EventMesh which, as already explained above, relies on Apache RocketMQ to realize the underlying event-driven architecture. We then configured and deployed RocketMQ first and the Apache EventMesh server afterwards. Finally, to verify the correct functioning of the system,

[18] This definition is based on the study of control theory first introduced in the 1959 paper from "On the General Theory of Control Systems" by Rudolf E. Kalman [102].

we created a workflow application adapted to the event case, an application which was then used and extended in subsequent comparative tests. This application was created using Java, because it actually[19] is the only language supported by Apache EventMesh. To keep uniformity through tests we decided to build in Java also the application to be used to test service meshes.

At this point, as a further step, in order to improve our metrics detection tools, with particular reference to the need to make data persistent, we decided to replace Prometheus and Grafana with our own system, compatible with the Prometheus data format. The system we realized is written in Python and it is capable not only of collecting data directly from Kubernetes and to directly scrape metrics from infrastructural entities deployed in the cluster, such as Apache EventMesh server, RocketMQ components, and service mesh proxies, but it is also capable of synthetize statistics directly from the workflow application components, transforming, when needed, these data in Prometheus format. It must be noted finally that our system can work in a multi-cluster environment, that we decided to realize to simulate a real business scenario, by deploying in each cluster a server controlled by a client that runs in the local environment.

[19] The reference is to May 2022; the developers foresee Go and Rust as the next supported languages.

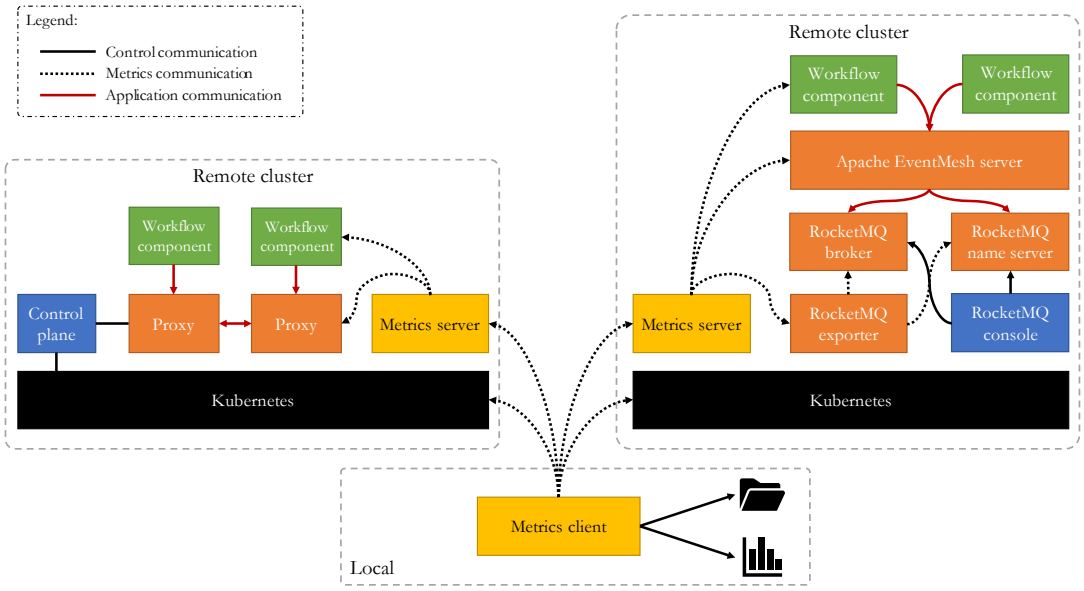


Figure 4-3. Overview of our project architecture: on the left the service mesh deployment and on the right the event mesh one. The local metrics client gathers data directly from the Kubernetes APIs and from each component through a metrics server deployed in each cluster.

As we already wrote, for the realization of microservices components, we used Java and Python and for this reason, according to the guiding principle, stated previously, of keeping resource usage low, we analyzed the sizes of the container images for these languages.

Tag	*Operating System*	*Size (MB)*
openjdk:8-alpine	Alpine 3.9	~ 105
openjdk-8	Debian 11	~ 526
python:3-alpine	Alpine 3.16	~ 47,6
python:3	Debian 11	~ 920
centos:7	CentOS 7.9.2009	~ 204

Table 4-1. Size of the container images available on Docker Hub[20].

As we can see from the Table 4-1, the images using Debian are bigger than the ones that use Alpine, so we decided to use this OS not only for the workflow applications

[20] Data updated to June 2022.

and the metrics server, but also for the components of the event mesh infrastructure, whose official images are based on CentOS.

5 PROJECT IMPLEMENTATION

In this chapter is described the realization of the workflow applications as well as of the metrics system and the infrastructural components of our project. We will start with the latter and then proceed in order with the workflow application and the metrics system, analyzing both single and multi-cluster deployments.

5.1 ISTIO

Istio, as stated earlier, is supported in multiple environments like Kubernetes and Nomad. We will now discuss about our istio setup restricting to Kubernetes.

5.1.1 SINGLE CLUSTER INSTALLATION

There are many ways of installing istio and we chose to do it by using `istioctl`, a command-line tool which provides rich customization of the control plane and of the sidecars for the data plane. With respect to other solutions, this tool does not only allow for controlling the istio installation, but also it has user input validation to help preventing installation errors and customization options to override any aspect of the configuration. `istioctl` supports the full IstioOperator API[21] via command-line options for individual settings or for passing a YAML file containing an IstioOperator custom resource (CR) and it can be installed through the following command:

```
$ curl -L https://istio.io/downloadIstio | sh -
```

5.1.1.1 CONFIGURATION OPTIONS

`istioctl` allows to install istio according to a configuration, expressed as a YAML file, that is termed *profile*. `istioctl` comes with some predefined profiles that is possible to visualize through the following command:

[21] Up to Istio 1.4, Helm was used as the primary tool for installing and upgrade istio. Version 1.4 of Istio started deprecating it and introduced istioctl with IstioOperator API, a Kubernetes operator, which provides a pattern to easly configure planes installation version and shape [104].

```
$ istioctl profile list
Istio configuration profiles:
    default
    demo
    empty
    external
    minimal
    openshift
    preview
    remote
```

As we can see from the previous output, there are several built-in options available. To choose the right one it is needed to look at the internal configuration each of those provides by using the sub-command `dump`. In addition, it is possible to compare two different profiles with the sub-command `diff`:

```
$ istioctl profile dump default
$ istioctl profile diff default demo > compare.yaml
```

In the previous code, with the first command it is possible to visualize the YAML file with all the configurations for the default profile; with the second command a file containing all the differences between the two profiles is generated.

We want now to describe each built-in profiles more in detail:

- ***Default*** deploys istiod control plane, adds an ingress gateway, and enables proxy auto-injection when namespaces or pods are labeled accordingly. Each pod in the data plane requires at least 128MiB and 100mcpu and the status is exposed on port 15020. As stated in the istio documentation [105], this profile is recommended for production deployments and for primary clusters in a multi-cluster mesh. In fact, it has the same settings as the *remote* profile.
- ***Demo*** is designed to test many features istio provides requiring just few resources. It enables not only istiod and the ingress gateway, but it also provides an egress gateway, disables autoscaling for ingress-egress and proxy requires at least 40MiB of memory and 10mcpu. Since it enables high levels of tracing and access logging, this profile is not suitable for performance tests.
- ***Minimal*** is designed from the *default* profile with the only difference of deploying just istiod control plane, without the ingress gateway.

- ***External*** can be used for configuring a remote cluster that is managed by an external control plane or by a control plane in a primary cluster of a multi-cluster mesh.
- ***Empty***, as the name suggests, deploys nothing and therefore it is useful as a base for custom profile configuration.
- ***Preview*** is used to showcase istio experimental features and for this reason stability, security, and performance are not guaranteed.
- ***Openshift*** is an adaptation of the *default* profile to be used with RedHat Openshift.

Among the options listed above, we have chosen to use the *default profile* in our project because, as already mentioned, it provides the possibility of preparing a production-ready configuration for both the single cluster and the multi-cluster case.

After choosing the profile, it is recommended to save the manifest data that `istioctl` uses to express all the details of the installation; so, it will be possible not only to inspect what exactly is being installed, but also to track changes to the manifest over time.

```
$ istioctl manifest generate --set profile="$ISTIO_PROFILE" > istio-manifest-
"$ISTIO_PROFILE".yaml
```

Code 5-1. Storing the istio manifest in a file. The `ISTIO_PROFILE` variable is expanded to `default`.

The last step before the installation required to verify that any prerequisites have been met in our Kubernetes cluster. Then we proceeded with the istio installation on the single cluster followed by the verification of the success of the operation by comparing it with the manifest previously generated and by checking if the cluster deployments are running:

```
$ istioctl x precheck
$ istioctl install --set profile="$ISTIO_PROFILE" -y
$ istioctl verify-install -f istio-manifest-"$ISTIO_PROFILE"
$ kubectl -n istio-system get deploy
NAME                   READY   UP-TO-DATE   AVAILABLE   AGE
istio-ingressgateway   1/1     1            1           90s
istiod                 1/1     1            1           2m6s
```

5.1.2 MULTI-CLUSTER INSTALLATION

As we have already anticipated in Chapter 3, istio is not bound to a single cluster, but it can span many clusters and provide the same capabilities across all of them. We decided to run our tests also in this scenario because many companies choose to adopt it [103]. The benefits it provides are:

- ***Improved isolation***: separated clusters can implement different business scopes while cooperating, enhancing isolation between domains.
- ***Failure boundaries***: errors and faults of configurations and operations for a cluster do not affect the others.
- ***Regulatory and compliance***: services that access sensitive data from other parts of the architecture can be restricted.
- ***Increased availability and performance***: one cluster can take over another in case of a failure, regional or not, or it is possible to route traffic to the closest cluster to reduce latency.
- ***Multi and hybrid cloud***: workloads can run in different environments, whether different cloud providers or hybrid clouds.

The multi-cluster implementation connects services across clusters in a way that is transparent to the apps, meanwhile maintaining all of the service mesh capabilities: fine-grained traffic management, resiliency, observability, and security for cross-cluster communication. To do that istio requires the following criteria:

- ***Cross-cluster workload discovery***: the API Server on each cluster must be accessible to others because the control plane uses it to discover the workloads in the peer clusters in order to configure the service proxies.
- ***Cross-cluster workload connectivity***: workloads must have connectivity between each other.
- ***Common trust between clusters***: cross-cluster workloads must mutually authenticate to enable the security features istio provides.

5.1.2.1 ISTIO MULTI-CLUSTER DEPLOYMENT MODELS

When installing as a multi-cluster, istio distingushes between *primary clusters*, which are the ones containing the istiod control plane, and *remote clusters*, which allow only the deployment of the data plane. The combinations of these two types of clusters define the models istio allows:

- ***multi-primary***: the control plane is installed on each cluster involved in forming the mesh.
- ***primary-remote***: the control plane, installed only on one cluster, observes the API Servers on each cluster for endpoints, providing in such way service discovery for workloads in each cluster involved in forming the mesh.

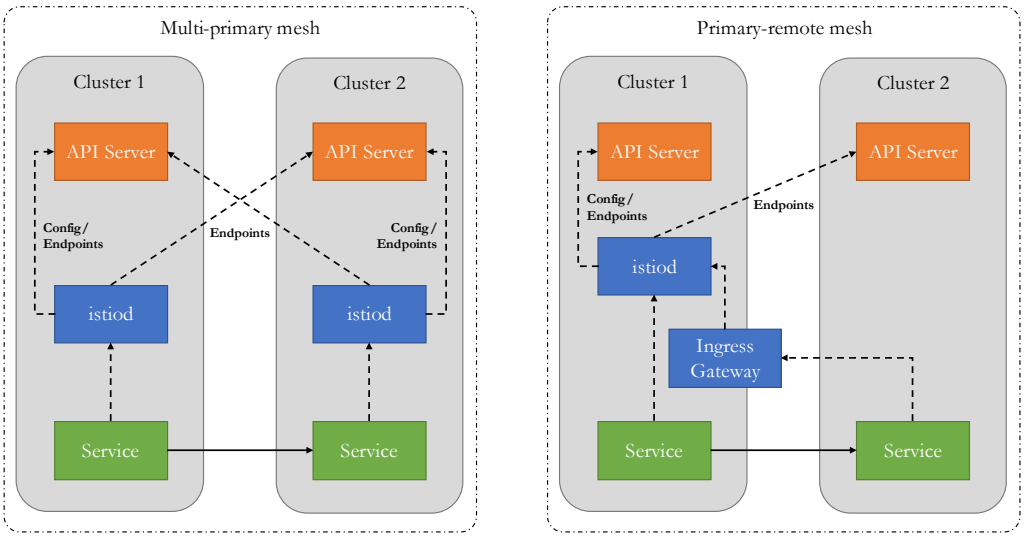


Figure 5-1. Difference between multi-primary and primary-remote control plane installations. In the first case each control plane observes the API Servers in both clusters for enpoints. In the second case the istiod in cluster 1 observes the API Servers in both clusters, while the services in cluster 2 reach the control plane in cluster 1 via an ingress gateway for east-west traffic. In this figure, both clusters reside on the same network, meaning workloads communicate directly (pod-to-pod) across cluster boundaries.

These deployment models differ mainly on two aspects: *availability* and *performance*. The primary-remote option has a single control plane managing the mesh and for this reason, it uses less resources on the remote clusters, but an outage in the primary one affects the entire mesh, undermining availability. On the other hand, the multi-primary

model ensures higher availability by replicating the control plane, but at the same time, it requires more resources.

Choosing a deployment model with respect to the other depends on the business needs, but in general it is possible to affirm that nowadays availability can be considered more important with respect to the performance criterion. For example, if we think to an online store, such as the one cited in the previous chapter, it must be always available to customers because every minute of it being down would cost the business a lot. Hence, high availability is important and we chose this last option to prepare the multi-cluster for our tests.

The profiles used to install istio on two clusters are the following:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  values:
    global:
      meshID: mesh1
      multiCluster:
        clusterName: cluster1
      network: network1
```

Code 5-2. YAML configuration for the first istio cluster.

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  values:
    global:
      meshID: mesh1
      multiCluster:
        clusterName: cluster2
      network: network1
```

Code 5-3. YAML configuration for the second istio cluster.

The previous code was used by `istioctl` to override some parameters in the default profile, before the installation process starts[22]:

[22] It is possible to verify the substitutions by inspecting the generated profiles using the `istioctl profile dump` command as in the single cluster case.

```
$ istioctl install --context="$CLUSTER1" -f cluster1.yaml
$ istioctl install --context="$CLUSTER2" -f cluster2.yaml
```

Once the istio control plane has been installed, it is needed, to permit it, to access to the Kubernetes API server of the remote cluster in order to discover cross-cluster workloads. To do that we needed to create remote secrets:

```
$ istioctl x create-remote-secret \
    --context="${CLUSTER1}" \
    --name=cluster1 | \
    kubectl apply -f - --context="${CLUSTER2}"
```

Code 5-4. Installation of a remote secret in cluster2 that provides access to API server in cluster1.

```
$ istioctl x create-remote-secret \
    --context="${CLUSTER2}" \
    --name=cluster2 | \
    kubectl apply -f - --context="${CLUSTER1}"
```

Code 5-5. Installation of a remote secret in cluster1 that provides access to API server in cluster2.

5.1.3 SIDECAR INJECTION

After the installation, in order to use the service mesh it is necessary to instruct istio to insert the sidecars in the data plane and it is possible to do this in two different ways. In the first case, labeling a namespace instructs the istiod injector to automatically insert a sidecar when a new pod resource is added in the target namespace; in the second case it is possible to manually indicate to the control plane to inject the sidecar by labeling the deployment when passing the YAML configuration to `kubectl`.

In our scripts we can use both methods, but in the final tests we preferred the last option; in fact, we created a Python script which can inject the correct metadata for both istio and Linkerd, allowing us to keep the deployment code cleaner:

```
$ ./inject.py deployment.yaml istio | kubectl -n $NAMESPACE apply -f -
```

`inject.py` uses pyYAML to read the deployment file, then adding to it the label `spec.template.metadata.labels:sidecar.istio.io/inject:true`.

5.2 LINKERD

Passing now to our Linkerd setup, we must preliminarily emphasize that, unlike istio, Linkerd is able to work only on Kubernetes, not supporting other platforms.

5.2.1 SINGLE CLUSTER INSTALLATION

Regarding the single cluster setup, the first step is necessarily to install the Linkerd command line interface, a tool that works similarly to `istioctl`, allowing the connection to the service mesh control plane:

```
$ curl --proto '=https' --tlsv1.2 -sSfL https://run.linkerd.io/install | sh
```

As we saw for istio, it is a good practise to run checks before proceeding with the installation of the control plane so to validate if the cluster does not have some incompatibilities or incorrect configurations:

```
$ linkerd check --pre
```

After the validation of the Kubernetes cluster, it is possible to proceed with the installation of the control plane of Linkerd and then check for deployment readiness and health:

```
$ linkerd install | kubectl apply -f -
$ linkerd check
```

The `linkerd install` command generates a Kubernetes manifest containing all the control plane resources to be applied. By default, the Linkerd control plane is installed in the `linkerd` namespace, but if needed it can be changed by passing the argument `--linkerd-namespace`.

The `linkerd check` command will verify any mismatch in the Kubernetes version, the ability to connect to the API server, and other aspects. It will wait for the Linkerd control plane pods to be available and for this reason it may take a long time to complete. At any point after the install, it is possible to run `linkerd check config` to

ensure that all the necessary resources of the control plane are available and correctly configured.

The injection is similar to the one we wrote about istio with the only exception that the injection annotation is:

```
spec.template.metadata.annotations:linkerd.io/inject:enabled
```

5.2.2 MULTI-CLUSTER INSTALLATION

Multi-cluster support in Linkerd requires extra installation and configuration on top of the single cluster installation decribed above. To keep uniformity along our project, we decided to define a multi-primary installation, like we did for the istio setup.

After the installation of the Linkerd control plane on the first cluster, we need to prepare a trust anchor that we use to secure the connection between the two clusters. The trust anchor allows the control plane to encrypt the requests that go between clusters and verify the identity of those requests. This identity is used to control access to clusters, so it is critical that the trust anchor is shared.

For our testing purposes it was sufficient to create a single trust anchor certificate shared between our two clusters with the following code:

```
$ kubectl --context="$CLUSTER1" -n linkerd get cm linkerd-config -
ojsonpath="{.data.values}" | yq e .identityTrustAnchorsPEM - > trustAnchor.crt

$ step certificate create root.linkerd.cluster.local root.crt root.key \
    --profile root-ca --no-password --insecure
$ step certificate create identity.linkerd.cluster.local issuer.crt issuer.key \
    --profile intermediate-ca --not-after 8760h --no-password --insecure \
    --ca root.crt --ca-key root.key

$ cat trustAnchor.crt root.crt > bundle.crt
```

At this point, we used the new bundle to upgrade the existing cluster and to install Linkerd on the new cluster together with the issuer certificate and key:

```
$ linkerd --context="$CLUSTER1" upgrade --identity-trust-anchors-file=./bundle.crt |
kubectl --context="$CLUSTER1" apply -f -

$ linkerd --context="$CLUSTER2" install \
  --identity-trust-anchors-file bundle.crt \
  --identity-issuer-certificate-file issuer.crt \
  --identity-issuer-key-file issuer.key | \
  kubectl --context="$CLUSTER2" apply -f -

$ linkerd --context="$CLUSTER2" check
```

After this step, it is possible to proceed with the installation of the multi-cluster control plane, using the following commands:

```
$ linkerd --context="$CLUSTER1" multicluster install | kubectl --context="$CLUSTER1"
apply -f -
$ linkerd --context="$CLUSTER2" multicluster install | kubectl --context="$CLUSTER2"
apply -f -
```

After the setup of the resources needed for the multi-cluster control plane, it is necessary to link the two clusters. This step consists of installing several resources in the source cluster including a secret containing a kubeconfig that allows access to the target cluster Kubernetes API, a service mirror control for mirroring services, and a Link custom resource for holding configuration. The linking of the two clusters is performed using the following code:

```
linkerd --context="$CLUSTER2" multicluster link --cluster-name "cluster2" | kubectl
--context="$CLUSTER1" apply -f -
linkerd --context="$CLUSTER1" multicluster link --cluster-name "cluster1" | kubectl
--context="$CLUSTER2" apply -f -
```

Finally, to check if the installation was correctly executed, beyond using the `check` command, we listed the gateways for each cluster to see if each of those could see the other:

```
$ linkerd --context="$CLUSTER1" multicluster gateways
CLUSTER   ALIVE    NUM_SVC  LATENCY_P50  LATENCY_P95  LATENCY_P99
cluster2  True           0          2ms          3ms          3ms

$ linkerd --context="$CLUSTER2" multicluster gateways
CLUSTER   ALIVE    NUM_SVC  LATENCY_P50  LATENCY_P95  LATENCY_P99
cluster1  True           0          1ms          3ms          3ms
```

5.3 APACHE ROCKETMQ

Proceeding now to the examination of the Apache RocketMQ setup, it must be noted that this project does not provide any official installation process in a Kubernetes cluster, but there exists a community-driven project, recognized by the Apache Foundation, which maintains an Operator to automatically take care of the installation, upgrade, and runtime maintenance processes. Although it could be considered useful to setup a simple cluster, this tool has many limitations such the fact that it runs only in the default namespace or that there are some scalability issues. Since we wanted greater control over the installation process, we decided to create our own.

5.3.1 CONTAINER IMAGE

The `apache/rocketmq:4.9.3` image from DockerHub is based on CentOS and, as we already mentioned in the previous chapter, we opted for our own image which is lighter and which allowed us to insert customized launch scripts. We describe now the Dockerfile we made.

```
FROM openjdk:8-alpine
ARG user=rocketmq
ARG group=rocketmq
ARG uid=3000
ARG gid=3000

RUN addgroup -g ${gid} ${group} \
 && adduser -u ${uid} -G ${group} -D ${user}
...
USER ${user}
```

Code 5-6. Beginning of the RocketMQ Dockerfile.

Since Apache RocketMQ is a central component for the event mesh, we decided to run it as a user just named `rocketmq`, avoiding the use of `root`. Then we proceeded with the installation of this software, verifying if it was correctly done, and finally, adding our launch scripts.

```
RUN set -eux ; \
    apk --no-cache add curl gnupg ; \
    curl -L https://archive.apache.org/dist/rocketmq/${ROCKETMQ_VERSION}/rocketmq-
all-${ROCKETMQ_VERSION}-bin-release.zip -o rocketmq.zip ; \
    curl -L https://archive.apache.org/dist/rocketmq/${ROCKETMQ_VERSION}/rocketmq-
all-${ROCKETMQ_VERSION}-bin-release.zip.asc -o rocketmq.zip.asc ; \
    curl -L https://www.apache.org/dist/rocketmq/KEYS -o KEYS ; \
    gpg --import KEYS ; \
    gpg --batch --verify rocketmq.zip.asc rocketmq.zip ; \
    unzip rocketmq.zip ; \
          mv rocketmq*/* . ; \
          rmdir rocketmq-*  ; \
          rm rocketmq.zip rocketmq.zip.asc KEYS

COPY scripts/ ${ROCKETMQ_HOME}/bin/

RUN chown ${uid}:${gid} ${ROCKETMQ_HOME}

RUN mv ${ROCKETMQ_HOME}/bin/runserver-customize.sh ${ROCKETMQ_HOME}/bin/runserver.sh
\
 && chmod a+x ${ROCKETMQ_HOME}/bin/runserver.sh \
 && chmod a+x ${ROCKETMQ_HOME}/bin/mqnamesrv ; \
    mv ${ROCKETMQ_HOME}/bin/runbroker-customize.sh ${ROCKETMQ_HOME}/bin/runbroker.sh
\
 && chmod a+x ${ROCKETMQ_HOME}/bin/runbroker.sh \
 && chmod a+x ${ROCKETMQ_HOME}/bin/mqbroker

RUN export JAVA_OPT=" -Duser.home=/opt" ; \
    sed -i
's/${JAVA_HOME}\/jre\/lib\/ext/${JAVA_HOME}\/jre\/lib\/ext:${JAVA_HOME}\/lib\/ext/'
${ROCKETMQ_HOME}/bin/tools.sh

ENV PATH "$PATH:${ROCKETMQ_HOME}/bin"

WORKDIR ${ROCKETMQ_HOME}/bin
```

Code 5-7. Installation of Apache RocketMQ and copying the launch scripts.

5.3.2 SINGLE CLUSTER DEPLOYMENT

To distribute the containerized application, we proceeded to the creation of a Kubernetes infrastructure, initially a vanilla setup, moving then to Google Kubernetes Engine (GKE), a more advanced system. The deployment consists of the following components that share the same RocketMQ container image:

- A master ***broker*** deployment with its configuration.

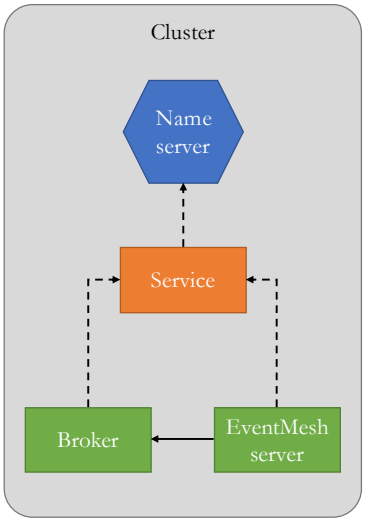


Figure 5-2. Single cluster deployment schema.

- A ***name server*** with an associated Service that allows to access to the name server without the explicit need of the IP address.
- A ***console*** application which allows to *control the deployment*, adding, for example, new topics.

We will now describe each of these Kubernetes objects in more detail.

5.3.2.1 NAME SERVER

The name server is the central component in the RocketMQ setup and it comprises a `Deployment` and a `Service`. The code of the first is shown below.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rocketmq-name-service
  labels:
    environment: research
    app: rocketmq
    role: name-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rocketmq
      role: name-service
      ...
      containers:
      - name: name-service
        image: framctrdh/rocketmq:4.9.3
        command: ["sh", "mqnamesrv"]
        imagePullPolicy: IfNotPresent
        resources:
          requests:
            cpu: "250m"
            memory: "512Mi"
          limits:
            cpu: "500m"
            memory: "1Gi"
        ports:
        - containerPort: 9876
          name: main
          protocol: TCP
        volumeMounts:
        - mountPath: /home/rocketmq/logs
          name: namesrv-log
          subPath: logs/namesrv
      volumes:
      - emptyDir: {}
        name: namesrv-log
```

Code 5-8. Deployment of the RocketMQ name server.

As we can see, the deployment is univocally identified through the `app` and `role` labels. The first indicates the application used, while the second which role it has in the cluster. The name server has just one replica and it is launched with the standard configuration through the command `mqnamesrv` and by default the TCP communication happens through the port 9876.

The access to the name service is mediated by a `Service` resource, as shown below, and this allows to communicate with it through its own name instead of the IP address.

```
apiVersion: v1
kind: Service
metadata:
  name: rocketmq-name-service
spec:
  selector:
    app: rocketmq
    role: name-service
  ports:
  - port: 9876
    name: main
```

Code 5-9. Service definition for the RocketMQ deployment.

5.3.2.2 BROKER

The broker requires, with respect to the name server, some configuration which is kept in the ConfigMap shown below.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: broker-config
...
data:
  BROKER_MEM: " -Xms2g -Xmx2g "
  broker-common.conf: |
    namesrvAddr=rocketmq-name-service:9876
    flushDiskType=ASYNC_FLUSH
    brokerRole=ASYNC_MASTER
```

Code 5-10. Configuration of the RocketMQ broker.

The `BROKER_MEM` variable is required by the broker to tune the JVM performance settings [106]. Specifically, it defines the initial and minimum of the Java heap size (`-Xms`) and the maximum size of the memory allocation pool (`-Xmx`).

In addition, the ConfigMap specifies a file, named `broker-common.conf`, which contains all the data related to the broker specific configuration: the name service address with the associated port, the broker role (master or slave), and the flush type which defines if the messages need to be stored on disk for high availability (`SYNC_FLUSH`) or kept just in memory to increase performance (`ASYNC_FLUSH`).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rocketmq-broker-master
  labels:
    environment: research
    app: rocketmq
    role: broker
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rocketmq
      role: broker
    ...
    spec:
      containers:
      - name: broker-master
        image: framctrdh/rocketmq:4.9.3
        command: ["sh", "mqbroker", "-c", "/home/rocketmq/broker/conf/broker-
common.conf"]
        imagePullPolicy: IfNotPresent
        resources:
          requests:
            cpu: "250m"
            memory: "2Gi"
          limits:
            cpu: "500m"
            memory: "4Gi"
        env:
        - name: BROKER_MEM
          valueFrom:
            configMapKeyRef:
              name: broker-config
              key: BROKER_MEM
        ports:
        - containerPort: 10911
          name: main
          protocol: TCP
        volumeMounts:
        - mountPath: /home/rocketmq-broker/logs
          name: broker-log
          subPath: logs/broker-master
        - mountPath: /home/rocketmq-broker/store
          name: broker-store
          subPath: store/broker-master
        - mountPath: /home/rocketmq/broker/conf/broker-common.conf
          name: broker-config
          subPath: broker-common.conf
```

```
      volumes:
      - emptyDir: {}
        name: broker-log
      - emptyDir: {}
        name: broker-store
```

Code 5-11. The broker deployment.

5.3.2.3 CONSOLE

The console application allows to take control over the RocketMQ cluster from command line and it is defined by a `Deployment` which creates a pod running indefinitely, consuming just few resources. It is possible to connect to the pod through the following command:

```
$ kubectl exec -ti $(kubectl get pod -l app=rocketmq -l role=console-admin -o
jsonpath="{.items[0].metadata.name}") -- sh
```

After the connection, it is possible to execute many operations, such as adding a topic to the default RocketMQ cluster, verifying the insertion, as shown below:

```
$ sh mqadmin updateTopic -c DefaultCluster -t TEST-TOPIC-TCP-ASYNC -n rocketmq-name-
service:9876
$ sh mqadmin topicList -n rocketmq-name-service:9876
```

5.3.3 MULTI-CLUSTER DEPLOYMENT

The multi-cluster deployment is similar to the single cluster with the only exception for the following changes:

- Broker and name server deployments have each one 2 replicas, one for each cluster. The console does not require to be replicated, but, to simplify the scripts responsible for the deployment of the workflow application, the console has 2 replicas.
- The name servers are accessible through public IP because they are deployed on different clusters.

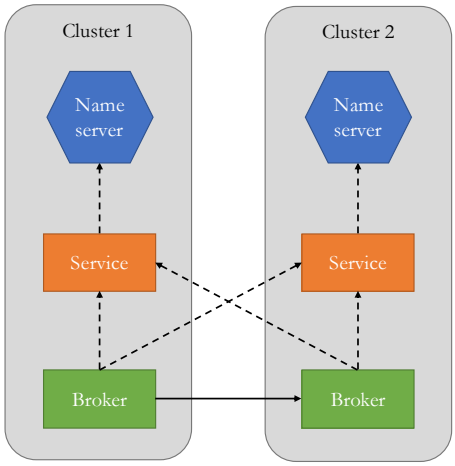


Figure 5-3. Apache RocketMQ multi-cluster schema.

More in detail, the name servers have a `Deployment` and a `Service`, but with respect the the single cluster case, the `Service` is a LoadBalancer, which publicly exposes an IP address to access the name server. Since the broker needs the name servers IP when booting, there is the need to wait until the public IP becomes available as presented by the code below.

```
while [ -z "$(kubectl -n eventmesh get svc rocketmq-name-service -
o=jsonpath='{.status.loadBalancer.ingress[0].ip}')" ]
do
    sleep 2
done
```

Subsequently, we deployed the brokers, inserting both addresses of the server names in each of these. In many YAML files we use custom defined variables (whose name is indicated between angular parenthesis), that we then substitute with their values before the deployment, using the BASH `sed` command. An example is given by the following code showing the replacement of the `< ADDR >` variable with the name servers public IPs.

```
cat cluster-2/broker-config-fast.yaml | sed "s/< ADDR >/${NAMESRV}/g" | kubectl -n
eventmesh apply -f -
cat cluster-2/broker-fast.yaml | sed "s/< ADDR >/${NAMESRV}/g" | kubectl -n
eventmesh apply -f -
```

5.4 APACHE EVENTMESH

After the deployment of Apache RocketMQ, we proceeded with the one of Apache EventMesh, creating a custom Docker image and subsequently the deployments for single and multi-clusters.

5.4.1 CONTAINER IMAGE

The Docker image is based on Alpine and it is built by downloading the official Apache EventMesh binaries that contain also the `start.sh` script that we set as the container launcher, after its adaptation to the Alpine shell.

```
FROM openjdk:8-alpine

ARG em_version
ENV EVENTMESH_VERSION ${em_version}
WORKDIR /opt

RUN wget https://github.com/apache/incubator-
eventmesh/releases/download/v${EVENTMESH_VERSION}/apache-eventmesh-
${EVENTMESH_VERSION}-incubating-bin.tar.gz -O - | tar -xz \
 && mv apache-eventmesh-${EVENTMESH_VERSION}-incubating eventmesh-
${EVENTMESH_VERSION} \
 && chmod +x eventmesh-${EVENTMESH_VERSION}/bin/*.sh \
 && rm eventmesh-${EVENTMESH_VERSION}/bin/start.sh

COPY bin/start.sh eventmesh-${EVENTMESH_VERSION}/bin/

WORKDIR /opt/eventmesh-${EVENTMESH_VERSION}/bin

ENV DOCKER true
CMD sh start.sh

EXPOSE 10000
EXPOSE 10105
```

Code 5-12. Apache EventMesh Dockerfile.

5.4.2 KUBERNETES DEPLOYMENT

Regarding the deployment of Apache EventMesh, it must be initially noticed that it is the same for the single and multi-cluster environments. In fact, the EventMesh server uses the same deployment configuration in both cases: the only element that could be different is the address of the RocketMQ name server, however Apache EventMesh requires just the name server of the cluster it resides in.

The first element for the deployment of the Apache EventMesh runtime is the ConfigMap, shown below, that contains all its configuration properties. We enabled the TCP communication, required by the workflow applications, and the metrics, exposed in Prometheus format on the port 19090.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: eventmesh-config
data:
  eventmesh.properties: |-
    eventMesh.server.idc=DEFAULT
    eventMesh.server.env=PRD
    eventMesh.server.cluster=COMMON
    eventMesh.server.name=EVENTMESH-runtime
    eventMesh.sysid=0000
    eventMesh.server.http.port=10105
    ############ eventMesh tcp configuration
    eventMesh.server.tcp.enabled=true
    eventMesh.server.tcp.port=10000
    eventMesh.server.tcp.readerIdleSeconds=120
    eventMesh.server.tcp.writerIdleSeconds=120
    eventMesh.server.tcp.allIdleSeconds=120
    eventMesh.server.tcp.clientMaxNum=10000
    # client isolation time if the message send failure
    eventMesh.server.tcp.pushFailIsolateTimeInMills=30000
    # rebalance internal
    eventMesh.server.tcp.RebalanceIntervalInMills=30000
    # session expire time about client
    eventMesh.server.session.expiredInMills=60000
    # flow control, include the global level and session level
    eventMesh.server.tcp.msgReqnumPerSecond=15000
    eventMesh.server.session.upstreamBufferSize=20

...

    #metrics
    eventMesh.server.metrics.enabled=true
    eventMesh.metrics.prometheus.port=19090
    eventMesh.metrics.plugin=prometheus

...
```

Besides this ConfigMap, there is another one containing the file `rocket-client.properties` with the RocketMQ name server address.

The second element is the Apache EventMesh server `Deployment` object which has one replica per cluster, requires the configuration files provided by the ConfigMap, and opens ports for communication and metrics.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: eventmesh-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: eventmesh-server
  template:
    metadata:
     labels:
       app: eventmesh-server
    spec:
      containers:
      - name: eventmesh-server
        image: framctrdh/apache-eventmesh:1.4.0-alpine
        imagePullPolicy: IfNotPresent
        ports:
          - containerPort: 10000
            name: tcp
          - containerPort: 10105
            name: http
          - containerPort: 10205
            name: grpc
          - containerPort: 19090
            name: metrics
        volumeMounts:
          - mountPath: /data/app/eventmesh/conf/eventmesh.properties
            subPath: eventmesh.properties
            name: eventmesh-config
          - mountPath: /data/app/eventmesh/conf/rocketmq-client.properties
            subPath: rocketmq-client.properties
            name: rocketmq-client-config
      volumes:
      - name: eventmesh-config
        configMap:
          name: eventmesh-config
      - name: rocketmq-client-config
        configMap:
          name: rocketmq-client-config
```

Code 5-13. Apache EventMesh server deployment file.

Finally, the third element is a `Service` that allows the workflow applications to access the EventMesh runtime server through its name inside the cluster, as shown below.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: eventmesh-service
  labels:
    app: eventmesh-service
spec:
  selector:
    app: eventmesh-server
  ports:
    - port: 10000
      name: tcp
    - port: 10105
      name: http
    - port: 10205
      name: grpc
    - port: 19090
      name: metrics
```

Code 5-14. Apache EventMesh service useful for applications to access the event mesh through the service name.

5.5 WORKFLOW APPLICATION

At this point we want to move from the explaination of work done for the infrastructural components to the applicative part of the project and, more specifically, to the workflow that consists of a series of echo applications.

The workflow application is formed by a set of elements each of those can be one of the following types:

- ***Trigger*** which starts the communication and sends messages that will be exchanged along the application chain.
- ***Action*** which executes an echo to the previous component and at the same time forwards the message to the next component.
- ***Conclusion*** which is the last component to receive messages.

While there must be only one trigger element and one conclusion, the number of action elements can vary. The configuration of each element is illustrated in a file describing the behavior of each component, as we will see in more detail in the next chapter: it contains the role, the name associated to the component in the cluster (rootname), the number of send/receive operations it can do each second, and the communication properties.

```
[trigger]
rootname=trigger
operation.ops.start=10
operation.ops.end=100
communication.hostname.to=action01
communication.port.to=10091
communication.port.from=10090
```

Code 5-15. Example of trigger element configuration.

The applicative part is divided into two different applications, one for the synchronous and the other for the asynchronous communication. To keep uniformity between those two implementations, we wrote both applications in Java, since Apache EventMesh SDK supports only this programming language; furthermore, we decided to adopt TCP as the communication protocol for both applications.

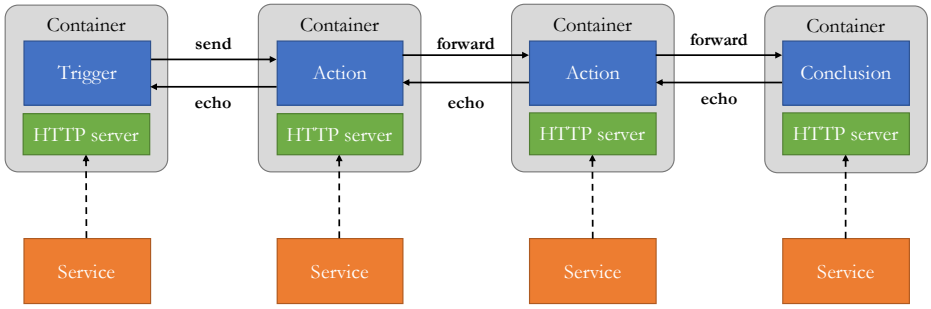


Figure 5-4. Schema of the workflow application used during tests. Each container comprises a workflow application and a web server for statistics, that are accessible through a Service.

5.5.1 SYNCHRONOUS APPLICATION

Starting with the synchronous application we used for testing the service meshes, we will describe the codebase first and then continue with containerization and deployment.

5.5.1.1 CODEBASE

The codebase is organized into two parts: on one hand, the `main` which reads configuration from environment variables, prepares and starts the communication; on

the other hand, the communication objects, shown below, which make up the server and the client.

The TCP echo client creates a Socket and then sends a message, waiting for the latter to be echoed. It must be noted that, during the deployment, each workflow entity can be scheduled at a different time and so it can happen that a client component starts before the server, resulting in a connection failure. For this reason, the client attempts multiple times to connect to the server.

```
while( !clientSocket.isConnected() && attempts > 0) {
    try {

        clientSocket = new Socket(host, port);

        reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
        writer = new BufferedWriter(new
OutputStreamWriter(clientSocket.getOutputStream()));
    ...
    attempts--;
}
```

Code 5-16. TCP client init connection.

```
writer.write(message + "\n");
writer.flush();

String ret = "";
if((ret = reader.readLine()) != null) {

    logger.info("Received message: " + ret);
}
```

Code 5-17. TCP client echo.

On the other side, the server waits for connections and then, when the message is received, it echoes it to the client.

```
serverSocket = new ServerSocket(port);
...
socket = serverSocket.accept();

reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
writer = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
```

Code 5-18. TCP server waiting for connections.

```
StringBuilder message = new StringBuilder();

String line = "";
while((line = reader.readLine()) != null) {

    writer.write(message + "\n");
    writer.flush();

    message.append(line);
}

socket.close();
```

Code 5-19. TCP server echo.

The forwarding of messages from one workflow entity to another is accomplished by combining a server and a client: the former receives a message which is then passed to the client which sends it to the next component.

To support constant as well as variable amount of requests per second, each workflow entity creates a `DurationCalculator` object which evaluates the time between one operation and another.

```
this.duration = totalDuration;

this.start_wait = 1000 / start_ops;
this.end_wait = 1000 / end_ops;

this.start = System.currentTimeMillis();
...
if(start_wait == 0 || start_wait == end_wait) {

    return start_wait;
}
// check for elapsed time > duration
long elapsed = System.currentTimeMillis() - start;
double t = elapsed < duration ? elapsed / (double)duration : 1.0;
return Math.round((1.0 - t) * start_wait + t * end_wait);
```

Code 5-20. Evaluation of the thread sleeping time between one operation and the other. The variable time is calculated with a linear interpolation.

5.5.1.2 CONTAINER IMAGE

The building of the container image requires to package the project in JAR file which is then copied inside the container image together with the shell script used for the application launch.

```
./mvnw package

docker build -f Dockerfile -t framctrdh/servicemesh-workflow:latest . && \
docker push framctrdh/servicemesh-workflow:latest
```

Code 5-21. Shell script that builds the project and the Dockerfile.

```
FROM framctrdh/stats-base:latest

COPY tmp /opt/workflow
COPY bin/start.sh /opt
```

Code 5-22. Dockerfile of the workflow application. We used a custom image containing the HTTP server for the statistics.

```
#!/bin/sh

java -cp workflow:. -jar workflow/*.jar it.workflow.Workflow && \
./start-stats.sh
```

Code 5-23. Launcher of the application. After the applications exits, it starts an HTTP server for statistics.

5.5.1.3 DEPLOYMENT

The deployment of the workflow application is the same for both the single and multi-cluster environments with the only difference, in the second case, that the conclusion component is deployed on a different cluster.

The Deployment and the Service resources which are used for the application workflow, have custom variables that are expanded with the values taken from the configuration files already mentioned (cf. Code 5-15).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: workflow-< NAME >
spec:
  replicas: 1
  selector:
    matchLabels:
      app: workflow
      role: < ROLE >
      name: < NAME >
  template:
    metadata:
     labels:
       app: workflow
       role: < ROLE >
       name: < NAME >
    spec:
      containers:
      - name: workflow-< NAME >
        imagePullPolicy: Always
        env:
        - name: DURATION
          value: "< DURATION >"
        - name: ROOTNAME
          value: "< NAME >"
        - name: ROLE
          value: "< ROLE >"
        - name: OPS_START
          value: "< OPS_START >"
        - name: OPS_END
          value: "< OPS_END >"
        - name: HOST_TO
          value: "< HOST_TO >"
        - name: PORT_FROM
          value: "< PORT_FROM >"
        - name: PORT_TO
          value: "< PORT_TO >"
        - name: STATS_FILENAME
          value: "stats.log"
        image: framctrdh/servicemesh-workflow:latest
        command: ["sh", "start.sh"]
        ports:
          - containerPort: < PORT_FROM >
            name: tcp-server
          - containerPort: 8080
            name: app-stats
          - containerPort: 15020
            name: istio-metrics
```

Code 5-24. Deployment of the workflow application. It sets the environment variables, starts the application executing the custom script, and opens one port for the workflow and the others for exposing metrics.

```
apiVersion: v1
kind: Service
metadata:
  name: < SERVICE_NAME >
spec:
  selector:
    app: workflow
    role: < ROLE >
    name: < NAME >
  ports:
  - port: < PORT_FROM >
    name: tcp-server
  - port: 8080
    name: app-stats
  - port: 15020
    name: istio-metrics
```

Code 5-25. Service of the workflow application. It is used by clients to communicate with servers and by the metrics server to gather metrics.

5.5.2 ASYNCHRONOUS COMMUNICATION

Passing now to the asynchronous application designed to be used with Apache EventMesh, it must be noticed that this, although constructed similarly to the previous one, has some differences, some of which are more significant, that we will describe in the following.

5.5.2.1 CODEBASE

Regarding the codebase, the difference with the previous case is due obviously to the different way of communication.

This project requires the Apache EventMesh Common and Java SDK libraries which expose APIs to connect to the EventMesh runtime server. Both producer and consumer, defined in our application codebase, require an instance of `EventMeshTCPClientConfig` class that specifies the configuration of EventMesh TCP client. The `host` and `port` fields must match those of the EventMesh server.

```
public class PublisherTCP {
    public PublisherTCP(UserAgent userAgent) {

        UserAgent clientUserAgent = MessageUtils.generatePubClient(userAgent);

        EventMeshTCPClientConfig clientConfig = EventMeshTCPClientConfig.builder()
                .host(eventMeshIP)
                .port(eventMeshTCPPort)
                .userAgent(clientUserAgent)
                .build();

        client   = EventMeshTCPClientFactory.createEventMeshTCPClient(clientConfig,
EventMeshMessage.class);
        client.init();
}
```

Figure 5-5. Initialization of the EventMesh TCP publisher in the workflow application. The procedure is the same for the TCP consumer.

The publisher client has an `asyncPublish` method which wraps the `publish` method implemented by the `EventMeshTCPClient` class, which accepts the message to be published and a timeout value, adding some properties to the message, such as the topic.

```
public void asyncPublish(String message, String topic, Map<String, String>
properties) {

    EventMeshMessage eventMeshMessage = new EventMeshMessage();

    if(properties != null) {
        for(Map.Entry<String, String> property : properties.entrySet()) {

            eventMeshMessage.getProperties().put(property.getKey(),
property.getValue());
        }
    }

    eventMeshMessage.setTopic(topic);
    eventMeshMessage.setBody(message);

    client.publish(eventMeshMessage, client_timeout);
}
```

Code 5-26. Publish method of the workflow TCP client.

On the other hand, the TCP consumer implements the `ReceiveMsgHook` class with its `handle` method, called by Apache EventMesh when a new message for a certain topic is ready to be consumed. The subscription happens through the `subscribe` method which accepts: the topic, a `SubscriptionMode` (`CLUSTERING` or `BROADCASTING`), and the

`SubscritptionType` which indicates if the subscription should be asynchronous or synchronous.

```
public void asyncSubscribe(String topic, IWorkflowActor actor) {

    this.actor = actor;
    handler.setActor(actor);

    client.subscribe(topic, SubscriptionMode.CLUSTERING, SubscriptionType.ASYNC);
    client.registerSubBusiHandler(handler);
    client.listen();
}
```

Code 5-27. Subscribe method of the workflow TCP client. The handler requires an actor which implements the echo action.

```
@Override
public Optional<EventMeshMessage> handle(EventMeshMessage msg) {
    actor.action(msg.getBody());
    return Optional.empty();
}
```

Code 5-28. Handler method of the TCP client. It retrieves the message and passes it to the echo function, implemented as an actor action.

5.5.2.2 DEPLOYMENT

The containerization and the deployment are really similar to the synchronous case, however the communication between the workflow components requires to add topics to the broker cluster. For this reason, the `deploy.sh` script has a phase in which it adds them through the Apache RocketMQ console application.

```
    kubectl -n $NAMESPACE exec $rocketmq_console -- \
        sh mqadmin updateTopic -c DefaultCluster -t $topic \
        -n rocketmq-name-service:9876
```

Code 5-29. Execution of the topic addition to the Apache RocketMQ default cluster.

5.6 METRICS

After the description of the applicative part, let's now face the last part of our project which is the one related to the metrics. As we already know from the previous chapter, all the data related to the tests are collected by a custom application, compatible with the Prometheus format.

The metrics are gathered by a *client application* running locally on our computers, which obtains data from various sources, such as the Kubernetes API or our own metrics *server application*, then storing them in files which are finally aggregated into a single table for graphs generation.

The metrics client can monitor multiple Kubernetes clusters by creating for each of them an object that keeps all the data related to the cluster resources. These data are collected periodically, usually every few seconds, and stored by a single `Persistence` object.

```
while not stop:
    for cluster in clients.keys():
        now = datetime.datetime.now()
        persist.set_collection(cluster=cluster, coll=now)

        client.detect_worflow_apps(namespace)

        client = clients[cluster]
        # CPU and memory
        client.get_nodes_resource_usage()
        client.get_containers_resource_usage(ns=namespace)

        # Storage
        client.list_persistent_volume_claims(ns=namespace)

        client.request_metrics()
        client.request_stats()

    time.sleep(scrape_interval)
```

Code 5-30. Loop function that periodically gathers metrics for each cluster. For every execution the Persistence object sets a collection (snapshot) labeled with the date it was taken.

5.6.1.1 KUBERNETES METRICS CLIENT

The metrics client uses the official Kubernetes client library for Python. It loads the configuration for a certain cluster from the default file in the system[23], which corresponds to executing `kubectl --context=`. Then, it loads APIs needed to gather information from the Kubernetes API Server and, more specifically, from the Kubernetes metrics addon.

[23] It is possible to display in the shell all the clusters executing: `kubectl config get-contexts`.

```
class KubeMetricsClient:
    def __init__(self, cluster = "default"):
        api_client = config.new_client_from_config()

        if cluster != "default":
            api_client = config.new_client_from_config(context=cluster)

        self.core_v1_api = client.CoreV1Api(api_client)
        self.custom_objects_api = client.CustomObjectsApi(api_client)
```

Code 5-31. Initialization of the metrics client for a certain cluster.

At first, the `KubeMetricsClient` class, by accessing the data provided by the Kubernetes metrics addon, gathers the resources usage of each node in the cluster, formatting it according to the Prometheus format by using a custom object.

```
def get_nodes_resource_usage(self, human_readable = False):
    api = self.custom_objects_api

    try:
        resource = api.list_cluster_custom_object(group="metrics.k8s.io",
version="v1beta1", plural="nodes")

        f = formatter.PrometheusFormatter()
        for node in resource["items"]:
            node_name = node["metadata"]["name"]
            node_cpu = utils.parse_cpu(node["usage"]["cpu"])
            node_mem = utils.parse_mem(node["usage"]["memory"])

            f.insert_elem(("node", "cpu", "usage"))
            f.insert_label("node", node_name)
            f.insert_value(utils.raw_unit(node_cpu))

            f.insert_elem(("node", "memory", "usage"))
            f.insert_label("node", node_name)
            f.insert_value(utils.raw_unit(node_mem))

            logging.debug("Node res usage\n" + str(node_name) + ":\n - cpu: " +
utils.readable_unit(node_cpu) + "\n - memory: " + utils.readable_unit(node_mem))

        self.persistence.store(cluster=self.cluster, name="nodes",
content=f.format())
...
```

Code 5-32. Collection and formatting of the node resource usage.

The metrics client gathers data also for each container in the target namespace of the cluster. In this way, in the case of service meshes, we are able to differentiate workflow application metrics from their associated sidecar proxies. The code, similar to the previous, is shown below.

```
def get_containers_resource_usage(self, ns = "default", human_readable = False):
    api = self.custom_objects_api

    try:
        resource = api.list_namespaced_custom_object(group="metrics.k8s.io",
version="v1beta1", namespace=ns, plural="pods")

        f = formatter.PrometheusFormatter()
        for pods in resource["items"]:
            pod_name = pods["metadata"]["name"]

            for container in pods["containers"]:
                container_name = container["name"]
                container_cpu = utils.parse_cpu(container["usage"]["cpu"])
                container_mem = utils.parse_mem(container["usage"]["memory"])

                f.insert_elem(("container", "cpu", "usage"))
                f.insert_label("pod", pod_name)
                f.insert_label("namespace", ns)
                f.insert_label("container", container_name)
                f.insert_value(utils.raw_unit(container_cpu))

                f.insert_elem(("container", "memory", "usage"))
                f.insert_label("pod", pod_name)
                f.insert_label("namespace", ns)
                f.insert_label("container", container_name)
                f.insert_value(utils.raw_unit(container_mem))

        self.persistence.store(cluster=self.cluster, name="containers",
content=f.format())
    ...
```

Code 5-33. Collection and formatting of the containers resource usage.

Subsequently, the metrics client collects data from each application in the cluster. These data that are divided into *metrics* (already in Prometheus format) and *stats* (not in Prometheus format).

```
def init_metrics_server(self, ns = "default", server_name = "simple-metrics-server",
server_path = "/metrics"):
    services = self.core_v1_api.list_namespaced_service(ns)

    # retrieve metrics server IP in the cluster

    self.metrics_server_url = "http://" + server_ip + ":" + str(server_port) +
server_path
    logging.info("Metrics server address is " + self.metrics_server_url)

    self.in_cluster_metrics = dict()
    self.in_cluster_stats = dict()
```

Code 5-34. Initialization of the registries for the application metrics and stats, and retrieval of the metrics server public IP.

The metrics client keeps, for each of these data categories, a registry (in the form of a Python dictionary), containing the URL of each component that is present in the cluster, as shown by the code below for the Apache EventMesh case.

```
def init_eventmesh_registry(self, em_ns = "eventmesh"):
    self.in_cluster_metrics['rocketmq'] = {'hostname': ''.join(('rocketmq-
exporter.', em_ns, '.svc.cluster.local')), 'port': '5557', 'path': '/metrics'}
    self.in_cluster_metrics['eventmesh'] = {'hostname': ''.join(('eventmesh-
service.', em_ns, '.svc.cluster.local')), 'port': '19090', 'path': ''}
```

Code 5-35. Registry containing the endpoint URLs for Apache EventMesh and Apache RocketMQ.

Finally, the metrics client gathers the data from each component in the registry and stores the result, formatting them according to Prometheus format in the case of *stats*:

```
for endpoint in self.in_cluster_metrics:
    addr = self.in_cluster_metrics[endpoint]
    metrics = requests.get(url=self.metrics_server_url, params=addr,
timeout=REQUEST_TIMEOUT)

    if metrics.text == 'Error':
        logging.warning("Cannot reach " + endpoint + " in the cluster.")
    else:
        self.persistence.store(cluster=self.cluster, name=endpoint,
content=metrics.text)
```

Code 5-36. Gathering of metrics.

```
f = formatter.WorkflowFormatter()
for app in self.in_cluster_stats:
    addr = self.in_cluster_stats[app]
    stats = requests.get(url=self.metrics_server_url, params=addr,
timeout=REQUEST_TIMEOUT)

    if stats.text != 'Error':
        f.parse(name=addr['hostname'], text=stats.text)

for actor in f.format():
    self.persistence.store(cluster=self.cluster, name=actor['name'],
content=actor['content'])
```

Code 5-37. Gathering and formatting of stats.

5.6.1.2 KUBERNETES SERVER

As already mentioned, the *metrics client* uses a *metrics server*, deployed in a dedicated namespace (termed monitoring) in each Kubernetes cluster, to gather metrics and stats related to each microservice. The metrics server is a simple web server build upon

Flask[24]. It receives requests from the metrics client, which inserts inside the HTTP request the URL parameters of the target application. These data are assembled by the metrics server and used to scrape the remote application in the cluster. The result is returned to the client.

```
@app.route("/metrics")
def server_metrics():
    address="http://" + request.args.get('hostname') + ":" +
request.args.get('port') + request.args.get('path')

    result = 'Error'
    try:
        r = requests.get(url=address, timeout=REQUEST_TIMEOUT)
        if r.status_code == 200:
            result = r.text
        else:
            logging.warning("Request failed with status: " + r.status_code)
    except:
        logging.warning("Request failed.")

    return result
```

Code 5-38. Metrics server core method.

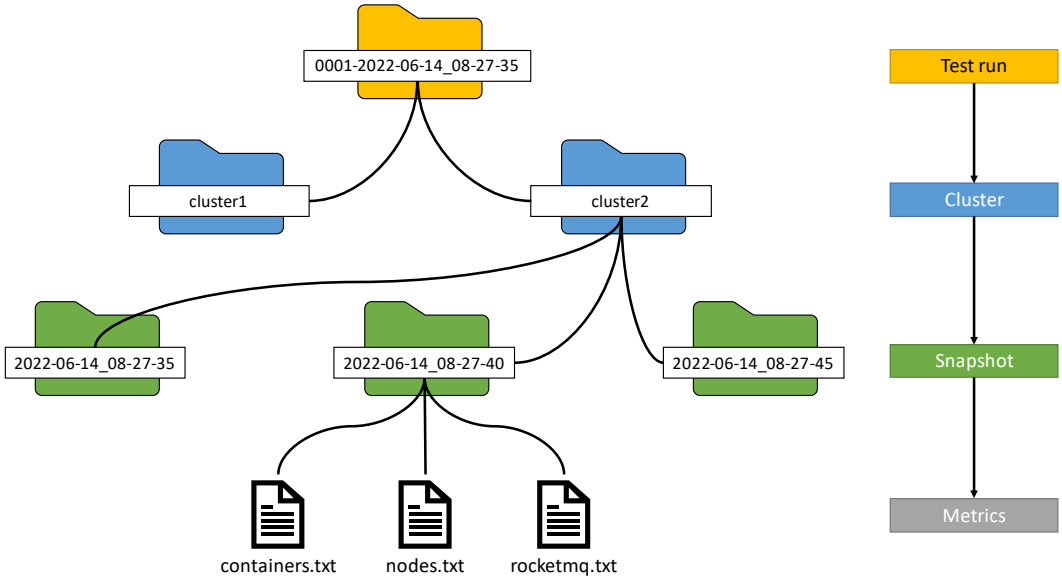
For security reasons we decided to avoid running the application as a root user, as we did for the case of Apache RocketMQ Docker image.

5.6.1.3 PERSISTENCE

Regarding the persistensce of the data, it is important to point out that Prometheus stores data as a time series: streams of timestamped values belonging to the same metric and the same set of labeled dimensions. For simplicity, we decided to store all the metrics as files and, to keep the data ordered, we organized them according to a precise structure.

Each test run defines a root folder named with the date the test was started. This directory contains all the clusters involved in the test as folders: each folder contains the snapshots, also named collections, of the timeseries as directories. Finally, each of these directories contains the files related to each metric gathered.

[24] Flask is a lightweight web application framework for Python.



Code 5-39. Folder structure of the metrics gathering.

5.6.1.4 METRICS GATHERING

To simplify the gathering of the metrics, we collect data in Prometheus format when possible. In particular, sidecar proxies and the Apache EventMesh runtime server expose data in this way without the need to add any element to the cluster. Instead, Apache RocketMQ requires an *exporter* component, which returns rich metrics about the RocketMQ deployment in Prometheus format. Finally, each workflow application logs every send/receive operation in a file. When the test finishes or the duration timer runs out, an HTTP server built using SpringBoot starts, allowing the metrics server to retrieve all the operation data for each workflow component.

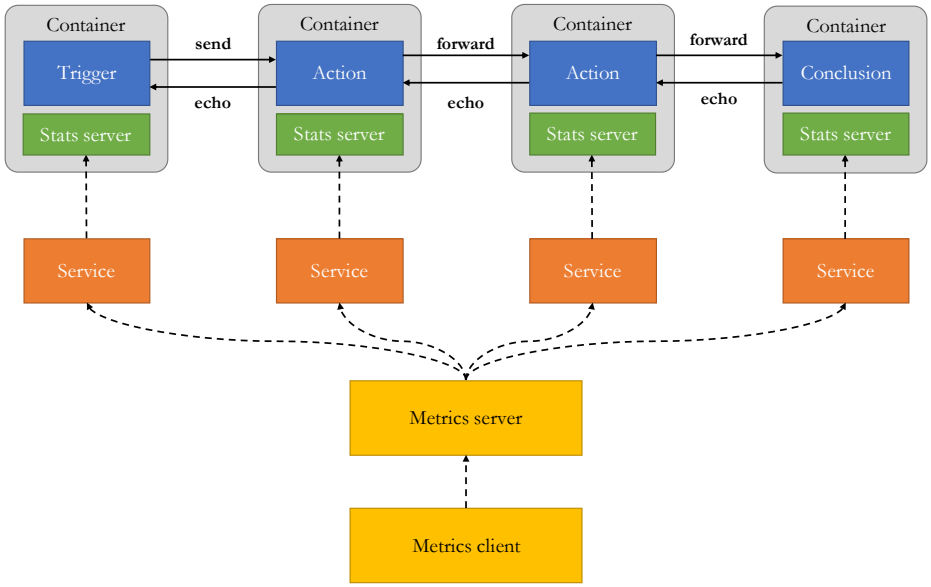


Figure 5-6. The metrics server and the workflow applications.

6 EXPERIMENTAL RESULTS

In this chapter we want to focus on the results of our experimentation, starting from the illustration of how we configured the tests to then continue with their analysis.

The tests we carried out were aimed at evaluating the performance of the various infrastructural components, which are Istio, Linkerd, and Apache EventMesh (together with Apache RocketMQ), when they are used for the operation of workflow applications in cloud environments. In particular, monitoring consumption of resources of each component under a load of first constant and subsequently incremental requests, we evaluated with these tests the functioning of more mature technologies, such as service meshes, and newer ones, such as event mesh, both in a single cluster and in a multi-cluster environment.

6.1 TEST CONFIGURATIONS

As already mentioned in the previous chapter, the configuration of each test is illustrated in a file, whose syntax derives from the one Java uses for the Properties [107], [108].

Inside each configuration file, it must be specified, first, the total *duration* of the test, then the workflow components, each of which starts with its *role* followed by the *configuration attributes* for the same component. The duration attribute can be written in human readable syntax since the script, which will deploy all the components according to the specified configuration, will evaluate it using a Python script to convert the timing to milliseconds, as Java uses to.

```
import pint

duration = sys.argv[1]
ureg = pint.UnitRegistry()
t = ureg.Quantity(duration)
print('{0.magnitude:.0f}'.format(t.to('millisecond')))
```

Code 6-1. Python script to convert from human readable time unit to milliseconds using the Pint package[25].

We decided to test our project in two different ways: in the first, a constant load with ten requests per second for two minutes; in the second, an incremental load from one to one thousand messages per second for five minutes. Since the load is generated by the Trigger component, the latter is the only one affected by this configuration, as can be read from the code below.

```
duration=2minutes

[trigger]
rootname=trigger
operation.ops.start=10
operation.ops.end=10
communication.hostname.to=action01
communication.port.to=10091
communication.port.from=10090

[action]
rootname=action01
operation.ops.start=100
operation.ops.end=100
communication.hostname.to=action02
communication.port.to=10091
communication.port.from=10091

[action]
rootname=action02
operation.ops.start=100
operation.ops.end=100
communication.hostname.to=conclusion
communication.port.to=10091
communication.port.from=10091
```

[25] The conversion requires Pint, a Python package to define, operate, and manipulate physical quantities, used also by the metrics client to correctly interpret metric values.

```
[conclusion]
rootname=conclusion
operation.ops.start=100
operation.ops.end=100
communication.port.to=10091
communication.port.from=10091
```

Code 6-2. Configuration for the test of constant load used for service meshes.

```
duration=5minutes

[trigger]
rootname=trigger
operation.ops.start=1
operation.ops.end=1000
communication.hostname.to=action01
communication.port.to=10091
communication.port.from=10090

[action]
rootname=action01
operation.ops.start=1000
operation.ops.end=1000
communication.hostname.to=action02
communication.port.to=10091
communication.port.from=10091

[action]
rootname=action02
operation.ops.start=1000
operation.ops.end=1000
communication.hostname.to=conclusion
communication.port.to=10091
communication.port.from=10091

[conclusion]
rootname=conclusion
operation.ops.start=1000
operation.ops.end=1000
communication.port.to=10091
communication.port.from=10091
```

Code 6-3. Configuration for the test of incremental load used for service meshes.

The tests for the asynchronous workflow application are the same, except for the configuration of each workflow component.

```
...
[trigger]
...
communication.to=INTERACTION-TRANSACTION-ASYNC-V1_0_0-WORKFLOW-ACTION1

[action]
...
communication.from=INTERACTION-TRANSACTION-ASYNC-V1_0_0-WORKFLOW-ACTION1
communication.to=INTERACTION-TRANSACTION-ASYNC-V1_0_0-WORKFLOW-ACTION2

[action]
...
communication.from=INTERACTION-TRANSACTION-ASYNC-V1_0_0-WORKFLOW-ACTION2
communication.to=INTERACTION-TRANSACTION-ASYNC-V1_0_0-WORKFLOW-ACTION3

[conclusion]
...
communication.from=INTERACTION-TRANSACTION-ASYNC-V1_0_0-WORKFLOW-ACTION3
```

Code 6-4. Differences of the configurations used for event mesh. In the communication attributes are indicated the topics.

6.2 SINGLE CLUSTER TESTS

For the execution of the tests for the single cluster case, it was created a Google Kubernetes Engine (GKE) cluster with the following characteristics:

- Kubernetes version: 1.21.11-gke.900.
- Zone: europe-west8-a (Milan, Italy).
- Total cores: 8 vCPUs.
- Total memory: 32 GB.

Regarding the nodes, we created a pool of 4 nodes, each of which with the following properties:

- Node container image: container-optimized OS (cos_containerd).
- Node type: n2d-standard-2.
- Node resources: 2 vCPUs (AMD EPYC) and 8GB of memory.

As mentioned above, we executed two different tests, one with constant and the other with incremental load, indicating both the metrics of each single component, as well as their mean.

6.2.1 RESULTS OF TESTS WITH CONSTANT LOAD

Starting from the constant load, first we will proceed with the analysis of the use of the resources of the workflow application and of the infrastructural components, to then move on to the latency of each single message as the number of messages sent per second varies.

6.2.1.1 WORKFLOW APPLICATION

As can be seen in the Figure 6-1 and Figure 6-2, the workflow application designed to work with the event mesh consumes in general few more CPU than the synchronous application, with a peak at the beginning. However, from the analysis of the memory usage illustrated in Figure 6-3 and Figure 6-4, it is evident that the event mesh workflow application consumes about four times the memory than the synchronous case. This is probably due to the Apache EventMesh Java SDK. The final peaks in all the graphs are due to the boot of the Spring based web server which exposes stats to our metrics system.

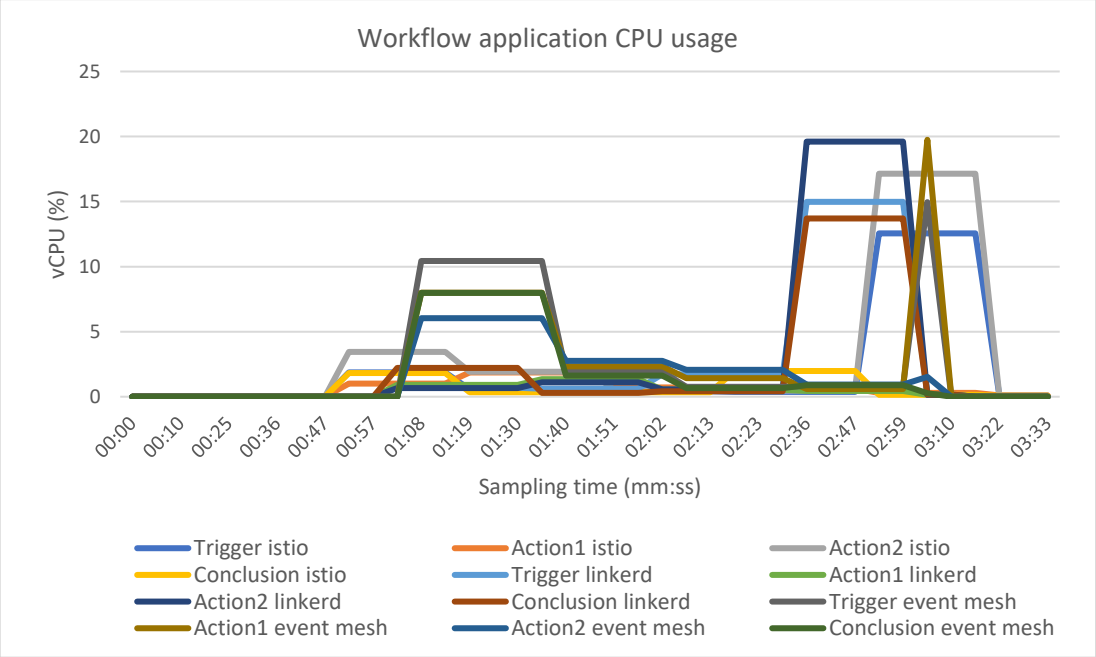


Figure 6-1. In the graph it is represented the trend of the CPU consumption for each workflow component for synchronous and asynchronous communication.

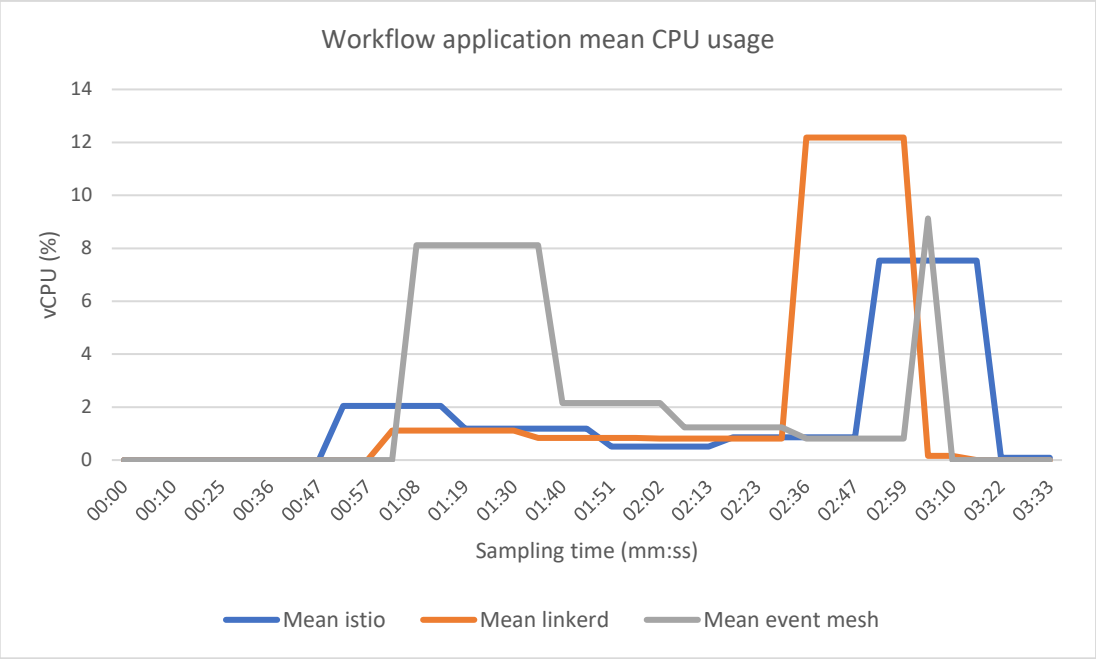


Figure 6-2. The graph represents the mean of the CPU usage of each workflow component for synchronous and asynchronous communication.

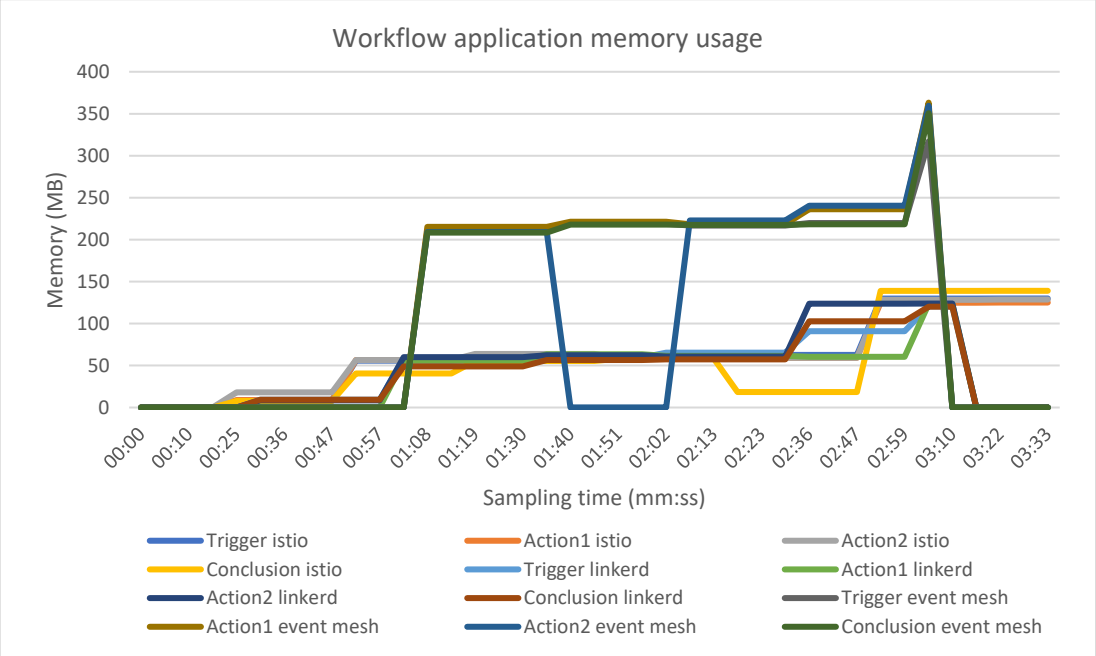


Figure 6-3. In the graph is represented the memory usage for each workflow component.

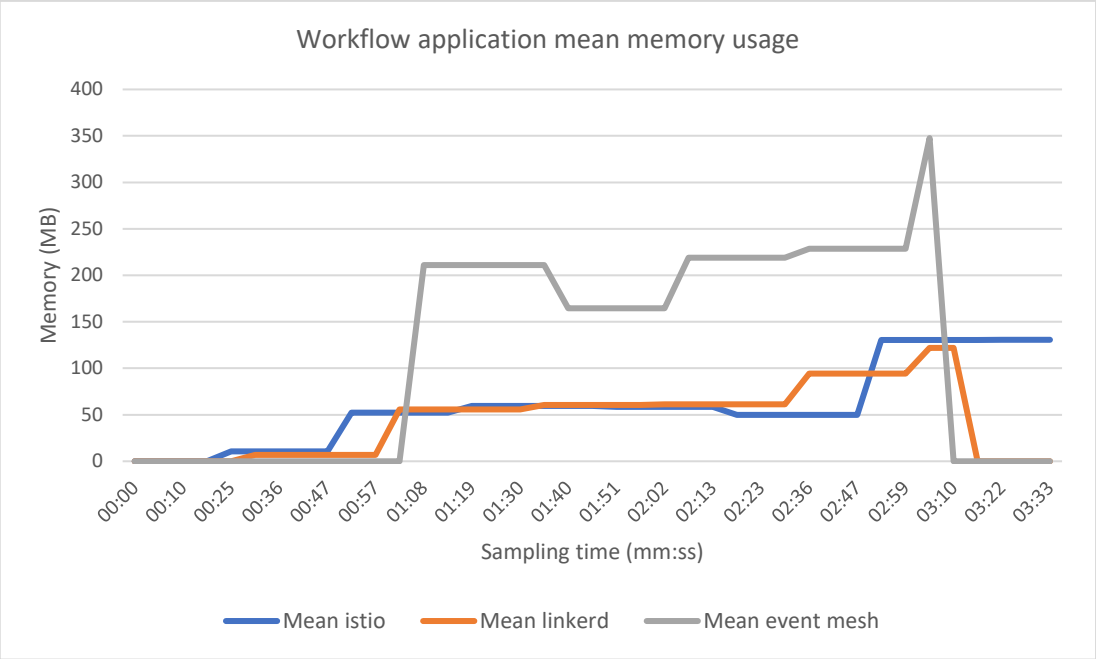


Figure 6-4. In the graph is shown the mean of the memory consuption for synchronous and asynchronous workflow application.

6.2.1.2 *INFRASTRUCTURAL COMPONENTS*

We now report the graphs relating to the results obtained for the infrastructural part of our project.

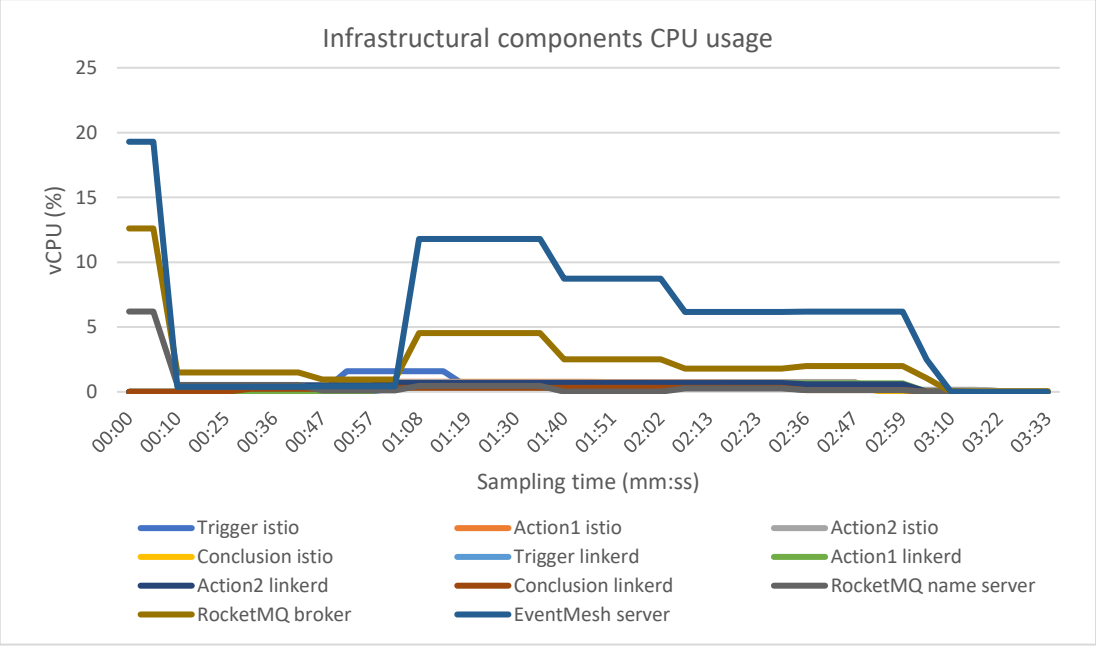


Figure 6-5. The graph represents the CPU usage for each mesh component.

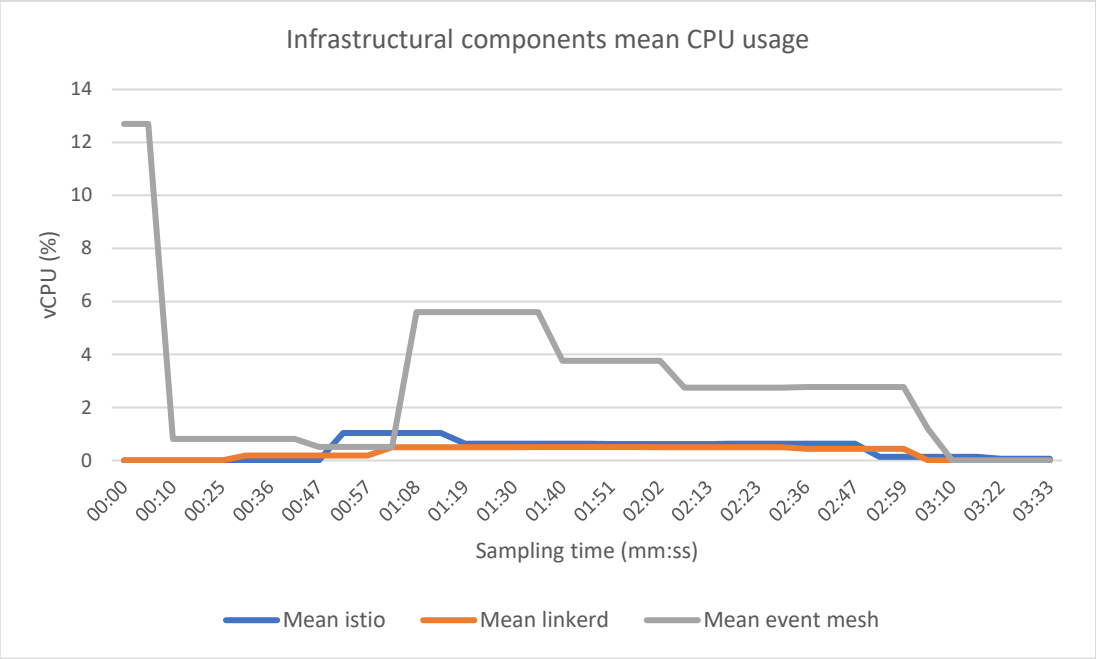


Figure 6-6. In the graph is represented the mean CPU consumption of the service and event mesh components.

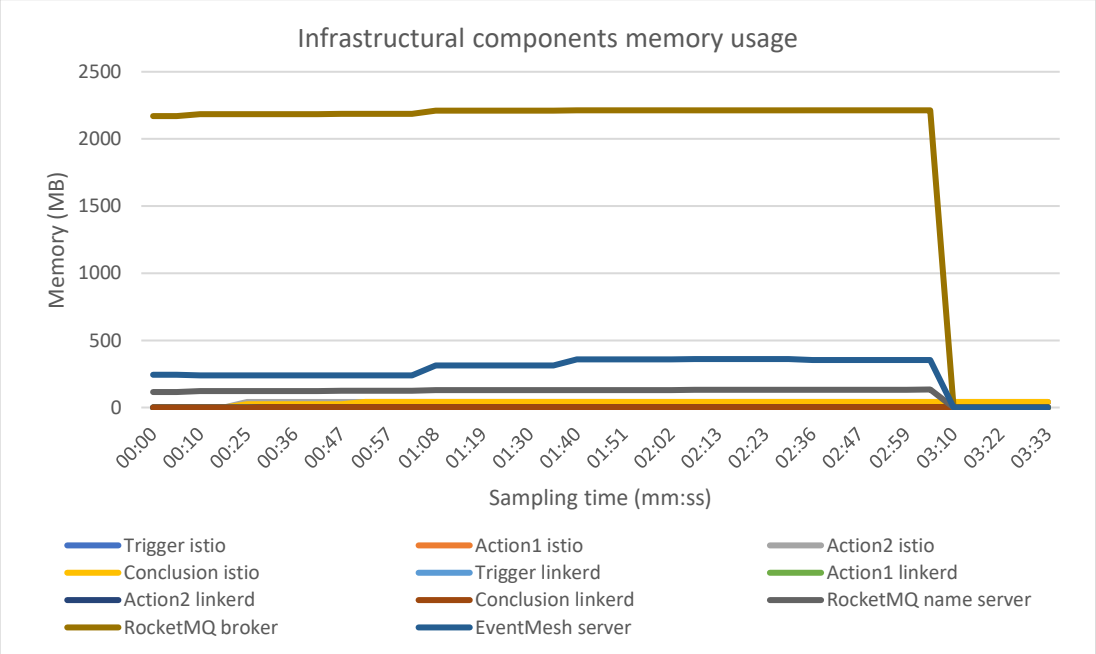


Figure 6-7. The graph represents the memory usage of each mesh component.

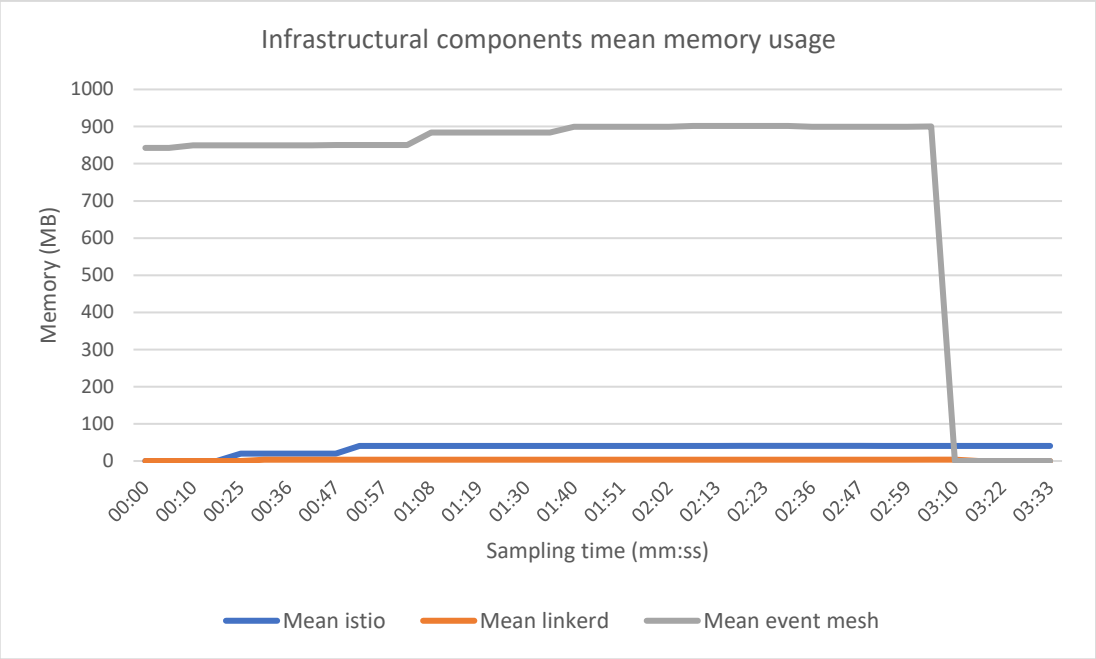


Figure 6-8. The graph represents the mean memory usage of the service and event mesh deployments.

As it is evident in Figure 6-5 and Figure 6-6, Apache EventMesh server is the component which uses the greatest amount of CPU, followed by Apache RocketMQ broker, while the service mesh proxies consume just a few amount of CPU, negligible with respect to the event mesh deployment. Regarding the memory consumption showed in Figure 6-7 and Figure 6-8, the Apache RocketMQ broker is the component which uses the largest amount, requiring more than 2GB. Trying to reduce the requested memory range of the JVM and setting limits below 2GB in the Kubernetes `Deployment` resource results in an Out-of-Memory error, as shown below.

```
$ kubectl -n eventmesh get pods
NAME                                      READY   STATUS      RESTARTS   AGE
eventmesh-server-694b54f9c9-59b6k         1/1     Running     0          100s
rocketmq-broker-master-78b68fb4b9-grhnf   0/1     OOMKilled   4          102s
rocketmq-console-admin-79546b9cdc-6dh7t   1/1     Running     0          102s
rocketmq-name-service-7bfc679884-n2nnf    1/1     Running     0          104s
```

Code 6-5. Apache RocketMQ broker failure when the maximum memory is under 2GB.

On the other hand, it is interesting to focus specifically on service meshes, which cannot clearly be seen in the previous graphs because they use far less memory than the Apache RocketMQ broker. It is noticeable from Figure 6-9 and Figure 6-10, that

the Envoy proxy used by istio requires more memory (about 40MB) with respect to linkerd2-proxy (less than 5MB). This result confirms what was stated by the Linkerd developers about the better resource consumption of their proxy with respect to Envoy [109].

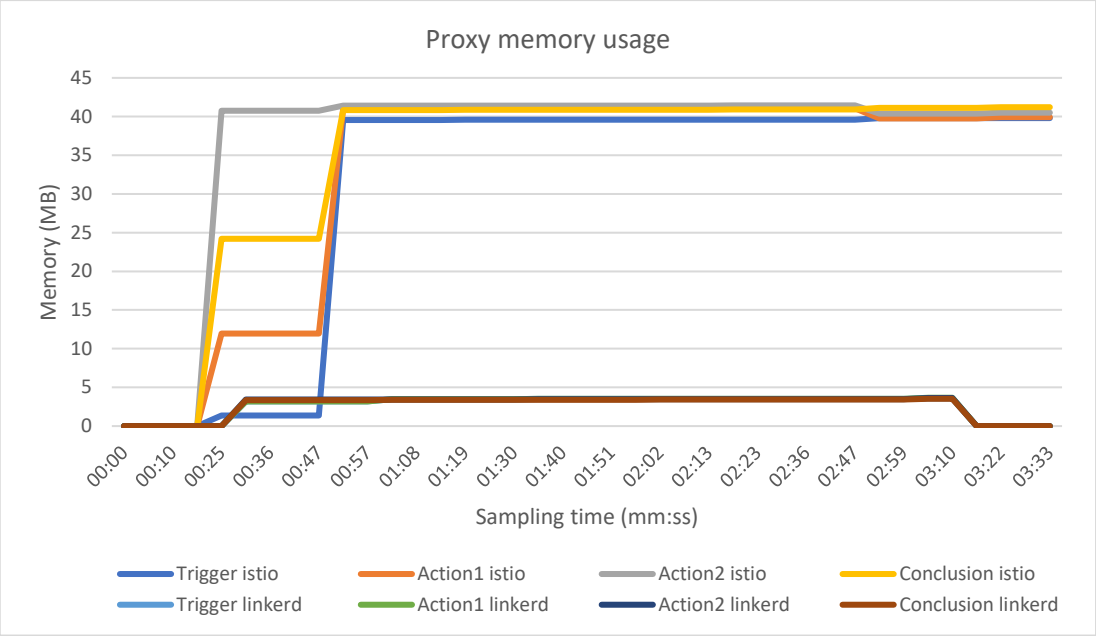


Figure 6-9. The graph shows the memory usage difference between istio and Linkerd.

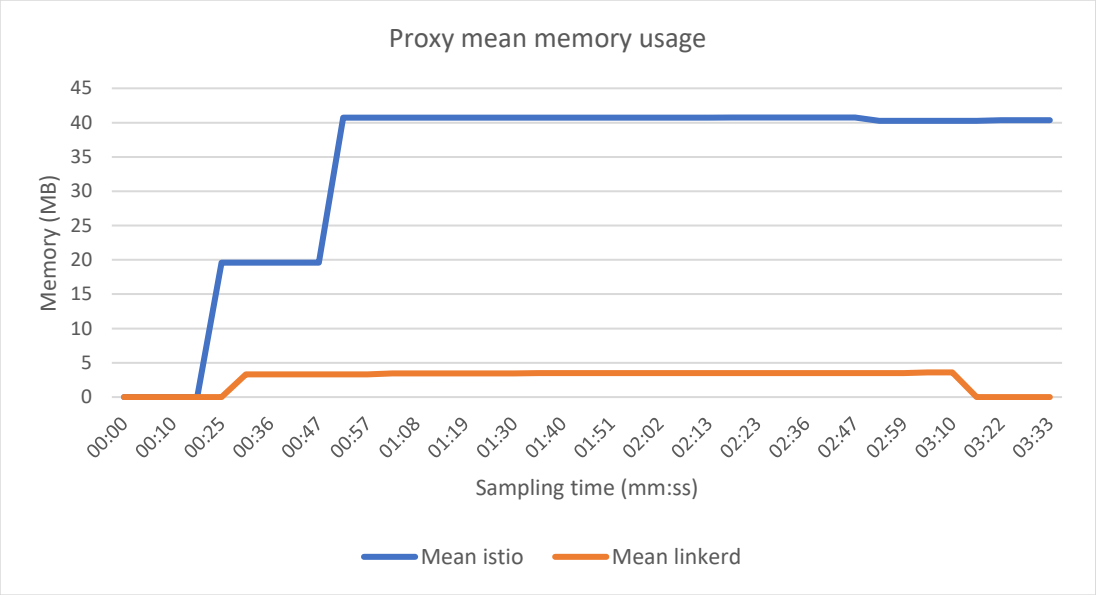


Figure 6-10. The graph summarizes the difference between the two proxy implementation showing the mean memory consumption.

6.2.1.3 MESSAGE LATENCIES

Before proceeding with the incremental load tests, we want to show now the latencies of messages measured at the ends of the workflow application.

As it is possible to notice in the Figure 6-11, there is a significant message latency for the first 200 messages with a serrated trend in the case of event mesh. Identifying the exact cause of this trend is difficult, however we can hypothesize that the Apache RocketMQ broker keeps the messages in a cache, sending them together subsequently to consumers, gradually adapting the cache to the number of requests per second.

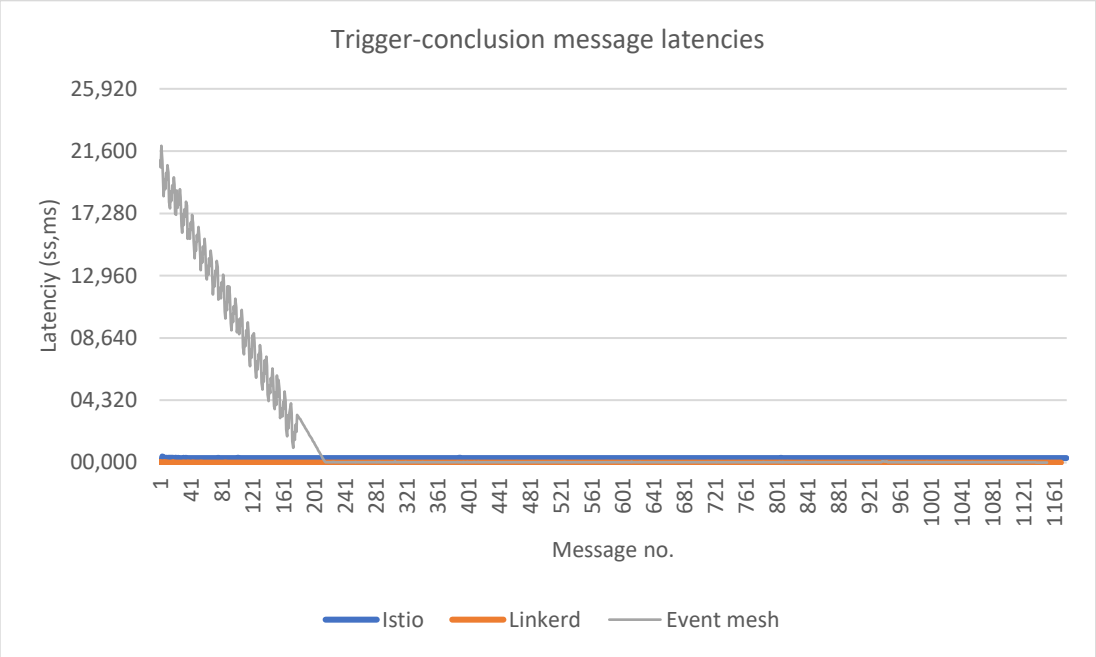


Figure 6-11. The graph shows the message latencies for each mesh.

Looking instead just at the service mesh proxies in Figure 6-12, it is possible to notice that the delay in which the messages incur, although always contained (less than half a second), is greater for istio Envoy (about 320 milliseconds) with respect to linkerd2-proxy (between 5 and 15 milliseconds).

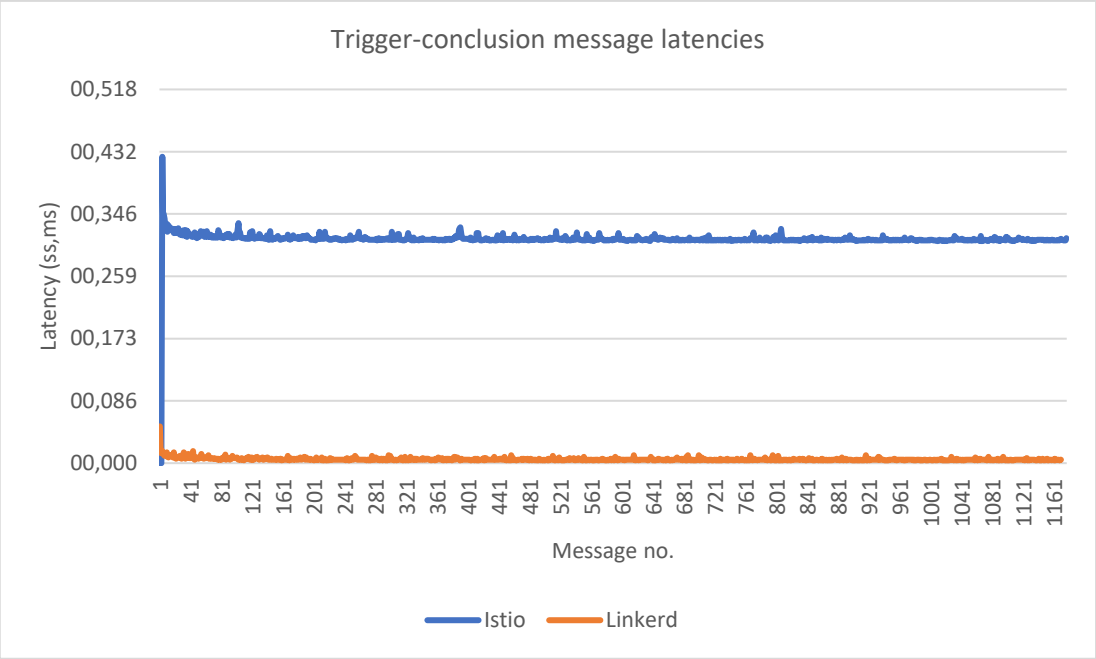


Figure 6-12. The graph compares the message latencies only between the two service mesh implementations used in our project.

6.2.2 RESULTS OF TESTS WITH INCREMENTAL LOAD

We will now illustrate the tests carried out with incremental load following the same order used for the description of the tests with constant load.

6.2.2.1 WORKFLOW APPLICATION

The tests for the workflow application confirm what we already saw in the constant load case. In particular, it is possible to observe in Figure 6-13 and Figure 6-14 that the CPU consumption is in general low while the Figure 6-15 and Figure 6-16 show the greater the memory usage of the event mesh deployment with respect to the one required by the components used to test the service meshes. However, it is possible to notice that the synchronous applications increment their memory consumption as the number of requests per second increases, while the event mesh applications, which rely on the Apache EventMesh SDK, require almost the same amount of memory. The final peaks are due to the boot of the Java Spring application used to gather metric information of each workflow application.

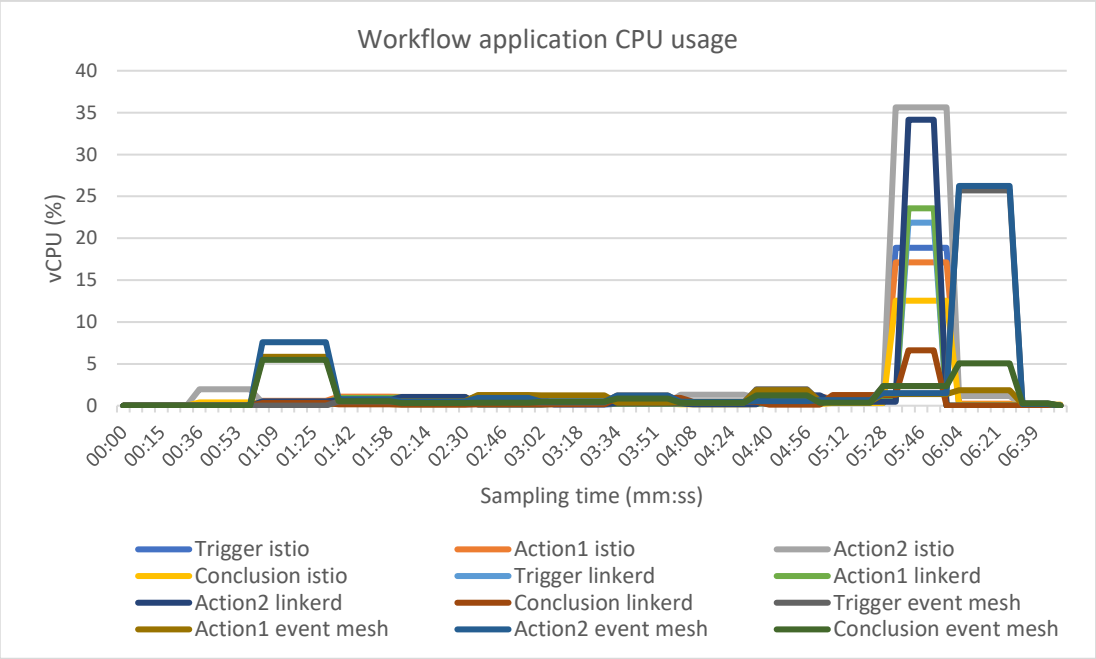


Figure 6-13. The graph represents the CPU usage of each workflow component.

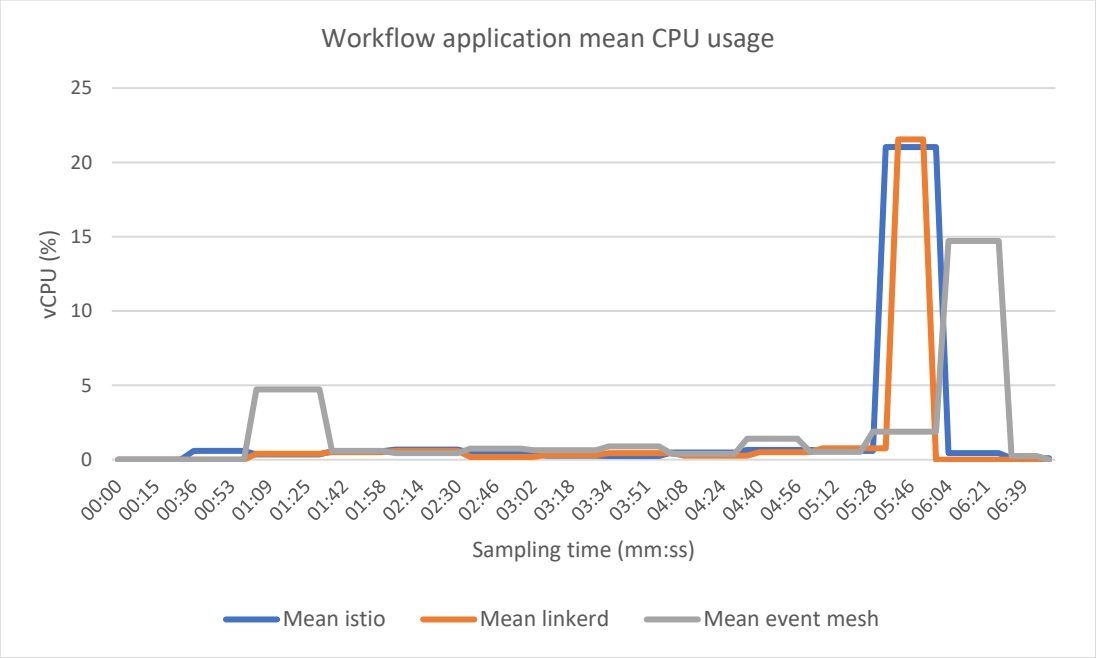


Figure 6-14. In the graph is represented the mean of the CPU usage of synchronous and asynchronous workflow application.

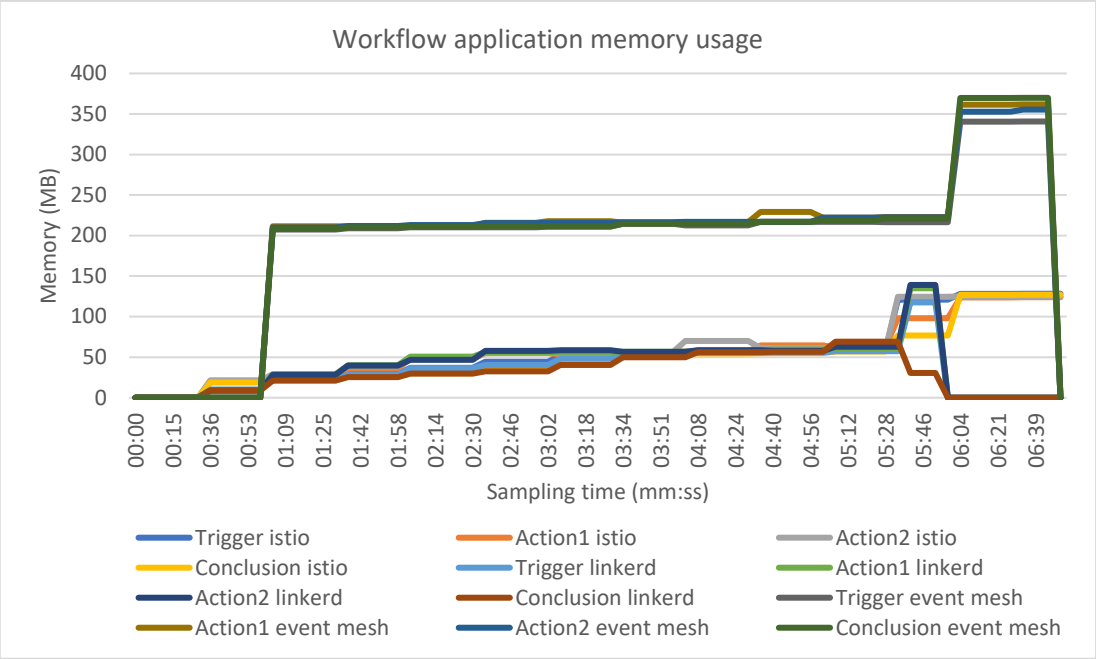


Figure 6-15. The graph represents the memory usage of each workflow application.

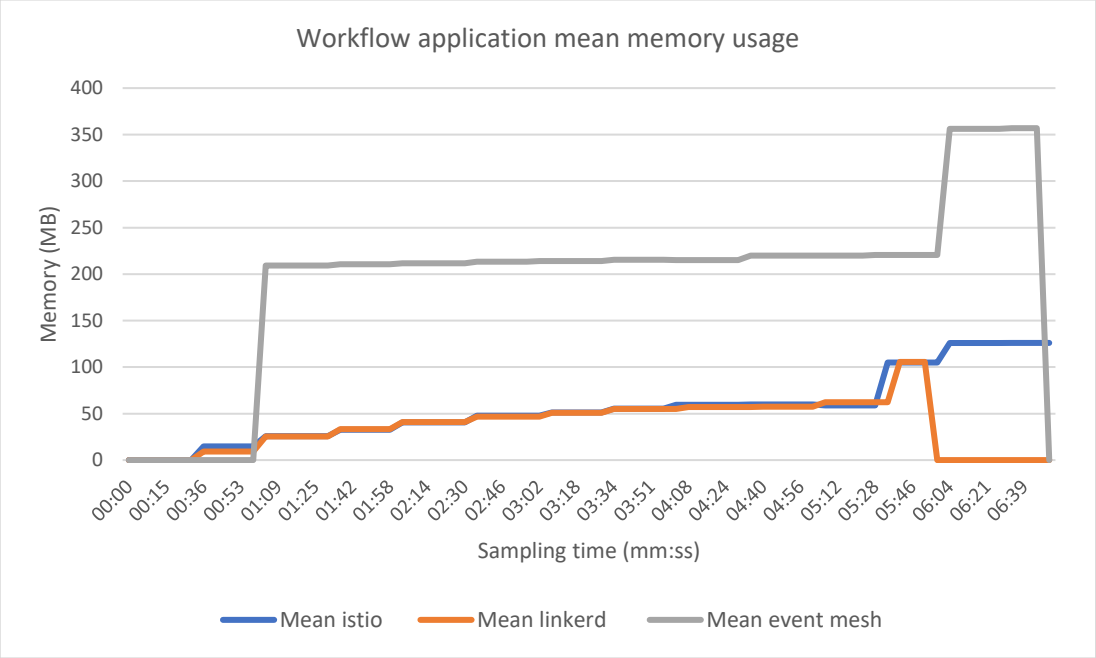


Figure 6-16. The graph shows the trend of the memory usage of the different meshes.

6.2.2.2 *INFRASTRUCTURAL COMPONENTS*

In the case of the infrastructural components, it is possible to notice that it is really similar to the constant load case: the Figure 6-17 and Figure 6-18 show that the CPU

consumption is almost below 5%, with the highest usage from the Apache EventMesh server, followed by the Apache RocketMQ broker, while the service mesh proxies consume a very low percentage. The Figure 6-19 and Figure 6-20 show that the memory consumption of the Apache RocketMQ broker is significantly higher (more than 2GB) than the service mesh proxies (from 5 to 40 MB) and the one required by Apache EventMesh (from about 235 to 359 MB).

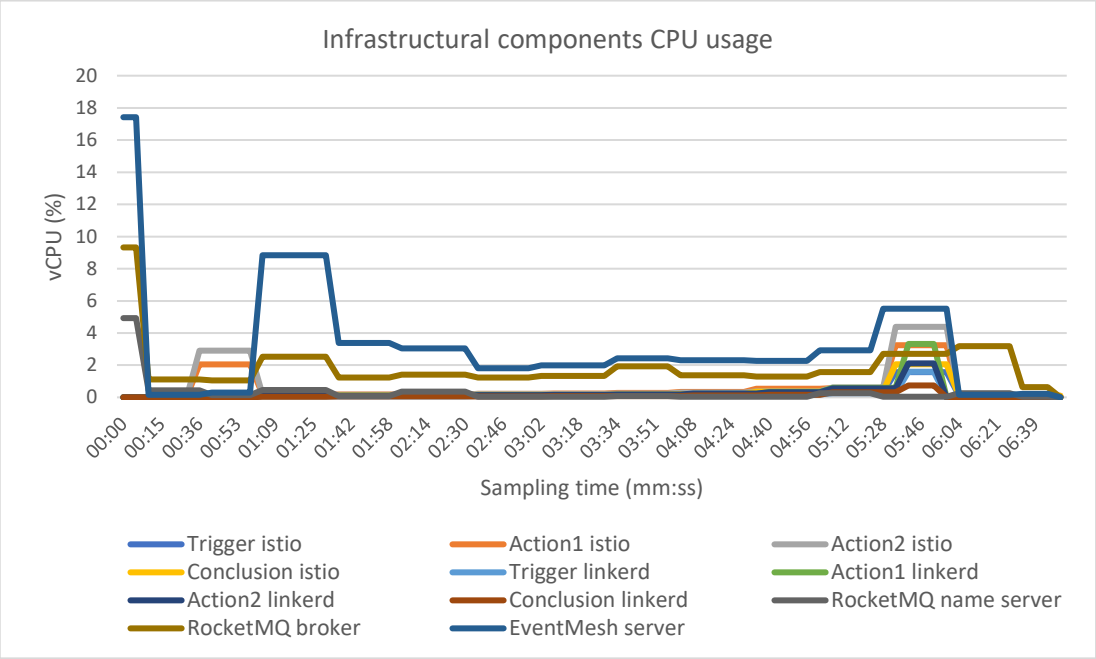


Figure 6-17. In the graph is illustrated the CPU usage for each different mesh component.

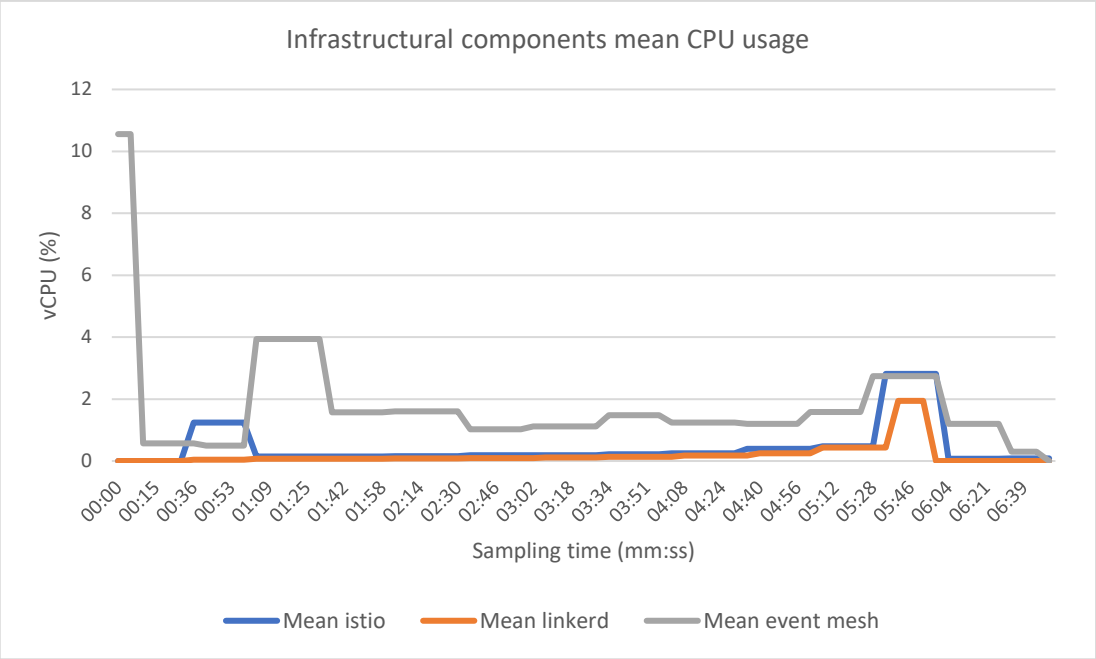


Figure 6-18. The graph illustrates the mean CPU usage of the mesh components.

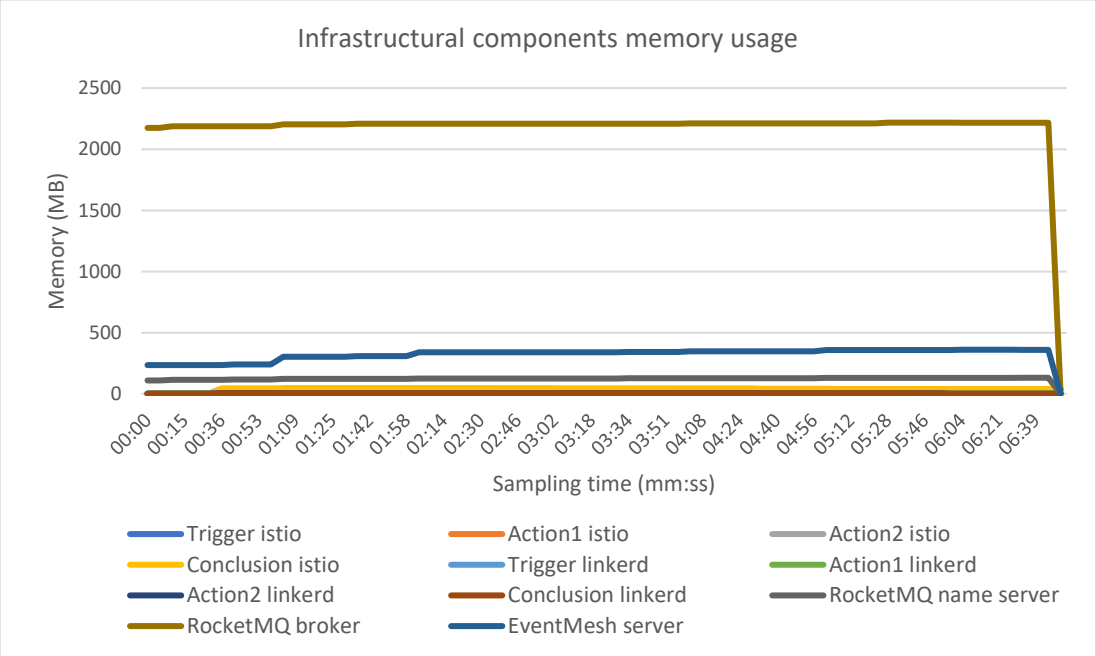


Figure 6-19. The graph shows the memory usage for each mesh component.

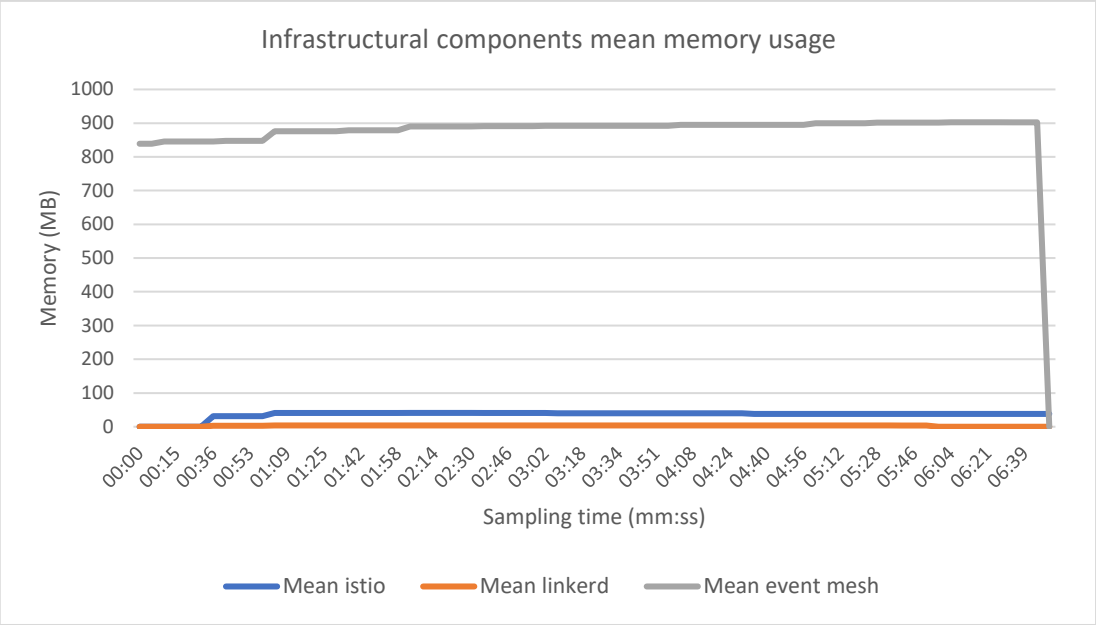


Figure 6-20. In the graph is represented the mean memory usage for the service and event mesh.

6.2.2.3 *MESSAGE LATENCIES*

The message latencies of the increasing load test in the Figure 6-21 show that the Apache EventMesh broker reduces the latency from a maximum of about 19 seconds to under one second in just 20 messages (ten times less than the constant load case).

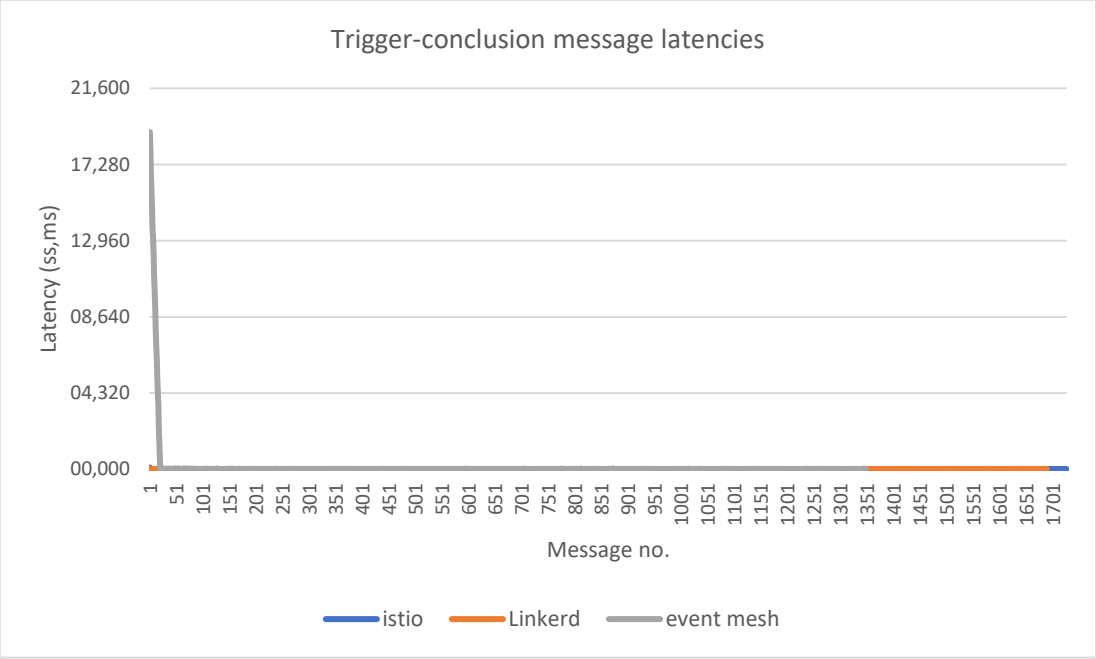


Figure 6-21. The graph represents the latencies for service and event meshes.

6.3 MULTI-CLUSTER TESTS

After the report of the tests conducted in a single cluster, we want to describe now a more advanced scenario, which consists of two GKE clusters, each of which has the same configuration of the single cluster case (16 vCPUs and 64GB in total).

Like for the single cluster, we repeated also in this case the two different tests, one with constant and the other with incremental load, leaving the Trigger and the Action components in the first cluster and moving the Conclusion to the second cluster. For the setup of the infrastructural components, we used the same configuration as it was described in the previous chapter.

6.3.1 RESULTS OF TESTS WITH CONSTANT LOAD

Starting from the constant load, we will proceed, as in the single cluster case, with the analysis of the use of the resources of the workflow application and of the infrastructural components, to then move on to the examination of the latency of each single message as the number of messages sent per second varies.

6.3.1.1 WORKFLOW APPLICATION

The workflow application follows about the same trend as in the single cluster case with more memory usage in the case of the event mesh workflow application. In particular, it is possible to observe in Figure 6-22 and Figure 6-23 that the workflow application designed to work with the event mesh consumes about the same CPU than the synchronous application, except for the peaks at the start and the end of the graphs. However, the analysis of the memory consumption in Figure 6-24 and Figure 6-25 shows that the memory required by the asynchronous applications is significantly higher (more than 200MB) than the one required by the synchronous application (around 50MB). The final peaks are related to the boot of the Java Spring application used to expose test results of each component to our metrics system.

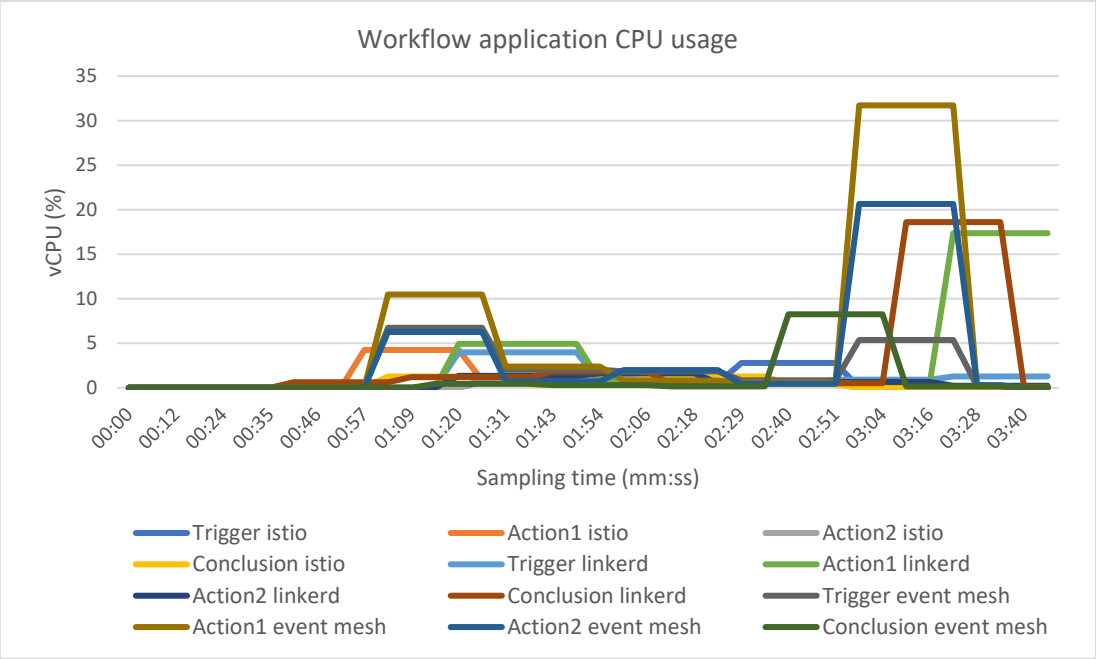


Figure 6-22. The graph represents the CPU usage for each workflow component.

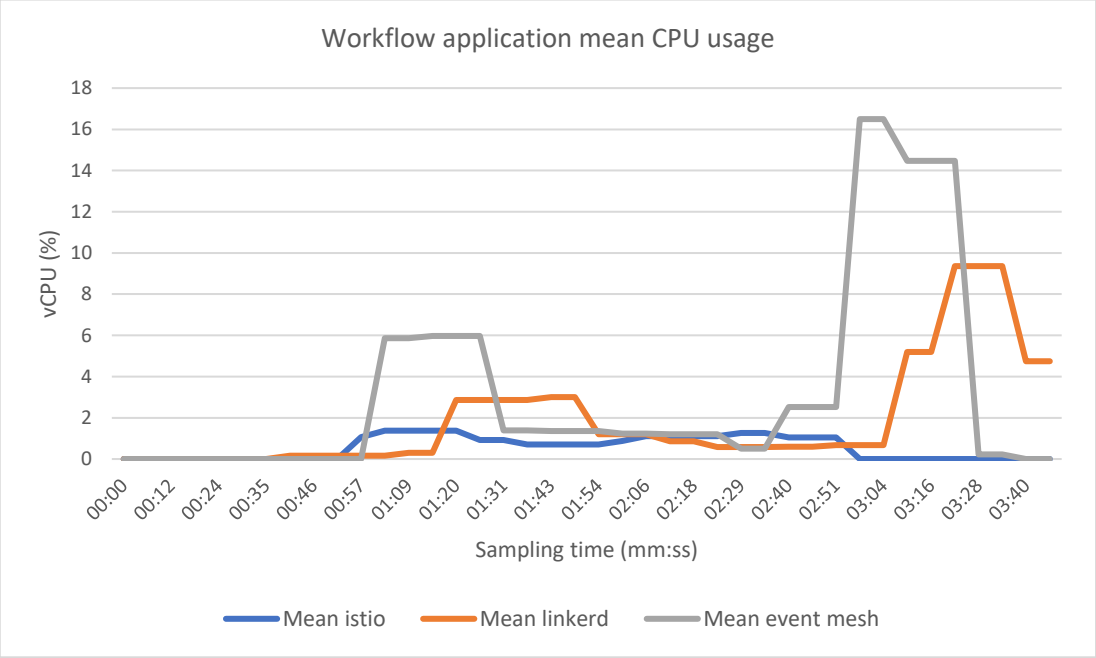


Figure 6-23. The graph represents the mean CPU usage of the mesh components.

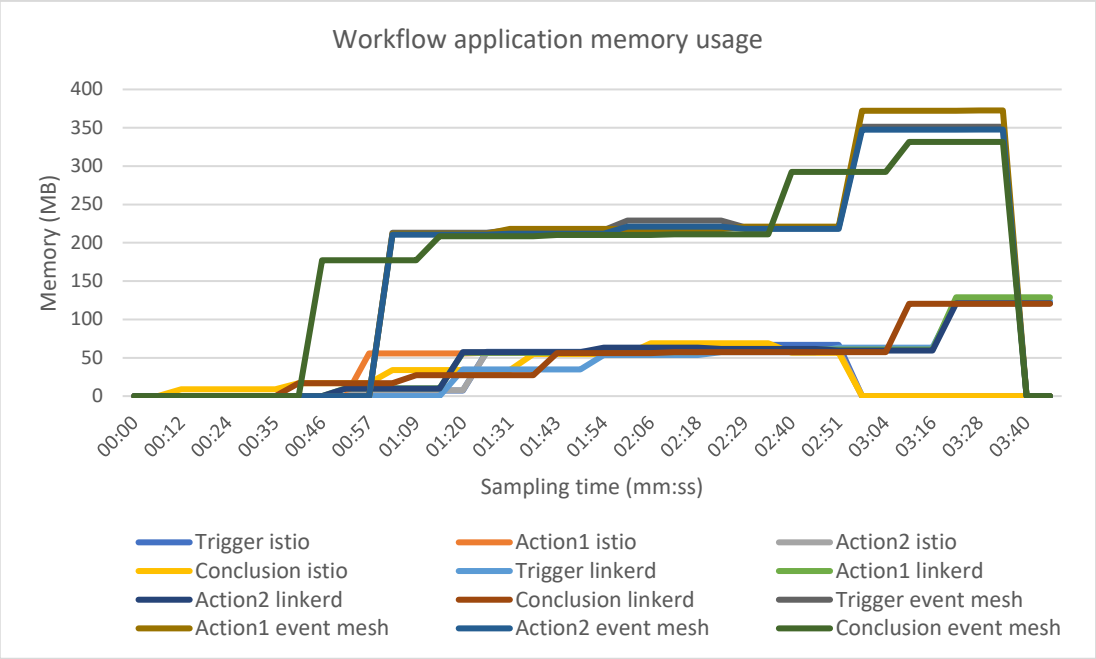


Figure 6-24. In the graph is represented the memory usage for each workflow component.

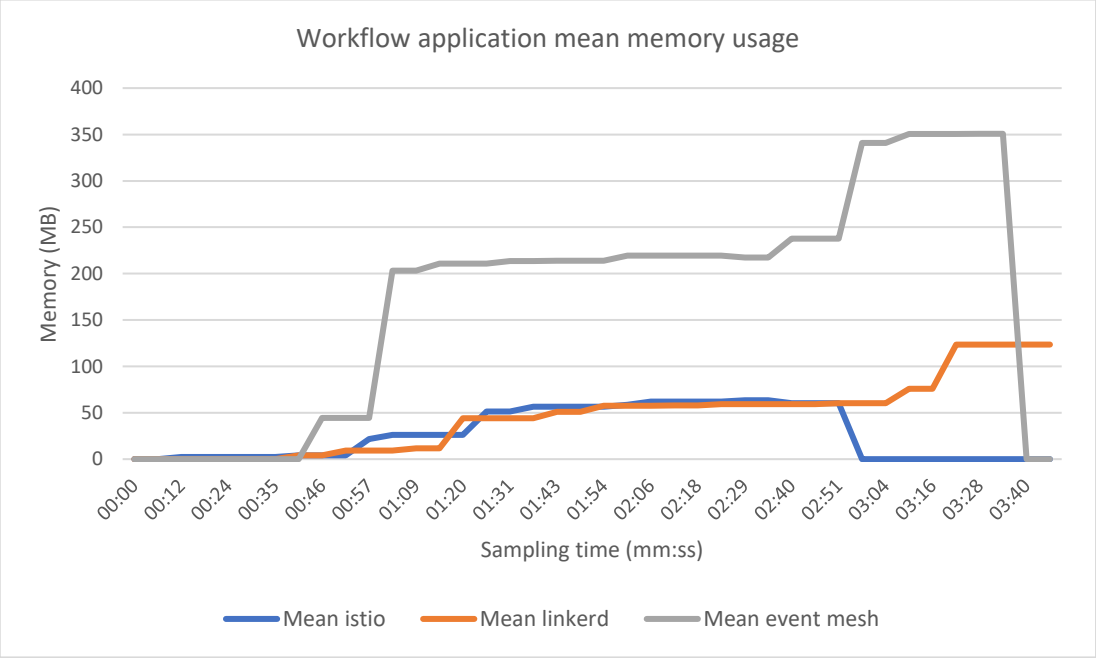


Figure 6-25. Graph representing the mean memory usage for service and event mesh deployments.

6.3.1.2 *INFRASTRUCTURAL COMPONENTS*

From the analysis of the Figure 6-26 and Figure 6-27, regarding the infrastructural components, it is possible to notice that, in the event mesh case, the CPU usage is

slightly higher in the first cluster, since there are three workflow components relying on the EventMesh server. On the other hand, the CPU required by the service mesh proxies is always significantly lower than the event mesh deployment. Instead, the Figure 6-28 and Figure 6-29 show that the memory usage of the event mesh components is very high and it is almost the same for both clusters, with the first being just a little higher than in the second one.

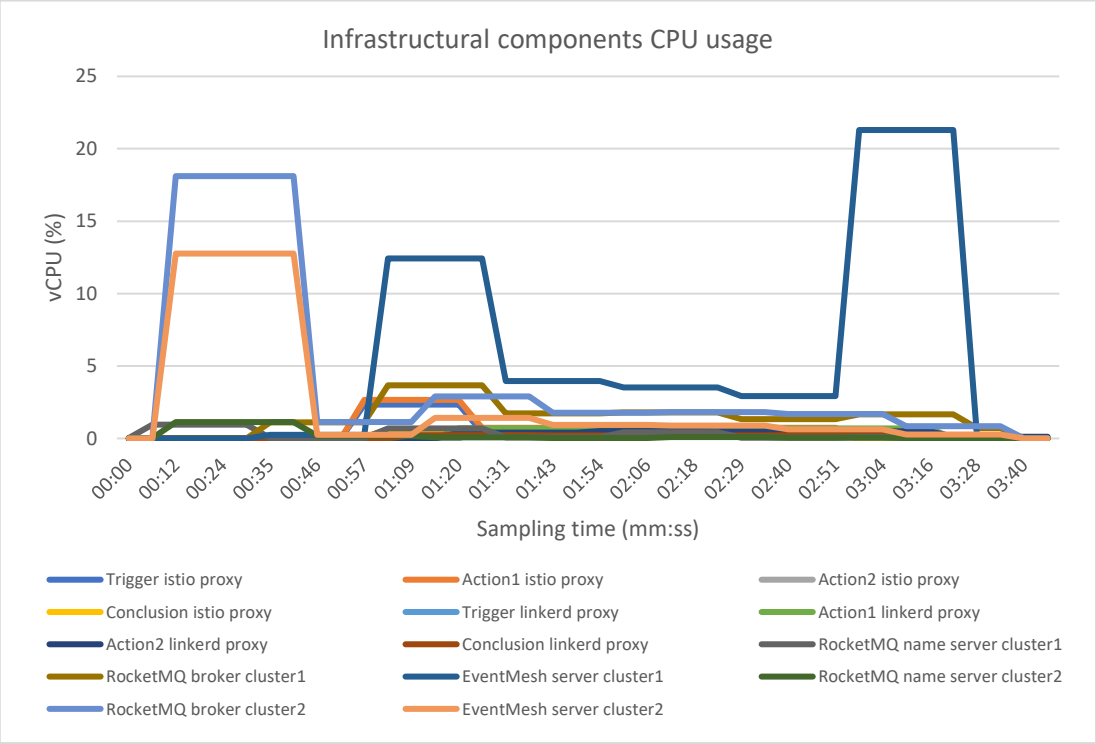


Figure 6-26. The graph represents the CPU usage of each mesh component.

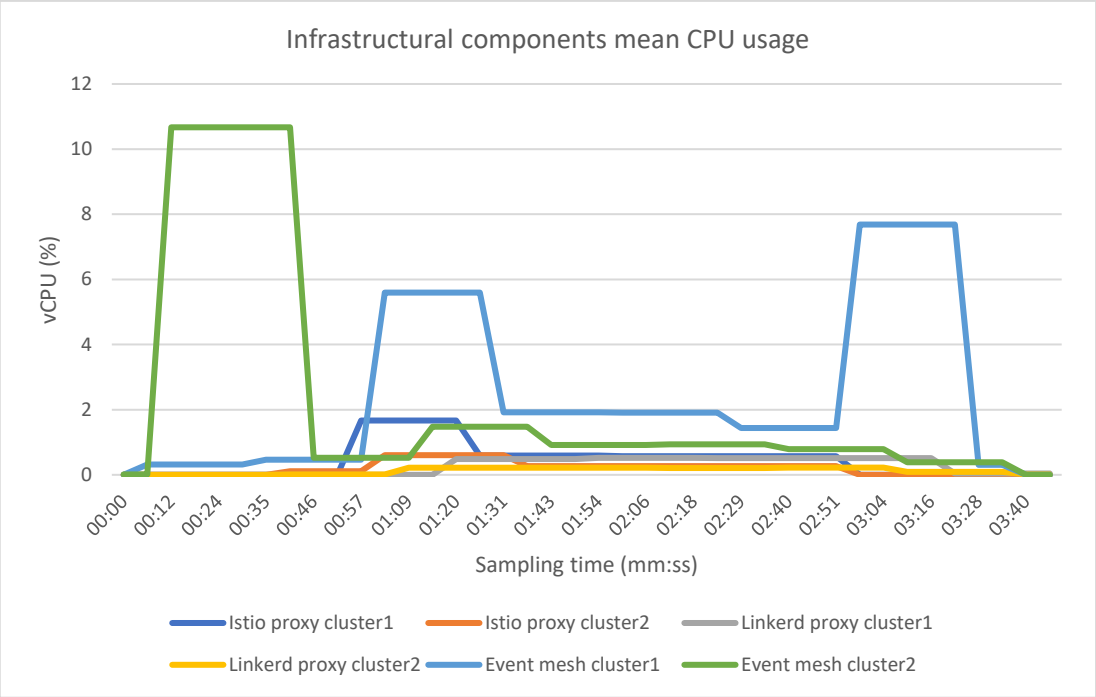


Figure 6-27. In the graph is represented the mean CPU usage of the service and event mesh components.

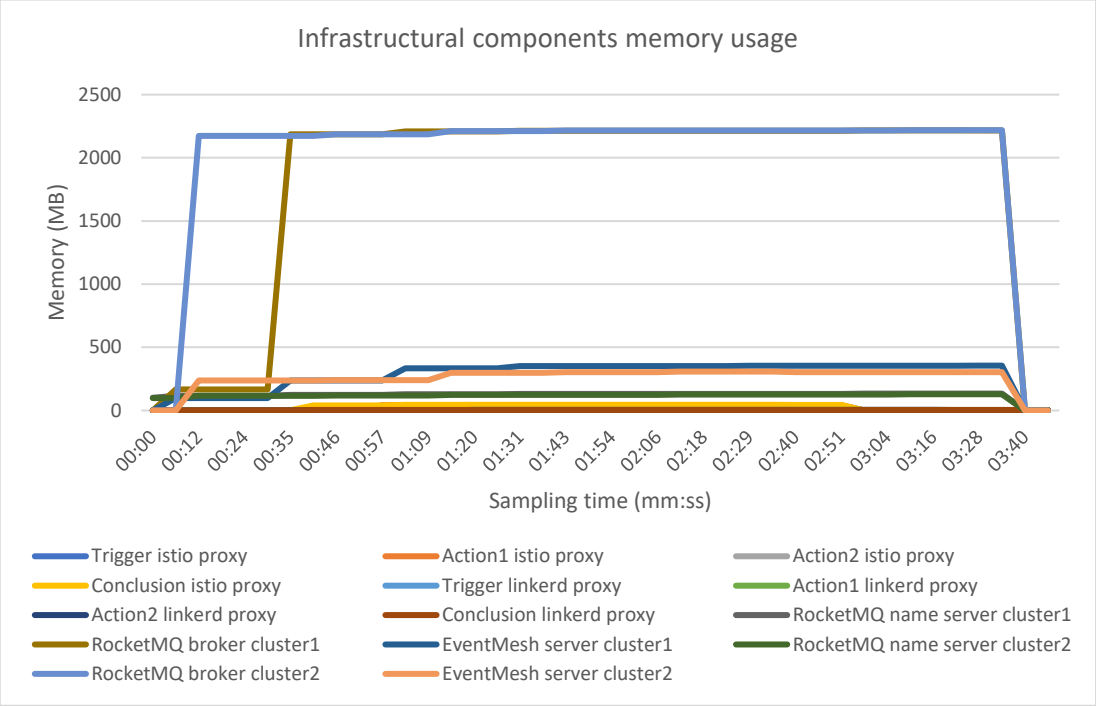


Figure 6-28. Graph representing the memory usage of the mesh components for each cluster.

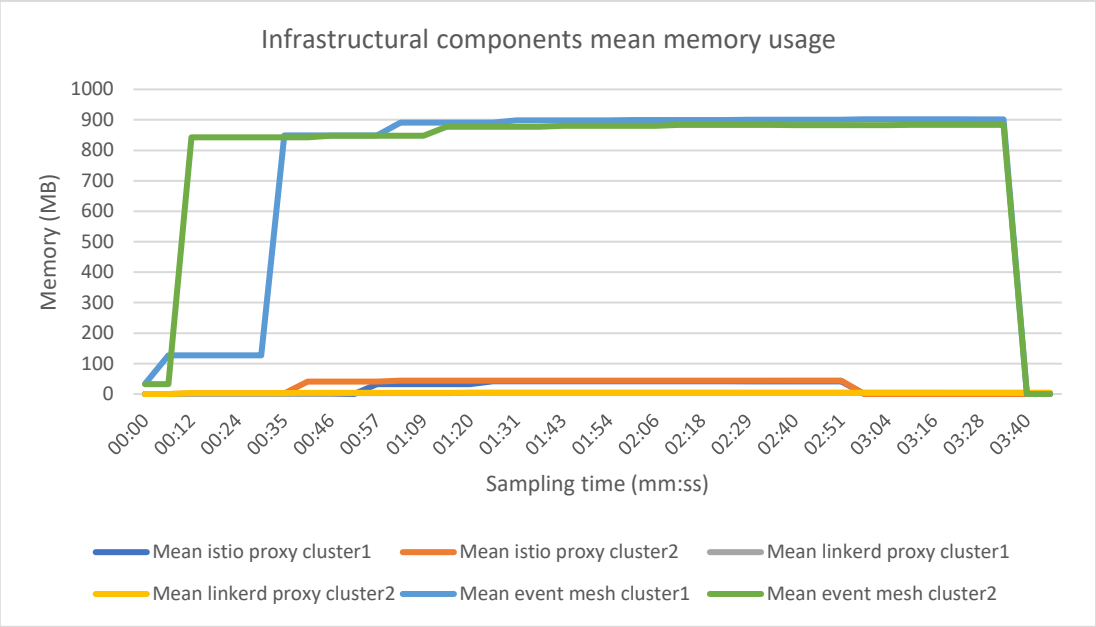


Figure 6-29. The graph shows the mean memory usage of the mesh components for each cluster.

Regarding the service mesh deployments, it is possible to notice their low memory usage when compared to the one consumed by the event mesh deployment. Focusing only on service mesh proxies, the Figure 6-30 confirms that each Envoy proxy uses much more memory (more then 40MB) with respect to linkerd2-proxy (about 3MB), as we already showed in the single cluster tests.

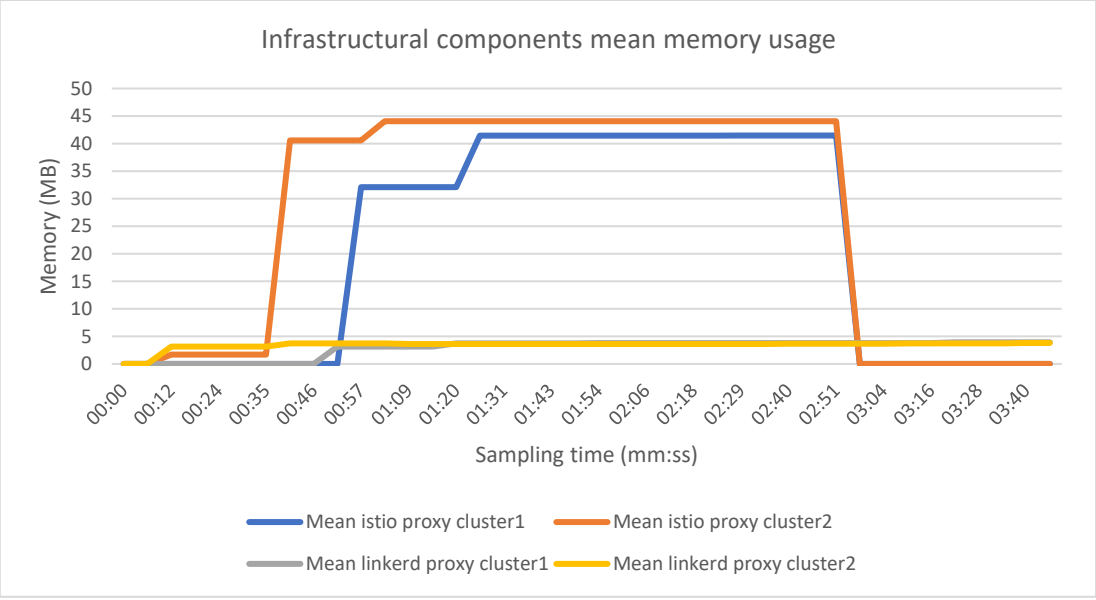


Figure 6-30. The graph shows a comparison of the memory usage between istio and linkerd proxies.

6.3.1.3 *MESSAGE LATENCIES*

When looking at the the latency for each message exchanged along the chain between the Trigger and the Conclusion, while istio keeps latency less than about 200 milliseconds and Linkerd around 10 milliseconds, as illustrated in Figure 6-32, Apache EventMesh is not capable of forwarding all the messages through the workflow application chain, and in general the latencies are really high, from about 2 seconds to more than 25 seconds, as shown in Figure 6-31. Like in the single cluster case, it is possible to recognize a segmented pattern for the latencies in the event mesh case.

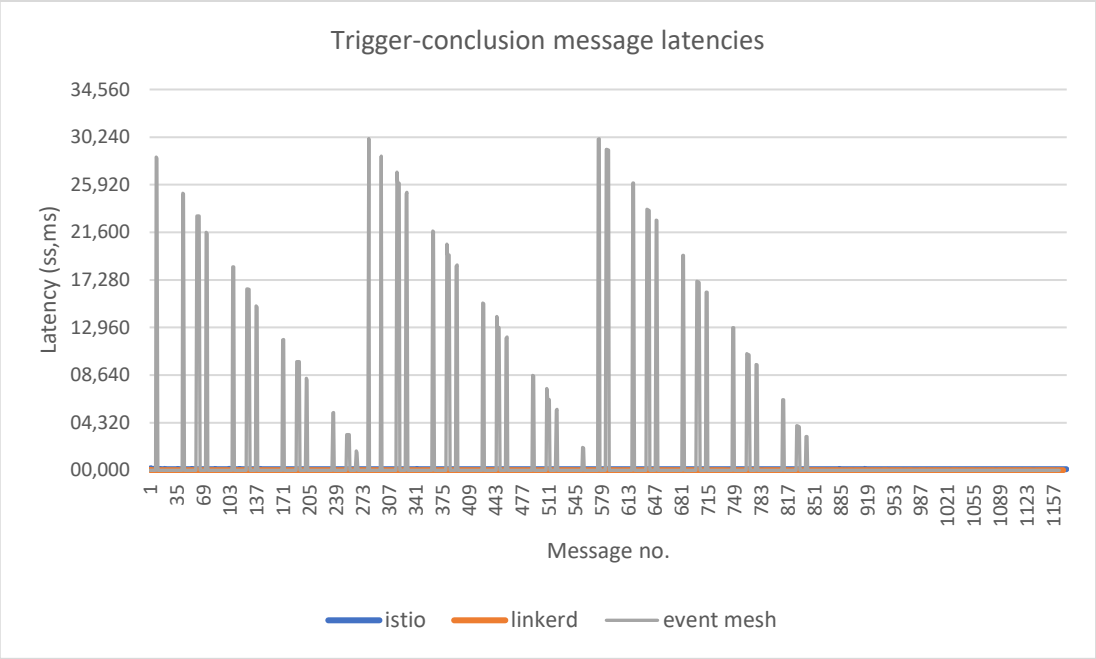


Figure 6-31. The graph represents the message latencies for the service and the event mesh deployments.

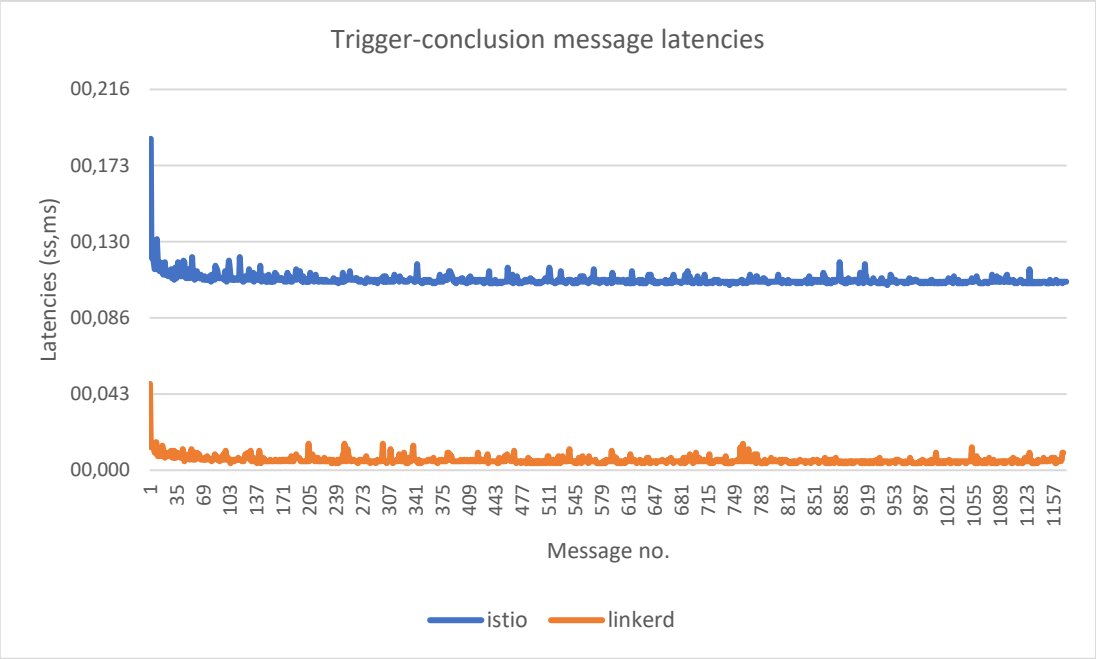


Figure 6-32. In the graph is represented the message latencies for service meshes.

6.3.2 RESULTS OF TESTS WITH INCREMENTAL LOAD

We will now analyze the tests carried out with incremental load still using the same order utilized for the description of the tests with constant load.

6.3.2.1 WORKFLOW APPLICATION

Looking at the Figure 6-33 and Figure 6-34, it is possible to notice, with respect to all the previous cases analyzed, that the asynchronous applications use less CPU. As we will see in the next analysis, it is due to the fact that only few messages reached the Conclusion component. Instead, it is possible to observe in the Figure 6-35 and Figure 6-36 that the memory usage remain higher for the event mesh case with respect to the synchronous applications, which require from about 17 to 55 MB.

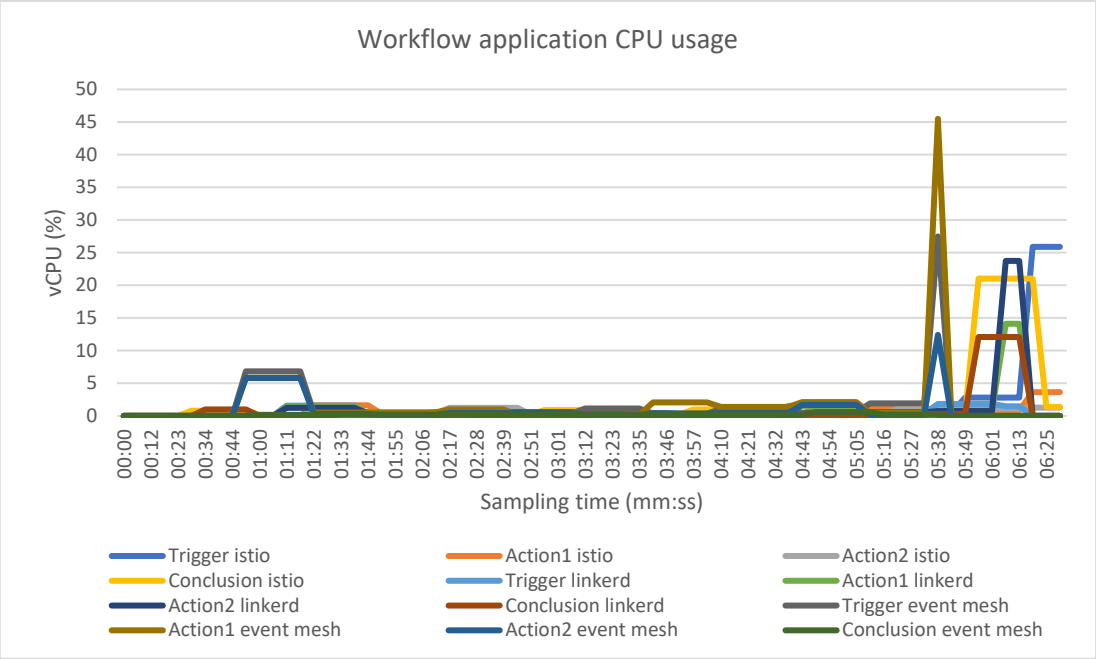


Figure 6-33. The graph represents the CPU usage of each workflow component.

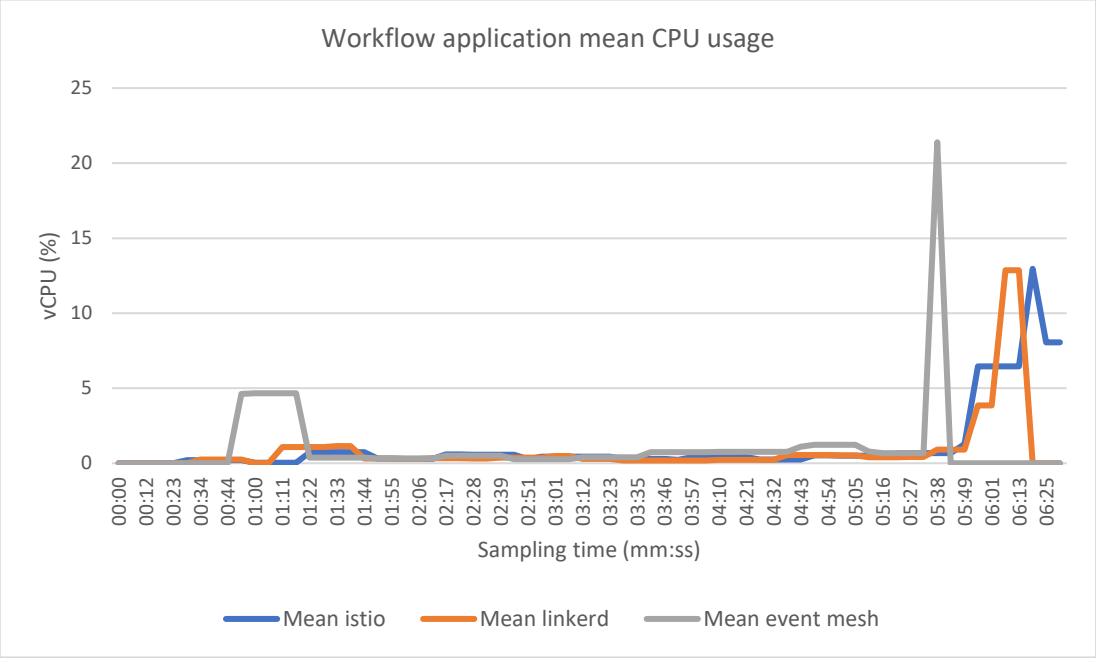


Figure 6-34. In the graph is represented the mean CPU usage for the synchronous and asynchronous workflow applications.

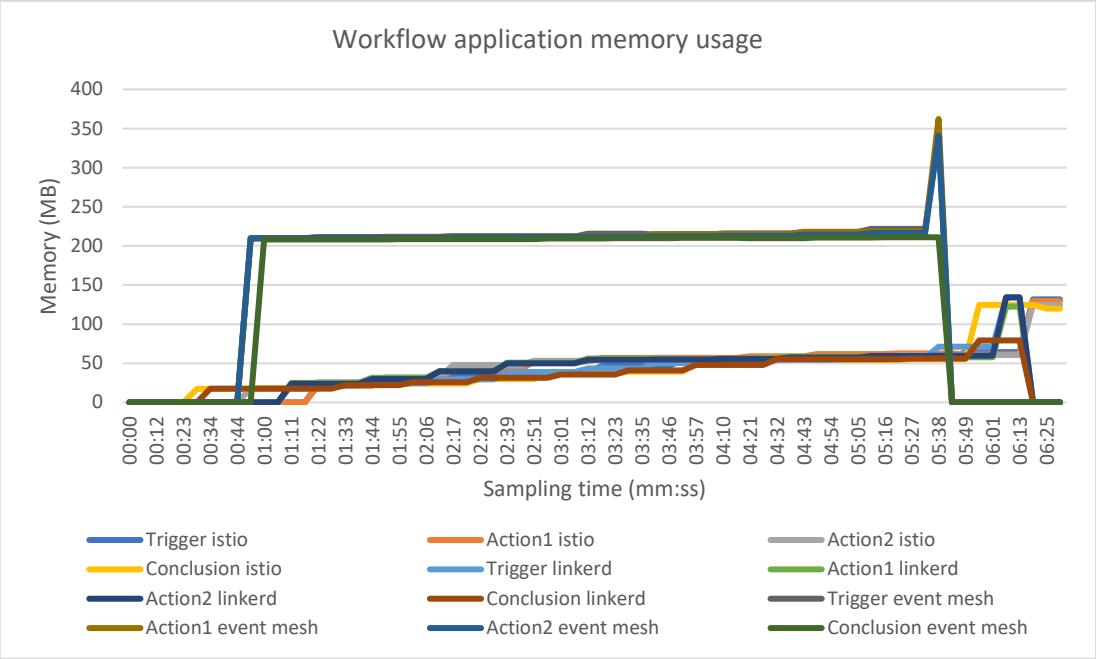


Figure 6-35. The graph represents the memory consumption of the workflow components.

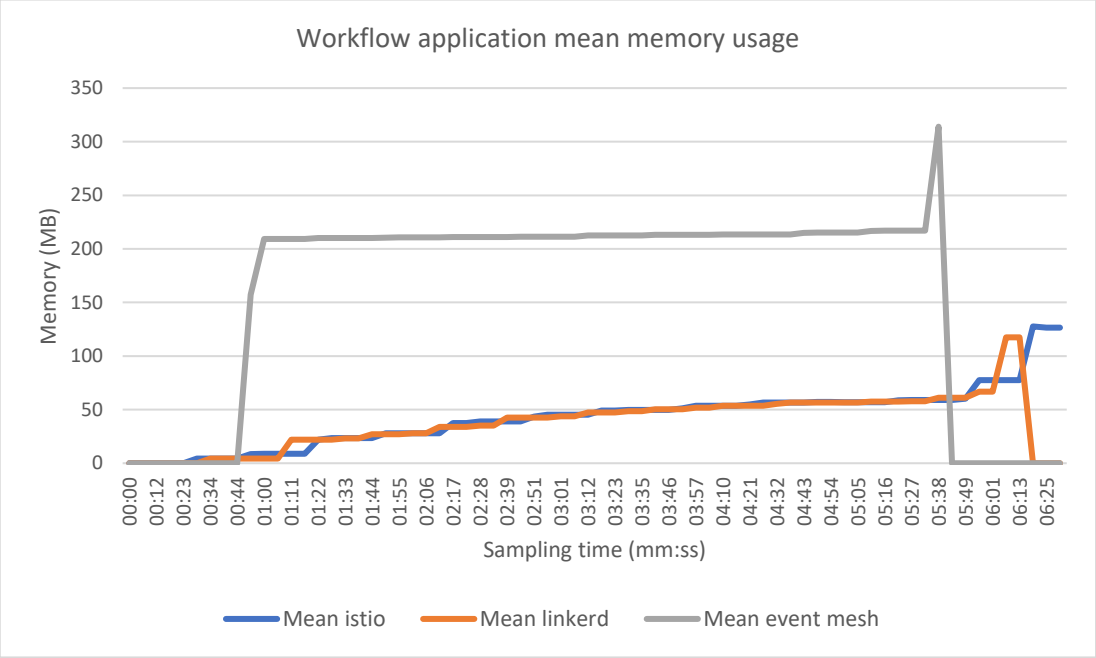


Figure 6-36. The graph shows the mean memory usage for the synchronous and asynchronous workflow applications.

6.3.2.2 *INFRASTRUCTURAL COMPONENTS*

The visualizations provided by the infrastructural components resource usage confirm what we observed above: in Figure 6-37 and Figure 6-38 it is possible to notice that the event mesh deployment consumes less CPU than all the previous tests, while in the Figure 6-39 and Figure 6-40 the memory usage of the Apache RocketMQ brokers is always very high with respect to the service mesh proxies.

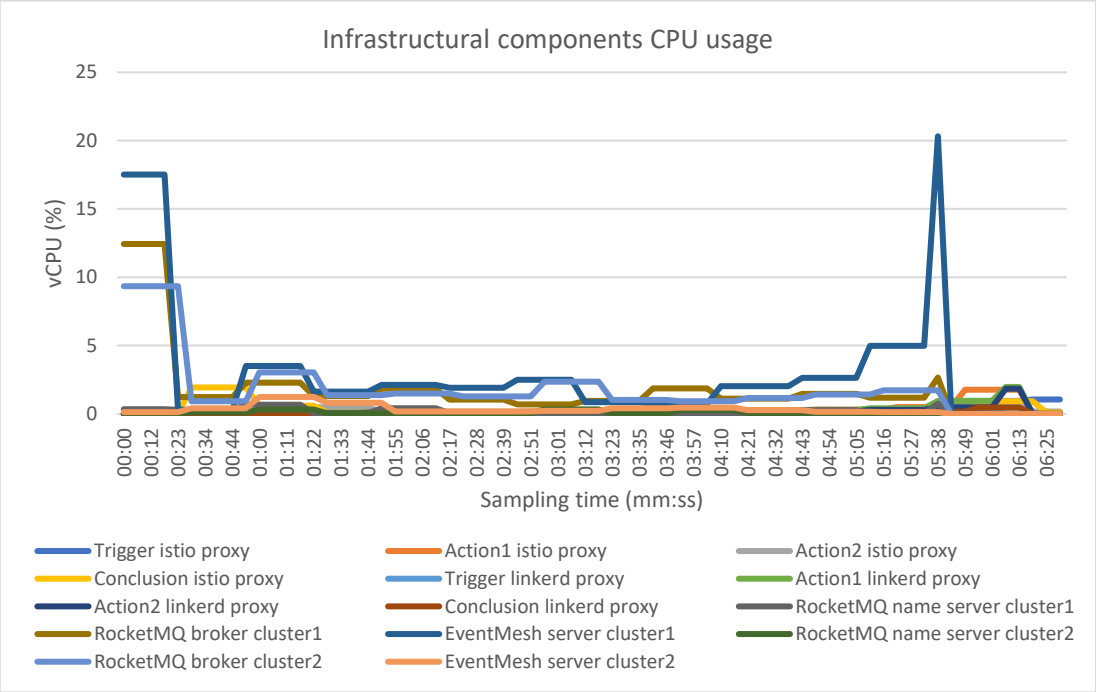


Figure 6-37. The graph shows the CPU usage of each mesh component.

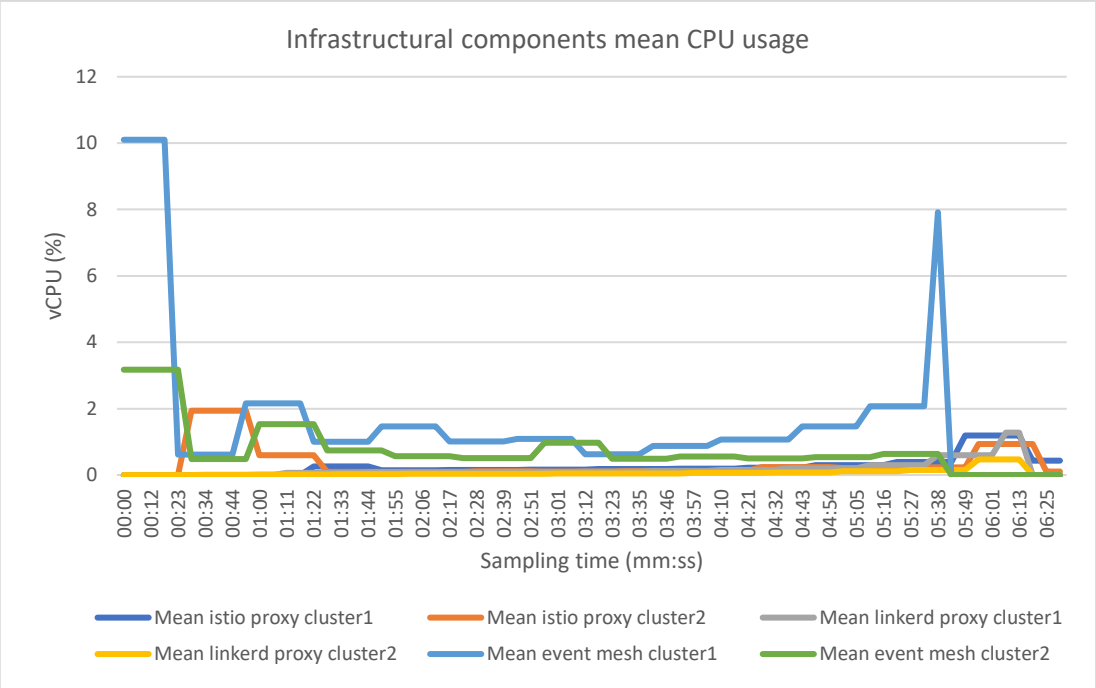


Figure 6-38. The graph represents the mean CPU usage of mesh components for each cluster.

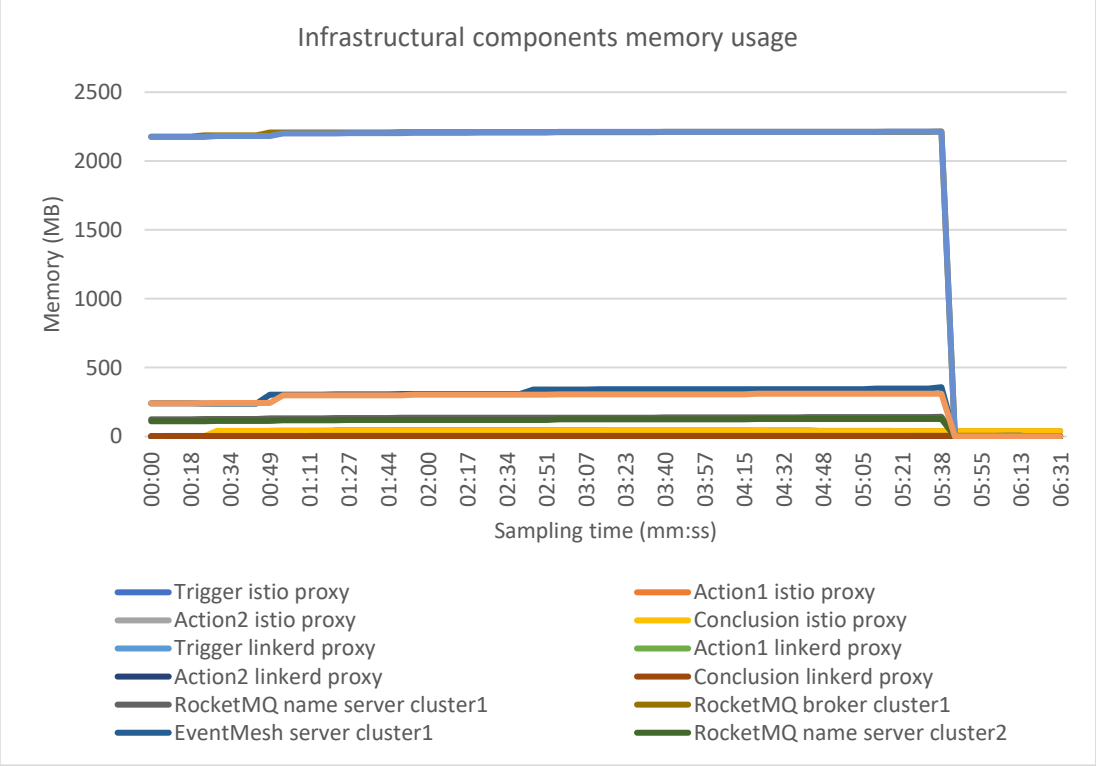


Figure 6-39. The graph represents the memory usage of each mesh component.

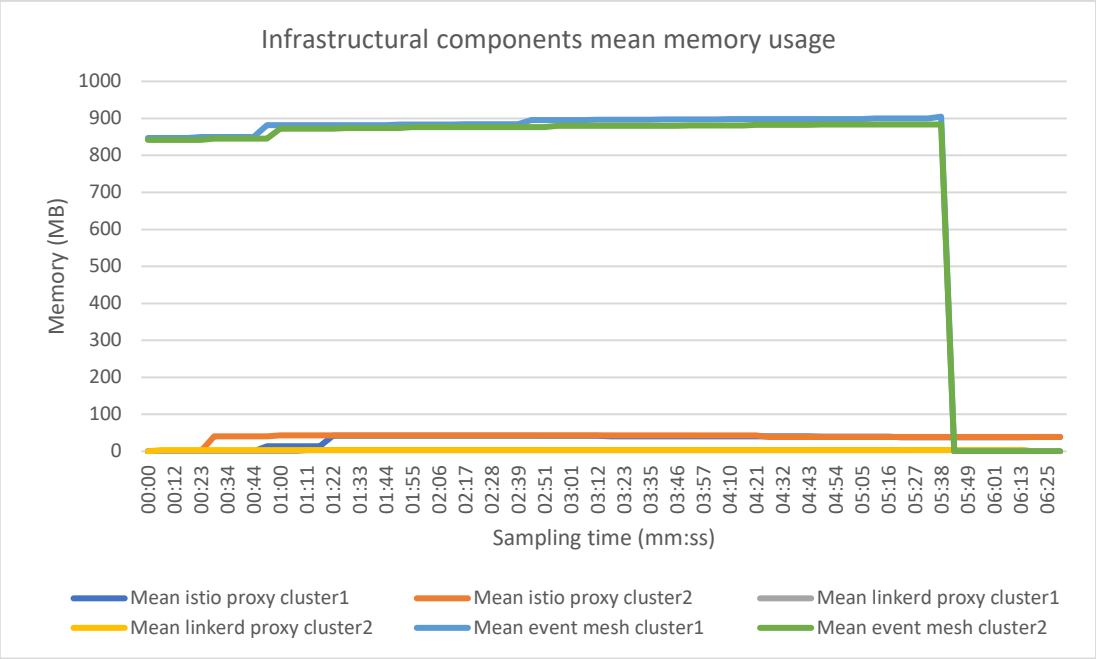


Figure 6-40. The graph represents the mean memory usage of each mesh component for each cluster.

6.3.2.3 MESSAGE LATENCIES

The analysis of the latency for each message in Figure 6-41 shows clearly the malfunctioning of the event mesh technology, comprising both Apache EventMesh and Apache RocketMQ, with only a few messages reaching their destination and with the highest latencies we observed in all the tests.

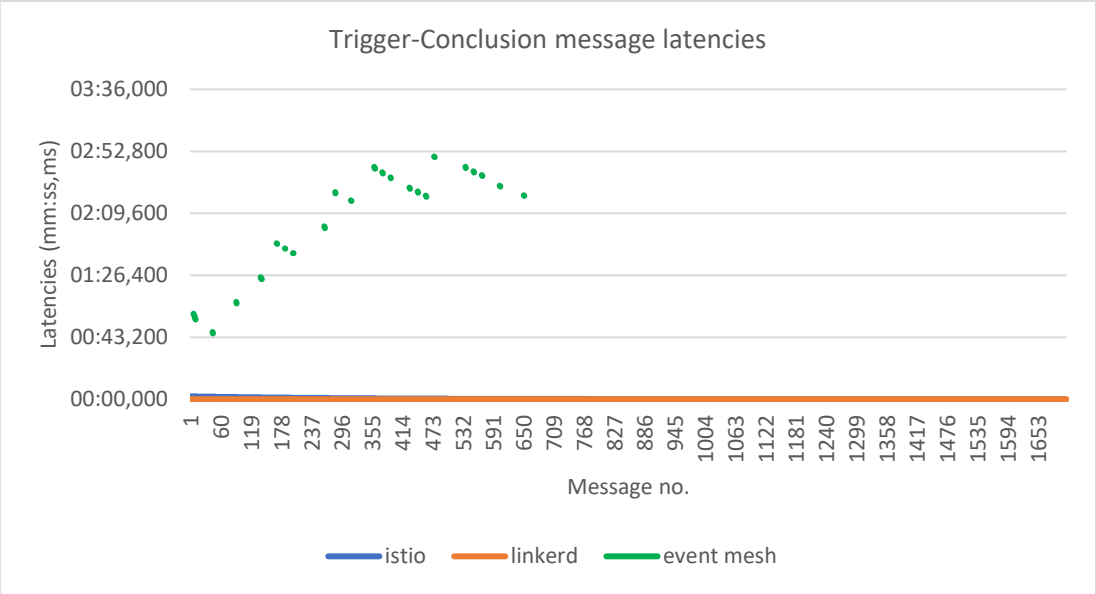


Figure 6-41. The graph shows the message latencies for each mesh.

Finally, in Figure 6-42, which is focused only on service meshes, istio shows an interesting trend as the number of messages per second increases: it starts with high latencies, more than 2 seconds, slowly optimizing its performance. On the other hand, Linkerd performs impeccably, with no messages lost and with latencies in the range of about 10 to 20 milliseconds.

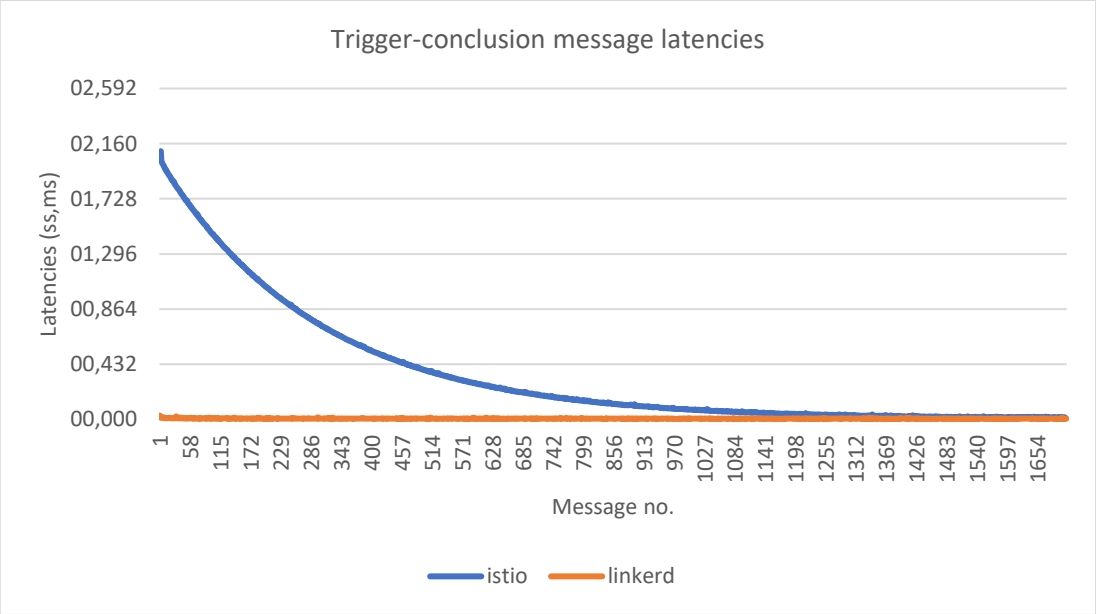


Figure 6-42. The graph represents the message latencies for service meshes.

6.4 FINAL CONSIDERATIONS

In conclusion, the experiments carried out showed the maturity of the service mesh, highlighting, on the contrary, the strong limitations of Apache EventMesh.

Istio and Linkerd make it possible to create a stable and efficient system to support synchronous communication. In particular, Linkerd has demonstrated that it requires much less memory than istio, which allows it to better exploit the resources of a cluster. In addition, in all the tests, Linkerd never lost a message, even in the multi-cluster scenario, in which it showed, when tested with increasing messages per second, its high performace, keeping latencies to the minimum. However, beyond the performance considerations, Linkerd showed a minor issue consisting in not allowing Kubernetes to capture logging output from each pod. In other words, the command `kubectl logs` did not output anything, with respect to the other meshes.

On the other hand, it became clear that Apache EventMesh is still an immature project, despite being based on Apache RocketMQ, which has existed in a stable form for some years. The latter, in particular, required much more resources than the single proxies of the service mesh: in fact, we noticed the large use of memory by the broker, which in a real business application would require huge resources, with a possible consequent increase in costs[26]. Furthermore, we have noticed in the multi-cluster case a significant decrease in performance together with the loss of many messages along the chain of workflow applications.

It must finally be considered that Apache EventMesh does not officially offer any tool to help automate the installation of the event mesh, as both istio and Linkerd do, but it is necessary to manually implement the entire mesh deployment process, adapting it to each specific environment. And this also applies to Apache RocketMQ.

[26] It should be noted that, although it uses more resources, the event mesh deployment is shared by multiple microservices, while in the case of service meshes, each microservice has associated a sidcear proxy. For this reason, only from the point of view of the resource usage, the deployment of event mesh could become convenient only if there are dozens of microservices in the system.

7 CONCLUSIONS

In the last few years, IT systems have been characterized by significant innovations, mostly driven by the growing cloud infrastructure and microservices adoption, which allow organizations to deliver new capabilities for their respective business. Modern application development on cloud infrastructure requires engineering people to solve old problems, such as latency, security, and network reliability, which can become very difficult to manage for fast growing business when there are many different services which can be autoscaled or can fail unexpectedly. In this scenario, mesh technologies emerged to support business application development providing connectivity, reliability, security, and observability at infrastructure level.

In particular, service mesh technologies are becoming more and more popular, and have gone through significant innovations and several new architecture trends, technology capabilities, and new projects have emerged in the ever evolving mesh space. On the other hand, event meshes are more recent, offering fewer options than service meshes, but are part of modern event processing which is fast becoming a foundation of today information society. Modern event processing emerged in the late 1990s, but it has been for many decades the foundation, just to name a few areas, for discrete event simulation, weather simulation and forecasting, networks and the internet, and all the manner of information gathering and communications. Since then, many companies have been developing and marketing event processing products, so that complex event processing has become an established market area for commercial applications.

Due to the importance of these technologies for enterprises, the work covered by the thesis had the aim of creating a microservices system to test workflow applications supported by some of these modern technologies. Our analysis first concerned the theoretical study of service, event, and data meshes, subsequently focusing on the examination of specific implementations. Finally, for the realization of our project we selected some technologies with particular characteristics and we compared them focusing mainly on the performance side, evaluating their functioning in a public cloud

environment within both a single Kubernetes cluster and in a multi-cluster scenario. In particular, in the field of service meshes, we used istio and Linkerd, given their open-source nature and given that these two technologies offer different data plane proxy implementations. Regarding event meshes, we used a recently developed project termed Apache EventMesh, which depends on Apache RocketMQ.

In our results, istio and Linkerd confirmed what we analyzed in the theory: they are performant and reliable mesh solutions. In a real business case with strict requirements of speed, reliability, and security, both service meshes demonstrate to be the right solution, with Linkerd that showed the best results in our tests, and which, in confirmation of our consideration, a recent report showed with a general positive trend of adoption, surpassing even istio in Europe and North America [110]. On the contrary, Apache EventMesh appears as a not yet mature technology, due to many problems in operation and significant difficulties during the installation process.

However, the study of meshes is not only important from the point of view of the use that can currently be made of these technologies, but also from the perspective of their possible future developments which appear ever more numerous. For example, in recent years, the cloud adoption by different organizations has transformed from single cloud solution to multi-cloud, and supporting diverse workloads (transactional, batch, and streaming) is becoming critical to realize a unified cloud architecture. Some service mesh solutions, such as istio, are moving in this direction, supporting heterogeneous infrastructures (bare metal, virtual machines, Kubernetes).

It must be considered also that other mesh vendors are focusing on the development of new offerings for their service mesh implementations, such as “service mesh as a service”. For example, Buoyant announced the release of a SaaS application called Buoyant Cloud that allows the customer organizations to take adantage of managed service mesh with the on-demand support features for the Linkerd service mesh.

But further considerations must also be made with regard to another topic which is that of data management. In fact, data is becoming larger and more ubiquitous and it is no more feasible for enterprises to rely on one large data store, shared among the different

organization areas. During the years, some enterprises have decentralized their data, pushing data storage, models, and management into different business units. However, data analythics has remained a more centralized activity. In this field, a robust and well-defined data communication layer can help organizations on facing data complexity, focusing on business functionality, instead of querying needs of other bounded contexts, and therefore the event meshes, which are a natural evolution of event-driven architectures to handle these large and diverse data sets, will gain great importance. And indeed, in the future there could be a convergence between event and data meshes, which was named event-driven data mesh. Some of the capabilities that are emerging in this field were pioneered in the synchronous data world and could be replicated in the asynchronous context: event-driven data should be easely discoverable and easy to catalog in a self-service manner; analytical data should be traceable from origin to consumption.

It must also be noted that there is interest also in applying service meshes to this field, with discussions about emerging architectural patterns for implementing event-driven messaging support within a service mesh. In this last context, the work toward supporting Apache Kafka within Envoy proxy attracted a fair amount of attention.

In conclusion, we believe that meshes will gain more and more importance in a context in which companies, as part of their digital transformations, are building new applications, and modernizing legacy ones, to leverage cloud native technologies that enable consistent and reliable development, deployment, management, and performance across cloud environments and across cloud vendors, including on-premises infrastructure. For this reason, the analysis of these technologies and in particular the evaluation of their performance will be increasingly important, allowing to identify their advantages and possibly their limits and therefore for companies to choose the technology that best suits their business needs.

BIBLIOGRAPHY

[1] T. Erl, R. Puttini, and Z. Mahmood, *Cloud computing: concepts, technology, & architecture*. Upper Saddle River, NJ: Prentice Hall, 2013.

[2] J. Arundel and J. Domingus, *Cloud native DevOps with Kubernetes: building, deploying, and scaling modern applications in the Cloud*, First Edition. Beijing ; Boston: O'Reilly, 2019.

[3] D. Comer, *The cloud computing book: the future of computing explained*, First edition. Boca Raton: CRC Press, 2021.

[4] P. M. Mell and T. Grance, 'The NIST definition of cloud computing', National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-145, 2011. doi: 10.6028/NIST.SP.800-145.

[5] B. Gregg, *Systems performance: enterprise and the cloud*, Second. Boston: Addison-Wesley, 2020.

[6] M. Armbrust *et al.*, 'A view of cloud computing', *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010, doi: 10.1145/1721654.1721672.

[7] K. K. Hiran, R. Doshi, T. Fagbola, and M. Mahrishi, *Cloud computing master the concepts, architecture and applications with real-world examples and case studies*. BPB Publications, 2019.

[8] 'Definition of Community Cloud - Gartner Information Technology Glossary', *Gartner*. https://www.gartner.com/en/information-technology/glossary/community-cloud

[9] J. Mulder, *Multi-Cloud Architecture and Governance: Leverage Azure, AWS, GCP, and VMware vSphere to build effective multi-cloud solutions*. Birmingham: Packt Publishing Limited, 2020.

[10] F. Klaffenbach, M. Klein, and S. Sundaresan, *Multi-Cloud for Architects: Grow your IT business by means of a multi-cloud strategy*. Birmingham Mumbai: Packt Publishing Limited, 2019.

[11] P. Zikopoulos and C. Bienko, *Cloud Without Compromise: Hybrid Cloud for the Enterprise.* Sebastopol, CA: O'Reilly Media, Incorporated, 2021.

[12] K. Hwang, G. C. Fox, and J. J. Dongarra, *Distributed and cloud computing: from parallel processing to the Internet of things*. Amsterdam ; Boston: Morgan Kaufmann, 2012.

[13] B. Familiar, *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*. Berkeley, CA: Apress, 2015.

[14] G. Agarwal, *Modern devops tips, tricks, and techniques manage, secure, and enhance software development in the public cloud with cutting-edge tools and solutions*. Packt Publishing, 2021.

[15] J. Rossberg, *Agile project management with Azure DevOps: concepts, templates, and metrics*. New York, NY: Apress, 2019.

[16] J. Davis and K. Daniels, *Effective devOps: building a culture of collaboration, affinity, and tooling at scale*, First edition. Beijing ; Boston: O'Reilly, 2016.

[17] S. Newman, *Monolith to microservices: evolutionary patterns to transform your monolith*, First edition. Beijing [China] ; Sebastopol, CA: O'Reilly Media, Inc, 2019.

[18] R. Mitra and I. Nadareishvili, *Microservices: up and running: a step-by-step guide to building a microservices architecture*. O'Reilly Media, 2021.

[19] A. Davis, *Bootstrapping microservices with Docker, Kubernetes, and Terraform: a project-based guide*. Shelter Island, NY: Manning, 2021.

[20] C. Carneiro and T. Schmelmer, *Microservices From Day One: Build robust and scalable software from the start*. Berkeley, CA: Apress, 2016. doi: 10.1007/978-1-4842-1937-9.

[21] J. Bugwadia, 'Microservices: Five Architectural Constraints | nirmata', *nirmata.com*, Feb. 02, 2015. https://nirmata.com/2015/02/02/microservices-five-architectural-constraints/

[22] S. Newman, *Building microservices: designing fine-grained systems*, Second edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2021.

[23] M. Fowler, 'Microservice Trade-Offs', *martinfowler.com*, Jul. 01, 2015. https://martinfowler.com/articles/microservice-trade-offs.html

[24] K. Indrasiri and S. Suhothayan, *Design patterns for cloud native applications: patterns in practice using APIs, data, events, and streams*, 1. Auflage. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2021.

[25] T. Ziadé, *Python microservices development: build, test, deploy, and scale microservices in Python*, First published. Birmingham Mumbai: Packt Publishing, 2017.

[26] 'What's a Linux container?', May 11, 2022. https://www.redhat.com/en/topics/containers/whats-a-linux-container

[27] ‘Understanding containers’, Dec. 10, 2019. https://www.redhat.com/en/topics/containers

[28] S. P. Kane and K. Matthias, *Docker: up and running: shipping reliable containers in production*, Second edition. Sebastopol, CA: O’Reilly Media, Inc, 2018.

[29] A. T. Velte, T. J. Velte, and R. C. Elsenpeter, *Cloud computing: a practical approach*. New York: McGraw-Hill, 2010.

[30] Canonical, ‘For CTO’s: the no-nonsense way to accelerate your business with containers’. Canonical, 2017.

[31] T. Erl, *Service-oriented architecture: concepts, technology, and design*. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, 2005.

[32] N. Poulton and P. Joglekar, *The Kubernetes Book*, 2021 Edition. Self-published, 2021.

[33] S. R. Goniwada, *Cloud native architecture and design: a handbook for modern day architecture and design with enterprise-grade examples*. s.l.: APress, 2022. doi: 10.1007/978-1-4842-7226-8.

[34] M. Lukša, *Kubernetes in action*. Shelter Island, NY: Manning Publications Co, 2018.

[35] B. Burns, J. Beda, and K. Hightower, *Kubernetes: up and running: dive into the future of infrastructure*, Second edition. Beijing: O’Reilly, 2019.

[36] G. Sayfan, *Mastering Kubernetes - Third Edition*. Packt Publishing, 2020.

[37] S. Trewin, *The DataOps Revolution: Delivering the Data-Driven Enterprise*, 1st ed. Boca Raton: Auerbach Publications, 2021. doi: 10.1201/9781003219798.

[38] M. Schwartz, ‘The data-driven enterprise’, *AWS Executive Insights*, p. 18.

[39] E. Strod, ‘What is a Data Mesh?’, *DataKitchen.io*, Aug. 03, 2021. https://datakitchen.io/what-is-a-data-mesh/

[40] ‘What is a data lake?’, *RedHat*, Sep. 16, 2019. https://www.redhat.com/en/topics/data-storage/what-is-a-data-lake

[41] Z. Dehghani, ‘How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh’, *martinfowler.com*, May 20, 2019. https://martinfowler.com/articles/data-monolith-to-mesh.html

[42] Z. Dehghani, *Data mesh: delivering data-driven value at Scale*. Sebastopol, CA: O'Reilly Media, 2022.

[43] Z. Dehghani, 'Data Mesh Principles and Logical Architecture', *martinfowler.com*, Dec. 03, 2020. https://martinfowler.com/articles/data-mesh-principles.html

[44] B. Schmeling and M. Dargatz, *Kubernetes Native Development: Develop, Build, Deploy, and Run Applications on Kubernetes*. Berkeley, CA: Apress, 2022. doi: 10.1007/978-1-4842-7942-7.

[45] B. Burns and C. Tracey, *Managing Kubernetes: operating Kubernetes clusters in the real world*, First edition. Sebastopol, CA: O'Reilly Media, 2018.

[46] L. Sun and D. Berg, *Istio Explained: Getting Started with Service Mesh*. Sebastopol, CA: O'Reilly Media, 2020.

[47] H. Adkins, B. Beyer, P. Blankinship, P. Lewandowski, A. Oprea, and A. Stubblefield, *Building secure and reliable systems: best practices for designing, implementing, and maintaining systems*, First edition. Beijing [China] ; Boston [MA]: O'Reilly, 2020.

[48] L. Calcote and Z. Butcher, *Istio: up and running: using a service mesh to connect, secure, control, and observe*, First edition. Sebastopol, CA: O'Reilly Media, Inc, 2019.

[49] M. Klein, 'Service mesh data plane vs. control plane', *Medium*, Oct. 10, 2017. https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc

[50] L. Calcote, *The Enterprise Path to Service Mesh Architectures: Decoupling at Layer 5*, Second Edition. Sebastopol, CA: O'Reilly Media, 2020.

[51] J. Song, 'Why Do You Need Istio When You Already Have Kubernetes?', *The New Stack*, Mar. 18, 2021. https://thenewstack.io/why-do-you-need-istio-when-you-already-have-kubernetes/

[52] Z. Jory, 'Why Service Meshes, Orchestrators Are Do or Die for Cloud Native Deployments', *The New Stack*, Feb. 22, 2019. https://thenewstack.io/why-service-meshes-orchestrators-are-do-or-die-for-cloud-native-deployments/

[53] O. Etzion and P. Niblett, *Event processing in action*. Greenwich, 74° w. long: Manning, 2011.

[54] A. Bellemare, *Building event-driven microservices: leveraging organizational data at scale*, First edition. Beijing Boston Farnham: O'Reilly, 2020.

[55] J. Schabowsky, ‘Event-Driven Microservices’. Solace white papers. [Online]. Available: https://solace.com/resources/white-papers/wp-download-event-driven-microservices-lp

[56] ‘What is an event mesh?’, Aug. 19, 2021. https://www.redhat.com/en/topics/integration/what-is-an-event-mesh

[57] ‘What is an Event Mesh?’, *Solace*. https://solace.com/what-is-an-event-mesh/

[58] ‘The Event Mesh: A Primer’. Red Hat, 2021.

[59] H. Shaffner, ‘Comparing and Contrasting Service Mesh and Event Mesh’. Solace white papers.

[60] W. Morgan, ‘The Service Mesh: What Every Engineer Needs to Know about the World’s Most Over-Hyped Technology’, *Buoyant blog*. https://buoyant.io/service-mesh-manifesto/

[61] A. Murphy, ‘The Hidden Costs of Service Meshes’, *The New Stack*, Nov. 04, 2021. https://thenewstack.io/the-hidden-costs-of-service-meshes/

[62] ‘Announcing Istio 1.5’, *Istio Official Blog*, Mar. 05, 2020. https://istio.io/latest/news/releases/1.5.x/announcing-1.5/

[63] Z. Varga, ‘Istio telemetry V2 (Mixerless) deep dive’, *Banzai cloud blog*, Apr. 12, 2020. https://banzaicloud.com/blog/istio-mixerless-telemetry/

[64] N. C. Mendonca, C. Box, C. Manolache, and L. Ryan, ‘The Monolith Strikes Back: Why Istio Migrated From Microservices to a Monolithic Architecture’, *IEEE Softw.*, vol. 38, no. 5, pp. 17–22, Sep. 2021, doi: 10.1109/MS.2021.3080335.

[65] S. Dake, ‘Istio 1.5 introduces simpler architecture, improving operational experience’, *IBM Developer*, Mar. 04, 2020. https://developer.ibm.com/blogs/istio-15-release/

[66] C. Box, ‘Introducing istiod: simplifying the control plane’, *Istio Official Blog*, Mar. 19, 2020. https://istio.io/latest/blog/2020/istiod/

[67] C. Posta, ‘Istio as an Example of When Not to Do Microservices’, *Software Blog*, Jan. 08, 2020. https://blog.christianposta.com/microservices/istio-as-an-example-of-when-not-to-do-microservices/

[68] B. Sutter and C. Posta, *Introducing Istio Service Mesh for Microservices*. Sebastopol, CA: O'Reilly Media, 2019. [Online]. Available: https://developers.redhat.com/books/introducing-istio-service-mesh-microservices

[69] V. Noronha, 'How to Write Envoy Filters Like a Ninja! — Part 1', *Medium*, May 28, 2019. https://blog.envoyproxy.io/how-to-write-envoy-filters-like-a-ninja-part-1-d166e5abec09

[70] 'xDS configuration API overview — envoy 1.20.0-dev-46ab91 documentation', *Envoy documentation*. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/operations/dynamic_configuration

[71] 'xDS REST and gRPC protocol — envoy 1.20.0-dev-755e28 documentation', *Envoy documentation*. https://www.envoyproxy.io/docs/envoy/latest/api-docs/xds_protocol

[72] W. Morgan, 'Introducing Conduit', *Linkerd Official Website*, Dec. 05, 2017. https://linkerd.io/2017/12/05/introducing-conduit/

[73] 'Architecture', *Linkerd documentation*. https://linkerd.io/2.10/reference/architecture/

[74] E. Weisman, 'Under the hood of Linkerd's state-of-the-art Rust proxy, Linkerd2-proxy', *Linkerd official blog*, Jul. 23, 2020. https://linkerd.io/2020/07/23/under-the-hood-of-linkerds-state-of-the-art-rust-proxy-linkerd2-proxy/

[75] A. Khatri and V. Khatri, *Mastering service mesh: enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul*. Birmingham: Packt Publishing, 2020.

[76] 'Consul by HashiCorp - Agent', *Consul Documentation*. https://www.consul.io/docs/agent

[77] 'Consul Reference Architecture | Consul', *HashiCorp Learn*. https://learn.hashicorp.com/tutorials/consul/reference-architecture?utm_source=consul.io&utm_medium=docs

[78] 'Consul by HashiCorp - C vs Envoy', *Consul Documentation*. https://www.consul.io/docs/intro/vs/proxies

[79] 'Consul by HashiCorp - Envoy Integration', *Consul Documentation*. https://www.consul.io/docs/connect/proxies/envoy

[80] ‘Consul by HashiCorp - Telemetry’, *Consul Documentation*. https://www.consul.io/docs/agent/telemetry

[81] ‘Ingress Gateways on Kubernetes’, *Consul Documentation*. https://www.consul.io/docs/k8s/connect/ingress-gateways

[82] ‘Consul by HashiCorp - Sync Kubernetes & Consul’, *Consul Documentation*. https://www.consul.io/docs/k8s/service-sync

[83] A. Rawdat, ‘Introducing NGINX Service Mesh’, *NGINX*, Oct. 12, 2020. https://www.nginx.com/blog/introducing-nginx-service-mesh/

[84] ‘Architecture | NGINX Service Mesh Docs’, *NGINX Docs*. https://docs.nginx.com/nginx-service-mesh/about/architecture/

[85] J. Urquhart, *Flow architectures: the future of streaming and event-driven integration*, First edition. Beijing Boston Farnham Sebastopol Tokyo: O’Reilly, 2021.

[86] ‘NATS - Request-Reply’, *NATS Docs*. https://docs.nats.io/nats-concepts/reqreply

[87] ‘OSM Docs’, *Open Service Mesh Documentation*. https://docs.openservicemesh.io/

[88] H. Thorsten, ‘Getting Started With Microsoft Open Service Mesh’, *OSM Docs*, Aug. 28, 2020. https://thorsten-hans.com/getting-started-with-microsoft-open-service-mesh/

[89] ‘Ingress’, *Linkerd documentation*. https://linkerd.io/2.10/features/ingress/

[90] T. Rampelberg, ‘Architecting for Multicluster Kubernetes’, *Linkerd official blog*, Feb. 17, 2020. https://linkerd.io/2020/02/17/architecting-for-multicluster-kubernetes/

[91] P. Pradeep, ‘Born at China’s WeBank, now incubating in the ASF - Introducing Apache EventMesh’, *The Stack*, Jun. 29, 2021. https://thestack.technology/apache-event-mesh/

[92] ‘EventMesh Workflow | Apache EventMesh’, *Apache EventMesh Docs*. https://eventmesh.apache.org/docs/design-document/workflow/

[93] ‘EventMesh Runtime Protocol | Apache EventMesh’, *Apache EventMesh Docs*. https://eventmesh.apache.org/docs/design-document/runtime-protocol/

[94] ‘Service Provider Interface | Apache EventMesh’, *Apache EventMesh Docs*. https://eventmesh.apache.org/docs/design-document/spi/

[95] ‘RocketMQ Architecture’, *Apache RocketMQ Docs*, May 12, 2022. https://rocketmq.apache.org/docs/rmq-arc/

[96] ‘Core Concept’, *Apache RocketMQ Docs*, May 12, 2022. https://rocketmq.apache.org/docs/core-concept/

[97] ‘Best Practice For NameServer’, *Apache RocketMQ Docs*, May 12, 2022. https://rocketmq.apache.org/docs/best-practice-namesvr/

[98] ‘Best Practice For Broker’, *Apache RocketMQ Docs*, May 12, 2022. https://rocketmq.apache.org/docs/best-practice-broker/

[99] F. P. Brooks, ‘No Silver Bullet Essence and Accidents of Software Engineering’, *Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987, doi: 10.1109/MC.1987.1663532.

[100] T. Gupta, ‘Analyzing Polyglot Microservices’, *Capital One Tech*, Dec. 19, 2018. https://medium.com/capital-one-tech/analyzing-polyglot-microservices-f6f159a1a3e7

[101] C. McKenzie, ‘What is polyglot programming?’, *TechTarget Network*, Sep. 2015. https://www.techtarget.com/searchsoftwarequality/definition/polyglot-programming

[102] R. Kalman, ‘On the general theory of control systems’, *IRE Trans. Automat. Contr.*, vol. 4, no. 3, pp. 110–110, Dec. 1959, doi: 10.1109/TAC.1959.1104873.

[103] R. Maloku and C. Posta, *Istio in action*. Shelter Island: Manning Publications, 2022.

[104] F. Budinsky and M. Ostrowski, ‘Introducing the Istio Operator’, *Istio Blog*, Sep. 14, 2019. https://istio.io/latest/blog/2019/introducing-istio-operator/

[105] ‘Installation Configuration Profiles’, *Istio Documentation*. https://istio.io/latest/docs/setup/additional-setup/config-profiles/

[106] ‘JVM options’, *Oracle Docs*. https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/java.html

[107] ‘Java Properties File Format’, *Oracle Docs*. https://docs.oracle.com/cd/E23095_01/Platform.93/ATGProgGuide/html/s0204propertiesfileformat01.html

[108] ‘Java Properties file syntax’, *IBM Docs*, Mar. 21, 2022. https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/was/8.5.5?topic=SSEQTP_8.5.5/com.ibm.websphere.nd.multiplatform.doc/ae/rxml_prop_file_syntax.html

[109] W. Morgan, ‘Why Linkerd doesn’t use Envoy’, *Linkerd official blog*, Dec. 03, 2020. https://linkerd.io/2020/12/03/why-linkerd-doesnt-use-envoy/

[110] W. Morgan, ‘Linkerd surpasses Istio adoption in Europe and North America with 118% growth in 2021’, *Linkerd official blog*, Feb. 16, 2022. https://linkerd.io/2022/02/16/linkerd-istio-adoption/