AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI
KATEDRA INFORMATYKI

PRACA DYPLOMOWA

APPLICABILITY OF PROGRAMMABLE NETWORK DEVICES IN SERVICE MESH ARCHITECTURES

Autor:	Arkadiusz Kraus
Kierunek studiów:	Informatyka
Typ studiów:	Stacjonarne
Opiekun pracy:	dr inż. Sławomir Zieliński

Kraków, 2022

Abstract

Service mesh is an emerging trend because of a plethora of possibilities it offers. It is based on some concepts, like control and data plane separation, that are in the core of mature technology - Software Defined Networks (SDN). The similarity between those two concepts has already been noticed and L2-L7 SDN has been proposed. This thesis attempts to take advantage of programmable network switches in order to achieve service mesh functions in new or existing systems. The objective is to review existing features of programmable devices and their applicability in service meshes, as well as to implement and evaluate some concepts using them. The work includes: implementation of service mesh functions using OpenVSwitch, integration of proposed solution with Kubernetes, and investigation of other tools and solutions like data plane programmability to extend the proposed base. The results show that a service mesh with limited functions is possible to achieve and that its performance is around two times better than the existing service mesh.

Acknowledgments

I would like to express gratitude to my supervisor, dr inż. Sławomir Zieliński, for his guidance, commitment and patience. His knowledge of network technology was of great help in creating this thesis and extending knowledge.

Contents

1. Introduction

Nowadays, applications play an important role in our daily life. Some of them save our lives, others help us manage finances or simply provide entertainment. The size of an average application has increased rapidly in recent years. To make them available all the time, people started building massive data centers and clouds.

Obviously, making everything bigger is a source of new challenges and technologies. The best examples are monolith applications that are hard to maintain, so people started dividing them into microservices and later organizing them into meshes. For data centers, it turned out that traditional networking is not sufficient and we need to have more control over traffic.

To solve the mentioned problems, two interesting concepts have been raised in recent years: service mesh and programmable networks together with data plane programmability. The first allows developers to focus on business logic instead of infrastructure by delegating all infrastructure-oriented code to other components. Traffic management, observability, and security are built-in functions, which really attract companies to migrate their existing architectures. The second concept, SDN, helps to programmatically manage data center networks from a central place rather than manually plugging in a new cable each time a new machine is deployed. Centralization of the management drastically increases the productivity and even makes large network fabrics like public clouds possible to build. Both concepts have been used successfully in many projects and are the core of modern applications.

In this thesis, I am going to research the possible usage of programmable networks in the implementation of service meshes and their possible integration with existing container orchestrators.

1.1. Project goals

Although the concept of Service Mesh is a really useful architecture for most applications, it is not always easy to introduce it to existing environments. Additionally, improved maintainability and observability of a set of applications also adds overhead to communication between services.

The solution presented in this thesis aims to leverage programmable networks to implement the functionalities of meshes within the network. This goal will be investigated in this thesis from three points of view:

- implementation of existing Service Mesh concepts using Software Defined Networks (SDNs),
- possibility of extracting data from high-level protocols, such as HTTP that is needed to implement some of the functions or creating a new, structured, and parseable by switches HTTP-like protocol,
- compatibility of the designed solution with container orchestrators.

Given the fact that service mesh is a very broad concept, for the first of the listed points, some basic concepts of has been chosen. For each of them, the goal is to provide a proof-of-concept of implementation that uses programmable switches or to research whether it is possible to achieve it. Replacing a component that needs to be run together with each service with a programmable switch would result in better performance and reduced resources needed for an application to run, and, in the end, in lower costs.

Some data needed for service mesh is passed using HTTP protocol in the less structured way than in, for example, TCP. Data plane programmability is not designed to handle this type of packets. Moreover, it can be encrypted so it would be impossible to read them on programmable switches. So, the second point focuses on the research on how to read them or to design a new protocol without such problems.

Last but not least, to make it usable in existing infrastructures mostly based on container orchestrators such as Kubernetes, the solution must be possible to be implemented in them. The third point focuses on this possibility and the proof-of-concept implementation that provides some functions in Kubernetes. However, the use of programmable switches may also lead to introducing service mesh features to systems which are not deployed leveraging container orchestration.

2. Background

The thesis topic contains two valuable and promising concepts: Service Mesh and Programmable Networks. In this section, a detailed description of both is provided.

2.1. Programmable Networks/SDN

Software Defined Network (SDN) is an approach that revolutionized the way computer networks are created and managed. It allowed us to basically code the network and its behavior from a central place, making networks much easier to manage. The whole concept is based on two rules:

- Separation of the data and control plane.
- Manage all devices using a controller through a southbound interface.

However, the concept did not appear out of nothing and is the result of an evolution that lasted many years. The most important part here is probably network virtualization, but before it, according to [11] and [17], we can differentiate three main parts of the way: active networking, data/control plane separation, and OpenFlow.

Active networks were researchers' answer to the problem of introduction and standardization of new protocols on network devices in the 1990s. At that time, the process between the idea of a new protocol and its general public availability was very time-consuming due to proprietary software and complicated standardization rules. The research aimed to make network devices programmable such as PCs. Communication and programming of the device was possible through the exposed API, which allowed the use of resources such as processing, storage, or packet queues. Two programming models evolved for active networks:

- capsule mode - execution code was transferred to the devices through the in-band data packets
- programmable router/switch model - execution code provided out-of-band to the device

The safety and error handling of the code execution was provided by the virtual machine that runs a program in a safe environment on an active node.

As with every technology, active networking has some benefits and drawbacks. To the first category belongs performing more processing in a network, which reduces the amount of computation on the end devices and the possibility to develop new protocols without changing hardware. Because of the use of advanced and high-level languages, such as Java, the programs also gained interoperability on many devices, even from different manufacturers. In the drawback section, we can include "network ossification", over-tuning a network for a specific use case, and vendor-specific programming models, which impede the development of general components, such as firewalls. "Network ossification" means the need to implement the OSI standard on such devices if one wants only to perform small improvements to this stack.

Active networking had an important impact on SDNs, because of facing similar problems which then transformed into approaches such as Network Virtualization Functions (NVFs), which are used in nowadays networking.

The second part, the concept of separating data and control plane, emerged in the early 2000s as a result of the need for better approaches to network management. The rapidly growing scale of the network, especially the provider networks, forced engineers to move the processing of packets to the hardware level and management of processing rules to the software level. Moreover, a number of devices to control encouraged people to create a single point of control and interfaces to manage devices from a single controller. Both concepts were later adopted in SDNs.

First attempts to split those layers were presented in ForCES and the 4D model. The first was a standard for communication between the control and the data plane, but was not widely adopted. One reason for this was that the application needed to rely on existing routing protocols, which limited its possibilities. The latter proposes the split not only into 2 layers but into 4. and was a broader concept than the ForCES. It stated a separation into 4 logical layers:

- data plane - processing packets based on rules,
- discovery plane - collecting traffic measurements,
- dissemination plane - configuring packet-processing rules,
- decision plane - converting network-level objectives into packet-handling state, included controllers.

The original 4D paper stated that the network was very fragile and proposed a new approach. However, the proposed model was not widely implemented. Some outcomes of it, like latency, security, and high availability, are still present in the SDNs.

Today, the term SDN is often used interchangeably with OpenFlow, which was the next step on the road to SDNs. Separation of control and data plane is at the core of it, but OpenFlow standardized data plane rules and API that is used for a control plane. OpenFlow itself is a protocol that enables to control switches from the central controller. It is based on the concept of flows. Each flow is identified by a set of bits from a packet header (like the source MAC, destination IP, source TCP port) and a proper action (such as drop, flood, forward) which is performed when a packet is matched to the flow. The success of OpenFlow was based on the fact that it was deployable on existing hardware and provided a much larger set of rules to describe flows.

However, the greatest progress was made by the concept of network virtualization. This idea abstracts a network from the physical devices on which it runs. It created the possibility of running many logical networks on the same physical architecture. Although it is a much older concept than SDN, network virtualization turned out to be a prominent use case for it.

SDNs to provide network virtualization are nowadays widely used in data center networks, in which a physical architecture is used to create a separate network for each tenant. It may be used in many other places, such as campus networks, but this is the most common commercial usage.

On the other hand, the concept of virtualization, or to be more specific, containerization, had a significant impact of the SDN development since it allowed one to simulate architecture and test SDN control application on a virtual infrastructure because of decoupling the control plane from the data processing plane.

2.2. Data Plane Programmability/P4

In 2014 paper [3] was published proposing the P4 programming language, which the authors named OpenFlow 2.0. They focused on the fact that OpenFlow is designed only on a hardcoded set of fields of the TCP/IP network stack. To enable the design of the new protocols, P4 does not limit itself to the known stack, but enables creating a data-plane program which processes packets according to a given code and then, based on the parsed set of fields, makes a decision using rule tables. Following the SDN terminology, it is a language that is used to create programs in the data plane.

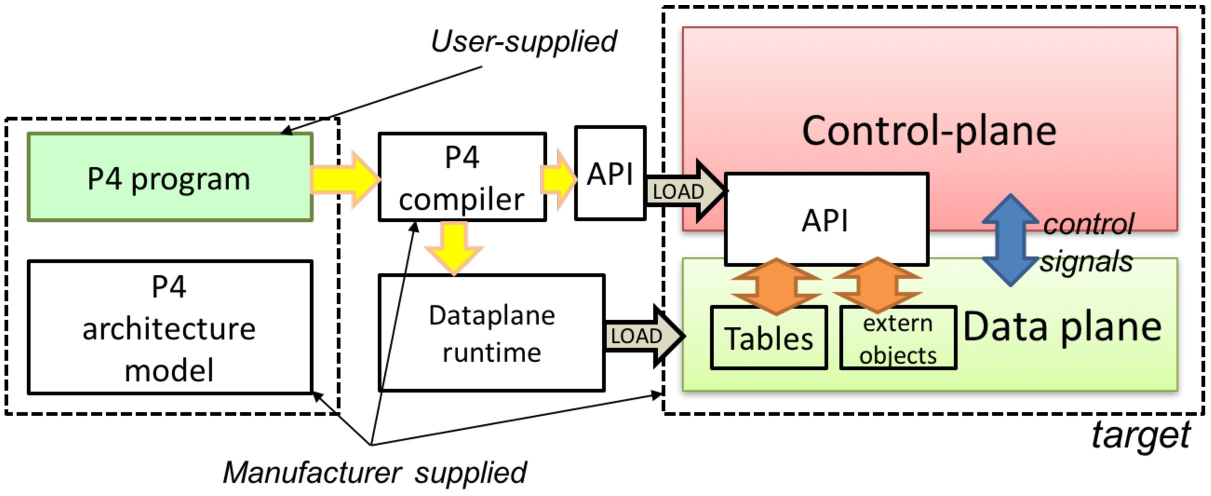

Figure 1: Programming a target with P4 from [47]

P4 is designed in a way that works with different targets, as shown in Figure 1. A target is a packet processing system that can execute P4 programs. P4 not only specifies a language, but also provides the ecosystem with tools and concepts needed to successfully run the created program. Concepts include:

- P4 architecture model - it is a contract between a program and a target, which defines programmable blocks and interfaces of them
- P4 compiler - provides target-specific implementation of the P4 program as well as data plane API which is used to control this layer
- Data plane runtime - a runtime which processes packets according to a P4 program
- Control plane - controller which modifies switch tables according entries
- externs - these are the objects and functions that can be used by the P4 program but are specific for the target that they run on.

A target provider is responsible for defining the architecture and providing a hardware or software framework with tools and a compiler implementation. There are a variety of targets, such as network interface cards, FPGAs, software switches, and hardware ASICs. Each target also provides match-action pipeline abstraction, which defines a structure of programs that are supported by devices. A pipeline is a sequence of unit blocks that directly maps to the code structure. Each unit is responsible for performing a key lookup by its attributes and matching the right action. The reference architecture for now is Protocol-Independent Switch Architecture (PISA), which abstraction should be able to handle all protocols. This approach is a generalization of past approaches which performed specific actions based on specific rules, the simplest example of it is routing table.

The reference software switch is named Behavioral Model (currently in v2 version) - bmv2. It provides the most common switches on which users can test and debug their programs. This switch is not production-ready and has much worse latency and throughput than, for example, OpenVSwitch.

Although the technology is fairly new, it has been successfully leveraged in a plethora of solutions. The current state and usages are well described in [23]. Thanks to this technology prototyping of new protocols has newer been easier what makes it really promising. New functionalities does not require waiting for vendor to implement them or replacing a device if it is not longer supported.

2.3. Microservices

In the past years, there is a number of examples of applications that have evolved very quickly from small projects with basic functionalities to large systems with thousands of users. However, many of them remain maintained as a single code base, which is called a monolith. Although this approach has some advantages, such as the simple deployment of a single package or the simplified end-to-end testing, it has many more drawbacks. First of all, development became much more complicated since it is hard to understand such a large structure and seemingly a very easy change might end up with a lot of code changes. Cooperation on such projects became very difficult as well as scalability of such an application. In addition, to apply one simple fix, one needs to redeploy the entire application. More disadvantages are described in [44].

As a solution to the above problems, the Service-oriented Architecture (SOA) concept emerged, which proposes splitting a monolith application into many small interconnected services. Each service is self-contained unit that should implement a small set of features focused on a single domain/functionality, such as payments. Moreover, each service exposes a well-defined API that allows other services or clients to access them via an exposed interface. This approach is much more scalable and easier to maintain because all services are self-contained, so can be modified not affecting the others. Moreover, services can be reused across many applications and do not have to be created from scratch each time.

Microservices are the next step in evolution and the modern implementation of SOA used to built distributed applications. Microservices are more of an implementation strategy, while SOA is more business-oriented. Such an architecture is built of many fine-grained processes that communicate over platform-agnostic network protocols. Each service can be created using different technologies and is loosely coupled to the others.

This approach is very flexible in line with modern agile practices in software development and DevOps. It focuses on scalability and continuous deployment. If one functionality is overloaded, the number of instances and resources can be increased for this specific microservice, which costs administrators much less than scaling the whole application. Moreover, teams can work separately on different functionalities after agreeing on a service interface and continuously develop their parts of the system and updating it in the separately.

However, microservices are not without drawbacks. Incorrect microservice architecture (for example, creating too many fine-grained services) may lead to maintenance troubles and performance issues. First of all, they need to communicate through RPC (usually HTTP), which is much less resilient and slower than function invocations in a single app. Calls go through the network and are vulnerable to packet loss or network delays. In addition, other problems of distributed systems, such as distributed transactions, apply. Furthermore, the deployment of a system of microservices is far more complex because of the many dependencies between components. Last but not least, debugging process is more complicated in such systems since it requires analyzing all dependencies, which makes observability aspect of the network really important.

There are also some other problems that must be solved for the system to work. For example, how one microservice can find another one or how to secure communication between them.

2.3.1. Service discovery in existing microservices-based infrastructures

One of the main and most important problems in microservices-based infrastructures is how a service can discover where its dependency is deployed [45]. Without it, the microservices cannot operate together, so this concept is described in detail here and in the following sections. The problem appears because of the fact that services should not be deployed statically, every time to the same machine and port because it reduces scalability and is vulnerable to host failures. Static deployments are much harder to maintain due to the scale of such architectures. Manually entering the address for all clients of the service would be very painful. When services are deployed dynamically, each time a service is deployed, it gets a different address and port. The component that stores information when services are available is called service registry. It allows to find the current address of the service by name. Each service registers itself on startup, or it is registered by an external registrar so that others can find it. Over the years, some ways have evolved to provide service discovery, but not all of them are based on registry. The one that uses a registry can generally be divided into two classes: client-side and server-side. In solutions that belong to the first class, the client is responsible for querying the service registry before making a call. In the server-side approach, the client makes a call to a specific component (such as the load balancer or proxy) that is responsible for communication with the service registry. The most popular approaches today are the following:

- Client-side,
- Load Balancer/API Gateway (server-side),
- Sidecar (Service Mesh, server-side),
- Other: SSDP, ZeroConf.

Client-side

The first concept that may appear when one wants to translate a name into an IP address is the well-known DNS. However, its simplicity implies some problems with this approach. First of all, typical A records can store only IP address, which means that there is no way to use non-default port or have more than one service on one machine when ports are dynamically assigned.

A much better way might be to use SRV record types [37]. They can store port and address, but usually there is no support for querying those records in HTTP client implementations. It adds a custom boilerplate code to query these record types. Moreover, its complexity increases when service wants to find another service by, for example, a specific tag like "v2".

A different type of record, which could potentially be used, is TXT, which can store arbitrary information for an entry. It would be a developer choice on what is stored in the record. However, due to the lack of required stucture it would lead to many misconfigurations.

Service registries, such as Consul, in addition to other ways of querying them, implement responding to DNS SRV queries. This approach is more lightweight than solutions using high-level protocols, but DNS has some limitations since it was not designed for this goal.

To avoid limitations of DNS, service registries can also provide their own API to lookup information about services. Such an API offers much more functionalities and can be extended without changing a many-year-old standard like DNS. Usually it is a typical REST API. However, the drawback of this solution is that there is no established standard for all APIs, so each service registry has a different one. It makes a migration from one service registry to another much more difficult because an implementation swap is needed in all services.

Load balancer/API gateway

The first example of a server-side approach is a dedicated API gateway or load balancer. The service is added to the gateway on a dedicated route or port, and this is a gateway that is responsible for querying the service registry and binding the current dynamic address of a service to the well-known address. Adding services which should be followed by load balancer can happen in a manual or dynamic manner. The first assumes that an operator manually puts services in the load balancer configuration. The dynamic approach can have many implementations, for example, it can be based on tags added to the registry. Load balancer queries services that contain a specific tag. Tags can also contain information about the route through which they should be accessible. After the services to follow are determined, the load balancer follows their real address and redirects traffic to them. The fact that all traffic must pass through this gateway is worth noticing, which puts it under a heavy load. The most common gateways are HAProxy [19] and NGINX [38]. Doing server-side service discovery on switches is a method in between load balancer and sidecar proxy. There is no one load balancer for all services and no dedicated proxy for each service. These responsibilities are taken by the closest switch.

Sidecar proxy

Sidecar is a common pattern in microservices. It is a component that runs alongside a service and is responsible for performing some infrastructure-related operations. It removes the need to repeat the implementation of such operations in each service. The sidecar communicates with the service via standardized network protocols. The common sidecar is a proxy that is responsible for properly routing traffic, which includes service discovery. Sidecar proxy pattern is very similar to the API gateway approach with the slight difference that there is no single proxy, but each service has its own proxy. The discovery process is delegated to the proxy, which then uses other methods to find a dependency. To the service proxy exposes some ports for local connection. Each port is dedicated to a different dependency. It consumes more resources than the previous approach, but there is no single component that can be overloaded. This concept is widely used in service meshes.

Other
There are other ways in which we can perform service discovery, but they are not as popular as those mentioned in the previous section. An example of them is the Simple Service Discovery Protocol, which is used in the Universal Plug and Play standard. This protocol does not require a central place to store entries, but leverages a multicast address to publish and discover services. For REST services, there exists an implementation of this protocol that can be found in [50]. The other option is ZeroConf, which is based on mDNS and DNS-SD and aims to create a network that is self-configurable and provides the automation address assigned and service discovery. However, both were designed for network devices, which may not be the most effective way for microservices.

2.4. Service Mesh

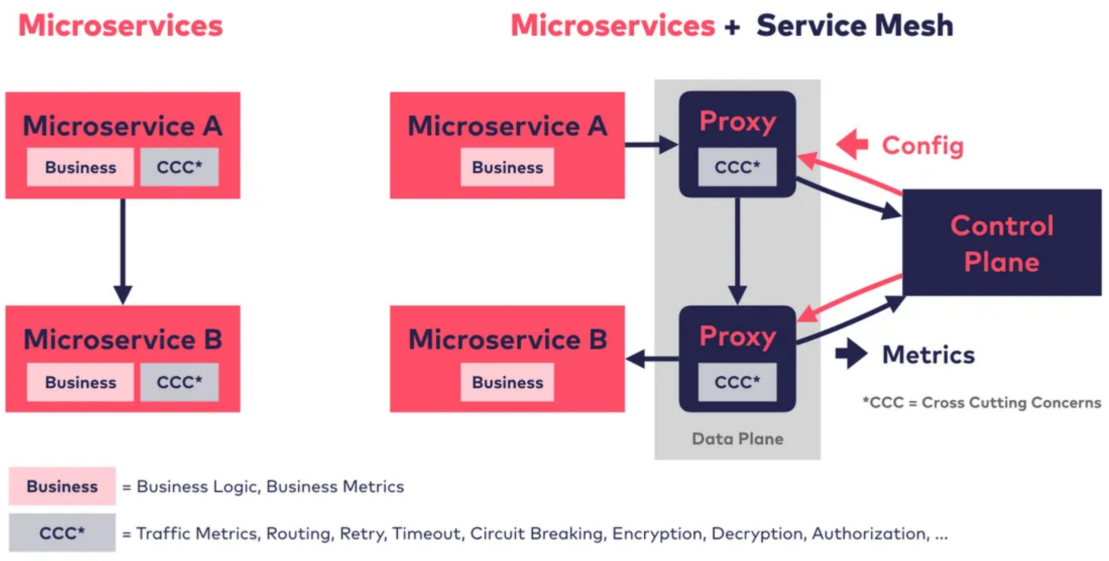
Microservices
Microservices + Service Mesh
Microservice A
Business
CCC*
Microservice B
Proxy
Config
Control Plane
Metrics
*CCC = Cross Cutting Concerns
Data Plane
Business = Business Logic, Business Metrics
CCC* = Traffic Metrics, Routing, Retry, Timeout, Circuit Breaking, Encryption, Decryption, Authorization, ...

Figure 2: Service mesh concept. Source: [46].

The next step after microservices is a service mesh. It can be defined as an additional layer of services (called infrastructure or control layer) which takes the responsibility for common infrastructure operations (such as service discovery, providing security, collecting metrics, etc.) from business services. It lets developers focus on business logic instead of infrastructure. Features such as observability, traffic management, and security can be added without modification of existing code.

The service mesh is split into two layers: control plane and data plane. The first one contains infrastructure components and services, while the second contains business sidecars. Sidecars communicate with the control plane as well as other sidecars to provide service mesh features. The concept is presented in Figure 2.

Service meshes are nowadays tightly coupled to container orchestrators, such as Kubernetes or Nomad. They are described in the next section. The solution researched in this thesis may also lead to introducing service mesh concepts to old systems that are not deployed using orchestrators.

2.4.1. Concepts

Below there is a description of key concepts that are realized by service mesh. They are generally divided into three categories: traffic management, observability, and security.

Traffic management

Traffic management refers to the set of rules that controls how network traffic flows between services. There are many pieces that build this concept: circuit breaking, timeouts, retries, canary rollouts, and staged rollouts.

Circuit breaking refers to the electrical circuit which can be opened or closed. Opening circuit means notifying other clients that one of the services is down and there is no point connecting to it. It can help, for example, in a situation when service is overloaded and needs to recover by finishing current work. Opening circuit will effect in not adding more work to it.

Timeouts and retries are fairly simple concepts. Timeout means that service took too long to respond and respond to the client that there is no point in waiting for it. Retries allows to automatically perform the same request again without client's request to repeat it.

Rollouts, both canary and staged, are responsible for routing only specific traffic (e.g. with a specific header) or part of traffic (e.g. 50%) to different instances of services. Canary rollouts are used to test new versions of the service by routing only chosen traffic to it. Staged rollouts allow to incrementally move more and more traffic to the new version of the service.

Observability

Concepts that allow developers to observe network behavior fall into this category. It includes service metrics, distributed tracing, and access logs. All of them help developers to quickly debug issues as they appear.

Metrics provide a high-level overview of the network focused on monitoring and understanding what is generally happening in the network. It is important to note that it is almost impossible to detect short (measured in milliseconds or seconds) tendencies in the network. For example, the default (and recommended) scrape interval for Prometheus [42] is 1 minute. Usually they give information about network's or particular services behavior like: error rates, memory and cpu usage, response time, number of objects waiting in the queue etc.

Distributed tracing allows for more detailed debugging, but narrowed down to a single problem (for example, finding the reason for error code). Such systems let us track the request from the source in the network through all the dependencies it reaches. It lets developers to easily debug the source of errors or find bottlenecks in the network.

Full request logs can be gathered through access logs mechanism in service mesh. It allows to collect full metadata from the source to the destination, which enables auditing of a service behavior.

Security

The last concept is nowadays of a greatest importance. When it comes to service mesh to this very broad category, we can include: secure and encrypted service-to-service communication without possibility of man-in-the-middle attack, access control through authorization and authentication, and auditing tools.

Traffic encryption is used to prevent man-in-the-middle attacks. Today, the standard is to use TLS to communicate between a client and a server. A control plane contains a Certificate Authority that is responsible for generating certificates used in secure communication.

Access control covers problems related to question: who or what has permissions to access a given resource. At the core of it there are authentication and authorization mechanisms integrated with service mesh. The clue is to provide fine-grained policies specifying what can be accessed by who. Usually, it is achieved through a mutual TLS (mTLS).

A fundamental concept of mutual authorization and authentication is an identity by which allow/deny decisions are made. Parties exchange their credentials that contain identity information. It is supported by functions such as secure naming, which checks if server is authorized to run on a target machine.

There are two types of authentication: request and peer. Request refers to user authentication, while peer refers to machine-to-machine authentication and verification of a client making connection.

After both sides have successfully authenticated and are sure of the identity of the other party, authorization policies must be enforced. They contain information on whether a client party should be able to run a given workflow. Usually, proxies on both sides stand as Policy Enforcement Points that check and validate these rules. The rules might be of a different granularity, for example: host, service, or route.

In addition, service mesh should provide auditing tools that can be used to diagnose potential security incidents. The tools should provide us with audit records containing information: who, what, and at what time.

2.4.2. Data plane

In the core of data plane there is a concept called sidecar proxy, which is much broader than the one that only does service discovery. It is a component which is run together with each service and is fully transparent to the service. It intercepts its network traffic, by providing a simple API for communication with service, taking responsibility for communication between services by doing service discovery, routing, TLS encryption, etc. It is called sidecar since it runs alongside the application instead of inside it.

The leading solution in this field is Envoy [35]. It is a high-performance proxy, designed to run alongside the service in service mesh architectures. It abstracts the environment for business service by providing a common set of features in a platform-agnostic way. Other well-known proxies, such as NGINX or HAProxy are less commonly used in this field.

In the past the proxies were needed to be configured manually. To enable automatic updates, an API was needed to control them. Using this API, control plane can configure a proxy. Each of the most common providers has its own API. Envoy exposes one called xDS [10].

2.4.3. Control plane

Control plane contains services that provide the features of the service mesh. As an example, it can contain a service collecting metrics, authentication server, service registry, etc. According to the configuration and current states, it control proxies in data plane through an API.

Control plane is also what is sometimes referred to as a service mesh. Frameworks like Istio are responsible mostly for components in control plane. The most popular ones are: Istio, Linkerd, envoy-control-plane, and Consul Connect.

2.4.4. Migration stories

Although the idea of a separation between architectural components and business logic components is very useful for both developers and operations teams, service mesh might consume a significant amount of time before it is used in production. Some companies have already performed such migration and published their conclusions in articles such as [24] and [43]. Real-life stories might be helpful in designing solutions that avoid problems they had.

The first of the articles [24] describes the migration of their large architecture containing hundreds of services of different maturity, where some were using old technologies. The team decided to create their own control plane which was based on java-envoy-control. The entire migration took about 10 months to be used in production. Worth noticing is the fact that one of the main reasons of introducing service mesh, despite the general service mesh goals, was to introduce observability into the network. The summary of findings, problems, and conclusions they had is:

- After successful adoption of Envoy as ingress proxy of service by replacing port values in service registry, they encountered problem with egress. It required modification of their services to explicitly call the proxy.

- They noticed that new control planes frameworks like Istio work well for new projects, however, it is harder to adopt them in the existing architectures with non-homogenous stack.

Another story [43] describes the research on which service mesh they should use and their later adoption of it. The team finally decided to use Istio. Similarly to the previous article, one of their main goals was to achieve the observability of the network. The conclusions from the article are:

- Although Istio is free to use, the service mesh is not free to run. They estimated that running proxy next to 1000 services would require 100 CPUs, which would be equivalent to a machine worth $3500.

- Unlike in the previous article, iptable rules were used, and the team notices that it slightly impacts latency.

2.4.5. Market state

CNCF Survey is an annual survey conducted among IT companies that are interested in cloud-native solutions. According to the survey from 2020 [6] around 50% of the participants are evaluating or using service mesh in production. The leading solution implemented in production in this category is Istio, which was used by 47% of companies. However, the other ones are not far from the leader, especially Linkerd and Consul, which have only around 5% less. The survey covers 10 different frameworks, which shows how popular the solution is. In addition, there are a number of companies that are during the evaluation process. One can also spot that the more a framework is evaluated, the less is used in production.

When comparing the results with the 2019 survey [5], there was a significant increase in the usage of service meshes in production. 2019 was the first year this concept was added to the survey, as interest in this technology was first shown then. Most of the companies were evaluating this solution at that time, so the increase in production usage in 2020 is a natural process.

According to Gartner's Hype Cycle 2021 it was placed after the peak of expectations closer to the trough of disillusionment [15].

2.5. Container orchestrators

In recent years, containers have definitely captured the market and are used by most of the companies participating in [6]. The increasing number of containers used in everyday workflows has caused the need for a system that simplifies their management. In 2015 Google published a paper [48] describing its large-scale container orchestration system. Then the system was open-sourced and renamed to Kubernetes, which became one of the most rapidly growing projects in recent years.

Container orchestration in the simplest words is a concept of managing, scaling, and maintaining containers across clouds and data centers. Although it might seem simple, it is a very complex solution which enables to control thousands of containers across many locations from a single place. It includes complex networking between them, computing, storage, scaling, security, and more.

2.5.1. Market state

According to the mentioned survey [6] the market is dominated by Kubernetes. It is used in production by 87% of the companies that participated in the survey. It is also implemented as a service on most of the major cloud platforms, such as Azure Kubernetes and GCP Kubernetes. There are some other solutions such as Apache Mesos or Nomad, but they are less advanced and mostly following the features from the leading one.

2.5.2. Kubernetes

Kubernetes is definitely the leading solution among container orchestrators. It is also very complex and has many features. The ones that seem to be important for the thesis work are described in this section.

Consul k8s

Consul k8s [20] is a plugin designed to easily adopt the Consul Connect service mesh to a cluster. It installs necessary services (like Consul service registry) in the network and allows to deploy new pods to a mesh.

When a new pod is deployed, Consul controller evaluates if it should be added to a mesh by checking configuration and pod annotations. If the result is positive, then the plugin injects an Envoy proxy next that is responsible for ingress and egress.

Its documentation promotes a mode that is called a transparent proxy. The main benefit of it is the fact that upstream do not have to be explicitly given to the service, but they leverage special network rules [7] to route traffic through proxy. In this mode, the proxy cannot be bypassed, which enhances security.

CNI plugins

CNI plugins are not tightly bound to Kubernetes but are a general standard. They are a specification of a way container interfaces should be configured for different kinds of networks. Networking may differ on different platforms or providers. CNI is a specification of a common interface and implementation might be provided by everyone. The project, which is developed under the Cloud Native Computing Foundation, also provides users with tools to create new plugins.

The process is as follows. All plugins share a common interface, which is part of the specification. When a container is created/removed/changed, it invokes method and runtime decides (according to configuration) which implementation is used and the right function is invoked.

There exists a plethora of implementations, for example, for docker overlay network, for specific cloud provider, for VXLAN etc. In this thesis, a pod needs to be attached to a network created by programmable switches (OpenVSwitch or P4 switches). There are two projects that allow to achieve this: Antrea [2] and ovs-cni-plugin [26]. The first one proposes much more functionalities some of which already implements service mesh functions such as traffic permissioning. But it can collide with the rules proposed in this thesis. The second one is much simpler and does not put any rules

Kubernetes Custom Resource Definition (CRD)

A custom resource provides a way to extend the Kubernetes API with new resources. By default, Kubernetes provides resources such as pods or replica sets. Custom resources are a way to customize a cluster with modules that provide new capabilities. They need to be implemented by operators.

Custom resource definitions allows to specify an API for a custom resources which is later on used to validate new resources.

Kubernetes operators

Operators are the way to extend Kubernetes capabilities without modifying existing code [33]. They provide a way to automate common workflows and also a way to implement the behavior of CRDs. The most common way to create an operator is to create a CRD and an associated controller. Controllers are separate programs that monitor cluster state and are responsible for achieving declared state or, in other words, providing implementation for declarative syntax.

There are many libraries that simplify the creation of operators, such as Kubebuilder. It allows to automatically generate a common code such as Kubernetes API subscription and provides a set of tools to communicate with a cluster API.

2.6. Related work

In article [52] the author notices similarities between service mesh and SDN related to the separation of a control plane from a data plane. However, he points out the fact that they operate at different levels of OSI/ISO layers. SDN focuses on the four lower layers, while service mesh is focused on layers four to seven. As an outcome of the article, it is stated that what should be developed in service mesh is an application layer similar to the one in SDN. Such application layer would allow one to programmatically control infrastructure when for example some components misbehave. There have been such attempts to create the Service Mesh Interface (SMI), but this solution is not widely adopted. Moreover, the author also presents the concept of controlling also hardware devices from the control plane.

The second result from the previous article is wider described in article [36] where control plane is used to also control F5 BIG-IP device to connect to an external database (not included in service mesh) and enable access only from authorized containers.

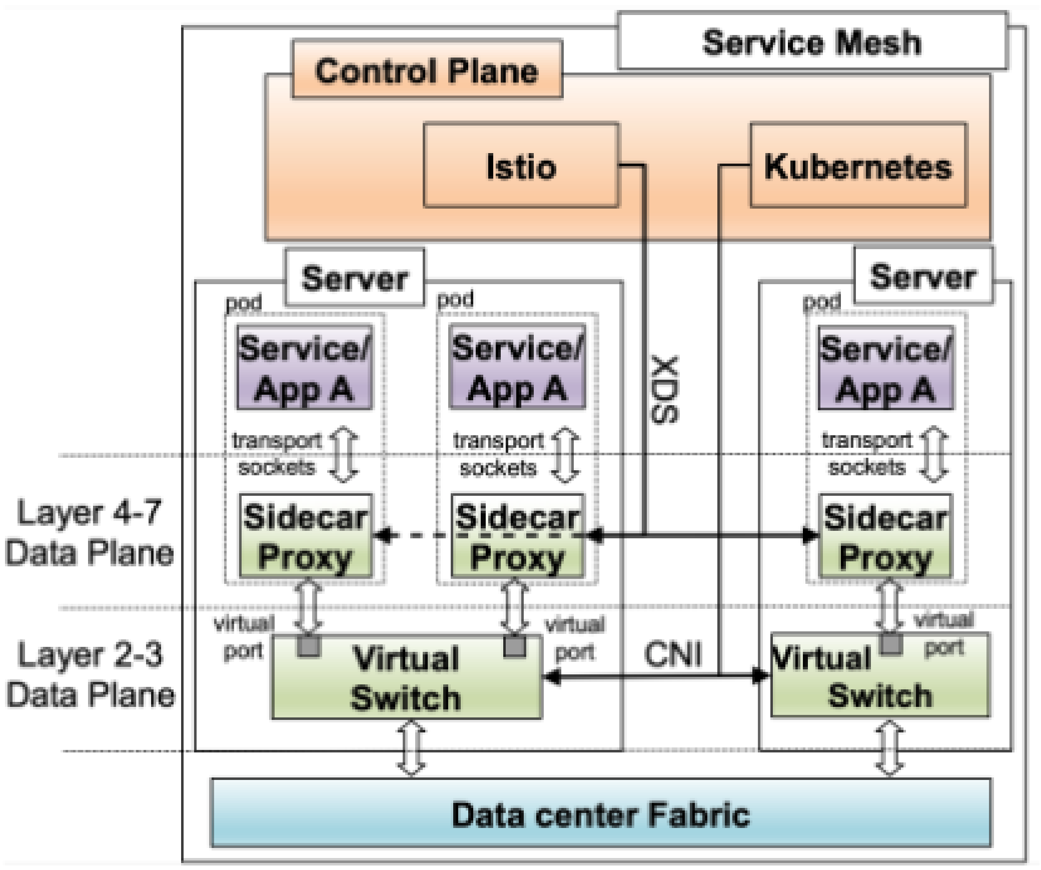

Figure 3: Current layers identified by the authors of [1]

The concept of SDN that operates only in layers 1-3 is challenged in paper [1]. The authors describe a concept of L2-L7 SDN as the next big challenge. The article had three goals: identify sidecar as L4-L7 data plane and Istio as a control plane, provide use cases for L2-L7 SDN and propose the research structure. Figure 3 comes from the article and shows the current split of the typical network stack into separate data planes in different network layers.

The first of their points is proved by showing that Envoy can be abstracted by a match-action pipeline, known from SDN data layer. For the example usages, the authors provide web services and network virtualization functions. In the end, they propose the research on further connecting SDN and service mesh into one solution.

The networking between containers can also be achieved by a software technology different from a programmable network, which can also inspect and modify packets on the host. For example, such technology can leverage eBPF [14], which allows to create programs operating in kernel space of the system. An example of it is Cilium [4] which enables not only network connectivity between processes or containers, but also observability and security of such connections. As an example, it provides features like: load balancing, multi-cluster connectivity, metrics, identity-aware observability, encryption, audit. Especially features that are security-oriented might be harder to achieve on switches.

For Kubernetes there exists a CNI plugin that implements a subset of features needed by service mesh. The plugin is named Antrea [2]. It is based on OpenVSwitch and connects containers to it. Moreover, it provides a way to control which services should be able to communicate (L4 intentions) and exposes a network metrics and provides a way to visualize them. Both features are needed in the service mesh. However, it does not relate to the service mesh, but describes itself as a very powerful plugin with many useful features.

3. Problem statement

As the previous section has shown, service mesh is a very promising concept. However, according to market state this solution already passed through the biggest excitement that might be caused by the complexity of it. Definitely it is not a simple solution, made of a plethora of different concepts, across traffic management, observability and security, so, as [43], [24] and [16] shown, migration from the microservices to a mesh is not an easy task and might be time consuming. Especially in huge environments that contain hundreds or thousands of services.

There is a common point in all linked articles - running sidecar for each of the existing services. Revisiting a large number of sometimes old services to run sidecar for them might be time-consuming and not possible to do at once. Furthermore, [43] mentions the fact that the usage of resources is noticeably higher. Indeed, additional sidecars need a lot of resources. The article points that, despite the framework being free to use, users need to pay for additional resources.

On the other side, there is a concept of programmable networks, which was evolving for a long time and is naturally more mature than service mesh. After a long time, SDN reached a point in which they are widely and successfully used in data centers across the world.

Given the above, the first thesis goal is to prove that it is possible to build a data plane which includes programmable switches without the need for running a sidecar proxy for each service along with control plane responsible for controlling the switches. Since service mesh is a broad concept, in this thesis, the layer will be limited and responsible only for some features focusing mostly on a service discovery. The solution might compete with Envoy proxy, which is the base of almost all data planes nowadays.

Today, most microservices-based applications are deployed using container orchestrators, as was described in the previous chapters. Most of current service mesh frameworks are tightly coupled to one of them, so the work would be pointless without the integration with existing ones. This defines the second hypothesis of the thesis: the proposed solution can be integrated with existing container orchestrators.

Last but not least, the network stack that service mesh is based on is made of traditional protocols. Mostly on HTTP, which is less structured than lower protocols and cannot be easily parsed by programmable switches. However, some of the functions of service mesh are based on, for example, HTTP headers.

This leads us to the third thesis problem: programmable switches can read information from high-level protocols, which enables implementing more features of service meshes and leads to improving service mesh network performance and resigning from proxies. If it is not possible, then it is possible to propose a new protocol which extends the existing ones. The extension protocol should take advantage of data plane programmability, which significantly reduces the cost of any new protocol.

Each hypothesis and problem are described in a separate section. Section 4.1 contains research on the first goal, 4.3 covers the second hypothesis, and the third problem is described in 4.2. Section 5 contains the summary of the results achieved in the previous sections.

4. Solution

4.1. Implementation of service mesh concepts using programmable switches

This section shows how different concepts of service mesh can be implemented using programmable networks, which answer the first thesis goal. Each subsection contains a description of an implementation or a research of one feature. Together, the features are able to create a service mesh. Server-side service discovery is implemented and described in detail.

4.1.1. Application

In order to present and test the ideas from the thesis, a microservice-based application is needed. To avoid creating a new example application, an application provided by HashiCorp, commonly named HashiCups [21], has been chosen to demonstrate the results. I strongly believe that it is a great example to work on because it is a piece of well-written software, divided into 4 microservices using different technologies, that reflects architecture used in real applications. Moreover, it is strongly used by the company to present its solutions and specifically Consul Connect, which is HashiCorp's service mesh solution.

The application is a typical online shop that enables the user to create an account, view products, and buy them. It is split into 4 pieces: frontend, public api, product api, and payments api. The frontend is created leveraging Next.js framework and talks only to public api, which contains the main application logic. Public api is written in GO and realizes its functions using two more dependencies: product api and payments api. The first dependency is also written in GO and uses postgres database to store its products. The second is created using Java and Spring framework and does not have additional dependencies.

To simulate real deployment using a container orchestrator, in the reference architecture in this thesis, the application has been spread among 4 hosts, which is presented in Table 1. All services are assigned with arbitrary ports, not typically used. Such a deployment reflects the random distributions in which one host contains more than one service, and other dependencies are deployed on different hosts. Additionally, there is a host that contains a service registry.

Hostname	Services	IP	MAC
H1	Frontend, Public API	192.168.1.11	00:00:00:00:00:01
H2	Product API	192.168.1.12	00:00:00:00:00:02
H3	Payments API	192.168.1.13	00:00:00:00:00:03
DB1	Postgres	192.168.1.21	00:00:00:00:00:04

Table 1: Services distribution on the hosts

The proposed architecture is also presented in Figure 4

4.1.2. Physical topology

The architecture is implemented using the Kathara framework. All hosts are connected using programmable switches (OpenVSwitch or P4 Simple Switch), creating a small hierarchical

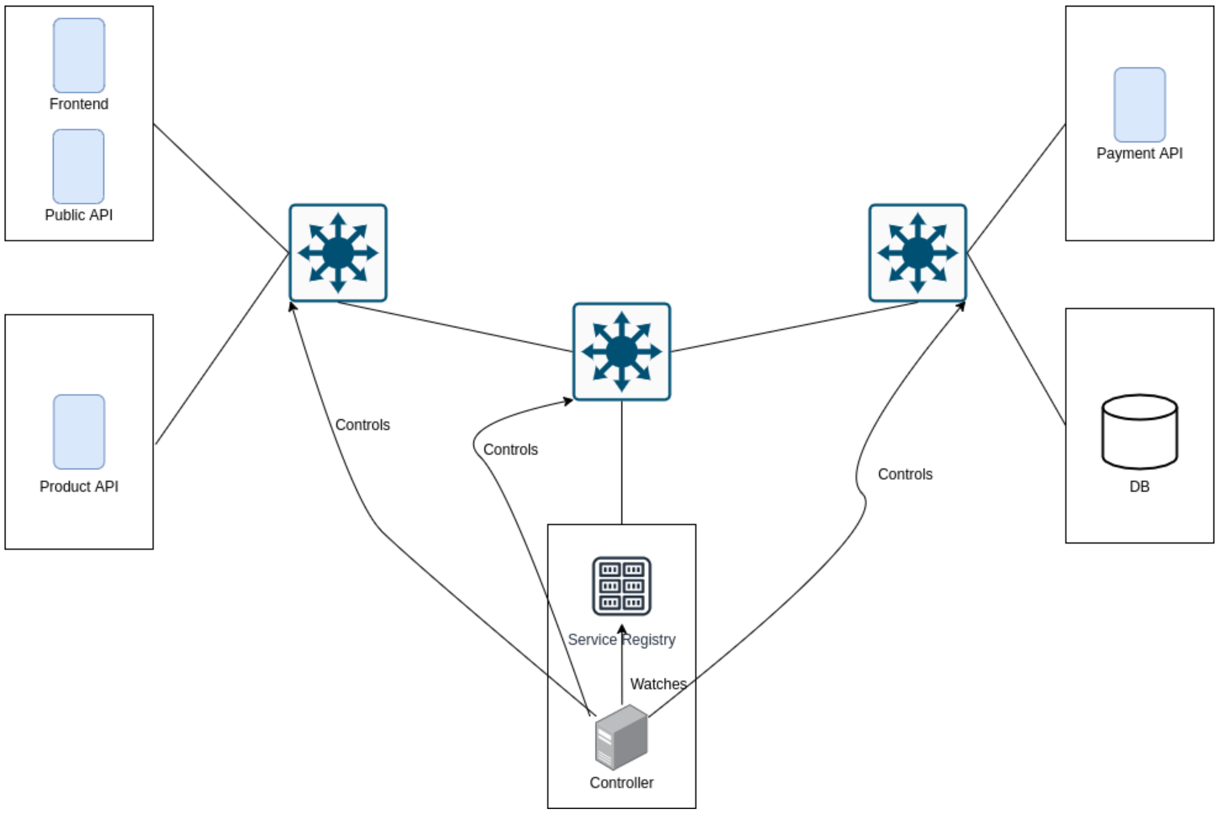

Figure 4: Service mesh lab architecture

structure. Hosts are simulated using docker containers. A Netkit topology is identical to the one in Figure 4. What is worth noticing is that in the current architecture a switch is connected to hosts, not to particular services.

4.1.3. Concepts

Server-side service discovery

The first concept that could be supported by programmable networks is a server-side service discovery. In this model, a service sends a request to an address on which a dependency is visible to this service. The address is pointing to a load balancer or a proxy which is responsible for routing traffic to the real dependency. As the goal is to replace a separate component and performs in-network discovery, the responsibility should be transferred to switches.

Switches themselves cannot do additional queries to the service registry, but can contain switch table entries with the routes to particular services. The entries have to be built on the basis of the service registry's database. The SDN controller is responsible for synchronizing a registry's state and the rules. The idea is to watch services registrations and then, knowing the switches topology, built routes and entries for each switch.

Services still need to have an address (IP + port) that they call to reach the destination (upstream). As there is no physical component and switch does not have an IP address, the virtual one can be chosen and handled by the switches. The table entries are based on this virtual address. The first problem that appears here is that a host needs to know a MAC address for virtual IP. The simplest but not optimal solution is to put static ARP entries on the host that maps the virtual address to the random MAC. The better solution would be to make switches or other component answering ARP queries for this address. Unfortunately, it is not possible to achieve it using OpenFlow. We would need to adopt P4 data-plane programmability to be able to achieve ARP responses.

Moreover, packets are sent to the virtual IP but physically they can be addressed to any host. Because of it the destination MAC address is not known. It is a switch, what makes the decision, so it also needs to put the right ARP address so that destination host does not reject the packet at this level.

When a packet is successfully sent to the network from the host and it reaches the first switch, the destination virtual IP and port need to be translated to the real one base on the rules built by the controller. It is not a hard task to do using OpenFlow rules, but the second problem is that the translation cannot be one-way. If it is one-way, the TCP connection cannot be established because of a different address in the response. The example of incorrect communication and how the packet changes is presented in Table 2.

Packet	Packet type	Device	Source IP	Source port	Destination IP	Destination port
1	SYN	H1	IP1	P1	VirtualIP	Virtual port
1	SYN	Switch	IP1	P1	IP2	P2
1	SYN	H2	IP1	P1	IP2	P2
2	ACK	H2	IP2	P2	IP1	P1
2	ACK	Switch	IP2	P2	IP1	P1
2	ACK	H1	IP2	P2	IP1	P1

Table 2: Example packet flow with connection tracking

In the presented flow because of the fact that the ACK packet comes with a different source address than H1 expects, it drops the packet and connection is not established. Because P1 being ephemeral and not known upfront, the switch cannot prepare rules to translate it back.

Listing 1: Template for server-side service discovery OpenFlow rules

```
ovs-ofctl add-flow {switch.name}
"priority=50,tcp,in_port={port},ip_dst={VIRTUAL_PROXY_IP},
tp_dst={local_bind_port},
action=ct(commit,zone=1,nat(dst={dst_address}:{dst_port})),
mod_dl_dst:{dst_host.mac},output:{via_port}"

ovs-ofctl add-flow {switch.name}
"priority=50,tcp,in_port={via_port},ct_state=-trk,
action=ct(table=0,zone=1,nat)"

ovs-ofctl add-flow {switch.name}
"priority=50,tcp,in_port={via_port},ip_dst={src_host.ip},
ct_state=+est,ct_zone=1,action={port}
```

The solution to this problem is to track connections like it is done in the widely known PAT pattern. Switch stores the table for the TCP connection, which is used to track the initialized sessions and translate addresses. OpenVSwitch allows this to be done by the conntrack module and nat function. By using it, the switch modifies not only the request but also the response so that the source host sees this as the response from the virtual ip address. The general pattern of rules for a single upstream of a single service is presented in Listing 1.

For testing, the virtual ip was assumed to be 192.168.1.250 and the host addressing presented in the table 1. The rules were built using a Python script that accepts the current consul state along with infrastructure state. It needs to know by using which ports services are connected and their addresses: IP and MAC. This code can be easily integrated with the controller as it knows switches and can connect to Consul, so it is possible to rebuild it when state changes. The need for knowing the ports might be replaced using normal switch behavior in which the switch learns the MAC and ports. The description of the infrastructure and the builder code can be found in [29]. The set of rules presented in the appendix A was built to achieve the connectivity of services in the presented architecture.

These rules allow to successfully communicate services in the service mesh with one exception. There is no connection between services on the same host due to the fact that packets are not going back through switches. For example, a service on H1 sends a packet to the virtual ip, which is then translated to its dependency on H1. When the dependency sends the packet back it addresses it to H1 so the host does not send the packet through the switch, so the address is not translated.

It is a problem on a presented infrastructure, but could be avoided if each service was on a separate port on a switch, which is a way how, for example, CNI plugins works. It depends on the level at which we connect a switch. If we connect at the host level it is a problem, if we connect at the container level it will not be because each service has a different ip. A solution to the host-level connection would be to translate not only destination ip, but source ip as well.

Moreover, in real infrastructures, there is usually more than one instance, but in the current set of rules (and code of a rules builder) we always choose one instance. The clue of microservices is to have more than one instance. However, for the current set of rules it is pointless since all traffic will always go to the first instance. The solution is presented is section 4.1.3.

Client-side service discovery

In the previous section, the network was responsible for service discovery. The other approach to this problem is to resolve a dependency address directly on the client. An example of this is to use DNS.

In the paper [51] the solution was proposed to support answering DNS queries using P4 switches. Assuming that this is the way how service discovery is implemented in the microservices-based architectures the work might be possibly leveraged by in-network service mesh. In the solution switches can observe the packets going through it and cache them for a much faster answer in the future.

The important thing in the service mesh is that the records can change quickly. According to the paper controller updates the TTL every second, which might be used to set the TTL to zero when the record changes. The switches should be properly adjusted to the new state.

However, the solution has some drawbacks. The authors mention problems with handling variable-length fields, which lead to limitations on the size of packet that could be parsed. This limitation might be not sufficient to handle long records in the existing solution because following the standards, DNS packet might be 512B long. In modern implementation EDNS0 [49] the allowed size might be larger.

Moreover, the previous solution was designed to work with OpenVSwitch. This requires changing the switches in the network to the P4 switch. The possible problem with it is that too heavy workload when switches implement many features might affect their performance. OpenVSwitch is also a more mature technology.

The proposed implementation was not tested as part of this thesis and is only a suggestion of the possible use of the given work.

On the other hand, DNS is not the only possible way, and there are more options for client-side service discovery. Some services do it using the registry API using HTTP protocol. However, it requires parsing such packets, which is covered later on in this thesis.

Load Balancing

Usually there is more than one instance of services to distribute the load, or in case any of the instances are down, the other are able to handle requests. Moreover, more popular is the concept of an automatic autoscaling app to automatically add additional instances when the load increases. The rules created in Section 4.1.3 have a specific target. The lack of a way to load-balance the traffic would make such a solution useless.

To handle such a situation, one might want to take advantage of the group feature offered by OpenFlow [40]. There is the possibility of defining a group of buckets and specifying how traffic should be distributed to the buckets. It supports hashing of the 5-tuple (source ip, destination ip, source port, destination port, transport protocol) to handle packets fragmented by IP protocol. Each of the buckets is just a set of actions, the same as in a normal OpenFlow rule. This option should provide the possibility of loading-balancing a traffic very close to the source of it.

The template for the rules would have a structure presented in listing 2.

Listing 2: Structure for load balanced flows

```
ovs-ofctl add-group {switch.name} "group_id=100
type=select selection_method=dp_hash
bucket=ct(commit,zone=1,nat(dst={dst_address}:{dst_port})),
mod_dl_dst:{dst_host1.mac},output:{via_port1}
bucket=ct(commit,zone=1,nat(dst={dst_address}:{dst_port})),
mod_dl_dst:{dst_host2.mac},output:{via_port2}"

ovs-ofctl add-flow {switch.name} "table=0,tcp,in_port={port},
ip_dst={VIRTUAL_PROXY_IP},tp_dst={local_bind_port},
actions=group:100"
```

Network metrics

The knowledge of how services interact with others is useful for debugging issues, migrating services to new versions, finding bottlenecks, and units that speak a lot to others. It is an important part of the observability, which is not provided by the default network stack. Performing this on the service mesh would be a great benefit of this solution.

By default, OpenVSwitch tracks flow-oriented metrics. For each flow, it saves the number of packets that matched the flow, as well as number of bytes in those packets. Because the rules created in Section 4.1.3 are created for a pair service - upstream, from these statistics it is possible to deduct metrics needed to achieve such observability.

To easily gather these metrics from all switches and store them in a central place Prometheus [42] might be used. It is a monitoring system that pulls metrics from targets at specified intervals and stores them in a time-series database. To convert metrics from a format in which OpenVSwitch shares them to the format in which Prometheus is able to scrape OpenVSwitch Prometheus exporter [9] may be used. Because Prometheus tracks how the number changes over time, based on those metrics user is able to gain time-oriented characteristic of the traffic. A similar approach was taken in the Antrea plugin [2], which exposes traffic metrics and provides dashboards to visualize and organize metrics.

In-network telemetry/Request tracing

The next step after observing how services interact with each other is the ability to track a single request through the different services to discover in which service is a problem or bottleneck. In articles cited in section 2.4.4 authors mention that one of the most important features was in-network telemetry and tracing.

OpenVSwitch can provide service-level metrics, but by them it is not possible to track relations between specifc requests between many pairs of services. Usually such a tracing is achieved by adding a special HTTP header in an originating service and passing it to the other dependencies. However, such an approach would again require reading data from high-level protocols. Moreover, OpenVSwitch statistics do not track other fields than matched in flow so it would be impossible to get identifier in statistics.

P4 offers in-band network telemetry which helps to track the request, but microservices would need to be aware of such system and pass flow identifier when it calls its dependencies.

Route permissions/Intentions

Under route permissioning concept we defined a concept answering if a pair of services is able to communicate. The rules are defined upfront, so a controller should know which pairs should reach each other. Those rules have to be later implemented in the switches.

It is definitely possible to achieve this when communication is defined in a binary way (can communicate on all routes/cannot communicate). However, usually the rules include specific routes. It leads to the problems from section 4.2 since the route need to be extracted from the HTTP packet.

Canary deployments

Canary deployments are based on HTTP headers which lead us to the problems from section 4.2. If this problem is solved, then the flows that are created based on following structure might be used:

Listing 3: Structure for canary deployments flows

```
table=1,priority=100,http_header["HEADER"]=v2,
actions=mod_ip_dst=<ip>,mod_tp_dst=<port>,table=2
```

4.2. Reading information from application-level protocols on switches

As was shown in section 4.1 many functions of service mesh are driven by information contained in HTTP header. Reading this information is needed for the switch to make a decision about where to route traffic. For example, if one wants to implement canary deployment, they need to read header containing a version of service to use.

OpenFlow does not support creating flows based on HTTP headers. P4 as a more advanced solution might be used for this goal. However, it was designed to read fixed-length fields like IP and TCP and reading variable-length ones is more complicated, as was shown in [51], but might be possible as the other solutions show [18] [25].

As a part of this thesis, two approaches to this problem were defined:

- inspect HTTP packet using P4
- create a custom protocol that is friendly to parse for P4

The next two sections contain research on these two approaches.

4.2.1. Reading HTTP header using P4

The concept of reading unstructured application-level packets such as HTTP, HTTPS, or DNS is called deep packet inspection [18]. P4 was not designed for this goal - it can parse only well-structured headers such as IP or TCP. The reason for that is a lack of undesirable instructions for packet forwarding, such as loops. It complicates using programmable switches as, for example, application-layer firewalls or service mesh proxies, since they need to access data like a route or a header.

Some work has been already done in order to read this information from high-level protocols. It was covered in the papers [18] and [25]. This thesis might benefit from these papers.

Article [18] presents an idea of reading the destination URL of a packet to filter traffic to malicious sites, so that the switch would work as an application-level firewall. To achieve this, authors split a packet into three parts - data from Ethernet, IP and TCP/UDP, data before URL and URL itself, and tries to statistically determine the size of each part based on OS, browser, protocol, etc.

The solution is very limited to the specific usage and might be hard to generalize. It reads specific data that is on a well-known place in a packet. Although it shows that it is somehow possible to read something, it is not possible to leverage this solution in reading HTTP headers, since their position might differ in the packet. The URL is possible to be read even from the HTTPS packet if it uses SNI.

Much more promising is the concept presented in [25]. It shows an idea of implementing Deterministic Finite Automata (DFA) to perform string searching, which hardly leverages the architecture benefits of a device on which it is run such as TCAM memory. It states that despite the additional overhead on the number of times that the packet is processed, the speed results are really promising, and the algorithm can search for an arbitrary string which is defined at compilation time. This assumption is sufficient for the service mesh since things like HTTP header names are known upfront. The solution is able to even search for a limited number of regular expressions.

The drawback that the authors notice is the fact that the algorithm will not find a match if it spans multiple packets. This should not cause a problem for reading headers since they are placed at the beginning of the packet. Later, to properly route the next packets, a similar connection tracking mechanism as that used in section 4.1.3 might be used. Moreover, because it looks for the given string, it might be harder to read data such as status code since they do not start with the description of the field like headers. However, it has a finite number of states, so all possible values could possibly be evaluated or the solution from [18] could be used since it is placed in a specific place.

A different approach has been presented in the previously mentioned article [51]. It reads variable-length data from DNS packets. It defines a separate parser state for each length of the URL. It has a limitation to be able to read only 64B so it disqualifies it from reading data in headers, but might be leveraged to read method and status code from HTTP headers which are placed at the beginning of it.

The P4 specification also defines modules called externs, which are the external pieces of code that might be implemented for a specific architecture and called from P4. The problem with it is that they are not independent of the device on which they run. As they are implemented in high-level language, it might be possible to perform string searching there, but it may not provide more value than the solution from [25], which is not bound to the device.

However, the biggest problem in all the described solutions is the fact that content can be encrypted, which successfully prevents the content from being read by switches, no matter which solution is used. Some kind of a solution to that is to force packets leaving a service to be decrypted and encrypt the content when it passes to a different switch by leveraging VPN tunnels between them.

Another thing that might cause a problem is the character casing. According to [39] HTTP headers are case-insensitive, so strings cannot be compared byte by byte. Usually, a case-insensitive comparison is done by lower casing all characters before, but such an operation is not trivial on switches.

4.2.2. P4-friendly HTTP-like protocol

The previous section has shown that it is possible to read data from well-known application-level protocols. Unfortunately, none of the solutions is natively supported, which creates limitations of those solutions. This section aims to propose an extension of the current protocols that would be friendly to parse by programmable switches.

The idea is based on the addition of an additional HTTP header that is structured and possible to read by switches. The header would go before the HTTP packet and store data needed by the service mesh, such as a status code or some headers. Moreover, the header should not be encrypted by SSL to avoid a problem with encrypted data. Because of that, no sensitive data should be carried there.

Another problem is that not all fields are of fixed length. Fields such as route or headers differ in length. There are different approaches to handling this kind of data. Here, only two of them are mentioned:

- Fix the size of all fields of one kind. Because it has to match the longest possible word it would lead to a big space waste when words are shorter.

- Store the length and the value. The length precedes the value so that a parser knows how many bytes it should read. However, it is not natively supported by switches as shown in [51].

Both solutions have their own mentioned drawbacks. Because the goal of this thesis is to focus on performance, the second approach has been chosen. This thesis only proposed the protocol, as it focused on the implementation of other parts. The protocol was not evaluated.

To better explain the the proposed header an example HTTP packets (request and response) from listings 4 and 5 can be used.

Listing 4: Example HTTP request

```
GET /api/list HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Accept: application/json
```

Listing 5: Example HTTP response

```
HTTP/1.1 200 OK
Host: localhost:8080
Connection: keep-alive
Content-Type: application/json
Content-Length: 722

..body..
```

The header built by the proposed algorithm and containing information from packets might be organized in the following way:

- type - (0 or 1) - defines whether the packet is a request or a response. Can be stored on one bit.
- status code - response-only (200) - although there are 5 spaces of HTTP status codes [39] (1xx, 2xx, 3xx, 4xx, 5xx) the possible values are sparse, so this field could be compressed. For the sake of simplicity, 10 bits to store status code without compression can be assumed. Empty
- method - request-only (GET) - there are 8 possible values [39] that can be encoded with consecutive numbers. To allow extension methods, the 4 bits should be assumed for it.
- length of route (LoR) - request-only (9) - stores the route (/api/list in this example) length.
- number of headers (NoH) - (3 for the request, 4 for the response in the examples) - stores how many headers it contains.
- route - request-only (/api/list) - HTTP route which is of length defined in LoR.
- headers - HTTP headers extracted from the structured header. Their count should match the NoH field. Each header has the following structure (example of Content-Type):

- length of label (12)
- length of value (16)
- label (Content-Type)
- value (application/json)

This structure should provide enough information to successfully route traffic through a service mesh. The main drawback of it is that the structured header must be provider by the client, which makes connection inside service mesh, which requires modification of existing services and common libraries in all programming languages, which makes it really hard to achieve.

4.3. Integration with container orchestrators

In the background and problem sections, it was stated that the work has to easily integrate with existing container orchestrators because they are the foundation of today's applications. The following section contains the proof-of-concept integration of the proposed solution with an orchestrator.

Kubernetes is the most mature orchestrator and is supported by a large community nowadays, so I decided to use it to create an example of integration. The solution takes advantage of the operator pattern [33] together with the OVS CNI plugin [26], which was described in the background section 2. Additionally, the Consul k8s [20] plugin has been utilized to install the Consul service registry within a cluster.

The proposed solution, except for using two existing extensions: Consul k8s and the OVS CNI plugin add two new custom components, which are designed to manage the state of the cluster. The first is the OVS service mesh controller [30] that monitors the pods in the cluster and creates OpenFlow rules for them. The second is a simple pod that exposes an REST API to control Open VSwitch.

4.3.1. Solution architecture

The concept is presented in Figure 5. Despite the fact that the concept is made up of many parts, the idea is fairly simple.

The working architecture in the figure consists of two nodes (more specifically kublets), on which Kubernetes is installed and are joined into a cluster. OpenVSwitch is installed on both nodes, and one bridge is created. Bridges are connected into a full mesh (in this specific example, only one connection is made between two hosts). Containers from the default namespace also connect to this bridge. A separate namespace is created for the controller and proxy, which connects to the OpenVSwitch socket on a specific host. Moreover, the Consul registry is added to the cluster.

The first component, Consul, is installed on a cluster using the Consul k8s plugin. The default mode of the plugin does not fit well to the solution presented in the thesis because it spawns a proxy next to each pod. To change this behavior, there exists a flag that allows one to disable it. For the proposed solution, switches are taking responsibilities of the proxy, so the default behavior is turned off. Consul only stores information about services, which is used by the controller to create rules.

The next already existing part, which is used in the example, is OVS CNI plugin. CNI plugins are responsible for connecting the container interfaces to the right bridge. OVS plugin is created to connect pods through Open VSwitch.

In the first design of the solution, the idea was to put the logic of creation and insertion of OpenFlow rules for upstream in the plugin. However, after the analysis, it appeared that the controller the uses operator pattern. CNI plugin is designed to work on a lower level and handles container's interface actions such as add, remove, or change. At this level, the information about a service packed in container should not passed. What is more, it does not have information about service lifecycle.

For app containers on a single host, there is a bridge created to which they connect. The convention is to have the same bridge name on all hosts, so that containers are not bound to any

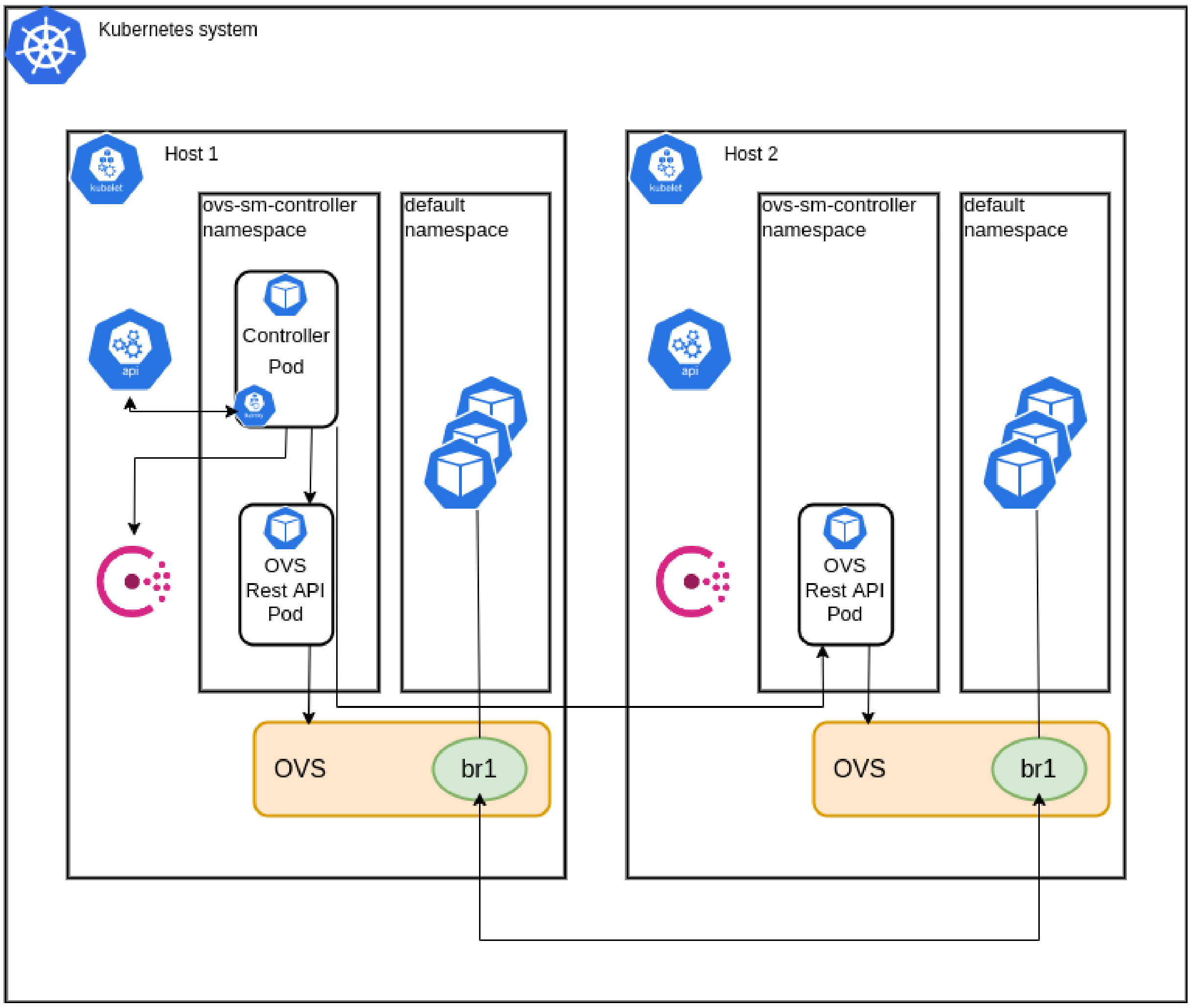

Figure 5: Integration with Kubernetes

host. Bridges on different hosts are connected using VXLAN interfaces. In the current design the bridges create fully-connected mesh (each switch is connected to all others). The optimal connection schema is not part of this thesis.

The controller is the core of the solution and the most complex element of it. It is responsible for the state of a network to become as stated in job specifications. It has the following functions:

- watching Kubernetes state and getting pod data from Kubernetes API,
- registering new pod in Consul,
- creating OpenFlow rules using pod specification and Consul state,
- adding the rules to the right switch.

The last part is the OVS proxy, which emerged after the controller-switch communication problem appeared (it is better described in the next section). On a high-level, it is deployed to all hosts and provides the REST API to communicate with a local instance of OpenVSwitch.

4.3.2. Implementation

The implementation of existing parts of the proposed solution is published on GitHub and is linked here as a reference [30].

Controller

In the presented example, the Controller is built using Kubernetes operators and, more specifically, the Kubebuilder framework [31], which provides a set of tools for easier creation of controllers.

The first responsibility is mostly built in the Kubebuilder framework. It provides tools and libraries to communicate which Kubernertes API. Both watching and fetching data is done in one line using the framework. Moreover, for the controller deployment as a pod in a cluster it automatically adds kube-rbac-proxy which exposes a port to a controller by which it can communicate easily with cluster. The proxy is responsible for authorization in Kubernetes RBAC.

Using API the controller watches containers being added/changed/removed in to the Kubernetes cluster. There are some annotations that are used to specify how a pod should be processed by the controller:

- ‘ovs.servicemesh.arkadiuss.dev/consul-register‘ - declares if the pod should be registered in consul and by it, if it should be a part of service mesh

- ‘ovs.servicemesh.arkadiuss.dev/upstreams‘ - specifies with which services the pod can communicate and using which local port

- ‘ovs.servicemesh.arkadiuss.dev/ovs-cni-network-name‘ - states what is the name of the service mesh container network. It should be the same for all containers and might be replaced with ‘k8s.v1.cni.cncf.io/networks‘.

In the implementation of the second point, an existing solution called kube-consul-register [8] has been used as inspiration. In particular, the base concept of fetching pod specification from Kubernetes API and translating it into Consul registration as well as discovery of the consul service.

The current implementation of this part of the controller is far less advanced and provides only app registration without the possibility of deregistering it, when pods stop. Moreover, instead of three ways to discover the Consul instance in the cluster, the controller always communicates with the main instance.

The controller needs to know the IP address using which the container is attached to the network. It is simply provided by the ‘k8s.v1.cni.cncf.io/network-status‘ container annotation to which OVS CNI plugin inserts the current state. The more problematic thing is how the addresses should be assigned. The plugin implements the IPAM interface [13] that defines how the address is fetched. There are 3 types of assignment:

- static - IP address defined for each pod in the pod specification

- host-only - addresses are assigned from the pool which is local for each host

- dhcp - addresses are assigned from a global DHCP pool.

For simplicity in the example created for this thesis, the static type is used. However, in the desired solution addresses should come from dhcp pool to avoid conflicts (containers are distributed among different hosts, so host-only type is not sufficient) and for easier maintenance (static type would require a lot of synchronization with other containers).

The process of creating OpenFlow rules is analogous to the approach from the previous section 4.1. The only difference is that it is much harder to determine which port number of switch the controller should use. To solve it we can rely on NORMAL switch action which should flood the network when the first packet for a given mac is being processed and store the port in the table for further processing.

The fourth responsibility turned out to be more complicated. Although OpenVSwitch provides a way to access it remotely, presented in [41], it is not simple from a different host. The possible idea is to set the controller to ‘ptcp‘ mode in which it accesses remote connections. The problem with such approach is that I could not communicate with it when it was listening on an interface not attached to the bridge. The idea for a solution to this is to create a management vlan on this bridge leveraging the OVS CNI plugin. Only the controller would be attached to this vlan. The final obstacle is that the plugin does not assign an IP address to an interface that is connected to switch bridge. It would require modification of the plugin.

The way to overcome this problem is to implement an OVS proxy. The proxy is deployed to each host and exposes a simple REST API by which the command can be sent to the right host. The proxy then communicates directly with the local socket.

The controller is deployed as a regular pod to a cluster together with kube-rbac-proxy. Separate namespace is created for the controller and proxies.

OVS proxy

As stated above, the OVS proxy should be responsible for exposing a simple REST API on each host to access OpenVSwitch.

The concept of a proxy is very simple. Whenever the controller needs to insert entries into the OVS database, it finds the host on which a specific container is deployed. The proxy exposes a "well-known" port on a host to simplify discovery. When the right proxy is determined, the controller sends the command to the proxy, which passes it directly to the switch. The proxy exposes its service using the Kubernetes HostPort feature [32].

In the linked repository, proxy was not implemented, and rules were manually entered into the switches while testing. As an alternative, the proxy implemented in Antrea [2] can be used for this goal.

Traffic functions

There is also a problem of specifying some functions of the service mesh such as permissions or canary deployments.

The solution to that is the way it was implemented in frameworks like Istio. For example, when an administrator wants to specify a canary deployment for the service, they create a resource specification called "VirtualService" in which they enter rules for traffic. This resource is observed by the controller and enforced by the control plane on proxies. In the solution for this thesis, the controller would create flow rules for programmable switches that match the specification. Custom functions are defined using Custom Resource Definitions.

Category	Function	Status
Traffic management	Server-side service discovery	✓
	Client-side service discovery	●
	Load balancing	●
	Canary/Staged rollouts	❍
	Retries	✕
	Timeouts	✕
	Circuit breaking	✕
Observability	Network metrics	●
	Tracing	✕
Security	L4 traffic permission rules/intentions	●
	L7 traffic permission rules/intentions	❍
	Encryption	✕
	Authentication and authorization	✕

Table 3: Service mesh functions implemented/researched in this thesis. Meaning of symbols:
✓ - implemented proof-of-concept, ● - researched - possible to implement,
❍ - researched - needs high-level protocols data, ✕ - not reasearched/hard to implement

5. Conclusions and results

This thesis investigates the possible usage of programmable switches in the service mesh. More specifically, leveraging the features of programmable switches and data plane programmability to extend SDN possibilities to higher layers than traditional layers 1-3 would result in creating L2-L7 SDN. Since service mesh is a very broad concept, only some features of it were investigated in this thesis. This section contains conclusions from research and evaluation.

Section 4.1 shows that the greatest achievements are in connectivity between service and traffic management. It is possible to achieve service discovery using only OpenVSwitch. Table 3 contains a detailed list of what was done at which level.

Many of the functions require reading information from high-level protocols as described in Section 4.2, this part was only researched and not implemented but is needed to achieve: canary and staged rollouts since it needs information from HTTP headers, retries, timeouts, and circuit breaking because of the need for status code. In addition, request tracing can be achieved either by a custom solution that leverages data-plane programmability or information from HTTP headers.

The most complicated to realize are the functions focused on security. Switches cannot decrypt or encrypt traffic, so they can only be based on encrypted tunnels while connecting switches. Traffic permission is route-oriented, but route is sent in an HTTP packet. Authentication and authorization are also based on information from HTTP headers.

Optimistic results are also achieved in Section 4.3, which shows that the solution can be integrated with container orchestrators. All necessary components are possible to be connected and can operate successfully together. The full implementation is complex but can be achieved as an extension of the existing proof-of-concept.

5.1. Performance tests

The implemented part of the solution presented in this thesis was compared with the Consul Connect model set-up in terms of network performance. For the tests, the already mentioned Hashicups app was used and deployed to two Kubernetes clusters. The first was set up to route traffic over programmable switches as described in sections 4.1 and 4.3 while the second was the default cluster offered by the Azure cloud - AKS.

Evaluation has been carried out using Fortio [12]. It is a tool that generates load on a target based on initial parameters. The user is, for example, able to define a request, repeat it chosen number of times, and specify level of parallelism. The tool will execute requests and create a histogram of the response times.

Both clusters had the same number of nodes - 2 - and used the same virtual machine size 'Standard_B2ms', which offers 2 vCPUs of 8 Gb of memory each. In the programmable network one, the nodes were connected using Azure virtual network, and the OVS CNI plugin was used to create a container network. Under the hood AKS uses Azure CNI which connects containers using Azure virtual network. The setup of both is automated and repeatable by using Terraform [22]. The code is available at [28].

Different test scenarios were prepared to compare network clusters under different conditions. There were two types of requests:

- Main page load (Frontend), where the traffic goes only through the nginx proxy and reaches frontend microservice, but the response size is bigger than in the next scenario.

- Products list load (API), where a call spans over multiple services, starting from proxy, through public-api, products-api, and reaching database in the end. The response is smaller in this case.

The other parameters that had a significant impact on response time were the number of simultaneous connections to the server and the distribution of pods on the nodes. Tests were conducted for 1, 8, 100 and 200 parallel requests. The fact that all pods are deployed on one instead of more nodes obviously reduces the response time since communication happens locally and there is no need to communicate over nodes. In each test 1000 requests were conducted. Moreover, to avoid latency related to the connection to the cloud, tests were sent from a container deployed with the app, which also was part of the service mesh.

Naturally, the way the application is written also impacts the response times, but in both scenarios, the same application was used and the same endpoints to make the comparison independent of the application.

The selected histograms are presented and compared below. The complete results are presented in Table 4. Figures 6 - 11 compare the results for the same scenario (Frontend, both pods on a single node) for both cluster and different number of parallel requests. The graphs show that the performance is better on the cluster that uses OpenVSwitches. When requests were run sequentially (Figures 6 and 7), the distributions resemble the right half of the natural distribution. The peak for Consul Connect is on the 5ms, whereas for programmable switches one is around 2.5ms. The result is two times better than using a separate proxy. It shows that at least half of the request time was spent in the network instead of service.

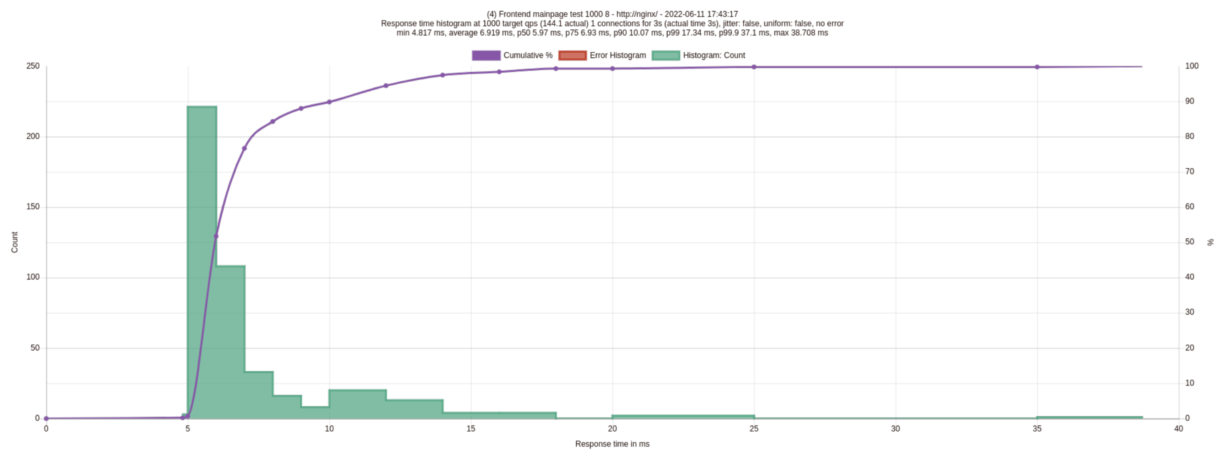

Figure 6: Consul Connect on AKS: 1 parallel Frontend request on a single node

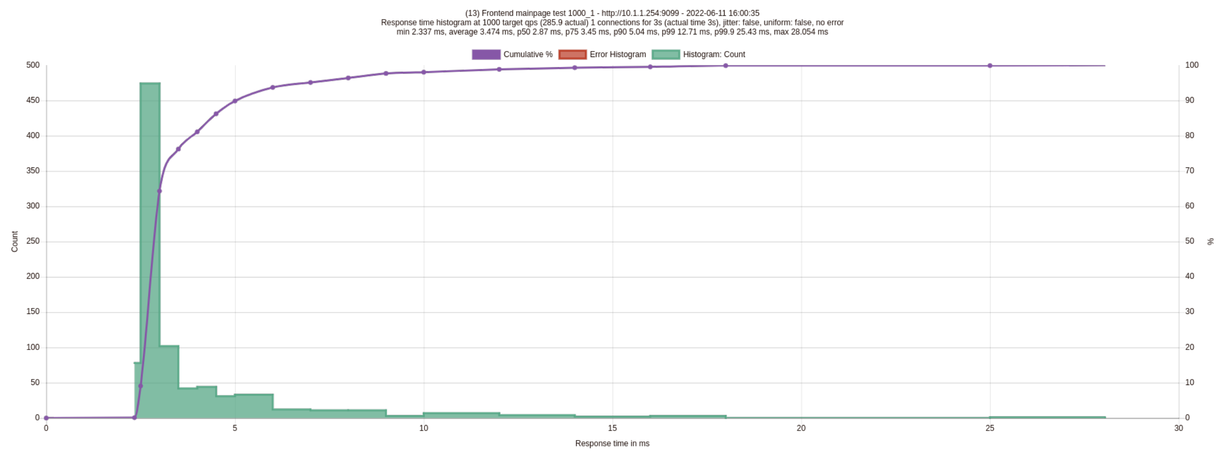

Figure 7: Service mesh based on OpenVSwitches: 1 parallel Frontend request on a single node

Figures 8 and 9 show the same scenario for 8 requests in parallel. The results do not resemble the normal distribution for Consul Connect and the times are much longer (average is 8 times longer for Consul Connect and 11 times for the OpenVSwitch cluster).

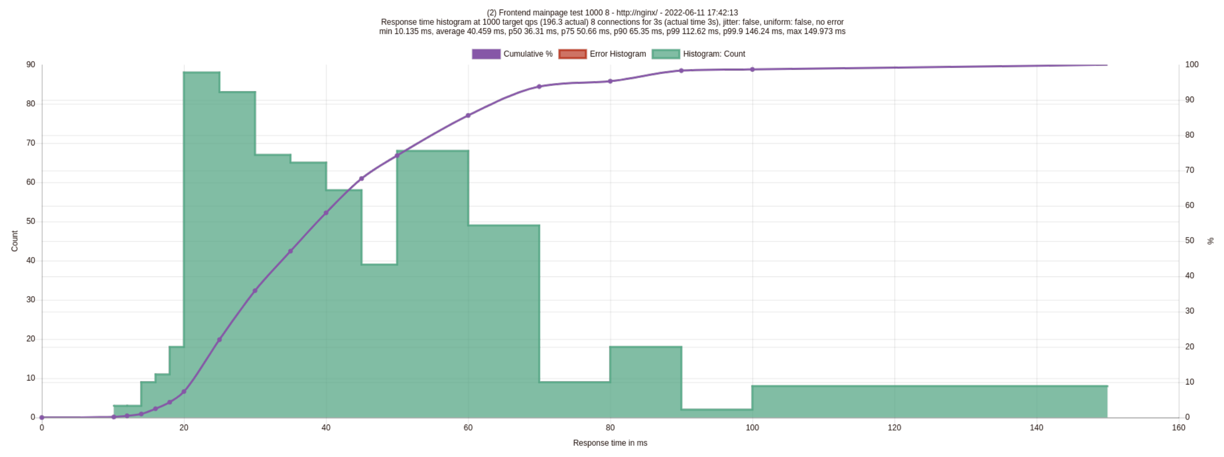

Figure 8: Consul Connect on AKS: 8 parallel Frontend request on a single node

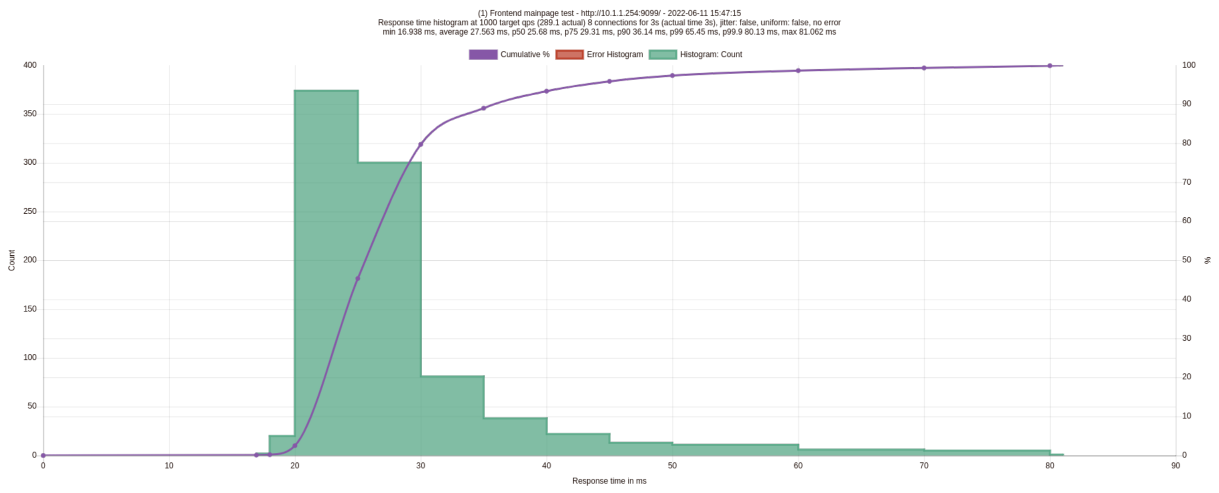

Figure 9: Service mesh based on OpenVSwitches: 8 parallel Frontend request on a single node

Figures 10 and 11 show the above scenario for 100 requests in parallel. The increase in average in comparison to 8 requests in parallel was this time similar for both clusters (around 10 times).

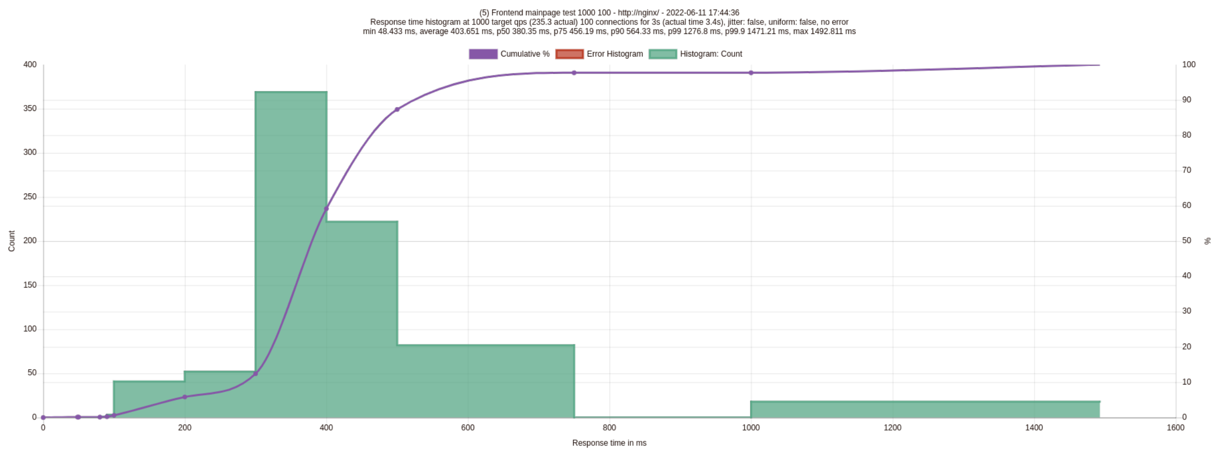

Figure 10: Consul Connect on AKS: 100 parallel Frontend request on a single node

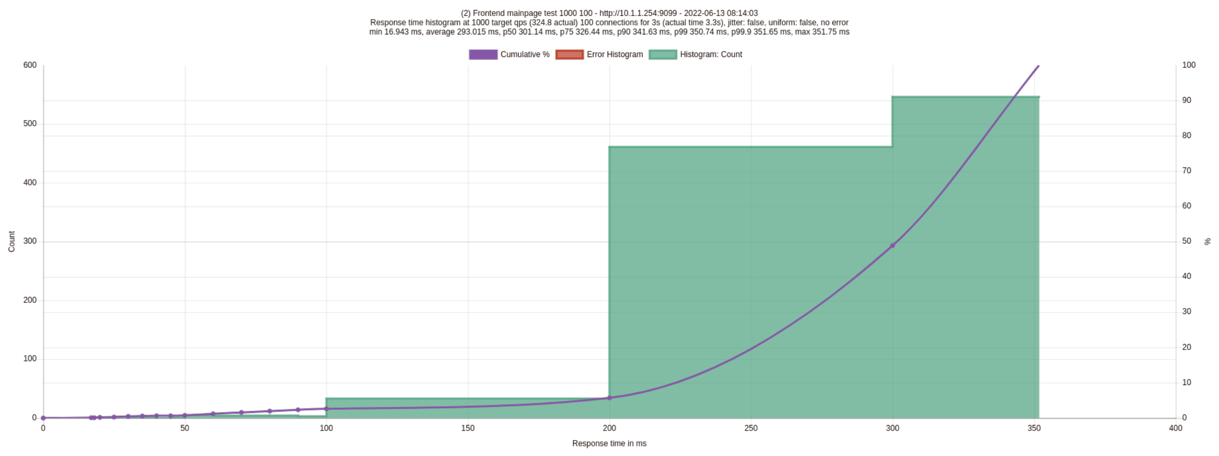

Figure 11: Service mesh based on OpenVSwitches: 100 parallel Frontend request on a single node

Another interesting result was shown by the first test of the Consul Connect setup. After the first attempt, the results were slightly worse than after the second and subsequent attempts. It is shown in Figures 12 and 13. It is probably a result of a service discovery and DNS requests that are performed in a lazy manner, so during the first request. In the programmable network cluster, this situation also occurred as shown in Figures 14 and 15. It is a result of the ARP queries that needs to be done at the beginning.

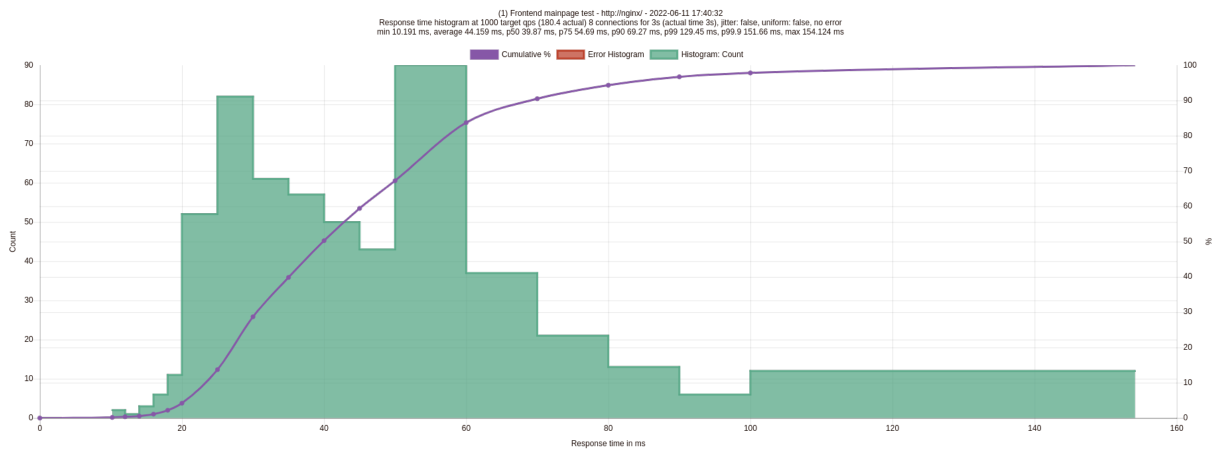

Figure 12: Consul Connect on AKS: 8 parallel Frontend request on a single node (first attempt)

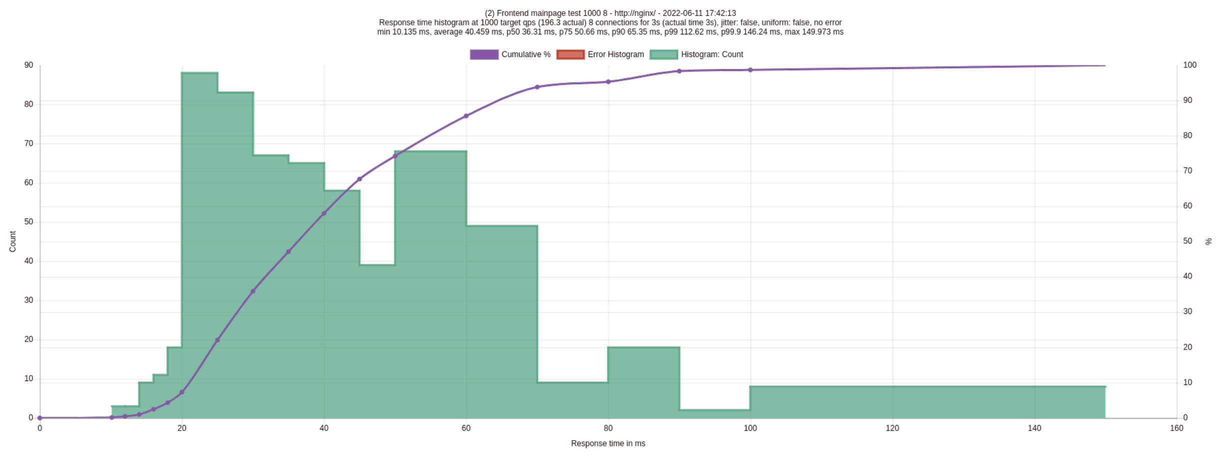

Figure 13: Consul Connect on AKS: 8 parallel Frontend request on a single node (second attempt)

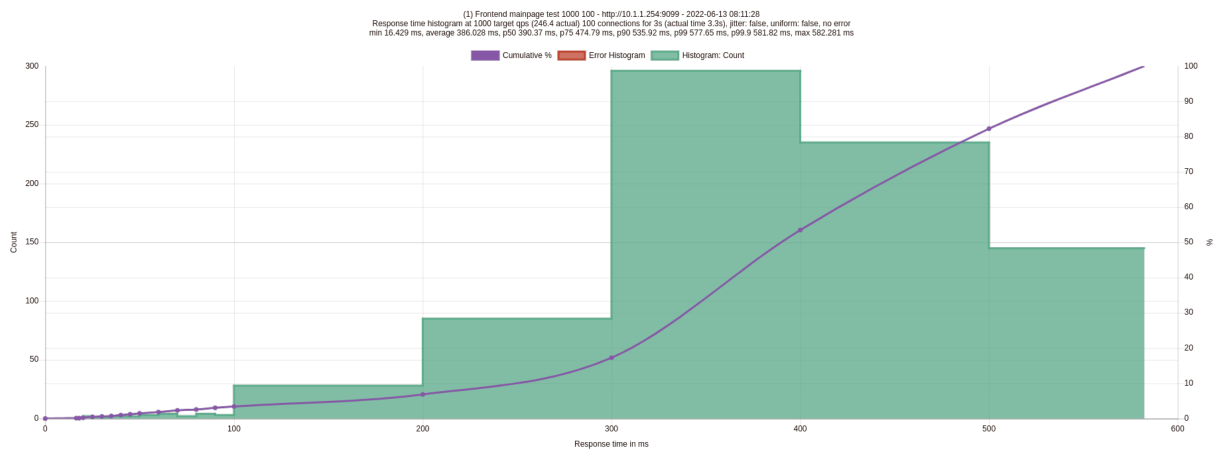

Figure 14: Service mesh based on OpenVSwitches: 100 parallel Frontend request on a single node (first attempt)

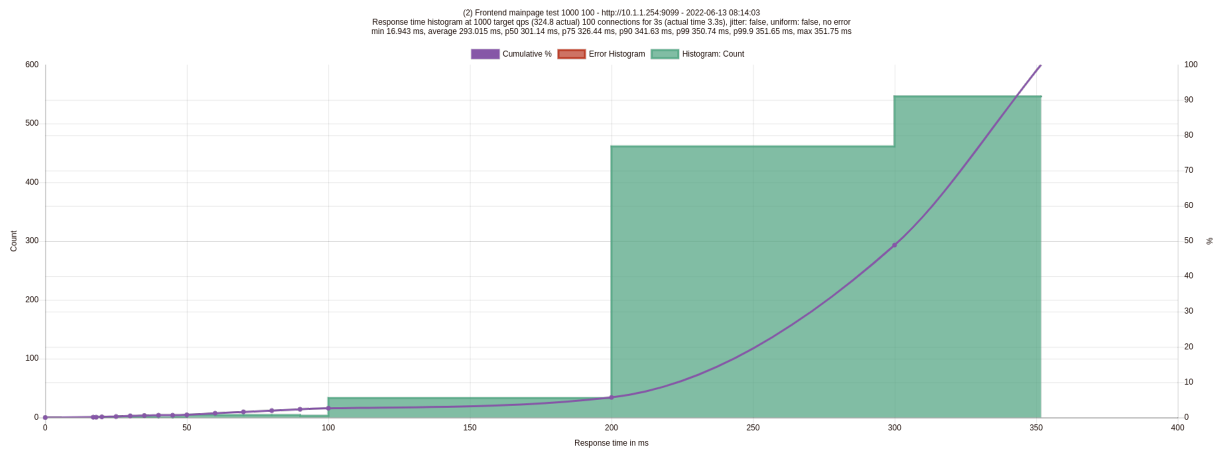

Figure 15: Service mesh based on OpenVSwitches: 100 parallel Frontend request on a single node (second attempt)

The summary of all tests performed is presented in Table 4. The frontend tests were performed on a single node, while the API tests were performed in both pods distributions, on a single node and on two nodes.

The results presented in the table are very promising. In almost all scenarios, the cluster based on programmable networks achieved two times lower response time for frontend test and 2-4 times better result for API tests. As expected, the more connections were initiated in parallel, the longer was the latency for both clusters. Notably, minimal times grow for AKS linearly, but for the proposed cluster they are much smaller, even for a higher number of simultaneous connections.

5.2. Conclusions

This thesis shows that it is possible to implement a service mesh that uses programmable switches to implement basic functions. However, it cannot be classified as the full L2-L7 solution, but its functions end at layer 4. The functions are very limited and focus mainly on traffic management and observability.

Test			Results					
Type	Pods	Parallel	Average [ms]		Min [ms]		Max [ms]	
			AKS	AZ-OVS	AKS	AZ-OVS	AKS	AZ-OVS
Frontend	SN	1	6.919	3.474	4.817	2.337	38.708	28.054
Frontend	SN	8	40.456	27.563	10.135	16.938	149.973	81.062
Frontend	SN	100	403.651	293.015	48.443	16.943	1492.811	351.75
Frontend	SN	200	835.139	486.481	65.233	28.284	2403.406	579.211
API	SN	1	14.456	3.441	11.044	2.853	38.489	24.785
API	SN	8	77.152	26.714	36.121	4.431	131.647	85.601
API	SN	100	923.501	359.112	186.77	4.833	3004.885	1371.172
API	SN	200	1577.674	676.223	119.774	4.719	4380.881	2698.787
API	MN	1	35.049	17.674	25.337	7.927	54.079	28.427
API	MN	8	59.52	24.732	37.173	11.727	98.314	59.183
API	MN	100	610.395	302.055	61.393	14.481	1959.487	1343.678
API	MN	200	1140.155	603.667	92.237	13.277	4003.147	2177.723

Table 4: Summary of the results. Shortcuts explanation:
Frontend - scenario in which the main page is loaded,
API - scenario in which coffee list is loaded which requires to go through multiple services,
AKS - test conducted on Azure AKS cluster,
AZ-OVS - test conducted on cluster based on OpenVSwitch deployed to Azure,
SN - services were on the single node,
MN - services were on multiple nodes

Using OpenVSwitch in the core of such network allows to achieve server-side client discovery, load balancing, service-level (L4) intentions, and network service-level metrics. Leveraging data plane programmability and the P4 ecosystem, all OpenVSwitch features (since data-plane programmability extends standard stack) plus the broader subset of features can be implemented including: client-side service discovery, in-network telemetry if supported by services by understanding and propagating identifiers. If the approaches to string lookup in the packet analyzed in the thesis are successful, then also features like staged rollouts, retries, timeouts, circuit breaking and L7 intentions are possible to achieve. It answers the first problem stated in the thesis.

The security aspects of the service mesh are the most complicated for programmable switches, since it requires encryption and data from high-level protocols. As the research shows, both things are hard to achieve, which complicates this problem.

The OpenVSwitch-based implementation achieves very promising results when analyzing response times. The latency was around 2x better than using Consul Connect with Envoy sidecars. However, achieving more features by leveraging P4 might degrade speedup because there is no production-ready implementation of P4 software switch, aggregation of so many features adds additional overhead, and string lookup which is needed to analyze traffic is not supported natively of these switches. Furthermore, the P4 ecosystem lacks some features needed to integrate it with container orchestrators, such as a CNI plugin that communicates with switches.

As an investigation of the third problem, an analysis of the work conducted on the problem of string lookup on programmable switches was done. This problem is barely solved right now, and only a few proof-of-concept solutions exist. Match-action pipelines are not designed to handle strings, and all solutions are workarounds to the problem. There is a chance to create a structured header containing necessary information, but it would require a modification of the clients.

Container orchestrators are the core of today's architectures, especially Kubernetes, which is highly extensible. Using the patterns defined for it, the service mesh based on OpenVSwitches can be implemented. The cluster with the necessary components can be automated. In cluster, the controller works together with the service registry to create rules for OpenVSwitches. It answers the second problem stated in this thesis.

However, the solution can provide a value even without an orchestrator. For older services that are no longer maintained and not extendable, switching the network to use programmable switches such as OpenVSwitch can provide the mentioned features, which can provide a great value to the old services.

5.3. Further work

This thesis resulted in the base for using programmable switches to implement service mesh. The first results are promising, but service mesh is a very broad concept and only part of it was researched here. Much more work would need to be done to make the proposed solution usable. Firstly, to make this solution competitive with existing service meshes, it would need to implement all functions, but to achieve this, there must be a support to process text on such switches. Recall that programmable switches are not designed to support it, because they should be able to analyze a packet in one go. Adding strings support would mitigate the speedup achieved in this thesis.

To overcome this problem, the interoperability of switches with sidecars could be investigated. The sidecar would be responsible for analyzing the handling of high-level protocols, preparing data in format convinient for switches and security. However, such an approach does not differ much from what is available today, and an additional sidecar can again cause higher latency, mitigating switches speedup.

In addition, also the possible extension of an existing Antrea plugin [2] would be considered as a better way to include service mesh functions. The potential problem is that the rules created by both solutions might not match and not work as expected together.

The proposed solution opens a way to implement the service mesh functions on hardware switches, which could further improve performance because dedicated devices are much faster. The problem with such an approach is that hardware switches can operate only at the host level rather than the container level, like OpenVSwitch. It would cause problems similar to the one in Section 4.1.

In the end, it is worth asking the question if the L2-L7 SDN would actually be beneficial for teams. A clear distinction between L2-L4 SDN and L4-L7 service mesh helps divide responsibility and having separate teams for each of them. One solution would be harder to maintain, which would mitigate better performance.

A. Open Flow rules for server side service discovery

The full set of rules to achieve connectivity of services in the architecture presented in Figure 4 and described in 4.1.3 is shown on listing 6. The rules were prepared based on schema from listing 1.

Listing 6: Open Flow rules for server-side service discovery

```
ovs-ofctl add-flow s1
"priority=50,tcp,in_port=1,ip_dst=192.168.1.250,tp_dst=9191,
action=ct(commit,zone=1,nat(dst=192.168.1.11:20863)),
mod_dl_dst:00:00:00:00:00:01,output:1"

ovs-ofctl add-flow s1
"priority=50,tcp,in_port=1,ct_state=-trk,
action=ct(table=0,zone=1,nat)"

ovs-ofctl add-flow s1
"priority=50,tcp,in_port=1,ip_dst=192.168.1.11,ct_state=+est,
ct_zone=1,action=1"

ovs-ofctl add-flow s1
"priority=50,tcp,in_port=1,ip_dst=192.168.1.250,tp_dst=9340,
action=ct(commit,zone=1,nat(dst=192.168.1.13:13427)),
mod_dl_dst:00:00:00:00:00:03,output:3"

ovs-ofctl add-flow s1
"priority=50,tcp,in_port=3,ct_state=-trk,
action=ct(table=0,zone=1,nat)"

ovs-ofctl add-flow s1
"priority=50,tcp,in_port=3,ip_dst=192.168.1.11,ct_state=+est,
ct_zone=1,action=1"

ovs-ofctl add-flow s1
"priority=50,tcp,in_port=1,ip_dst=192.168.1.250,tp_dst=9341,
action=ct(commit,zone=1,nat(dst=192.168.1.12:19572)),
mod_dl_dst:00:00:00:00:00:02,output:2"

ovs-ofctl add-flow s1
"priority=50,tcp,in_port=2,ct_state=-trk,
action=ct(table=0,zone=1,nat)"

ovs-ofctl add-flow s1
"priority=50,tcp,in_port=2,ip_dst=192.168.1.11,ct_state=+est,
ct_zone=1,action=1"

ovs-ofctl add-flow s1
"priority=50,tcp,in_port=2,ip_dst=192.168.1.250,tp_dst=9567,
action=ct(commit,zone=1,nat(dst=192.168.1.21:5432)),
mod_dl_dst:00:00:00:00:00:04,output:3"

ovs-ofctl add-flow s1
"priority=50,tcp,in_port=3,ct_state=-trk,
action=ct(table=0,zone=1,nat)"

ovs-ofctl add-flow s1
"priority=50,tcp,in_port=3,ip_dst=192.168.1.12,ct_state=+est,
ct_zone=1,action=2"
```

List of Figures

List of Tables

Listings

References

[1] Gianni Antichi and Gábor Rétvári. "Full-Stack SDN: The Next Big Challenge?" In: *Proceedings of the Symposium on SDN Research*. SOSR '20. San Jose, CA, USA: Association for Computing Machinery, 2020, pp. 48–54. ISBN: 9781450371018. DOI: 10.1145/3373360.3380834. URL: https://doi.org/10.1145/3373360.3380834.

[2] Antrea.io. *Antrea*. URL: https://github.com/antrea-io/antrea.

[3] Pat Bosshart et al. "P4: Programming Protocol-Independent Packet Processors". In: (July 2014). DOI: 10.1145/2656877.2656890. URL: https://doi.org/10.1145/2656877.2656890.

[4] Cilium. *Cilium*. URL: https://cilium.io/.

[5] Cloud Native Computing Foundation. *CNCF Survey Report 2019*. URL: https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Survey_Report.pdf.

[6] Cloud Native Computing Foundation. *CNCF Survey Report 2020*. URL: https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.

[7] Consul. *Transparent Proxy*. URL: https://www.consul.io/docs/connect/transparent-proxy.

[8] Tomasz Czekajlo. *kube-consul-register*. URL: https://github.com/tczekajlo/kube-consul-register.

[9] DigitalOcean. *Prometheus OpenVSwitch Exporter*. URL: https://github.com/digitalocean/openvswitch_exporter.

[10] Envoy. *xDS*. URL: https://www.envoyproxy.io/docs/envoy/latest/api-docs/xds_protocol.

[11] Nick Feamster, Jennifer Rexford, and Ellen Zegura. "The road to SDN". In: (Dec. 2013). URL: https://queue.acm.org/detail.cfm?id=2560327.

[12] fortio. *fortio*. URL: https://github.com/fortio/fortio/#fortio.

[13] The Linux Foundation. *CNI IPAM plugin*. URL: https://www.cni.dev/plugins/current/ipam/.

[14] The Linux Foundation. *eBPF*. URL: https://ebpf.io/.

[15] Gartner. *Gartner's Hype Cycle 2021*. URL: https://www.suntechnologies.com/data_factory/hype-cycle-for-monitoring-observability-and-cloud-operations-2021/.

[16] Google Cloud Architecture Center. "Supporting your migration with Istio mesh expansion". In: (). URL: https://cloud.google.com/architecture/supporting-your-migration-with-istio-mesh-expansion-concept.

[17] Paul Göransson, Chuck Black, and Timothy Culver. “Chapter 3 - Genesis of SDN”. In: *Software Defined Networks (Second Edition)*. Ed. by Paul Göransson, Chuck Black, and Timothy Culver. Second Edition. Boston: Morgan Kaufmann, 2017, pp. 39–60. ISBN: 978-0-12-804555-8. DOI: https://doi.org/10.1016/B978-0-12-804555-8.00003-X. URL: https://www.sciencedirect.com/science/article/pii/B978012804555800003X.

[18] Sahil Gupta et al. “Demo: Simple Deep Packet Inspection with P4”. In: *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. 2021, pp. 1–2. DOI: 10.1109/ICNP52444.2021.9651973.

[19] HAproxy. *HAproxy*. URL: https://www.haproxy.org/.

[20] HashiCorp. *Consul k8s plugin*. URL: https://www.consul.io/docs/k8s.

[21] HashiCorp. *Demo App - HashiCups*. URL: https://github.com/hashicorp-demoapp.

[22] Hashicorp. *Terraform*. URL: https://www.terraform.io/.

[23] Frederik Hauser et al. *A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research*. 2021. arXiv: 2101.10632 [cs.NI].

[24] Dariusz Jędrzejczyk. “Migrating to Service Mesh”. In: (May 2020). URL: https://blog.allegro.tech/2020/05/migrating-to-service-mesh.html.

[25] Theo Jepsen et al. “Fast String Searching on PISA”. In: SOSR ’19. San Jose, CA, USA: Association for Computing Machinery, 2019, pp. 21–28. ISBN: 9781450367103. DOI: 10.1145/3314148.3314356. URL: https://doi.org/10.1145/3314148.3314356.

[26] k8snetworkplumbingwg. *OVS CNI plugin*. URL: https://github.com/k8snetworkplumbingwg/ovs-cni/.

[27] Matt Klein. “On the state of Envoy Proxy control planes”. In: (Mar. 2020). URL: https://mattklein123.dev/2020/03/15/2020-03-14-on-the-state-of-envoy-proxy-control-planes/.

[28] Arkadiusz Kraus. *Automatization for performance tests*. URL: https://github.com/arkadiuss/programmable-network-in-service-mesh/tree/main/performance-tests.

[29] Arkadiusz Kraus. *Implementation of the server-side service discovery*. URL: https://github.com/arkadiuss/programmable-network-in-service-mesh/tree/main/labs/service-mesh-sd-full.

[30] Arkadiusz Kraus. *OVS Service Mesh Kubernetes Controller*. URL: https://github.com/arkadiuss/ovs-service-mesh-k8s-controller.

[31] Kubebuilder. *Kubebuilder*. URL: https://github.com/kubernetes-sigs/kubebuilder.

[32] Kubernetes. *ContainerPort api reference*. URL: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.19/#containerport-v1-core.

[33] Kubernetes Docs. *Operator pattern*. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[34] Wubin Li et al. "Service Mesh: Challenges, State of the Art, and Future Research Opportunities". In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2019, pp. 122–1225. DOI: 10.1109/SOSE.2019.00026.

[35] Lyft. *Envoy*. URL: https://www.envoyproxy.io/.

[36] Aspen Mesh. *Expanding Service Mesh without Envoy*. URL: https://aspenmesh.io/expanding-service-mesh-without-envoy/.

[37] Microsoft. *DNS SRV specificiton*. URL: https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-adts/c1987d42-1847-4cc9-acf7-aab2136d6952.

[38] NGINX. *NGINX*. URL: https://nginx.org/.

[39] Henrik Nielsen et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616. URL: https://www.rfc-editor.org/info/rfc2616.

[40] Open Networking Foundation. *OpenFlow Specification*. 2015. URL: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[41] Picos documentation. *Connecting to a controller*. URL: https://docs.pica8.com/display/picos2102cg/Connecting+to+a+Controller.

[42] Prometheus. *Prometheus*. URL: https://prometheus.io/.

[43] Romil Punetha, Adithya Menon, and Gaurav Arora. "Service Mesh in Production: An Istio Story". In: (Nov. 2020). URL: https://blog.cult.fit/engineering-design/service-mesh-story/.

[44] Chris Richardson and Floyd Smith. "Introduction to Microservices". In: *Designing and Deploying Microservices with NGINX*. May 2015.

[45] Chris Richardson and Floyd Smith. "Service Discovery in a Microservices Architecture". In: *Designing and Deploying Microservices with NGINX*. Oct. 2015.

[46] *Service Mesh*. URL: servicemesh.es.

[47] The P4 Language Consortium. *P4 specification*. URL: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[48] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[49] Paul A. Vixie. *Extension Mechanisms for DNS (EDNS0)*. RFC 2671. Aug. 1999. DOI: 10.17487/RFC2671. URL: https://www.rfc-editor.org/info/rfc2671.

[50] Troy Willmot. *RSSDP*. URL: https://github.com/Yortw/RSSDP.

[51] Jackson Woodruff, Murali Ramanujam, and Noa Zilberman. “P4DNS: In-Network DNS”. In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2019, pp. 1–6. DOI: 10.1109/ANCS.2019.8901896.

[52] Huabing Zhao. *What Can Service Mesh Learn from SDN?* URL: https://faun.pub/what-can-service-mesh-learn-from-sdn-1a4874edca03.