universität wien

BACHELORARBEIT / BACHELOR’S THESIS

Titel der Bachelorarbeit / Title of the Bachelor’s Thesis

„Resilient deployments with Istio service mesh“

verfasst von / submitted by

Ivan Varabyeu 01568715

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Bachelor of Science

Wien, 2020 / Vienna, 2020

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 033 521
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Bachelor Computer Science UG2002
Betreut von / Supervisor:	Amine El Malki BSc. MSc. Research Group Software Architecture

Contents

Abstract

The expanse of cloud computing technologies and movement to platforms as a service bring new challenges for developers. To stay efficient and to utilize most of the cloud features modern applications should be scalable, resilient and fast as in developing, so in testing and deploying to production. Some of the solutions are migrating from monolith to microservice architecture on already running projects or start to use cloud-native development patterns for completely new projects.

The splitting of monolithic application in several microservices introduces new challenges in software engineering processes. Extremely radical changes need to be done in operations departments to monitor, scale and deliver resilient workflow in the hole software life cycle. One of the most critical things to consider when running a complex distributed application is resiliency. In this thesis service mesh Istio running on top of the Kubernetes cluster will be introduced as a solution to provide visibility, control, security and fault tolerance to application deployment [11], [13]. The final goal is to demonstrate the possibilities of Istio and try out the resiliency features on the microservices application.

1 Introduction

The time of slow development cycles, deployments and support is gone. Users want to interact with services fast and without downtime. Cloud platforms have introduced a new advanced way to rapidly deliver results to clients. Migration to clouds also brought new challenges. Big monolithic applications were inefficient in scaling to custom loads [22]. This leads to the rethinking of the architecture of monolithic applications. Instead of packaging everything in one big project the idea with many independently developed and communicating with one another over network microservices came up.

Transition to microservices architecture helped to make application deployments more cloud friendly and made the fast code-to-market strategy possible. Automation, scalability and continuous delivery are among the most valuable attributes coming with these architectural changes in software engineering process [12]. All these factors and independence between microservices brought application resiliency on a completely new level [2].

Moving out from using virtual machines for deploying applications and the adoption of containers and automated deployments changed the scene one more time [12]. Containers are more lightweight and blazing fast in a startup in compare to virtual machines. The problem of delivering code from developers to the production environment is solved here by packaging applications and dependencies in images that run everywhere the same way.

Proper and efficient deployment strategies are crucial for microservices. Kubernetes container orchestration system provides all needed functions for the management of microservice applications. These include secret management, service discovery, horizontal scalability [17]. One of the problems is that it has no possibility to deal with network errors.

As the number of microservices grows developers and operations engineers lost the visibility of the deployment, control of communication inside the application. In this way, the overall availability of the service is falling. That is why the resiliency of microservices applications is very important. The failures take place on different levels: network, DNS, timeouts, internal exceptions [17]. Though it is almost impossible to eliminate all the failures, it is possible to tolerate them and to recover to maximize the availability of the application.

There are different approaches to overcome these challenges and one of them is to use service meshes to get full control over your microservices. The most valuable feature here is that very few changes or not at all should be added to the code of microservices. This also allows developers to focus only on the business logic of the application. Istio service mesh offers a complete solution to solve the complexity of distributed microservices applications [11].

In this work, the microservices application will be deployed on the Kubernetes cluster with installed Istio. The application itself was developed in the cloud computing course but was refactored and adopted to make the demonstration of Istio resiliency possibilities more visible. The resiliency of the deployment will be tested with load simulating and chaos testing. As a result of experiments - installation scripts and configuration files, graphics and console outputs of application behavior with and without Istio will be introduced.

The thesis has the following structure. In the first chapter different alternatives of service mesh architectures are discussed. Major idea, the theory about microservices, service meshes, Istio, and resiliency are introduced in the second section. Then the details of the implementation are described in the third chapter. Tests and the evaluation of the results are done in the last part.

2 Related work

There is already an intense need for service meshes for modern microservices applications. Many companies try to occupy this niche by developing their own implementations of service meshes solutions. So today with a big demand in getting observability and control over deployments there are also solutions with completely different architectures on the market. Most relevant issues that are covered by these architectures are security, tracing, observability, fault tolerance, fault injection, advanced routing. Libraries represent the most traditional way to include additional functionality to the application. Examples of such implementation are Hystrix and Ribbon from Netflix [19]. These libraries are used to get rid of network faults and not to implement code for communication inside application, but these should be developed and be up to date for each programming language in software stack of the company. This approach is not effective enough with microservice architecture because abuses polyglot idea of microservices. It also violates a principle of separation of business logic and communication and many changes in the code of microservices should be made.

Node agent represents the second way to deploy service mesh. The idea is to deploy a proxy agent on each node of the cluster, the same way Kubernetes

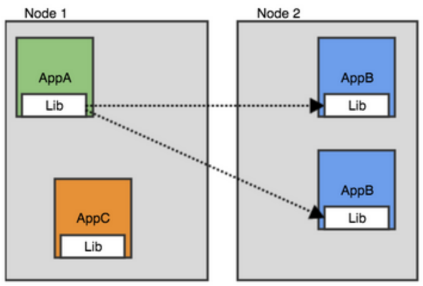

Figure 1: Service mesh based on libraries [19]

has kubelet on all nodes for registration purposes and to manage the pods [13]. An example of this type of architecture for service meshes is Linkerd [19]. As a disadvantage of this method, we can mention the existence of a one point of failure – node proxy. One failure in proxy will influence all the microservices deployed on this node. On the other hand this approach is more resource efficient [4].

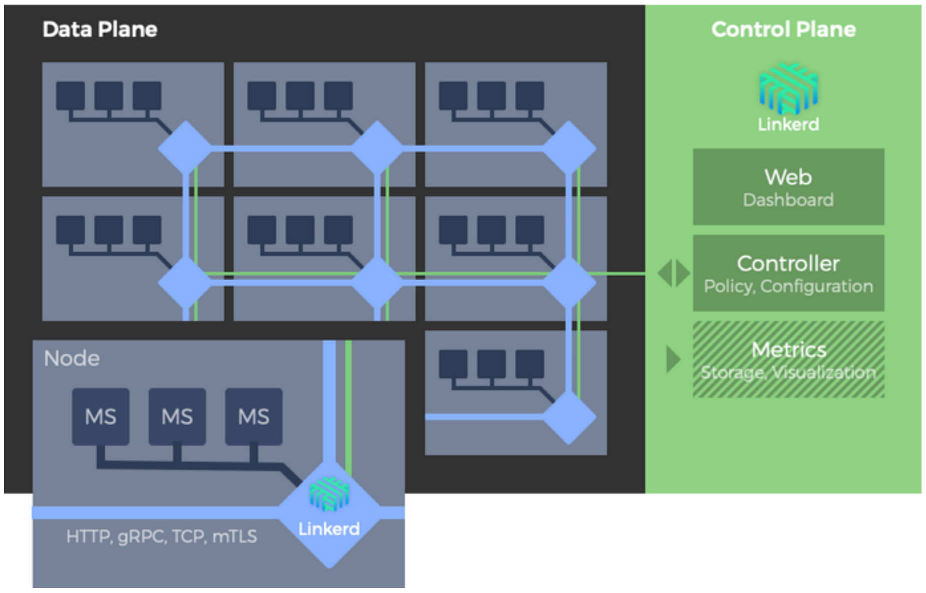

Figure 2: Linkerd architecture with node agents [4]

Sidecar proxy architectural pattern introduces another approach of integrating proxies in application deployment. In this kind of service mesh sidecars are inserted along each container so that every microservice pod has two containers inside: proxy and microservice itself. Examples of such architecture are Istio, Linkerd2, Consul. More details about this approach are covered in Istio chapter.

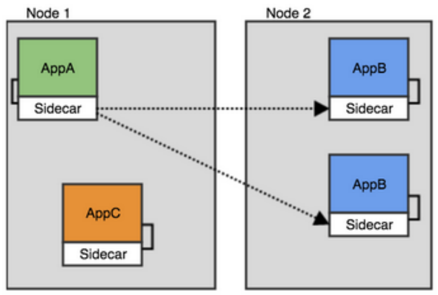

Figure 3: Service mesh based on sidecar proxies [19]

3 Major idea

There are plenty of tutorials online that utilize a sample application from Istio web site ("Bookinfo" application) to show typical service mesh and specific Istio features [11]. The idea of this thesis is to take an already implemented project, but not the one from Istio documentation, adapt it a little bit and provide a working demonstration of Istio resiliency features. In this way, it will be possible to see how difficult or easy it is to deploy a random application along with this service mesh.

The web application itself must be based on a microservices architecture. Trying to make focus on the operational part of the software engineering process and not focus on developing the service from scratch one of the solutions was to take a ready open source project from Github and deploy it with Istio [16].

After researching and trying out some of these projects the decision to take the application developed by myself in a cloud computing course was made. The text of the initial assignment can be found in the supplemental material. The application will be deployed in Kubernetes cluster with preinstalled Istio and configured sidecars auto injection for each pod. As Istio has plenty of resiliency functionalities, they will be examined one by one to provide a better overview of the results and also to minimize debugging time of possible deployment problems.

Some changes and additions were made to the original code of the web application. The initial commit in Github repository shows the start point of the project implementation. There you will also find Minikube and Istio installation scripts along with other developing environment scripts.

3.1 Microservices

Migrating from monolithic applications to microservices represents a challenge on itself [6]. There some reasons why one would like to completely redesign the production ready application. One of the reasons is that the updating cycle of the monolithic application is extremely slow. The work should be synchronized between different teams, that develop separate modules of the application and in the end, the functionality should not be lost [22]. The other reason is scalability. A monolithic application is just not efficient at scaling and can not provide the necessary velocity on load from the clients. As a result, we acquire unsatisfied users that costs the company much money.

Microservices represent an architectural pattern to split big monolithic applications in smaller independent services communicating with each other (often by means of HTTP requests). Each microservice is built around one small business logic. This architecture takes the maximum from the modern deployment automation facilities [7].

So by using only one service for one task without any shared libraries and dependencies a decent separation between business logic implementations can be achieved. This opens the road to horizontal scalability on purpose (e.g. high load on the Christmas period).

The idea of major microservices attributes can be compared with UNIX ideas [24]:

- one program – one task
- universal interface for all programs – exposed APIs
- programs communicate with one another – synchronous and asynchronous

Microservices are small, independently scaled and managed applications. Each of them performs its own unique and well-defined role, runs in its own process and communicates via HTTP protocol messaging and exposes APIs [23]. Ideally, one developer should be enough to understand the idea of one special microservice and maintain it [12].

Designing of microservice system needs different tools and processes: the application itself, infrastructure with virtualization for hosting, monitoring and logging for all communication, organizational structures (teams), development processes (continuous integration), deployment (continuous delivery), testing [9]. As it is incredibly challenging to reproduce errors in big distributed applications - logging is so important [8].

Each microservice is similar internally to the monolithic application. So it has not only the code but is a full featured normal web application [17]:

- application code or runnable program
- libraries
- processes (e.g. cron)
- data stores, load balancers, or other services

If we have many microservices in our fleet, comes up the question - what is the best way to package and deliver them from developers to testers and from testers to the production environment. Immutable images and containers resolve this problem [5]. Containers run applications that are packaged in images, virtual machines run containers [10]. These containers are executable artifacts that

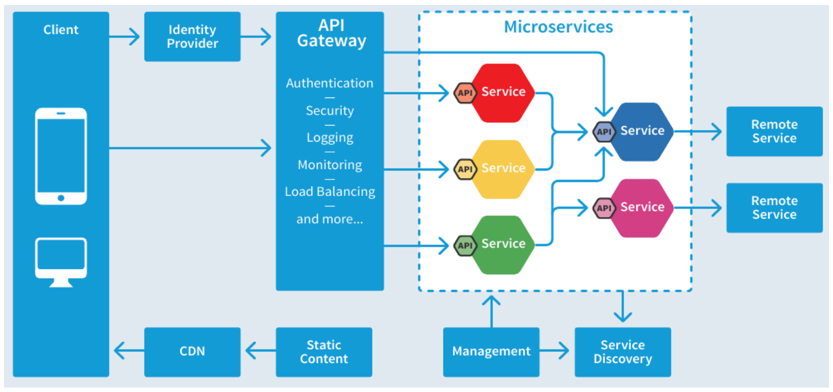

Figure 4: Web application with microservice architecture [23]

allow managing deployment by simply adding or removing a container from the current deployment [17]. But together with a scheduler containers provide an elegant and flexible approach that meets our two deployment goals: speed and automation.

Containers are extremely fast in start up because of the shared kernel with the host operating system. This can be a decent security issue. If one of the containers is compromised so are all the others. Virtual machines provide much better isolation, but remain resource-heavy, because of independent kernel running in each machine [10].

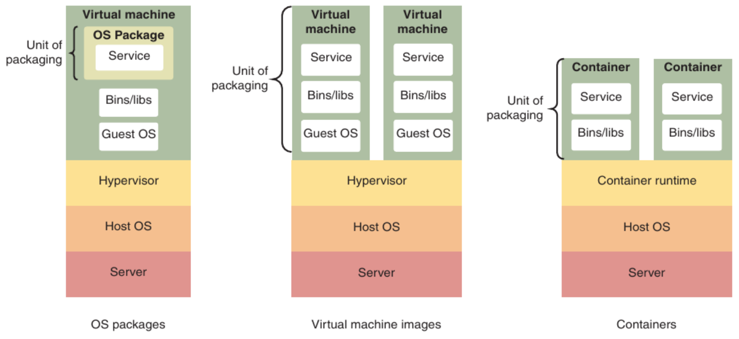

Figure 5: Comparison of packaging solutions [17]

Microservices architecture shows that containers dominate today on the market because allowing to package the application and its dependencies make it easier to run a polyglot stack. On the other side Kubernetes orchestration platform solves the following problems: managing, scaling and automating the deployment of the microservices across the cluster of worker nodes [20].

With the microservices benefits mentioned above, they bring high complexity to the deployment and maintenance of the running web application [1], [21]:

- lack of secure communication
- partitioned database architecture [3]
- unreliable communication [24]
- resiliency issues – cascade failures in distributed systems
- complicated transition from monolith to microservice [21]
- service discovery
- testing the complete system [3]
- faulty on the integration level [6]

To not lose the control and the overview of your microservice application is where service meshes come in play.

3.2 Service mesh

Modern web applications have very strict requirements such as availability (zero downtime) or fast response to requests [15]. Having a big number of microservices in your deployment makes the maintenance challenging. Operators should manage applications in large hybrid and multi-cloud deployments. With the demand to get more control and observability inside the network of running microservices the concept of service meshes appeared. Some of the current solutions to this concept were already discussed in related work.

A service mesh provides an opportunity to get full control over your microservices network uniformly and decoupled from the application code [20]. It focuses on networking between microservices (east-west traffic) rather than the business logic of the web application. A service mesh provides out of the box plenty of features now implemented in different separately managed ways: libraries for logging, API gateways for routing, certificates rotation for secure communication.

Service mesh can provide following functionalities depending on the implementation: service discovery, load balancing, resiliency and failure recovery, security (end-to-end encryption, authorization), observability (Layer 7 metrics, tracing, alerting, logging), routing control (A/B testing, canary deployments), API (programmable interface, Kubernetes CRD).

The most promising architecture of service meshes is based on sidecar proxy injections that works on top of Kubernetes cluster. Proxies handle all incoming and outgoing microservice traffic. Focus on the traffic between the services is what differentiates service mesh proxies form API gateways or ingress proxies, which center on requests from the outside network into the cluster [20].

3.3 Istio

Istio is described as a tool to connect, secure, control and observe services. The project is open source and was started by Google, IBM and Lyft [14]. It is a network of services that make the application. This service mesh is designed to add application-level observability, routing and resiliency to service-to-service traffic with almost no changes to the application code. Tracing, monitoring and logging give operators a full overview of the deployed microservice application.

Istio provides the following relevant for microservice architecture attributes: trusted communication, encryption of all internal traffic with mutual TLS enabled, tracing of requests (Jaeger), metrics (Prometheus), alerting and graphics (Grafana), visualization of the mesh topology (Kiali), advanced traffic management with routing and load balancing, communication and network resiliency, configuration API, policies to enable rate limiting, denials and white/black listing.

The only situation when some code changes will be needed is when tracing of calls between services should be enabled. This type of feature will need to add some custom headers propagation from service to service.

The traffic inside the web application between microservices is called "east-west". The opposite type of traffic is "north-south" and is referred to ingress and egress services.

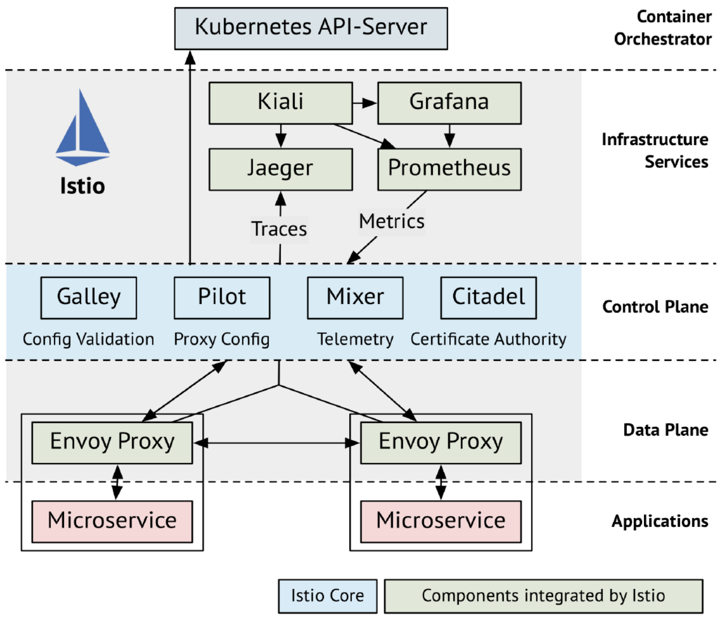

Figure 6: Istio architecture

Istio service mesh has two logical zones: a data plane and a control plane.

Data plane traffic is all the internal requests between the services of the deployed application. Envoy proxies are injected in each pod of the microservice as sidecars. Communication with service is possible only through these proxies. Traffic routing in Istio is managed purely on the data plane level. Mirroring of the traffic can be enabled for microservices. It is not efficient, but is beneficial for some particular situations, e.g. the service is read-only so enabling mirroring will not consume many computing resources.

Control plane traffic is used to configure and manage Istio components to reach a stable deployment of the mesh. It also manages the sidecars to allow routing and configures policies and collection of telemetry.

Control plane is also responsible for:

- automatic load balancing
- retries, circuit breakers, fault injection
- policies with access controls, rate limits and quotas
- telemetry for all traffic within and incoming and outgoing from a cluster requests
- authentication and authorization

Pilot talks to Kubernetes via an adapter to enable service discovery (can also work on top of Consul). It takes configured in Istio traffic rules and sends them to proxies in the data plane. The pilot works dynamically without restart needed and is also responsible for resiliency in the mesh. Citadel provides end-to-end encryption and user authentication. Mutual TLS configurations are done with its help. Galley holds, validates and distributes configurations. Mixer represents a layer between the Istio components, accompanying services and the infrastructure backends used for access control with policies and telemetry. All sidecars proxies ask mixer, if the request is allowed. Sidecar proxy (based on Envoy) adds behavior to application microservice without modifying its code. A combination of a proxy and a microservice is seen inside Istio as one logical unit. These proxies allow such Istio features as circuit breakers, health checks, fault injections, rich metrics, load balancing and others. Kiali helps to visualize the application to get an overview of what is deployed and who talks with whom. The primary purpose of Grafana is to give a graphical view about metric data, to create custom dashboards and trigger alerts [17]. It works with Prometheus as a backend. With Grafana fault injections and general behavior of the traffic load can be seen. Kiali and Grafana were widely used in the evaluation part of this thesis.

To configure traffic behavior inside service mesh Kubernetes custom resource definitions (CRD) are used. Istio provides the following resources for traffic management.

- Virtual services – how to route the traffic inside service mesh to a given destination. They are utilized to allow canary deployments. Routing is done based on various matching criteria (routing rule): weights, headers, URLs. Retries, timeouts and fault injection can be configured here.

- Destination rules – applied after routing is done. They consist of named subsets, traffic policies: circuit breaker and load balancing configurations.
- Gateways – used to allow incoming and outgoing traffic from the mesh. Each gateway is an Envoy proxy deployed on the edge of the mesh.
  - Ingress – used to expose service outside the cluster.
  - Egress – used to allow access to external calls outside the cluster. By default, all external traffic is restricted and needs to be enabled in service entry.
- Service entry – inherited automatically from Pilot, which takes service names and ports from Kubernetes service discovery. They also are used to add external services to the Istio registry (for egress destinations).

Gateway and service entries manage the incoming and outgoing traffic. Virtual service and destination rule handle the east-west traffic (inside service mesh).

Istio provides excellent features to manage your microservice application out of the box as a all-in-one solution and shows the need of service meshes in modern deployments. One of the goals of Istio service mesh is to put resiliency into the infrastructure.

3.4 Resiliency

In distributed microservices architecture one service can not await that all other services function without errors or that there are no network failures at all. Taking into consideration these aspects resiliency can be defined as the ability of a distributed system continues to respond to the client though there are network and service faults. Microservice must not block a request because then other microservices might also be blocked, the error propagates and cascade failure happens. Equally, simple network delays can cause such problems. Therefore it is necessary to be sure that one failed service does not bring down the entire system.

A resilient microservice application is one that can recover from failures at every level of the system: the hardware, the communication, the application and the microservice layer. There are several types of resiliency testing that can evaluate the fault tolerance of a deployment [8]. These are load and chaos testing.

Istio provides the following resiliency features: health checks, load balancing, delay injection, fault injection, timeouts, retries, rate limits, circuit breaker.

Services can use rate limits to protect themselves from spikes in load beyond their capacity to service [17].

There are two types of health checks: liveness and readiness probes [13]. They are crucial for system resiliency because the traffic should be forwarded

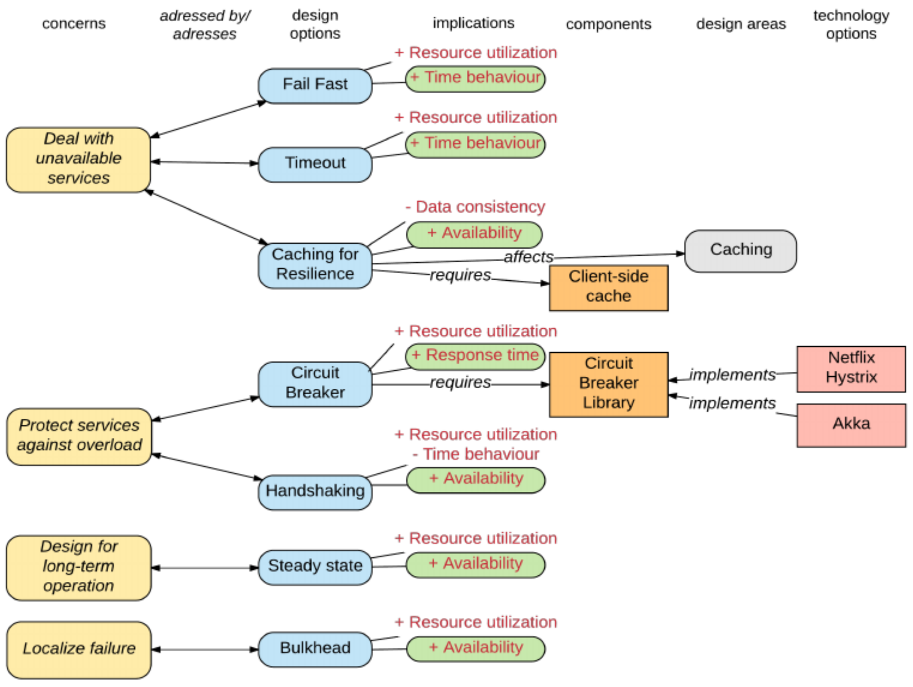

Figure 7: Possible failures and solutions for distributed applications [9]

only to healthy pods. Liveness probes help to determine if the application started and runs correctly. Readiness probes check if the application is ready to receive traffic for example after all configurations finished successfully [17]. Probes can be HTTP GET requests, "exec" scripts executed inside a container or TCP socket checks.

Though these are the mechanisms that belong to Kubernetes, they are still worth mentioning because Istio proxies allow these health checks to work seamlessly. One difference is that HTTP health checks work only with enabled mutual TLS. So some configurations on the side of Istio system namespace need to be done. "Exec" and TCP health checks work straight forward without any changes in Istio.

Load balancing in Istio provides some more sophisticated algorithms then native Kubernetes solution (round robin). They can be configured in destination rules:

- round robin - used by default
- random - random pods are taken for requests from load balancing pool
- least requests - least overloaded pod get new requests

Timeout helps to deliver fast responses to the client without waiting for a response from slow service. For better user experience, it is a good practice to fail fast than wait long for a response. The default value for a timeout in Istio is 15 seconds. It can be altered for each microservice individually in a virtual service configuration file. How to choose a proper timeout for calls depends on application and microservice. A small one can not be enough to process the request. A big one can slow the overall functionality of the system. Just waiting for slow responses needs much infrastructure resources (CPU, RAM). That is why timeouts are very important, and it is remarkably easy to configure them for service with Istio. The main challenge here will be to define the proper length of timeout. So infrastructure engineer needs to understand how the microservices application work or need to communicate with developers direct.

Retries policy repeats the failed request to get the response faster than return an error to the client and initialize a completely new request. By default, no retries are configured in Istio deployment, but it can be done in virtual services for each microservice. Typically developers take care of retries in application code, but Istio has built-in retry policies to configure and to make calls more resilient. Of course, with repeated retries, the load on the service will be more significant. This should be taken into consideration and could also be protected with a circuit breaker.

A circuit breaker pattern is used to protect the application from the failing microservice. It can be configured in Istio in destination rules for each microservice. There are two types of this pattern in Istio.

The first one works at the connection pool level and protects the microservice from overloading. It stops sending traffic to service if requests reach some limit defined in destination rule for this microservice.

The second type is outlier detection. If there are many replicas of microservice one of them can start returning errors (e.g. 50x). In this case, Istio will eject the problem pod from the load balancing pool for some time.

3.5 Demonstration

The primary result of this thesis will be a working demo to show the resiliency possibilities of Istio service mesh. The focus is made on the all-in-one solution. Project written in the cloud computing course is used as a microservice application. It is deployed in the single-node Kubernetes cluster that runs in Minikube with Virtualbox provider. Git repository contains all necessary scripts to install and start using Kubernetes with Istio in the development environment [18].

With the help of this demo, you can explore the basics of distributed applications and microservices, the concepts of modern application packaging, deployment, orchestration and monitoring. Docker files and Kubernetes manifests contain best practices from production ready deployments.

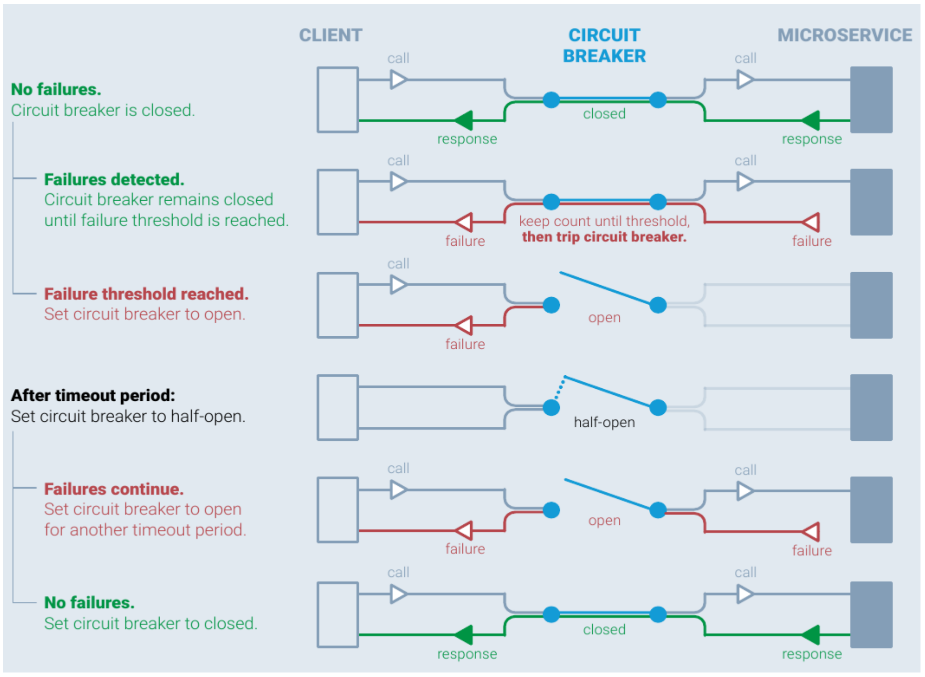
CLIENT
CIRCUIT BREAKER
MICROSERVICE
No failures.
Circuit breaker is closed.
call
closed
response
Failures detected.
Circuit breaker remains closed until failure threshold is reached.
failure
keep count until threshold, then trip circuit breaker.
Failure threshold reached.
Set circuit breaker to open.
open
After timeout period:
Set circuit breaker to half-open.
half-open
Failures continue.
Set circuit breaker to open for another timeout period.
No failures.
Set circuit breaker to closed.

Figure 8: Circuit breaker pattern [23]

4 Implementation

4.1 The Twelve Factors Application

5 Tables and Images

One of the great advantages of LaTeX is that all it needs to know is the structure of a document, and then it will take care of the layout and presentation itself. So, here we shall begin looking at how exactly you tell LaTeX what it needs to know about your document.

5.1 Images

Figure 9: Image Example

5.1.1 A Subsubsection

As one last example, this is how you can insert a sub-sub-section! Have fun writing your thesis with LaTeX!

References

[1] Alex Williams, B. B. *Applications and microservices with docker and containers*. The New Stack, 2016.

[2] Balalaie, A., Heydarnoori, A., and Jamshidi, P. Migrating to cloud-native architectures using microservices: An experience report. In *Advances in Service-Oriented and Cloud Computing* (Cham, 2016), A. Celesti and P. Leitner, Eds., Springer International Publishing, pp. 201–215.

[3] Chris Richardson, F. S. *Microservices From Design to Deployment*. NGINX Inc., 2016.

[4] https://glasnostic.com/blog/comparing-service-meshes-linkerd-vs-istio (accessed 07.03.2020).

[5] https://www.docker.com/ (accessed 07.03.2020).

[6] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. *Microservices: Yesterday, Today, and Tomorrow*. Springer International Publishing, Cham, 2017, pp. 195–216.

[7] https://www.martinfowler.com/articles/microservices.html (accessed 07.03.2020).

[8] Fowler, S. J. *Production-Ready Microservices: Building Standardized Systems across an Engineering Organization*. O'Reilly Media, 2017.

[9] Haselböck, S., and Weinreich, R. Decision guidance models for microservice monitoring. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* (April 2017), pp. 54–61.

[10] II, T. H. *Advanced Microservices: a Hand-on Approach to Microservices Infrastructure and Tooling*. Apress, 2017.

[11] https://istio.io/ (accessed 07.03.2020).

[12] Kratzke, N., and Quint, P.-C. Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. *Journal of Systems and Software 126* (2017), 1 – 16.

[13] https://kubernetes.io/ (accessed 07.03.2020).

[14] Lee Calcote, Z. B. *Istio: up and Running*. O'Reilly Media, 2019.

[15] Michael Hofmann, Erin Schnabel, K. S. *Microservices Best Practices for Java*. IBM Corp., 2016.

[16] https://github.com/davidetaibi/Microservices_Project_List (accessed 07.03.2020).

[17] Morgan, B., and Pereira, P. A. *Microservices in Action*. Manning, 2019. Microservices in Action.

[18] https://github.com/van15h/resilient_istio (accessed 07.03.2020).

[19] https://aspenmesh.io/service-mesh-architectures/ (accessed 07.03.2020).

[20] https://servicemesh.io/ (accessed 07.03.2020).

[21] Shadija, D., Rezai, M., and Hill, G. Towards an understanding of microservices. In *2017 23rd International Conference on Automation & Computing (ICAC)* (United States, 10 2017), Institute of Electrical and Electronics Engineers Inc.

[22] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., and Gil, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)* (Sep. 2015), pp. 583–590.

[23] Williams, A. *Guide to Cloud Native Microservices*. The New Stack, 2018.

[24] Wolff, E. *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 2016.