Programmable Proxy for Microservice communication in Multi-cloud environments

A THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

M.Tech

BY Shambhavi Pathak MT22067

Computer Science and Engineering

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY DELHI
NEW DELHI– 110020

May 21, 2024

Thesis Certificate

This is to certify that the thesis titled **Programmable Proxy for Microservice communication in Multi-cloud Environments**, submitted by **Shambhavi Pathak**, to the Indraprastha Institute of Information Technology, Delhi, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by her under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Rinku Shah

Rinku Shah
Thesis Supervisor
Assistant Professor
Dept. of Computer Science
IIIT Delhi, 110020

Place: New Delhi

May 21, 2024

Acknowledgement

First of all, I want to express my heartfelt gratitude to my supervisor Dr. Rinku Shah for providing every necessary supervisory and emotional support to pursue my M.Tech thesis research in a focused manner.

My family has supported me with unconditional love and encouragement, without which it wouldn't have been possible to accomplish this. I am also thankful to my peers at the Networks Research Lab for creating an atmosphere of research and collaboration. Special thanks to Alvin, Arjun, Sumit, Yukti, and Siddharth for coordinating with me throughout the thesis work.

Abstract

As enterprises migrate from single-cloud to multi-cloud architecture, they encounter challenges due to geographical dispersion, varying WAN characteristics, and diverse cloud policies. These challenges demand a re-evaluation of communication strategies by the developers to ensure reliability, security, and compliance. In response, our work presents a solution, Programmable Proxy, designed to address the dynamic nature of multi-cloud environments. My thesis focuses on the switch between HTTP and MQTT communication protocols; the proxy facilitates real-time adaptation based on service location, packet characteristics, and request types. Through this approach, programmable proxy optimizes communication mechanisms in alignment with evolving deployment requirements, enabling enhanced performance across diverse cloud infrastructures. This study contributes to the advancement of flexible, adaptive microservices architectures in the context of multi-cloud environments.

Contents

1 Introduction **1**

2 Background **2**
2.1 Microservice Architecture 2
2.2 Container Orchestration 2
2.2.1 Kubernetes 3
2.3 Service Meshes 3
2.3.1 Istio Service Mesh 3

3 Related work **4**
3.1 Skupper 4
3.2 Istio multi-cluster 4
3.3 Cilium cluster mesh 5

4 Design **6**

5 Implementation **9**
5.1 Programmable Proxy 10
5.2 Router 10

6 Evaluation **11**
6.1 Experimental results 12

7 Future work **14**

Chapter 1

Introduction

Cloud computing has changed the IT sector with respect to how to store, access, secure, and run our applications. However, accentuated growth demonstrates that low cost, fast elasticity, on-demand payment, and high availability proposals have become increasingly recurring choices for organizations worldwide, leading to multi-cloud adoption. A survey shows that about 90% of large enterprises are adopting multi-cloud infrastructure [1].

Multi-cloud environments are those in which multiple providers work cooperatively and synchronized. The growth of multi-cloud complexity resulted in the emergence of the concept of the sky computing paradigm [2], which discusses vendor lock-in where the customer becomes heavily dependent on a particular vendor's products, services, or technology, for example, Google provides TPUs which is good for ML training which is not provided by other vendors, Customer lock-in where vendor or service provider makes it difficult for customers to switch to alternatives, the difficulty of moving jobs to another cloud due to the expense of transferring the data, latency, and compliance consideration called data gravity [3] and regulatory compliance regarding where data can be stored, and computational jobs run as all cloud provider does not have datacenters in all countries. No business wants any critical part of its infrastructure tied to a single provider because such lock-in reduces its negotiating leverage and also makes the business vulnerable to large-scale outages at the provider. With the increase in the use of the multi-cloud, various multi-cloud management tools emerged like GCP's Anthos [4] and Azure's Arc [5].

The increase in multi-cloud adoptions by enterprises also occurred after the standardization of container orchestration platforms like Kubernetes, which helped enterprises deploy the application on the cloud in the form of microservices and enabled them to move from a single cluster to multiple clusters and cloud. This led to moving further and enabled a new ecosystem of solutions for multi-cluster deployment in compute[6][7][8], network [9][10], and storage [11]. All this work looks towards re-using the single cluster pattern towards multicluster. In the context of multi-cloud, communication between the services happens over WAN. WAN is unreliable and has non-deterministic characteristics [12]. There is a need to switch between communication protocol mechanisms based on network characteristics for better reliability yet meeting the application requirements. We propose Programmable Proxy, which performs so without any intervention from the developer or the deployer.

Chapter 2

Background

This section discusses commonly used terms and concepts in cloud computing, leading to the problem statement's motivation and the current work's objective.

2.1 Microservice Architecture

A software application's architecture is an ever-evolving aspect of the software industry domain. Shaping not only how applications are built and maintained but also how they adapt to changing technological landscapes and business needs. Monolithic architecture has been the default in the early days of computing.[2] In this architecture, the application is developed as a single indivisible unit, all processes are tightly coupled in a single code base, and the communication between them is via function calls. Microservice architecture has recently been widely adopted in the software industry. It is an architectural style where a software application comprises several different small, independent, and deployable services, each of which ideally performs one task. These services communicate with each other over well-defined APIs [1]. The shift in the architectural style was due to the following advantages of the latter over the former,

1. *Faster development*: In a microservice architecture, each service is developed independently, which implies that each can have its development team, speeding up the entire development lifecycle

2. *Easy to deploy*: In case of any modifications, only the modified service needs to be deployed, unlike in a monolithic architectural style

3. *Flexible scaling*: Allows to scale individual components, which means only the services required to handle increased traffic can be scaled.

4. *High availability*: Provides more resilience to the application; for example, in case of the failure of a service, other instances of the service could be spawned beforehand, avoiding any interruption to service usage.

The above advantages can be attributed to the services' independent and loosely coupled nature in the microservice architecture.

2.2 Container Orchestration

Cloud applications are widely developed using microservice architecture, and the deployment of the services is facilitated by containers, alternatively known as cloud containers. Cloud Containers can be defined as a software package containing the service code, its dependent libraries, and other dependent modules required to run the service in the cloud. There can be tens and thousands of such containers in an

application, and managing them can be cumbersome. For container orchestration, multiple solutions are being used in the Industry. Kubernetes is one of the popular container orchestrators. In the context of this paper, orchestration means managing several services that run together as part of a bigger workflow. This section will discuss an orchestrator's features and architecture, as our solution to multi-cloud communication is built on top of these concepts. For reference, we have considered Kubernetes[13] in particular.

2.2.1 Kubernetes

Kubernetes automates the deployment and maintenance of containers in the cloud; below are the major features that Kubernetes provides:

- Automatic Provisioning and Deployment
- Automatic scaling of containers up or down and load balancing
- Intelligent allocation of resources between containers
- Regular health checks of the application
- Service discovery

In conclusion, it automates the entire software development lifecycle.

2.3 Service Meshes

It is an infrastructure layer that sits on top of an application. It controls how different components of an application communicate with each other. In other words, it routes every request from one service to another. Although communication logic can be coded in the service code, when communication becomes too complex or there are frequent changes in the communication method, it is better to abstract out the networking logic to a separate component, which is what service mesh does. Istio[14] is one of the most popular service meshes used in the industry; hence, to discuss a service mesh's features and architecture, we use Istio as the reference.

2.3.1 Istio Service Mesh

Istio extends Kubernetes[13] to establish an application-aware network using a powerful envoy proxy. It provides the following features:

- Secure end-to-end communication within a cluster
- Network telemetry within a cluster
- Fine-grained control of traffic with fault injections, retries, failovers

Chapter 3

Related work

The solutions mentioned below provide layer 7 networking in the context of multi-cloud, but the only gap in them is that they will require developer or deployer intervention to specify or change the application layer communication protocol. None of them provides the flexibility of switching between the communication protocols on the fly.

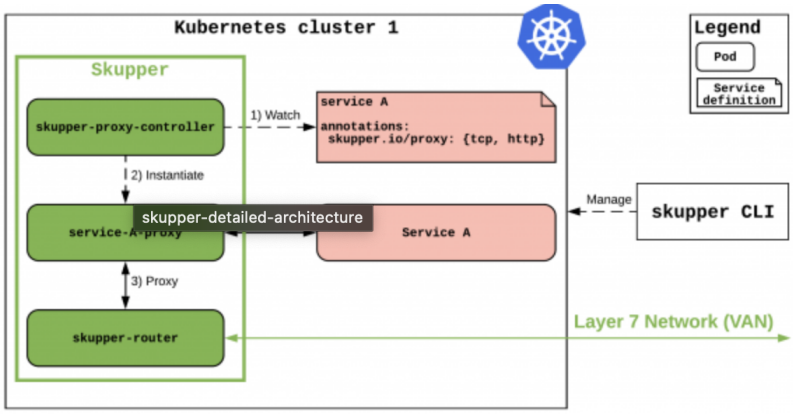

Figure 3.1: Skupper design

3.1 Skupper

It achieves multi-cloud networking by creating a VAN (Virtual Application Network)[9]. Key Components of Skupper

Proxy: The proxy component in Skupper handles the traffic routing between services in different clusters.

Router: It is responsible for establishing and maintaining the connections between clusters, ensuring that traffic is securely and efficiently routed. It is based on the Apache Qpid Dispatch Router. It creates a network of routers that dynamically route messages based on their destination, using a protocol called AMQP (Advanced Message Queuing Protocol). This ensures low-latency and high-throughput communication across clusters.

3.2 Istio multi-cluster

It enables cross-cluster communication using VPNs[15]. It abstracts the complexities of deploying and managing microservices, offering capabilities like global

traffic management, global security, and global observability. Key components include:

Envoy is a high-performance proxy used to handle and manage traffic between microservices. It is deployed as a sidecar container alongside each kubernetes pod.

The Control Plane is the core component of Istio and is responsible for managing configurations, policies, and certificates. Each cluster has its own Istio control plane, and the clusters can be connected to form a single mesh.

3.3 Cilium cluster mesh

Cilium provides connectivity to geographically distributed or within the same region clusters using VPNs or directly connected links. Cilium Cluster Mesh relies on a shared etcd or uses DNS-based service discovery to manage clusters. It leverages eBPF (extended Berkeley Packet Filter) to dynamically program the Linux kernel, offering advanced networking, security, and observability features. Key Components of Cilium:

Cilium Agent: The Cilium agent runs on each node in the Kubernetes cluster, managing networking andservice discovery load balancing. It uses eBPF to insert hooks directly into the Linux kernel, enabling high-performance networking and security enforcement. The Cilium agent is responsible for translating Kubernetes network policies into eBPF programs that run in the kernel.

Hubble: Hubble leverages eBPF for real-time visibility and monitoring, offering features such as flow tracking, metrics, and distributed tracing. It provides observability across all the clusters.

Chapter 4

Design

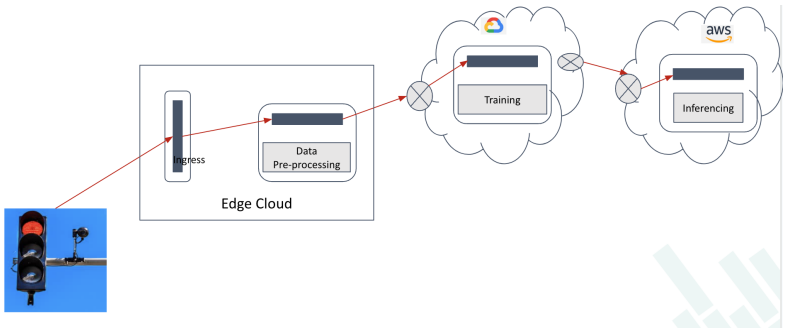

Figure 4.1: Road Traffic monitoring application in Multi-cloud environment

As mentioned earlier, transitioning to a multi-cloud environment presents several hurdles for organizations to overcome. These encompass communication, policy management, encryption, and data compression. In this report, we specifically tackle the issue of communication. To understand better, we will take an example application.

A Road Traffic Monitoring application in which cameras at Traffic lights capture the image and send it to the edge cloud for pre-processing; the pre-processed data is forwarded to GCP for data training and to AWS for inferencing.

Knowing that the inter-service communication in a multi-cloud environment will involve the traffic going through the WAN, which is unreliable in nature[16], and an application's services being a combination of high and low response times[17], there will be a need for diverse communication mechanism which provides reliability and suits the application requirements without any intervention from the developer or deployer.

The switch between communication protocols will require a middlebox in between[18]. The term proxy and middlebox has been used inter-changeably. The proxy placement choices varied from per pod to per cluster; below, we discussed the reasons for not finalizing per cluster or per pod proxy.

Why not per pod proxy?
If the sidecar approach is used to route the external requests, it will add an overhead on resource consumption and increase latency[19]

Why not per cluster proxy?
There are an average of 500 services per cluster [20], sharing the certificates of each

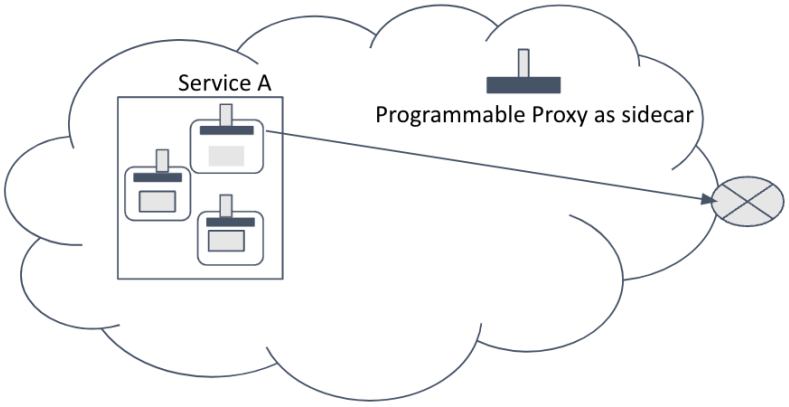

Figure 4.2: Proxy placement: Per pod proxy

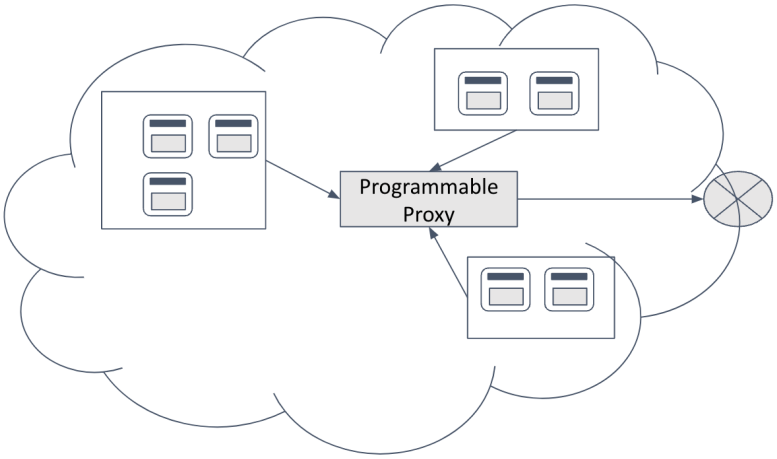

Figure 4.3: Proxy placement: Per cluster proxy

service will lead to putting too much trust in the infrastructure provider

The final placement of the proxy is considered as per service. This design is followed by skupper [9], an L7 multi-cloud solution.

The options of asynchronous communication addressability were as below

Per cluster topic:
By per cluster topic it means that each cluster is subscribed to a topic of it's own and in case a service of one cluster has to communicate to any service of a different cluster, the sender service will publish the request to the receiver cluster's subscribed topic. Considering the above, if the services of cluster c1 wants to communicate to the services of cluster c2, the requests may get queued longer than the timeouts and there can be an increase in the response times.

Per service topic:
When an application is deployed globally, there are many high traffic services which

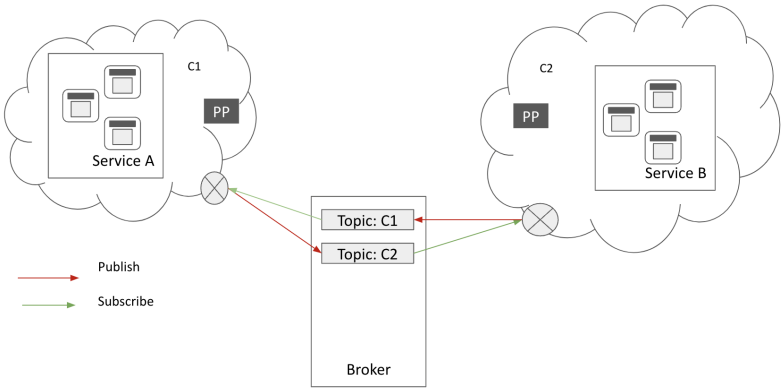

Figure 4.4: Asynchronous service addressability placement: Per cluster topic

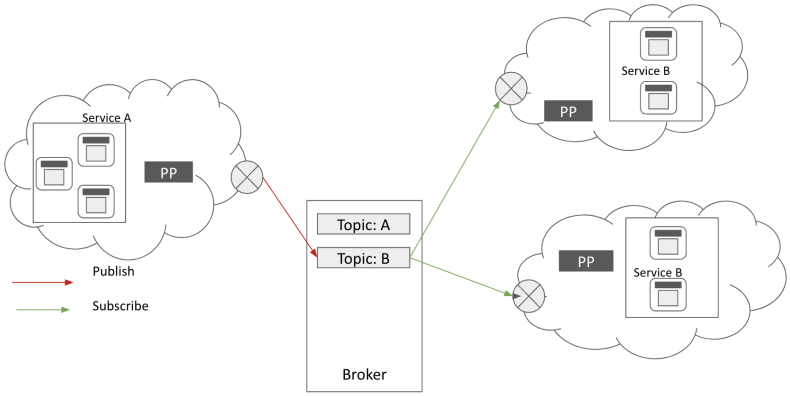

Figure 4.5: Asynchronous service addressability: Per service topic

have its instances deployed more than once[21]. If every instance of a service is subscribed to the same topic, the task of load balancing needs to be taken care by broker, which will be ineffective as the load balancing will follow the same rules as the subscription model for the particular topic [22].

Chapter 5

Implementation

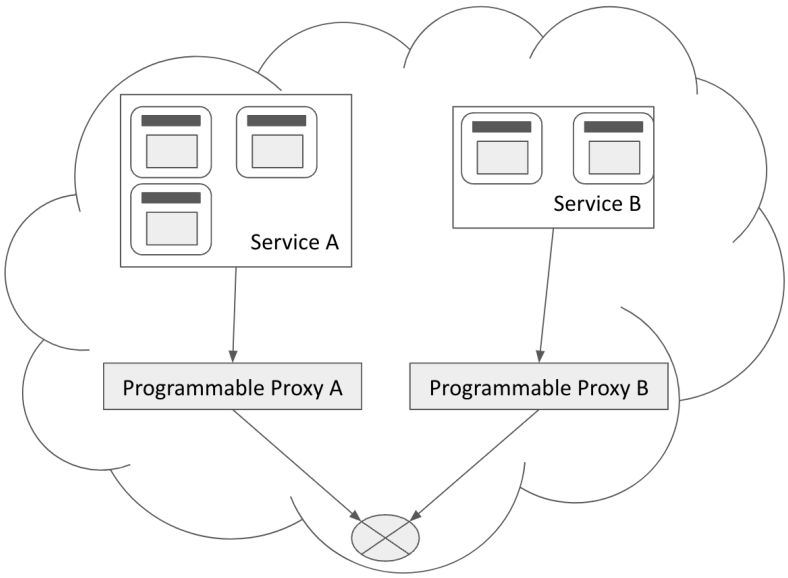

Figure 5.1: Proxy placement: Per service proxy

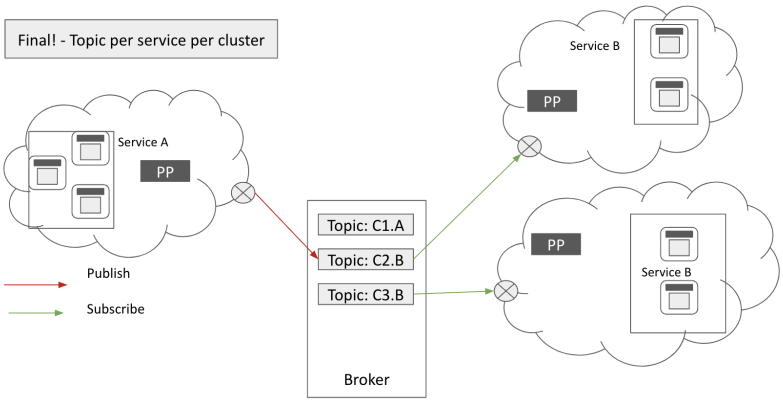

Figure 5.2: As placement: Asynchronous service addressability: Per cluster per service topic

The initial implementation of the Proxy is done in Go language [23]. Go outperforms other languages in terms of concurrency and network I/O by utilizing asynchronous system calls. Go's approach allows it to achieve a significant speedup in certain scenarios despite being slower in single-threaded performance [24].

5.1 Programmable Proxy

It acts as an HTTP server to the corresponding service. It is connected to the router or the gateway over three persistent TCP connections. One of them is used for sending TCP payload from the service to the gateway as an MQTT request, and the other one as an HTTP request. The third connection is for sending tcp payload from the gateway to the service.
Proxy adds two headers to the TCP payload, like **source pod IP** and **source service URL**, which is used on the receiver side proxy to form the response.

5.2 Router

It is connected to all the service's proxies of a cluster over persistent TCP connections. It is subscribed to the topics of the services inside its cluster. The router or the Gateway has the global knowledge of all the clusters as part of one application. It maintains a key-value storage for the asynchronous service addressability, with the key as the service URL and the value as the topic name.

On receiving the requests from its cluster's proxy, the router fetches the corresponding topic from its key-value storage in case of MQTT communication between the clouds. There are two scenarios for sending the response to the sender.

- Response from the broker: On receiving publish acknowledgment from the broker, the Router forms an HTTP response and sends it back to the proxy.
- Response from the receiver: On receiving a subscribe message for a service, forwards it as is to the corresponding proxy, and the proxy parses it to identify whether it is a response to existing acknowledgment from the broker; the Router forms an HTTP response and sends it back to the proxy.

On receiving the requests from its cluster's proxy, the router forwards to the respective service's cluster in case of HTTP communication between the clouds.

On receiving the subscribed message from the broker, it forwards the request to the corresponding service's proxy without parsing it.

Chapter 6

Evaluation

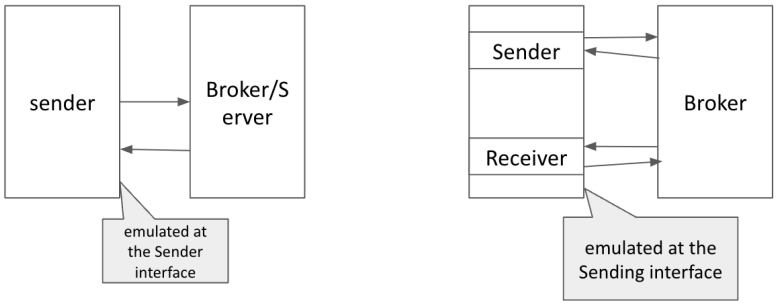

Figure 6.1: Evaluation setup for response from broker and response from receiver

Testbed: We built a testbed with two nodes. The client node has an Intel(R) Xeon(R) W-3335 CPU @ 3.40GHz 16-Core Processor, 32GB RAM. It runs Ubuntu 18.04.6 and uses Linux 5.4.0-150-generic. The server node has AMD Ryzen 9 5950X 16-core processor, 32GB RAM. It runs Ubuntu 18.04.6 and uses Linux 4.15.0-212-generic. The two nodes are connected using a 40 Gbps Netronome cable.

The test consisted of throughput testing. They were conducted for WAN(inter-cloud) setup. Each test consisted of 5 trials, and the numbers were reported as average of all the tests. In emulation of a WAN network, the bandwidth fluctuated between 15Mbps, 13Mbps, and 17Mbps[25] at the client node interface. For metrics collection and measurement, Grafana[26] and Prometheus[27] were configured.

Procedure: *HTTP:* Client Load Generator - Grafana K6[28]. Server: NGINX(v1.14)[29] (pinned to 1 core) Each client sends a request for a certain bytes payload. The server sends a 1-byte response to the client. A test was performed till network bandwidth saturation.
MQTT disguised as HTTP: Client Load Generator - EMQTT-Bench tool[30]. Server: EMQX Broker(v 1.4.15)[31] (pinned to 1 core) Each client sends a message of the corresponding HTTP packet size as the payload size with a Quality of Service set to 1. The server sends a 1-byte response to the client. The test was performed till network bandwidth saturation.

Objective: As part of the evaluations, we answer the below questions:

- *Performance of synchronous and asynchronous communication for emulated WAN environment for scenarios:*
  - *Response from Receiver*
  - *Response from Broker*
- *Initial prototype evaluation*

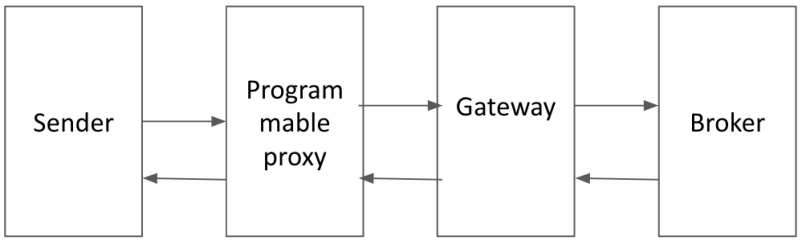

Figure 6.2: Evaluation setup for initial prototype of programmable proxy

6.1 Experimental results

Figures 6.4 and 6.3 show the throughput and latency for the following scenarios (1)Synchronous protocol (HTTP) requests with payload size 20B, 64B, 1000B (2) Asyncrhonous protocol (MQTT) messages with payload size equal to the HTTP packet size, as obtained from (1), response from broker (3) Asyncrhonous protocol (NATS messaging) with payload size equal to the HTTP packet size, as obtained from (1), response from receiver.

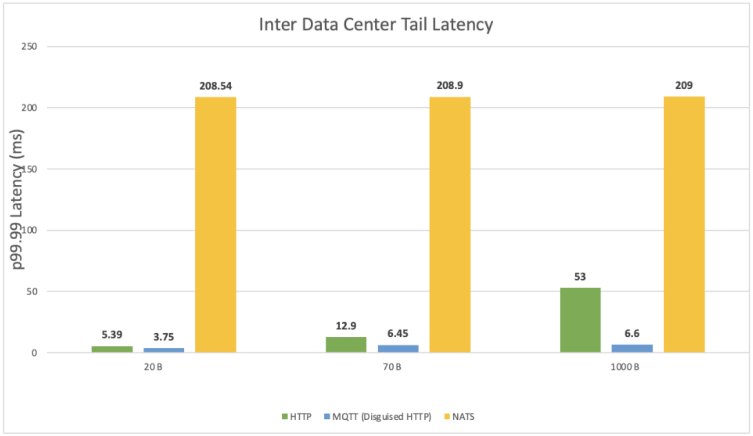

Figure 6.3: Tail Latency

Observation:

- MQTT has higher throughput as compared to other scenarios.
- Synchronous protocol has lower throughput as compared to asynchronous protocols
- Tail latency of Synchronous protocol is the lowest.

Inference:

- HTTP POST, PUT requests where the sender does not expect a response from the receiver can be substituted by the asynchronous mechanism with the response from the broker.

Initial prototype testing: For the current version of the programmable proxy the throughput achieved is 412 reqs/s where each request takes approximately 2ms.

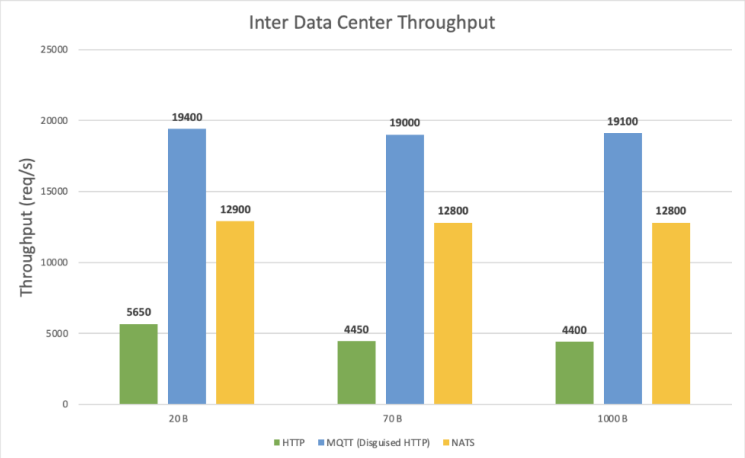

Figure 6.4: Throughput Results

Chapter 7

Future work

- Improve Programmable Proxy performance: Given the current, the connection between the proxy and the gateway is via a single persistent connection. To improve the performance, we can perform connection pooling.
- Test Programmable Proxy performance in a real-world scenario: The popular Deathstar Benchmark[32] provides several microservices applications. Currently, we have a setup of the Hotel Reservation application within a cluster. We aim to use it as a multi-cluster application and use it to test Programmable Proxy
- Find the metrics for the unreliable network (WAN): Given that the protocol selection will also be based on the network characteristics, find the correct parameters of the WAN environment to write the switching algorithm.
- Develop an algorithm to switch between the protocols: After having found out the network parameters, design an algorithm to switch between communication protocols based on it.
- Integrate more protocols in Programmable Proxy: Currently, the switching is between HTTP/1.1 and MQTTv5, extend it to support more L7 communication protocols

Bibliography

[1] "Hashicorp 2023 state of cloud strategy survey." https://www.hashicorp.com/state-of-the-cloud, 2023.

[2] I. Stoica and S. Shenker, "From cloud computing to sky computing," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 26–32, 2021.

[3] Z. Yang, Z. Wu, M. Luo, W.-L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F. S. Luan, G. Mittal, S. Shenker, and I. Stoica, "SkyPilot: An intercloud broker for sky computing," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, (Boston, MA), pp. 437–455, USENIX Association, Apr. 2023.

[4] "Introducing anthos: An entirely new platform for managing applications in today's multi-cloud world." https://cloud.google.com/blog/topics/hybrid-cloud/new-platform-for-managing-applications-in-todays-multi-cloud-world, 2019.

[5] "Azure arc." https://azure.microsoft.com/en-in/products/azure-arc, 2019.

[6] "Automate infrastructure on any cloud with terraform.." https://www.terraform.io/.

[7] "msc-api." https://github.com/kubernetes-sigs/mcs-api.

[8] "Overview - kubestellar.." https://docs.kubestellar.io/release-0.3.

[9] "Skupper - multicloud communication.." https://skupper.io.

[10] "T. s. community. :: Submariner k8s project documentation website.." https://submariner.io.

[11] "Planetscale. planetscale introduces the first true multi-cloud, multiregion database-as-a-service for mission-critical applications.." GlobeNewswire News Room, 2020.

[12] W. Li, D. Guo, K. Li, H. Qi, and J. Zhang, "idaas: Inter-datacenter network as a service," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1515–1529, 2018.

[13] "kubernetes." https://kubernetes.io/.

[14] "Istio." https://istio.io.

[15] "Install multicluster.." https://istio.io/latest/docs/setup/install/multicluster.

[16] W. Li, D. Guo, K. Li, H. Qi, and J. Zhang, "idaas: Inter-datacenter network as a service," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1515–1529, 2018.

[17] H. Aragon, S. Braganza, E. Boza, J. Parrales, and C. Abad, "Workload characterization of a software-as-a-service web application implemented with a microservices architecture," in *Companion Proceedings of The 2019 World Wide Web Conference*, WWW '19, (New York, NY, USA), p. 746–750, Association for Computing Machinery, 2019.

[18] P. H. Fong and S. A. Holdsworth, "Method of receiving a message processable by a component on one of plurality of processing threads," Jan. 11 2011. US Patent 7,870,560.

[19] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, *et al.*, "Dissecting overheads of service mesh sidecars," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pp. 142–157, 2023.

[20] "How many and how big services should kubernetes scale to?." https://github.com/kubernetes/kubernetes/issues/48938, 2017.

[21] "number of instances of same service?." https://discuss.kubernetes.io/t/multiple-deployments-of-same-app/16901, 2017.

[22] "emqx shared subscription?." https://www.emqx.io/docs/en/latest/messaging/mqtt-shared-subscription.html, 2017.

[23] "Deathstar benchmark.." https://go.dev/.

[24] D. Lion, A. Chiu, M. Stumm, and D. Yuan, "Investigating managed language runtime performance: Why JavaScript and python are 8x and 29x slower than c++, yet java and go can be faster?," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, (Carlsbad, CA), pp. 835–852, USENIX Association, July 2022.

[25] S. Suryavansh, C. Bothra, M. Chiang, C. Peng, and S. Bagchi, "Tango of edge and cloud execution for reliability," in *Proceedings of the 4th Workshop on Middleware for Edge Clouds & Cloudlets*, MECC '19, (New York, NY, USA), p. 10–15, Association for Computing Machinery, 2019.

[26] "grafana?." https://grafana.com/, 2017.

[27] "grafana?." https://prometheus.io/, 2017.

[28] "grafana?." https://github.com/grafana/k6, 2017.

[29] "Install multicluster.." https://nginx.org/en/.

[30] "emqx-bench?." https://github.com/emqx/emqtt-bench/tree/master, 2017.

[31] "Install multicluster.." https://www.emqx.io/.

[32] "Deathstar benchmark.." https://github.com/delimitrou/DeathStarBench.