Latency-aware Optimization of the Existing Service Mesh in Edge Computing Environment

ZHEN SUN
Latency-aware Optimization of the Existing Service Mesh in Edge Computing Environment

ZHEN SUN
Abstract

Edge computing, as an approach to leveraging computation capabilities located in different places, is widely deployed in the industry nowadays. With the development of edge computing, many big companies move from the traditional monolithic software architecture to the microservice design. To provide better performance of the applications which contain numerous loosely coupled modules that are deployed among multiple clusters, service routing among multiple clusters needs to be effective. However, most existing solutions are dedicated to static service routing and load balancing strategy, and thus the performance of the application cannot be effectively optimized when network condition changes.

To address the problem mentioned above, we proposed a dynamic weighted round robin algorithm and implemented it on top of the cutting edge service mesh Istio. The solution is implemented as a Docker image called RoutingAgent, which is simple to deployed and managed. With the RoutingAgent running in the system, the weights of the target routing clusters will be dynamically changed based on the detected inter-cluster network latency. Consequently, the client-side request turnaround time will be decreased.

The solution is evaluated in an emulated environment. Compared to the Istio without RoutingAgent, the experiment results show that the client-side latency can be effectively minimized by the proposed solution in the multicluster environment with dynamic network conditions. In addition to minimizing response time, emulation results demonstrate that loads of each cluster are well balanced.

Keywords

Edge Computing; Service Mesh; Istio; Weighted Round Robin; Load Balancing; Client-side Latency Minimization
Sammanfattning

Edge computing, som ett tillvägagångssätt för att utnyttja beräkningsfunktio-
ner som finns på olika ställen, används i stor utsträckning i branschen nuförti-
den. Med utvecklingen av kantdatabasen flyttar många stora företag från den
traditionella monolitiska mjukvaruarkitekturan till mikroserviceteknik. För att
gå bättre prestanda för de applikationer som innehåller många löst kopplade
moduler som distribueras bland flera kluster, måste service routing bland fle-
ra kluster vara effektiva. De flesta befintliga lösningarna är dock dedikerade
till statisk service-routing och belastningsbalanseringsstrategi, vilket gör att
programmets prestanda inte effektivt kan optimeras när nätverksförhållande-
a ändras.

För att ta itu med problemet som nämns ovan föreslog vi en dynamisk
viktig round robin-algoritm och implementerade den ovanpå den avancerade
servicenätverket Istio. Lösningen implementeras som en Docker-bild som he-
ter RoutingAgent, som är enkel att distribuera och hantera. Med agenten som
körs i systemet ändras Vikten av målrutingsklustret dynamiskt baserat på den
upptäckta interklusternätets latens. Följaktligen kommer klientsidans begäran
om omställningstid att minskas.

Lösningen utvärderas i en emulerad miljö. Jämfört med Istio utan agent
visar experimentresultaten att klientens latentitet effektivt kan minimeras av
den föreslagna lösningen i multicluster-miljö med dynamiska nätverksförhål-
landen. Förutom att minimera responstid visar emuleringsresultat att belast-
ningar i varje cluster är välbalanserade.
I would like to express my sincere gratitude to my examiner, Professor Mihhail Matskin, who kept helping me with professional advice and guidance.

My considerate, thoughtful and encouraging parents, they were always eager to know what I was doing and how I was proceeding. I am extremely grateful to the personal attention and care they give to me.

I am grateful to my boyfriend, Thomas Lee, who has provided moral and emotional support in my life. I love you forever.

I also would like to express thanks to my friends, Jiatong Zhao, Blake Wang, Claire Yang, who have supported me along the way and provided me with advice regarding academics and life.

Thanks for all your encouragement and support!
List of Figures

2.1 A comparison between monolithic and microservice architecture       8
2.2 Linkerd architecture ........................................... 12
2.3 Consul architecture ............................................. 13
2.4 Istio ecosystem ..................................................... 15

4.1 Kubernetes architecture ........................................... 26
4.2 Istio architecture .................................................... 27
4.3 Istio multicluster architecture .................................... 28
4.4 Overall architecture ............................................... 30
4.5 HTTPing microservice workflow .................................. 31
4.6 RoutingAgent workflow ............................................. 32

5.1 Emulated environment ............................................. 35
5.2 Bookinfo application .............................................. 37
5.3 Bookinfo service graph ............................................ 37
5.4 Average turnaround time .......................................... 39
5.5 Percentile of turnaround time .................................... 40
5.6 Network latency variation among different clusters ............ 41
5.7 Average turnaround time performance comparison ............. 42
5.8 Percentile of turnaround time .................................... 43
5.9 Network latency variation among different clusters .......... 44
5.10 CPU usage percentage ............................................. 45
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Edge computing and cloud computing comparison</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Netflix suite components</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Service mesh comparison</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Definition of mathematical model variables</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Inter-cluster latency and corresponding command execution time</td>
<td>31</td>
</tr>
<tr>
<td>5.1</td>
<td>The latency setting among different clusters</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>The deployments of Bookinfo application</td>
<td>36</td>
</tr>
</tbody>
</table>
Contents

1 Introduction 1
  1.1 Background ........................................ 1
  1.2 Problem ........................................... 2
  1.3 Purpose ........................................... 2
  1.4 Goal ............................................... 3
  1.5 Methodology ....................................... 3
  1.6 Sustainability and Ethics ......................... 4
  1.7 Delimitations .................................... 4
  1.8 Outline .......................................... 4

2 Background 6
  2.1 The Evolution of Cloud Computing .................. 6
    2.1.1 Monolithic Architecture ....................... 6
    2.1.2 Containerization ............................... 7
    2.1.3 Microservice Architecture .................... 7
    2.1.4 Edge Computing ............................... 9
  2.2 Container Orchestration ............................ 9
    2.2.1 Kubernetes ................................... 9
    2.2.2 Kubernetes Federation ........................ 10
  2.3 State of the Arts in Service Mesh .................. 11
    2.3.1 Linkerd .................................... 11
    2.3.2 Hashicorp Consul ............................. 12
    2.3.3 Istio ....................................... 14
    2.3.4 Netflix Suite ................................ 14
    2.3.5 Comparison Summary ........................... 16
  2.4 State of the Arts in Load Balancing ............... 16

3 Design 18
  3.1 Mathematical Discussion ............................ 18
3.1.1 Definition of Different Variables ........................................... 18
3.1.2 Optimization Objectives Definition ......................................... 19

3.2 Algorithms .................................................................................. 20
3.2.1 Weighted Round Robin ............................................................... 20
3.2.2 Latency-aware Weighted Round Robin ......................................... 21

4 Implementation .............................................................................. 25
4.1 System Architecture ..................................................................... 25
4.1.1 Kubernetes Architecture ............................................................. 25
4.1.2 Istio Architecture ..................................................................... 26
4.1.3 Overall Software Architecture ..................................................... 28
4.2 Mechanism .................................................................................. 29
4.2.1 HTTPing Service ..................................................................... 29
4.2.2 RoutingAgent .......................................................................... 32

5 Evaluation ....................................................................................... 34
5.1 Experiment Design ....................................................................... 34
5.1.1 Topology Setup ....................................................................... 34
5.1.2 Network Latency Emulation ......................................................... 34
5.1.3 Simulation Task ....................................................................... 36
5.1.4 Performance Measurement .......................................................... 38
5.2 Influence of Local Weight ............................................................... 38
5.3 Turnaround Time Performance ....................................................... 40
5.3.1 Average Turnaround Time ........................................................... 40
5.3.2 Percentile of Turnaround Time .................................................... 42
5.4 Load Balancing Performance ........................................................... 43

6 Conclusion ....................................................................................... 46
6.1 Discussion .................................................................................. 46
6.2 Future Work .............................................................................. 48

Bibliography ....................................................................................... 50
Chapter 1

Introduction

In recent years, with the exponential growth of mobile communication technology, edge computing has gained great attention and importance. Considering the communication structure, the design of software architecture has changed from the traditional monolithic style to a distributed micro-service style. Even though the micro-services pattern brings numerous benefits such as high scalability, fault isolation, and simple deployment. Due to the locality and computing capability limitation, there are still several aspects can be improved and optimized for the existing edge computing world.

1.1 Background

Considering the complex topology of micro-service software architecture, service mesh technologies have been adopted to more and more microservice based systems (e.g. Kubernetes)\[1\]. A service mesh offers a set of common functionalities for microservice based applications, such as communication, service discovery, routing, load balancing, security, and tracing. The main benefits include easier and faster development of applications, as well as integrated observability and manageability. At the same time, the mobile edge cloud is identified as one key technology to support services requiring low latency or massive data offload, e.g. intelligent transport and VR/AR. The mobile edge cloud consists of multiple sites at the edge of the network, which usually are geographically distributed and heterogeneous in their hardware or software setup. Emerging applications running on edge clouds are expected to be designed cloud-natively, such as microservice architectures. However, existing service meshes are mainly designed for centralized cloud environments within a single data center and have quite limited support for distributed/edge
cloud scenarios.

With traditional cloud computing environment and monolithic software architecture, all requests from users will be sent back to the data center and the response will be sent back after processing. The emergence of edge computing leads cloud computing from centralized, large-scale data centers to a multi-clusters setting comprised of a network of larger and smaller virtualized infrastructure running nodes[2]. The execution of a request can be treated as a service function chain consisted of different micro-services, and each micro-service may have multiple replica instances in different clusters. Therefore, the request routing becomes vital importance among micro-service instances residing in different clusters.

In this thesis, I address the problem of service routing and load balancing in a micro-service chain across multiple clusters.

1.2 Problem

Microservice architecture pattern has been adopted by many large Internet companies like Amazon, Netflix and Linkedin to scale their applications in a more efficient way[3]. A globally distributed enterprise-scale architecture consists of multiple geographically distributed clusters. Currently, most service mesh such as Istio, Consul are dedicated to the communication among services in a single cluster, which is not applicable for the geographically distributed multi-cluster scenario.

How can software developers improve the efficiency of inter-cluster service routing in a multi-cluster environment to minimize client-side latency?

1.3 Purpose

As Section 1.2 stated, existing service meshes do not take the inter-cluster latency into account, which will lead to longer response time for the user request. Alleviating communication overhead between different clusters is crucial for the minimization of client-side latency. Therefore, a service mesh taking inter-cluster latency and load status of different clusters into account when routing requests is important. In this thesis, we aim at improving the efficiency of inter-cluster service routing in the edge computing environment and minimizing the client-side latency.
1.4 Goal

To fulfill the purpose described in Section 1.3, practical and specific goals will be introduced in this section.

The high-level goal of this thesis work is to design an algorithm that dynamically update the weights in weighted round robin algorithm based on the detected latency among different clusters, implement the tool as a daemon including containers executing the designed algorithms together with containers measuring the inter-cluster latency, and evaluate the implemented tool in different emulated environment.

The specific goals of the thesis can be summarized as follows:

- Investigate the state-of-the-art of existing load balancing algorithms and service mesh tools, pros and cons will be summarized.
- Improve traditional weighted round robin algorithms by adding dynamic weight update function.
- Design algorithm which enables low latency service routing based on the measurement of inter-cluster latency.
- Implement the proposed solution as a daemon container that can be easily deployed and maintained.
- The implementation is expected to possess features:
  - Low client-side latency is guaranteed.
  - Inter-cluster latency is periodically measured and corresponding weights will be calculated and updated.
  - The implemented agent runs as a daemon without interruption of other running services.
- Setup an emulated environment that enables evaluation with different configurations.
- Summarize and analyze the evaluation results.

1.5 Methodology

Both quantitative and qualitative research methods will be applied in this thesis. This thesis studies existing tools and state of the art in the area of service
mesh and load balancing. Qualitative methods will be used in the design of the Dynamic Weighted Round Robin algorithm. This algorithm adopts cutting-edge ideas from several related literature and integrates them with the heuristic design based on reasoning and practical experience. Quantitative methods will be used in the section of evaluation to investigate how the implemented system performs in different emulated scenarios.

1.6 Sustainability and Ethics

With the emergent paradigm of edge computing, computation resource and storage can be extended to the places closing to end users to alleviate the latency caused by data transmission, which is crucial for time-constrained applications, such as Cloud Gaming or Auto-Drive[4]. The extensible service mesh algorithm and architecture are focusing on the minimization of client latency[5], which enables the sustainability of the system from an economic perspective. Additionally, implementing the system and corresponding tests are done by creating Virtual Machines on a single server, which will not cause pollution and waste of electricity. Concerning ethics, the research study of this thesis will not expose any credential data of Ericsson Research, it is based on an officially released tool called Istio[6].

1.7 Delimitations

Most of the existing service mesh such as Istio or Hashicorp Consul are constantly improved, contributors of Istio are dedicated to the communication and automated deployment in a multi-cluster scenario, therefore, features of the framework described in section 4.1 might be inaccurate and outdated in the time of reading. The experiment software architecture is based on Kubernetes Kind[7], which is also constantly improved by the community of Kubernetes. In the context of IoT, ‘Edge’ of edge computing refers to the computing infrastructure. In the context of this thesis, we use clusters instead of mobile devices to represent edge data centers. Emulation is used in evaluation considering the difficulty to build a real geographically distributed multi-cluster environment.

1.8 Outline

The rest of this thesis is organized as follows. Chapter 2 introduces the necessary background for understanding this thesis. It includes the evolution of
cloud computing, the introduction of container orchestration framework and the state of the arts in service mesh and load balancing area. Chapter 3 gives a mathematical model for the solution and explains the proposed dynamic weighted round robin algorithm in detail. Chapter 4 discusses the system architecture and the implementation details. Chapter 5 presents the design and setup of the experiments along with results and analysis. Finally, Chapter 6 concludes the thesis and summarizes the future work.
Chapter 2

Background

2.1 The Evolution of Cloud Computing

2.1.1 Monolithic Architecture

Before the emergence of microservice architecture, most companies and organizations used monolithic software architecture. The monolithic application is a single-tiered software application in which various components are integrated into a single program. The components may include business logic, authorization, database layer, and application integration, etc.

Traditional enterprise application consists of three parts: a database storing several tables which are usually managed by the relational database management system, a client-side user interface which is implemented by HTML and Javascript, and a server-side application. With monolithic architecture, those three parts will be packed up and deployed as a whole deployable application. To make any modifications to the system, developers need to build and deploy an updated version of the whole system, which increases the cost of deployment and maintenance. Besides, the application size will be too large to start up quickly, which will lead to the downtime of a system.

Even though the monolithic application is simple to develop and maintained at the initial stage, it is difficult to adopt new and advanced technologies. The entire application is affected by the changes in languages or frameworks, developers require more efforts to work around with the application details, which is both time-consuming and clumsy for development.
2.1.2 Containerization

Compared to full machine virtualization, containerization is a lightweight alternative. It can encapsulate an application in its environment without additional configuration and dependencies set up. Containerization provides many advantages of running an application inside a virtual machine, as the application can run smoothly on any suitable physical machine without complex settings.

Since containers share resources without a full operating system to underpin each app, applications can be run distributedly without launching an entire virtual machine, which saves time and resources. Currently, Docker container is the dominant role in this area, which is designed to run on everything from virtual machines to physical computers, servers, public clouds, etc.

Containers enable developers to configure predictable environments that are isolated from other applications. Containers can also setup software dependencies in advance, such as software libraries, specific versions of programming languages. From the developers perspective, a consistent running environment is guaranteed by the container wherever the application is ultimately deployed, which can also enhance the productivity.

2.1.3 Microservice Architecture

Microservice is an approach to develop a large application with several loose coupled modules/components. Each module in the system supports a specific function or business logic, and the communication between various microservices are managed by the service mesh. Since the services are independently deployable, each service can be implemented by different developers in different languages[8], which also help developers to achieve continuous integration and delivery.

With the emerge of containers, the development of software has undergone a tremendous change. In the container, the application components can be loaded with all binaries, dependencies, and configurations which are required to run the application. Simple deployment and management can also be guaranteed. Therefore, microservice is usually mentioned and used together with container technologies, such as Docker[9]. More and more big companies such as Netflix, Amazon, etc. adopt microservices architecture to build complex and scalable applications. A comparison of the monolithic and microservice architecture is shown in Figure 2.1.

Three major patterns are used in most microservice architectures, API Gateway, Service Registry, and Service Discovery. The seamless collabora-
Figure 2.1: A comparison between monolithic and microservice architecture

Source: https://www.bmc.com/blogs/microservices-architecture/

...tion between these three patterns along with the management of Kubernetes contribute to the smooth running of microservice based applications.

- **API Gateway** In a monolithic application, there is usually just one set of endpoints for the interaction between the system and users. Even though the endpoints can be replicated and load-balanced, it still brings limitations to the application. In the microservice architecture, a set of fine-grained endpoints are exposed by each microservice.

- **Service Registry** Service Registry is the tool to decide the location of service instances to forward the request to the corresponding service, which is the basis of service discovery and needs to be highly available and up-to-date.

- **Service Discovery** In a monolithic application, services trigger one another via language-level method or remote procedure calls. While in the microservice architecture, different services are running in a container or virtual machine, whose location changes dynamically. With service discovery, each service can be identified using a router which is registered with the service register server[10]. The server-side service discovery pattern is widely used currently[11].
2.1.4 Edge Computing

Edge computing is an emerging distributed computing paradigm that aims at bringing computation and data storage close to users. Compared to traditional cloud computing, edge computing addresses and solves the problem caused by the transportation of large amounts of data. Computation is mostly performed on distributed mobile nodes or decentralized data centers, which fulfills the low-latency requirements of several use cases.

During the past several years, many companies and industrial organizations have begun to integrate cloud into their operations to gain insights from a large amount of data which is beneficial to achieve key business results, including reducing unplanned downtime, improving productivity, and reducing energy consumption. Cloud still plays a key role in achieving new performance levels through the Industrial Internet of Things. In the Industrial Internet of Things, it requires a lot of computing power to effectively manage a large amount of data from machines. Therefore, edge computing will be essential for enabling client-side computation.

The main differences between cloud computing and edge computing are listed in Table 2.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Cloud Computing</th>
<th>Edge Computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server nodes location</td>
<td>Within the internet</td>
<td>At the edge of the local network</td>
</tr>
<tr>
<td>Client and server distance</td>
<td>Multiple hops</td>
<td>Single/multiple hop</td>
</tr>
<tr>
<td>Latency</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Delay Jitter</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Security</td>
<td>Non-locally controllable</td>
<td>Locally controllable</td>
</tr>
<tr>
<td>Location awareness</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Geographical distribution</td>
<td>Centralized</td>
<td>Dense and distributed</td>
</tr>
</tbody>
</table>

Source: [12]

Table 2.1: Edge computing and cloud computing comparison

2.2 Container Orchestration

2.2.1 Kubernetes

When software is developed in the microservice way, each software component is a deployable and scalable modular. It would be extremely time-consuming and labor-intensive if the whole software is set up and maintained
 manually. As stated in the official documentation, “Kubernetes is designed as a portable, extensible open-source platform for managing containerized workloads and services, which provides a container-centric environment.” It was open-sourced by Google and is built upon their experience of developing container orchestration systems[13].

Kubernetes Architecture has three main components: Master node, Worker/Slave node, and distributed key-value store(etcd). The master node works as an entry point for all administrative tasks and is responsible for the task assignment and health check for worker nodes. The worker node can be a physical server or virtual machine which executes tasks and is controlled by the master node. Together, several worker nodes with a corresponding master node construct a cluster that can be manipulated by the exposed API in master.

Kubernetes adds an abstraction layer on top of containers - pods. A pod is the smallest unit that users can deploy or create, which may contain either a simple container or a bunch of tightly coupled containers.

2.2.2 Kubernetes Federation

With datacenters spread across the world, developers are proactively searching for the ways they deploy their applications and services across multiple clusters. With multi-cluster structure, multiple advantages can be provided, such as high availability, resilience to single-point-failure, and low latency. However, the deployments of application and services across multiple clusters can be complex and problematic. Kubernetes Federation is designed to address these needs and simplify the process of deployment on the multi-cluster environment.

Kubernetes Federation V2 is a tool for managing applications and services in multiple Kubernetes clusters tracked by Kubernetes Cluster Registry. Federation makes it simple to handle the deployment in multiple clusters by providing two major building blocks: Sync resources across clusters and Cross-cluster discovery[14]. With those features, users can dynamically adjust replicas in various clusters a workload is deployed in. Although DNS records maintained by the Kubernetes enable the routing of external requests to one of the servicing clusters, it simply based on an intuitive strategy such as Round-Robin, etc.
2.3 State of the Arts in Service Mesh

As stated in [1], “Service mesh is a configurable, low-latency infrastructure which is designed to handle a large amount of inter-services communication among different nodes either in a single cluster or multiple clusters.” Container orchestration frameworks such as Kubernetes are designed for the scheduling and managing of different services, therefore, service mesh is more like complimentary tools for orchestration system to address the need for secure and fast inter-process communication. Meanwhile, service mesh provides a standardized and uniform way for runtime operations as Kubernetes did for standardized deployments.

Most service meshes are implemented by injecting a proxy program into running service instance, which is called sidecar. Some of service mesh are developed in alternative architecture, for instance, Netflix’s technology suite is an approach that provides various functions by a set of application libraries including Ribbon, Hysterix, Eureka, and Archaius[15]. In the following subsections, the unique features of several representative service mesh tools will be introduced, the drawbacks will be discussed as well, and finally, a thorough comparison between different service mesh will be given in Table.

2.3.1 Linkerd

Linkerd is developed as a tool to solve the problems existing in managing large scale production systems, in which the communication between different services is the main source of the complex and emergent behavior. Linkerd addresses these problems by providing a consistent and uniform abstract layer on top of the communication.

Linkerd started as a network proxy(v1.0) for enabling service mesh, which was developed in a JVM language. The overhead caused by the JVM is acceptable if users simply run one node agent per host, but it consumes too much memory if running a sidecar proxy in each pod, which is a common use case nowadays. Linkerd(v2.0), merged with Conduit[16], is an evolved version written in Golang and Rust specifically for Kubernetes. Compared to other service mesh designs, Linkerd has a tight default coupling between the data plane and control plane, which simplifies the configuration. The architecture of Linkerd is shown in Figure 2.2.

Although the memory and several other issues have been fixed in Linkerd by rewritten in Golang and Rust, there are still lot improvements that can be done from the stability and functionality perspective. For example, distributed
tracing is lacked in the latest version of Linkerd2, which is well-suited to debugging and monitoring modern distributed software architectures.

Figure 2.2: Linkerd architecture  

2.3.2 Hashicorp Consul  
Consul is a single binary offering both server and client functionalities and contains various capabilities for service catalog, configuration, TLS certificates, authorization and so on. Unlike Istio, Consul can work without collaboration with other additional systems, it still optionally supports external systems such as Vault to augment behavior. The independence of Consul enables it to be simply installed on any platform, including directly onto the machine. Consul is built upon Serf which provides a full gossip protocol that fulfills multiple purposes, Consul operates the service mesh in a fully decentralized way without any centralized control plane, which could become a bottleneck of the system.

An agent-based model is applied in Consul, therefore, each node in the cluster runs a Consul Client. The Client keeps a local cache that is efficiently updated from servers. With the fully distributed running Client agent, all secure service communication can be done in microseconds without external communication, which enables developers to implement connection enforce-
ment at the edge without communicating to the central servers. Istio forwards requests to a central Mixer service and needs to propagate updates through Pilot, which could be time-consuming.

The architecture of Consul is shown in Figure 2.3. The underlying Serf gossip protocol[17] is used to broadcast messages among clusters, which alleviate the overhead caused by communication between nodes and a centralized control plane. The new member in the cluster only needs to do full state sync over TCP when joining the cluster, UDP will be used afterward to enable gossip. Periodic random probing with a configurable interval is used to realize failure detection.

Figure 2.3: Consul architecture
Source: https://www.consul.io/docs/internals/architecture.html
2.3.3 Istio

Istio is an open source service mesh initially developed by Google, IBM and Lyft. Istio is built on top of the Envoy proxy, which acts as its data plane. Envoy is a proxy built-in application layer which is often used as a data plane in the service mesh. Envoy is implemented in C++ and designed to run as a sidecar alongside each workload[18]. Envoy proxy provides multiple features including dynamic service discovery, TLS termination, load balancing, fault injection, and circuit breakers, etc. Istio’s control plane runs on top of the proxies and consists of three major components, Pilot, Mixer, and Citadel. The detailed architecture will be introduced in Section 4.1.2.

Nowadays, Istio is one of the most popular open source service mesh, combined with other tools, it forms a multi-functional ecosystem which is shown in Figure 2.4.

2.3.4 Netflix Suite

As one of the earliest adopters of microservices, Netflix develops a whole suite of microservices-related infrastructure modules and document the journey along the way. Various Netflix components collaborate to accomplish an integrated microservice architecture pattern, the details can be found in Table 2.2.

<table>
<thead>
<tr>
<th>Operations Component</th>
<th>Netflix OSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Discovery Server</td>
<td>Netflix Eureka</td>
</tr>
<tr>
<td>Edge Server</td>
<td>Netflix Zuul</td>
</tr>
<tr>
<td>Central Configuration Server</td>
<td>Spring Cloud Config Server</td>
</tr>
<tr>
<td>Dynamic Routing and Load Balancer</td>
<td>Netflix Ribbon</td>
</tr>
<tr>
<td>Monitoring</td>
<td>Netflix Hystrix Dashboard and Turbine</td>
</tr>
</tbody>
</table>

Table 2.2: Netflix suite components

Netflix Eureka enables microservices to register themselves at runtime when they are launched in the system. Netflix Ribbon utilizes the information saved in Eureka to pinpoint appropriate service instances. If multiple service instances are found, load balancing will be applied to spread the requests among the available running instances. Ribbon run as an embedded component in each service consumer, which is similar to the envoy in Istio. Netflix Zuul works as the gatekeeper to the outside world in the system, which only
allows authorized external requests to pass through. Zuul can dynamically allocate ports to microservices as entry points, use Ribbon to find appropriate services and routes the external requests to corresponding service instances. Since the whole Netflix Suite was developed the launching of Container and Kubernetes, it is built on top of AWS EC2 primitives instead of Docker containers. Therefore, developers have to add a lot of Java libraries to their code to utilize the functionalities of Netflix Suite. This issue was addressed by the next generation code independent service mesh perfectly.
2.3.5 Comparison Summary

The detailed functionalities comparison is shown in Table 2.3. Since the Netflix Suite is not designed for a container-based microservice system and the Linkerd2 is the enhanced version for Linkerd, the comparison takes Istio, Linkerd2, and Hashicorp Consul into account.

<table>
<thead>
<tr>
<th></th>
<th>Istio</th>
<th>Linkerd2</th>
<th>Hashicorp Consul</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Sidecar</td>
<td>Sidecar</td>
<td>Sidecar</td>
</tr>
<tr>
<td>Platform</td>
<td>Kubernetes</td>
<td>Kubernetes</td>
<td>Any</td>
</tr>
<tr>
<td>Programming Language</td>
<td>Go</td>
<td>Go &amp; Rust</td>
<td>Go</td>
</tr>
<tr>
<td>Supporting Protocol</td>
<td>HTTP1.1/HTTP2/gRPC/TCP</td>
<td>HTTP1.1/HTTP2/gRPC/TCP</td>
<td>TCP</td>
</tr>
<tr>
<td>Default Data Plane</td>
<td>Envoy (Support others)</td>
<td>Native</td>
<td>Native</td>
</tr>
<tr>
<td>Auto Sidecar Injection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Encryption</td>
<td>Yes</td>
<td>Experimental Phase</td>
<td>Yes</td>
</tr>
<tr>
<td>Traffic Control</td>
<td>Label/content based routing, traffic shifting</td>
<td>None</td>
<td>Static upstream, prepared query</td>
</tr>
<tr>
<td>Tracing Integration</td>
<td>Jaeger</td>
<td>None</td>
<td>Pluggable</td>
</tr>
<tr>
<td>Paid Support</td>
<td>None</td>
<td>Yes</td>
<td>For advanced functionalities</td>
</tr>
<tr>
<td>Complexity</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

Source: https://kubedex.com/istio-vs-linkerd-vs-linkerd2-vs-consul/

Table 2.3: Service mesh comparison

Even though Consul is platform independent and fully decentralized, the multiple datacenters/clusters feature is supported only in the enterprise version. In this thesis, we choose the open source service mesh Istio as the platform for the implementation of the proposed algorithm and the following evaluation.

2.4 State of the Arts in Load Balancing

Authors in [19] propose the DAGOR overload control for the large-scale microservice architecture. The DAGOR efficiently handles the challenge raised by the subsequent overload, which may greatly influence (degrade) the system throughput even though Randomly performing load shedding can sustain the system with a saturated throughput. The overload problem is solved by the combination of overload detection and collaborative admission strategies, which efficiently minimize the latency and keep the system not overloaded at the same time.

Deval et al. propose a novel fair weighted affinity-based scheduling heuristic to address the problem of scheduling micro-services across multiple clouds[20]. According to the result analysis of the report, the scheduling strategy successfully reduce the total traffic generated and reduce overall turnaround time for the complete end-to-end service in service function chains. The latency minimization in [20] is considered from the service function chain perspective,
while in the real world scenario, most of the service chains are fixed and cannot be modified. The solution is limited due to the requirement of dynamically changed microservice chain.

In [21], authors propose a chain-oriented load balancing algorithm, namely COLBA. The solution is fully based on message queues, which achieve load balancing according to microservice requirements of chains to minimize turnaround time. Several simulation experiments are conducted, and the chain-oriented solution performs better than microservice-oriented and instance-oriented approaches. COLBA simultaneously takes the heterogeneity of requests and inter-chain competition into account, which is the highlight of this paper. However, this approach could not contribute too much to the industrial implementation due to the high time complexity.

Puthal et al. come up with a solution to address the security issue existing in load balancing which is normally ignored in most of the literature. In [22], authors propose an innovative secured and sustainable load balancing strategy for edge data centers. The technique consists of two major parts, the secure authentication of the edge data centers and a sustainable load balancing architecture which takes load status of edge data centers into account. This solution not only solves the load balancing problem but also protects the system from outsider attacks in an open and hostile environment.
Chapter 3
Design

This chapter introduces the mathematical discussion of problem and corresponding algorithm solution.

3.1 Mathematical Discussion

3.1.1 Definition of Different Variables

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I$</td>
<td>Set of edge data center(cluster)</td>
</tr>
<tr>
<td>$E_i$</td>
<td>The edge data center(cluster) with identification $i$</td>
</tr>
<tr>
<td>$J$</td>
<td>Set of Micro-services</td>
</tr>
<tr>
<td>$S_j$</td>
<td>The micro-service with identification $j$</td>
</tr>
<tr>
<td>$N_j$</td>
<td>The amount of instances of micro-service $S_j$</td>
</tr>
<tr>
<td>$I_j$</td>
<td>Set of instances for micro-service $S_j$</td>
</tr>
<tr>
<td>$I_{jk}$</td>
<td>The instance with identification $k$ of micro-service $j$, $k \in N_j$</td>
</tr>
<tr>
<td>$e_{jk}$</td>
<td>The location of $I_{jk}$, $e_{jk}$ belongs to one $E_i$</td>
</tr>
<tr>
<td>$L_{ij}$</td>
<td>Latency time between two different edge data centers $E_i$ and $E_j$</td>
</tr>
<tr>
<td>$T_{jk}$</td>
<td>The execution time of service $j$ on instance $I_{jk}$, which depends on the load balance situation of $e_{jk}$</td>
</tr>
<tr>
<td>$c_{jk}$</td>
<td>The load Status of instance $I_{jk}$ when handling requests from various users.</td>
</tr>
<tr>
<td>$C_{jk}$</td>
<td>The capacity of $I_{jk}$</td>
</tr>
</tbody>
</table>

Table 3.1: Definition of mathematical model variables

We consider a set of edge data centers $E_i$ hosting various micro-service instances. The latency overhead among those data centers is modeled as a
matrix of size $n \times n$ where the element $L_{ij}$ represents the latency time between the edge data center $E_i$ and $E_j$.

In this case we consider that micro-services are encapsulated and deployed in containers[23]. Each container only includes the instance of one micro-service and probably one sidecar proxy. Each micro-service $S_j$ has $N_j$ instances, $I_{jk}$ represents the $k$th instance of micro-service $S_j$. $c_{jk}$ stands for the hosting edge data center of instance $I_{jk}$. Since the computing resource of instances and load status of edge data centers are different, the execution time of $S_j$ on instance $I_{jk}$ are different, therefore, $T_{jk}$ represents the execution time of $S_j$ on instance $I_{jk}$. $C_{jk}$ is the capacity of service instance $I_{jk}$.

### 3.1.2 Optimization Objectives Definition

We define service chain as $C = \langle S_{s1}, S_{s2}, ..., S_{sm} \rangle$, which consists of $m$ micro-services and $S_{sx} \in J$. The service chain is an ordered list of services, which is dynamically changed based on the user requests. In order to response user’s request, the service chain needs to be executed in the fixed order.

Services in $\langle S_{s1}, S_{s2}, ..., S_{sm} \rangle$ are related in terms of consumption. A micro-service $S_{sx}$ consumes $S_{sx'}$, when the former one needs the results of the latter one to perform its task. For simplicity, we assume all services in the system are stateless, which just handle requests and return response without saving any information.

Our objective here is to minimize the total execution time of the service chain, which is the time

$$
\min T_{s1} + L_{s1s2} + T_{s2} + L_{s2s3} + ... + L_{s_{n-1}s_n} + T_{sn} \quad (3.1)
$$

$$
\min \sum_{i=1}^{n} T_{si} + \sum_{j=1}^{n-1} L_{sjs_{j+1}} \quad (3.2)
$$

Meanwhile, the total load in one instance should not exceed its capacity.

$$
c_{jk} < C_{jk} \quad \forall I_{jk} \in I_j \quad (3.3)
$$

Since the execution time $T_{jk}$ in service instance $I_{jk}$ depends on the load status $c_{jk}$, which has already been taken care of by Service Mesh. In the following section, we assume that the execution time $T_{jk}$ in different $I_{jk}$ are similar and propose a greedy-based algorithm, which makes the locally optimal choice at each step.
3.2 Algorithms

In this section, I will start with the introduction of the routing algorithm - Weighted Round Robin, which is adopted by Envoy Proxy. Then, the core algorithm Latency-aware Weighted Round Robin is defined.

3.2.1 Weighted Round Robin

Initially, the weighted round robin scheduling is designed to better handle server with different processing capacities. Patel and Dalal proposed a Low-latency Weighted Round Robin in [24] for achieving low latency and improved fairness in high speed networks.

Parameters and functions in the algorithm of weighted round robin are explained as follows[25]:

- $S$ A list of clusters that could receive and handle requests.
- $W(S_i)$ The assigned weight of cluster $S_i$.
- $j$ The server selected last time and $j$ is initialized with -1.
- $currentWeight$ The current weight in scheduling, and $currentWeight$ is initialized with 0.
- $max(S)$ The maximum weight of all the servers in $S$.
- $gcd(S)$ The greatest common divisor of all server weights in $S$. 
Algorithm 1: Weighted Round Robin

```
while true do
    j = (j + 1) mod n;
    if i == 0 then
        currentWeight ← currentWeight − gcd(S);
        if currentWeight <= 0 then
            cw ← max(S);
            if currentWeight == 0 then
                return NULL;
            end
        end
    end
    if W(S_i) >= currentWeight then
        return S_i;
    end
end
```

With the weighted round robin algorithm, the higher the weight of a cluster, the larger the proportion of service requests the cluster receives. However, in existing service mesh, the weights of clusters need to be manually set up and updated, which is static and inflexible.

### 3.2.2 Latency-aware Weighted Round Robin

Since the latency among different clusters may change over time and the solution adopted by Istio currently is a way of static routing, which means users have to manually modify the weight of different destinations. In this section, we propose an algorithm that can periodically update the weight of different destinations based on the detected latency without manual interaction, namely Latency-aware Weighted Round Robin (LAWRR). The algorithm consists of two major parts, expected latency calculation and weights update, which will be elaborated in the following subsections.

**Expected Latency Calculation**

Users prefer to get the response as quick as possible, therefore, this algorithm converts all the collected metrics into an expected latency. New weights are calculated based on the expected latency of each cluster, using a practical and efficient method proposed by Lukáš Poláček[26].

Parameters in the algorithm are explained as follows:
• $s$ Success rate of a cluster.

• $l$ The success latency, which is also the round-trip time in our circumstance.

• $p$ The penalty time for failure latency.

• $f$ The failure latency.

• $E$ The expected latency from the client’s perspective.

Suppose a cluster has a success rate of 50%, then half of the request sent to the cluster will be successfully handled and responded. With the probability of $\frac{1}{2}$ we need to do the first retry, with a probability of $\frac{1}{4}$ we need to do the second retry. The total amounts of retry can be calculated as follows:

$$\text{Total\_Retry} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... = 2 \quad (3.4)$$

In this circumstance, there are 1 request fails and 1 request succeeds on average. From the client’s side, the expect latency is $l + f$, which is the sum of 1 success latency and 1 latency latency. When a request fails, we assume that the client-side has a bad connection with the server. The penalty time should be added to the success latency to get a failure latency,

$$f = l + p \quad (3.5)$$

According to [27], $s$ success rate leads to $1/s$ times retry. The expected latency of cluster $i$ can be calculated as below:

$$E = l + f \times (1/s - 1) = l + (l + p) \times (1/s - 1) \quad (3.6)$$
Latency-aware Weighted Round Robin Algorithm

Algorithm 2: Latency-aware Weighted Round Robin Algorithm

\[
\text{every } \Delta T \text{ do}
\]
\[
\text{Collect Network Latency;}
\]
\[
\text{Calculate Expected Latency;}
\]
\[
\text{if } \text{success}_\text{rate} > \text{Threshold} \text{ then}
\]
\[
\quad \text{Move the server out of rotation;}
\]
\[
\text{end}
\]
\[
\text{if } \text{flag} = \text{true} \text{ then}
\]
\[
\quad W_{\text{local}} \leftarrow \text{LOCAL}_\text{WEIGHT}
\]
\[
\text{else}
\]
\[
\quad W_{\text{local}} \leftarrow 0
\]
\[
\text{end}
\]
\[
\text{foreach } \text{Cluster}_i \text{ in the Set of Clusters do}
\]
\[
\quad W_i \leftarrow 1/E_i \times W_{\text{remain}}
\]
\[
\text{end}
\]
\[
\text{Weight Normalization;}
\]
\[
\text{Apply New Weights Configuration;}
\]

In each round, the network latency between different clusters will be measured and saved. According to equation 3.6, we get the expected latency of each cluster as \( L = \{E_1, E_2, E_3, ..., E_n\} \). The success rate of each cluster is updated based on the logs if the success rate becomes smaller than the Threshold, this cluster will be temporarily moved out of the rotation for this service.

Flag value true means that local cluster hosts service which can handle the request. We assume the network latency between nodes in a local cluster is much smaller than the latency of inter-clusters. Therefore, a LOCALWEIGHT will be assigned to the local cluster first, which indicates that more requests will be handled by local service.

We have the expected latency for each cluster \( L = E_1, E_2, E_3, ..., E_n \), the weight of cluster is negatively correlated with its expected latency. The lower the latency, the higher the speed is and the higher the weight.

\[
W = \{W_1, W_2, W_3, ..., W_n\}
\]
\[
= \{W_{\text{remain}}/E_1, W_{\text{remain}}/E_2, W_{\text{remain}}/E_3, ..., W_{\text{remain}}/E_n\} \quad (3.7)
\]

The weight of each cluster is calculated as the production of remain weight and reciprocal of expected latency. The product may be not integer, the weight
normalization is needed after the calculation. Then, the configuration of the new weight will be applied to the system.
The proposed solution is implemented as a docker image that can be simply deployed as a running daemon without complicated configurations, namely RoutingAgent. The RoutingAgent is written in Golang. The RoutingAgent collaborates with Kubernetes and Istio to realize dynamic Weighted Round Robin. In the following sections, the architecture and mechanism of Kubernetes, Istio and the whole system will be roughly introduced first, the mechanism and implementation details of RoutingAgent will be elaborated afterwards.

4.1 System Architecture

4.1.1 Kubernetes Architecture

As introduced in section 2.2.1, Kubernetes orchestrates a cluster by two abstractions: master and worker node. There are three main components in Kubernetes master, which are API Server, Controller Manager and Scheduler. Their functions are introduced as follow:

- The API Server is the interactive portal between users and Kubernetes clusters. Kubectl is responsible for handling and validating commands input by users.

- The Controller runs several monitor process in the background to regulate the shared state of the whole cluster and executes routine tasks. When a change in configuration occurs, the controller pinpoints the change and starts working towards the new desired state.
• The Scheduler is the one who performs the configuration tasks and moves the whole cluster to a new state. When a change has been detected by the controller, the scheduler helps schedule pods among various nodes based on resource utilization.

The worker node contains two main components, which are kubelet and kube-proxy.

• The kubelet is the main service on a node, which takes the new configuration through the API Server and updates the state of pods hosts in the local node.

• The kube-proxy performs request routing in a cluster.

![Kubernetes architecture](https://x-team.com/blog/introduction-kubernetes-architecture)

Figure 4.1: Kubernetes architecture

**Source:** https://x-team.com/blog/introduction-kubernetes-architecture

### 4.1.2 Istio Architecture

Istio is a designed abstraction containing both a control plane and several data planes. The data plane is implemented with sidecar proxy container, which is Envoy proxy by default. Sidecar containers utilize the GRPC protocol to communicate with the control plane to optimize the pushdown model of changes
inside the cluster. Besides, the data plane is responsible for the inter-cluster and intra-cluster service routing and logging.

The control plane consists of four main components, namely Pilot, Mixer, Citadel and Galley.

- Pilot converts the routing rule into Envoy-specific configuration and propagates all information needed in the cluster, including services, endpoints and translated routing rules at run time.
- Mixer is an optional control plane component which is responsible for policy control and telemetry collection.
- Citadel is designed for the security purpose which takes charge of certificate insurance and rotation.
- Galley subscribes to resources in the underlying platform. When updated resources are pushed to the underlying platform, Galley will receive notification and forward the new configuration to Pilot.

Figure 4.2: Istio architecture

The proposed solution is targeted at addressing issues existing in dynamic inter-cluster service routing. The connection and communication among different clusters are implemented by Istio Gateway Connectivity with multiple control planes. For simplicity, the mechanism will be explained in an example with two clusters, which can be generalized into a multiple-cluster scenario.

Those two clusters in Figure 4.3 have identical Istio control plane respectively. Users are responsible for deploying services objects on each cluster with different version label which is essential for the intra-cluster routing. Services from remote clusters are created locally via ServiceEntries in name.namespace.global format. CoreDNS provides DNS resolution to find the address of a remote cluster which hosts the desired services. When a request needs to be sent to the remote cluster, the Envoy proxy in the pod will redirect the traffic to the remote gateway of another cluster. Once the request is received by cluster 2, local service routing rule will be applied. To enable mutual TLS communication across clusters, each cluster’s Citadel will be configured with intermediate CA credentials generated by a shared root CA.

Figure 4.3: Istio multicluster architecture
Source: https://istio.io/docs/setup/kubernetes/install/multicluster/gateways/

4.1.3 Overall Software Architecture

The implementation utilizes two main Daemon containers:
• The main module is responsible for tracking the latency between different clusters and dynamically calculating the weight for different clusters. The updated weights will be applied through the Kubernetes API, namely kubectl[28].

• The latency detection is rendered by another daemon container called httping[29], which measures the latency of the web server and network.

Our proposed solution runs in the Kubernetes master node just as the abstracted Istio control plane. The algorithm introduced in section 3.2 is executed periodically in the following steps:

1. RoutingAgent triggers the HTTPing service to retrieve the latest latency among different clusters.

2. The new weight will be calculated based on the tracked latency inside RoutingAgent and a new YAML configuration file will be generated.

3. The updated configuration will be applied to Kubernetes API Server via kubectl.

4. Galley received notification that new configuration has been set and forward the new configuration to Pilot.

5. Pilot converts the configuration into Envoy-specific configuration and propagates the latest configuration to all connected service containers.

### 4.2 Mechanism

#### 4.2.1 HTTPing Service

In microservice-based applications, software components are deployed in multiple places and servers. The services need to communicate with each other via different transport protocols, including HTTP, HTTP2, a message bus, TCP socket or even UDP. Different components of the system may influence the latency, including network transit, hardware, application design, etc. The inter-cluster network latency has a significant impact on the efficiency of microservice communication. Since the HTTP protocol is widely used in microservice communication, we will use the HTTPing service as an approach to measure the inter-cluster latency. Compared to Ping and Telnet, HTTPing can provide more accurate and granular information.
HTTPing is one basic tool developed based on Unix ping command, with which we can measure the minimum, average and maximum time between request and response. Unlike other latency indicators, HTTPing is designed to measure the network latency in application level\[30\]. Besides, failures to response can help to pinpoint a breakdown of services in the system. Compared to ICMP-based ping, HTTPing can work smoothly in an environment where ICMP is not allowed.

In the following proposed system, Docker containers created by Bret Fisher are used to host the HTTPing Service\[31\]. The HTTPing docker image is based on Debian Stretch, a lightweight minimal image, which will not bring to much overhead to the system. The workflow of calling HTTPing microservice in our system is depicted in Figure 4.5 and the steps are elaborated as follow:

1. RoutingAgent invokes HTTPing service through kubectl.
2. HTTPing sends ping packets different clusters through their exposed gateway address and port number.
3. The response for the ping packet is sent back to the HTTPing service.
4. RoutingAgent reads the logs from HTTPing, which contains the details of latency. Corresponding latency will be calculated based on the five times connections and pings.
Considering the HTTPing Service is implemented by measuring the interval between sending and receiving requests, the increase of inter-cluster latency also influences the execution time of HTTPing command. We have measured the execution time of HTTPing command in five different inter-cluster latency setup, which is shown in Table 4.1. Normally, connecting to a service across 2400km of distances will increase around 25ms to the latency[32]. The command execution time is still acceptable when the inter-cluster latency reaches 500ms, which is relatively high latency in real-world scenarios. Therefore, the overhead brought by HTTPing service is durable in our implementation.

<table>
<thead>
<tr>
<th>Latency (ms)</th>
<th>Command Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5.957</td>
</tr>
<tr>
<td>200</td>
<td>6.459</td>
</tr>
<tr>
<td>300</td>
<td>6.878</td>
</tr>
<tr>
<td>400</td>
<td>7.409</td>
</tr>
<tr>
<td>500</td>
<td>7.869</td>
</tr>
</tbody>
</table>

Table 4.1: Inter-cluster latency and corresponding command execution time
4.2.2 RoutingAgent

The RoutingAgent is implemented with Golang as a docker image, which is easily deployed and managed by Kubernetes. The RoutingAgent is running as a daemon like other daemon services of Kubernetes and Istio, collaborated with HTTPing service to update and apply the latest configuration YAML File. The workflow of RoutingAgent is depicted in Figure 4.6 and elaborated as follow:

1. RoutingAgent retrieves the configuration of different services and clusters from Kubernetes API server via kubectl. The information includes global services, HTTPing service pod’s name, federated clusters information, connected gateway address, and port number.

2. Istio uses label to locate different services, therefore, label will be paired
with the corresponding cluster address for the following weights updates.

3. RoutingAgent invokes HTTPing service to measure the latency among different clusters, the detailed procedure of measurement is introduced in Section 4.2.1. The expected latency will be calculated based on the collected metrics.

4. The weight is calculated based on the algorithm proposed in Section 3.2.2.

5. The old YAML configuration file is parsed and the weights of different clusters will be updated based on the results derived at Step 4.

6. Latest YAML configuration file will be applied and the corresponding services will be routed based on the new configuration.

7. System checks whether new clusters or services have been added. If there are any updates, the workflow will start from Step 1, if not, the workflow will start from Step 3 next round.

With the Docker-based RoutingAgent, the proposed latency-aware optimization for Istio is simple for deployment and maintenance. Constant check for service and cluster updates enables the solution to add the new services into monitoring without manual configuration.
Chapter 5

Evaluation

The proposed solution is evaluated with designed experiments in an emulated environment. Experiments are designed to focus on testing the system in various simulated scenarios. The emulation tools and system configuration will be introduced in the first section. Different metrics will be measured and analyzed in the following sections.

5.1 Experiment Design

5.1.1 Topology Setup

The multicluster scenario is emulated by Kind and Istio multicluster gateway connectivity. Kind[33] is the abbreviation of Kubernetes in Docker, which utilizes Docker container "nodes" to simulate multi-node clusters. Istio multicluster gateway connectivity has been introduced in Section 4.1.2.

Considering the limited computational resources and complexity of monitoring, we will set up an environment consisting of five fully connected clusters in the following experiments. We assume that the overhead of intra-cluster communication is much smaller than the overhead of inter-cluster communication. Therefore, each Kind container node represents a cluster in the following experiments (see Figure 5.1).

5.1.2 Network Latency Emulation

Nodes in the emulated multicluster environment are docker containers, therefore, the tool used for network latency emulation is Pumba[34]. Pumba is a powerful chaos testing tool for injecting chaos in Docker container, which can
also perform network emulation through delays packet loss, rate limiting and more[35].

In the following experiments, the inter-cluster latency configuration is initialized as values listed in Table 5.1. In the following experiments in Section 5.4 and Section 5.3, the inter-cluster latency will be dynamically altered to measure the system’s performance.

<table>
<thead>
<tr>
<th>Latency Among Clusters(ms)</th>
<th>Cluster 1</th>
<th>Cluster 2</th>
<th>Cluster 3</th>
<th>Cluster 4</th>
<th>Cluster 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster 1</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Cluster 2</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Cluster 3</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Cluster 4</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Cluster 5</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.1: The latency setting among different clusters
5.1.3 Simulation Task

The microservice system we used here for evaluation is the official example from Istio, which contains several services implemented by different languages and technologies. Some modifications were made for the system for the measurement of response time.

The system consists four kinds of microservice, a webpage interface called productpage implemented with python, a reviews service written in Java, a details function developed by Ruby and a Ratings service realized by Node JS. The workflow of the example system is shown in Figure 5.2.

- **Product Page** the productpage microservice invokes the reviews and details microservices to load the contents of web page.
- **Details** the details contains book information.
- **Reviews** the reviews microservice shows book reviews, which can also calls the ratings microservice. There are three versions of this microservice, v1 does not call the ratings, v2 calls ratings and displays it in black star format, v3 calls ratings and displays it in red star format. For the simplicity of the testing system, all reviews service will be deployed with reviews-v2 image.
- **Ratings** the ratings microservice contains book ranking information that related to the book review.

With the assistance of Kiali add-ons of Istio, we can derive the service graph of bookinfo application, which shows a node for each service in the service mesh but excludes all apps and workloads from the graph(See Figure 5.3). The arrow points in the direction of service invocation.

The deployments of the above example application in our emulated multi-cluster environment are listed as Table 5.2.

<table>
<thead>
<tr>
<th></th>
<th>Product Page</th>
<th>Details</th>
<th>Reviews</th>
<th>Ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cluster2</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Cluster3</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cluster4</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cluster5</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: The deployments of Bookinfo application
In the following experiments, the loading time of the product page will be considered as the response time, which triggers two microservice chains. All microservices will be randomly deployed in the multi-cluster environment including five fully connected clusters.
5.1.4 Performance Measurement

In the following experiments, three metrics will be collected and analyzed, including load status, average response (turnaround) time, and percentile of response time. The metric data will be collected by a powerful collection tool called Prometheus[36]. Grafana[37] is used with Prometheus to generate a dashboard with multiple panels to monitor the status of the system. In general, we choose turnaround time as the latency measurement metric and CPU usage as the load balancing measurement metric.

Turnaround time (TAT) is the total time taken between the submission of a request and the return of the complete output to the customer[38]. In our experiment, it is the time between we send a request for the product page and the complete loading of the page. Both average TAT and percentile TAT are used in the following evaluation. From Average TAT, we can observe the performance that represents the majority of the requests. However, the Average TAT could be bias and not sufficient to measure the performance in some circumstances. In the context of time-sensitive services, percentile turnaround time is used to measure how many customers request has been fulfilled in a given limited period.

To measure the load status of different clusters, CPU usage will be used as the metric for the load balancing evaluation[39]. In the following experiments, each Kind container represents a single cluster containing multiple services, the monitoring of the CPU usage of the container can effectively validate the load status of an emulated cluster.

5.2 Influence of Local Weight

According to the Latency-aware Weighted Round Robin algorithm (LAWRR) proposed in Section 3.2.2, we assume that the request will be handled faster in the local cluster and a variable called LOCAL_WEIGHT has been introduced. In this section, experiments will be conducted to validate the influence of different LOCAL_WEIGHT on the system’s performance.

We will use the same configuration and only the LOCAL_WEIGHT will be changed to derive different results. The metrics we used here are average turnaround time and percentile response time. Here, we use four LOCAL_WEIGHT for testing purposes. The results are shown in the following figures.

Figure 5.4 illustrates the change of average turnaround time of request handling in different LOCAL_WEIGHT configuration. Different values of Re-
quest Per Second(RPS) are used to validate whether the system can effectively handle requests when the throughput increases. Considering the intra-cluster delay is smaller than the inter-cluster one, the system with $LOCAL\_WEIGHT = 100$ performs better than systems with other configurations when the RPS is small. With the RPS gradually increase, we can find that the performance of the system with $LOCAL\_WEIGHT = 25$ is improved constantly, especially when the RPS equals 2000. With the increase of the RPS, the system with larger $LOCAL\_WEIGHT$ performs worse due to the overload of local service. Meanwhile, the system with smaller $LOCAL\_WEIGHT$ performs better since requests are evenly distributed among clusters, which will not cause overload. Among all those 4 configurations, the system with $LOCAL\_WEIGHT=50$ has the most stable performance, no matter in the low or high RPS environments.

From Figure 5.5, the sub-figure on the left side shows the 99 percentile turnaround tie and the sub-figure on the right side shows the 90 percentile turnaround time within different $LOCAL\_WEIGHT$ settings and RPS. Curves in Figure 5.5 have a similar tendency as the curves in 5.4, the turnaround time increases when there are more requests received per second by the system.

Considering the overall performance, the system with $LOCAL\_WEIGHT=50$ delivers the most stable performance in different environments with different RPS configuration. Therefore, in the following experiments, we will set $LOCAL\_WEIGHT$ to 50, which means half of the requests will be handled inter-
5.3 Turnaround Time Performance

In this section, we will use two metrics to evaluate the system performance, namely average turnaround time and percentile of response time. The average turnaround time will show the average performance of the system under different simulation of load status. While in the context of the time-sensitive services, many applications have extremely slow outliers, which should be excluded. The percentile is provided to show the performance of the system in most use cases.

In the following experiments, we choose to compare the performance of our solution to the performance of Istio without implemented RoutingAgent. The performance of Istio without RoutingAgent in an environment with consistent inter-cluster latency will be used as the benchmark. The inter-cluster latency variation for experiments are shown in Figure 5.6.

5.3.1 Average Turnaround Time

To compare the results of three experiments with different settings, we will use Figure 5.7 to see the tendency of turnaround time variation in different environments.

With the request per second ascending, the average turnaround time all has an increasing tendency. Compared to the benchmark, the average turnaround time of Istio without RoutingAgent slightly increases. When the RPS is smaller than 500, the performance of Istio with RoutingAgent is as good as the benchmark. In the same network condition simulation, our solution has a better performance in all RPS configurations.
Figure 5.6: Network latency variation among different clusters
From time 60s to 180s, both Cluster 3 and Cluster 4 has doubled latency, while Cluster 2 has normal latency from 60s to 120s and Cluster 3 has normal latency from 120s to 180s. In our solution, when the latency of Cluster 3 and Cluster 4 have been doubled, the system will automatically detect the variation and put more weights on the Clusters with normal latency. With the adjustment of the weights, more traffic will be forwarded to the cluster with lower latency, therefore, the turnaround time will not fluctuate even though the network condition changes.

### 5.3.2 Percentile of Turnaround Time

From Figure 5.8, we can see the 90 and 99 percentile turnaround time of the above experiments. From the left sub-figure, the performance of Istio with/without RoutingAgent is better than the average turnaround time, which means most of the requests can be handled in a limited time even though the network condition changes. However, in the right sub-figure, the Istio with RoutingAgent performs much better than the Istio without RoutingAgent. When the RQS is lower than 500, the performance of Istio with RoutingAgent
is even as good as the benchmark. With the RPS increasing, the single replica cannot efficiently handle all coming requests, therefore, the client-side takes more time to receive the response.

From the analysis of the average and percentile of turnaround time, the results show that the Istio with RoutingAgent in a multi-cluster environment with dynamic network condition performs well and the low client latency goal has been fulfilled.

5.4 Load Balancing Performance

Apart from low customer side latency, load balancing of clusters and microservice instances is another requirement for the system. In this section, load status will be evaluated by CPU usage of a cluster. The $\Delta T$ is set to 20s, therefore, the network latency will be collected every 20s and new weights will be updated. Various experiments will be conducted to validate whether the system fulfills the load balancing requirement.

Figure 5.9 shows the latency settings for the following experiments. The latency among different clusters is initialized as the values in Table 5.1. The latency between cluster 1 and cluster 3, cluster 1 and cluster 4 are doubled from 60s to 120s, after 120s, the latency drop back to 100ms. The latency between cluster 1 and cluster 5 is doubled from 180s to 240s, after 240s, the latency drops back to 100ms. The latency among other clusters are not changed. The load status of different clusters in the environment with above network latency variations are shown in Figure 5.10.

In this experiment, the Product Page is constantly visited by the script with the frequency of 500 requests per second(RPS) and the RoutingAgent update the weights every 20 seconds. Since the Product Page only has one replica in Cluster 1, the CPU Usage of Cluster 1 is higher than other clusters through the whole experiment. When the latency between cluster 1 and cluster 3, cluster 1
Figure 5.9: Network latency variation among different clusters
and cluster 4 increase from 60s, the weights of cluster 3 and cluster 4 decrease. Cluster 5 is the only one which contains Reviews service to handle the request sent from Product Page. Therefore, the CPU Usage of Cluster 5 increases slightly. The same situation happens to Cluster 2, which contains the only replica of Details except Cluster 3 and Cluster 4, therefore, the CPU Usage of Cluster 2 has the same tendency as Cluster 5.

From 120s, all inter-cluster latency switch back to 100ms, the CPU Usage of all clusters go back to the initial status. The latency between Cluster 1 and Cluster 5 increases to 200ms after 180s. Since Reviews and Ratings both have replicas in Cluster 3 and Cluster 4, the CPU Usage of Cluster 5 drops, the CPU Usage of Cluster 3 and Cluster 4 slightly increase. From 180s, the latency between Cluster 1 and Cluster 5 drop to 100ms, and all clusters load status go back to normal.

To summarize, Istio with RoutingAgent can effectively update the weights of target clusters and distribute loads when the network condition changes. Through the whole experiment, none of the cluster’s CPU usage exceeds 60%, which means the load balancing goal of our solution has been fulfilled.
Chapter 6

Conclusion

6.1 Discussion

Cloud computing is undergoing a rapid development both in academia and industry, which aims at providing distributed, virtualized and elastic resources as utilities to end users. With the emerge of microservice architecture, more and more companies move from traditional centralized architecture to a loosely coupled fully distributed architecture, which leads to the popularization of edge computing. For example, developers could deploy the services among different clusters with multiple replicas, and the user request could be handled by the closest cluster instead of propagating the request to the remote central data center.

Nowadays, many businesses are extremely demanding and require low latency, such as autonomous driving and Virtual Reality. The execution of a task may invoke various microservices hosted in different clusters. To minimize the client-side latency, several aspects of existing service mesh still need to be improved and optimized due to the locality and computing capability limitation. Among all optimization tasks, this thesis aims to propose a dynamic latency-aware routing strategy, which is essential for the minimization of client-side turnaround time in the edge computing environment.

The thesis investigates the related work in the field of existing service mesh solution and load balancing strategies. Based on the academic literature and open source implementation of the service mesh, a dynamic latency-aware weighted round robin algorithm is proposed and implemented on the top of Istio. The implementation is a Docker image called RoutingAgent, which can be simply deployed and run as a daemon in the system. Considering the latency among different clusters may change constantly due to various reasons,
the proposed algorithm utilizes the inter-cluster latency to dynamically update the corresponding weights of different clusters periodically. Compared to the static weighted round robin, our solution can make appropriate adjustments for the routing strategy when the network condition changed. We assume the intra-cluster latency is much smaller than inter-cluster latency, the parameter LOCAL_WEIGHT is introduced to adjust the proportion of global outbound requests and local outbound requests.

To evaluate the system, we set up a multicluster environment using Kubernetes in the Docker, in which all cluster containers are fully connected. Microservices for evaluation are deployed randomly in the multicluster environment by Kubernetes. From the results of experiments, the average and 90 percentile turn around time increase significantly when the Istio without dynamic RoutingAgent is used. With the implemented RoutingAgent, the turnaround time does not fluctuate too much when the network condition changed, all clusters meanwhile keep load balancing according to the monitoring resource usage.

Reviews of the goals of the thesis work are listed as follows:

- The state-of-the-art of existing load balancing algorithms and service mesh tools are investigated and summarized.
- Based on the weighted round robin algorithm adopted in Istio, a dynamic latency-aware weighted round robin algorithm is proposed and implemented.
- The designed module is implemented as a Docker image, which runs as a daemon and is simple to deployed and maintained.
- In the evaluation, an emulated multicluster environment is set up. Experiments are designed and conducted. Based on the analysis of the results from the experiments, the system performed great in terms of:
  - The user side latency remained stable when the network condition is changed.
  - All clusters kept load balancing when the routing strategy was changed based on the proposed algorithm

To summarize, a dynamic latency-aware round robin algorithm is proposed and implemented in this thesis. The solution aims at minimizing the client-side latency and keeping load balancing in a multicluster environment with frequently changing network conditions. With experiments designed and conducted, the solution is proven to fulfill those expectations.
6.2 Future Work

In this section, some potential improvements for the system are discussed. Some drawbacks of the designed system are also recorded and possible solutions are provided as well.

Data Plane Optimization

The proposed solution in this thesis is based on the control plane of service mesh. When the network condition is changed, the new configuration for different data plane will be generated and pushed, which may generate network overhead. Data plane optimization is based on the Envoy proxy in Istio, the latency measurement and documented by the Envoy proxy instead of the control plane and requests will be forwarded to clusters based on the load status and communication overhead.

Optimized Round Trip Time Measurement

The measurement of inter-cluster round trip time is conducted by the ping service, which requires sending and receiving packets periodically. Considering the network traffic generated by the ping service, the simple solution could be optimized from the round trip time measurement aspect. Harshicorp Serf uses a network tomography system to compute network coordinates for nodes in the cluster. The round trip time among different nodes can be simply done by mathematical calculation with the network coordinates. While the whole network coordinate approach is built upon the gossip protocol, it requires the multicluster environment connected by the Consul Enterprise version.

Ant Colony Optimization

Cloud task scheduling and request routing is an NP-hard optimization problem, and many meta-heuristic algorithms have been proposed and proved to solve it. The solution proposed in this thesis is heuristic which may be constrained by several assumptions. Ant colony optimization is a dynamic task scheduling algorithm, like other evolutionary algorithms. Many researchers applied ACO in various use cases to solve NP-hard problems including traveling salesman problem, graph coloring problem and get satisfactory results. In our case, the measured round trip time among different clusters can be used as the prior knowledge to boost the ACO. The effectiveness of ACO in the op-
timization of cloud task scheduling in a single cluster environment has been validated in [40].
Bibliography


