POLITECNINO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING

Politecnico di Torino

MASTER's Degree Thesis

Latency-aware task scheduling in the cloud continuum

Supervisors

Prof. GUIDO GUIDO

Prof. ALESSIO SACCO

Candidate

CRISTOPHER CHIARO

DECEMBER 2023

Abstract

In the evolving landscape of application deployment, containerization has emerged as a popular methodology across numerous domains. Kubernetes, recognized as the leading platform for orchestrating containers, effectively handles a wide array of devices but sometimes falls short in meeting specific user demands, especially since its scheduling decisions are primarily based on computational metrics. The growing trend towards distributed cloud infrastructures, which facilitate highly reliable and efficient solutions, necessitates a rethinking of Kubernetes' default scheduler.

This dissertation introduces an enhanced Kubernetes scheduler designed for multi-cluster environments, focusing on optimizing end-to-end latency to improve the overall user experience. This novel approach sets itself apart by being adept in a geographically varied setting, crafted to adhere to latency requirements as specified by users. It is adept at adhering to particular latency thresholds or selecting the cluster with minimal latency, based on individual user needs.

A significant advancement of this work is the integration of functionalities for managing multiple users, each with distinct and sometimes divergent latency needs. This aspect is particularly vital for user bases spread across different locations, as it allows the scheduler to make tailored decisions that optimize latency for each user, considering their unique network conditions and mobility.

Extensive testing has been conducted to evaluate the performance of this scheduler, with a focus on its latency awareness capabilities. These tests have consistently shown that our scheduler significantly outperforms the default Kubernetes scheduler and other existing solutions in terms of managing latency. This superiority is evident not only in standard operation scenarios but also in complex, real-world cloud computing applications, highlighting the scheduler's effectiveness and adaptability. The results affirm the potential of our scheduler as a robust solution for contemporary cloud-based systems, where latency management is crucial for user satisfaction and system efficiency.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, **Professor Guido Marchetto**, and my co-supervisor, **Alessio Sacco**, for their meticulous guidance and unwavering support throughout this journey. Their availability for clarifications and problem-solving on a weekly basis has been a pillar of my progress.

My heartfelt appreciation goes out to my family: to **Mum and Dad**, for their unwavering emotional and financial support, never making me feel like a burden; to my sister, **Simona**, who embodies all my hopes and to whom I wish an equally prosperous career; and to my brother, **Marco**, my best friend and companion, with whom every city feels like home.

I am immensely thankful to my colleagues, whom I regard as my second family. To **Mattia, Luca, Francesco, and Alessandro** - thank you for being true friends and for sharing the most memorable years of my life with me.

A special acknowledgment goes to **Danilo, Anna, and Federico**, with whom I embarked on my university journey and shared the initial life experiences as a student living away from home. I am also grateful to **Giovanni and Alessio** for accompanying me in France, ensuring that I never felt homesick for Italy.

Lastly, but most importantly, I dedicate this milestone to **Martina**, my partner and my inspiration. This achievement is as much yours as it is mine, a reflection of our shared dreams and hopes. Here's to a future filled with joy, love, and endless adventures together.

"It's not that I'm so smart, it's just that I stay with problems longer."
Albert Einstein

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

ML
Machine Learning

QoE
Quality of Experience

QoS
Quality of Service

K8s
Kubernetes

CNI
Container Network Interface

LAIS
Latency-Aware Intent Scheduler

EC
Edge Computing

API
Application Programming Interface

CPU
Central Processing Unit

RAM

Random Access Memory

VM

Virtual Machine

IoT

Internet of Things

RBAC

Role-Based Access Control

CIDR

Classless Inter-Domain Routing

HTTP

Hypertext Transfer Protocol

IP

Internet Protocol

DNS

Domain Name System

OS

Operating System

UI

User Interface

UX

User Experience

Chapter 1

Introduction

The digital transformation has shaped the way we live, work and interact, making network-based services central elements of our daily lives. From streaming a movie, to accessing a business platform, to navigating in real time on a digital map, we expect every service to respond to our requests immediately. However, behind the scenes, a critical challenge emerges: network latency. This subtle but crucial time interval, which elapses between the user's request and the response of the service, has assumed primary importance in the digital ecosystem.

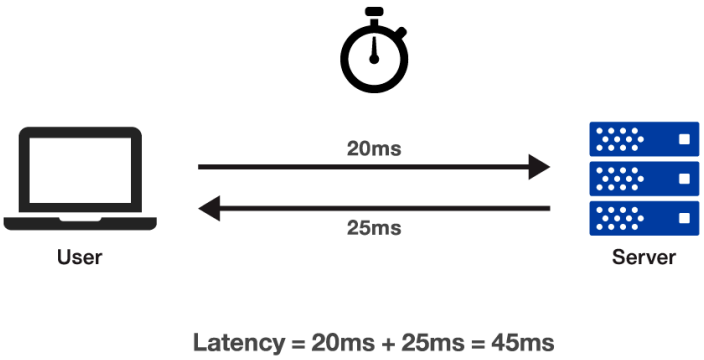


Figure 1.1: Network Latency

In a landscape where every millisecond of latency can mean a potential loss of customers or compromised user experience, traditional solutions often fall short. Optimal network latency management is not just about optimizing the data in transit, but also about where and how this data and services are processed and delivered.

Within this context, container orchestrators, tools designed to automate application deployment and management, play a key role. Among these, Kubernetes emerges as the undisputed leader. But, what makes it particularly suitable for

this task? The answer lies in its open source nature, which offers unprecedented flexibility in customizing its functions.

That's where the heart of our research fits in: the Kubernetes scheduler. The scheduler is responsible for deciding on which node in the system a particular "pod" (a group of one or more containers) should run. By modifying and optimizing this key component, it has the potential to directly affect the latency of the entire network, improving the end user experience.

In this thesis, we dive deep into the network latency issue and its intersection with pod scheduling in Kubernetes. We explore the opportunities and challenges presented by scheduler customization and how, through experimentation and innovation, we can get closer to an optimal solution to the latency dilemma.

1.1 Network latency: A Challenge for Efficiency in the Digital Age

In today's dynamic digital landscape, a seemingly small concept like network latency has assumed crucial importance. Latency, in simple terms, represents the time between a user requesting a service and the actual response from that service. While often measured in milliseconds, this delay can have huge implications, especially in critical applications such as medical, online gaming or real-time financial transactions.

Consider, for example, a telehealth app. A patient in Milan may need immediate medical advice from a doctor using a server-hosted app in New York. Given the geographical distance and the physical speed at which data can travel - the speed of light in optical fibers - one can already expect a base latency of around 50 milliseconds. But this is only the tip of the iceberg. Add to this the countless network devices such as routers, switches and gateways, each adding micro-delays. These delays accumulate, and during peak hours when networks are congested, the actual latency can be many times greater.

Several solutions have been proposed and implemented in an attempt to combat the latency problem:

- One of the most common strategies is to **horizontal scaling**, which involves adding more servers to spread the load and reduce congestion.
- Another popular solution is using **Content Delivery Networks (CDN)** which copy and distribute content to various servers in different parts of the world, trying to serve the user from the closest server. But, all of these solutions have their limitations.
- A last particularly interesting approach is **Edge Computing**, a paradigm in which data processing takes place as close as possible to the user. This

significantly reduces latency as data does not have to travel long distances. The idea is to move processing from centralized data centers to the edge of the network, closer to user devices.

In the context of this thesis, an even more innovative perspective has been adopted.While Edge Computing tries to bring data closer to the user through a static strategy, a dynamic method has developed.
The central idea is to **dynamically** bring the service or microservice closer to the user based on **real-time latency**, offering unprecedented responsiveness. This, in effect, creates "**Edge Computing on the fly**", ensuring that services are always positioned optimally in relation to the end user.

However, tackling such a complex challenge requires equally powerful and flexible tools. That's why the focus has turned towards **Kubernetes**. Originating as an open-source project, Kubernetes has quickly become the standard for container management and orchestration, and its **modularity** and **extensibility** allowed us to choose it among the various alternatives.

1.2 The Project Idea: A new scheduler from scratch

The **Kubernetes Scheduler** is a key component of the Kubernetes container orchestration system. Its primary function is to take the **Pods** that have been created and assign them to a node within the cluster. Customizing the scheduler in Kubernetes through the use of plugins is an extremely powerful feature, which allows you to adjust the scheduling mechanism according to specific metrics and rules, different from the default settings such as load balancing or round-robin algorithm. However, one of the unique challenges of our scenario is related to the network latency between the user and the cluster nodes. This metric is not available a priori during the scheduling phase, as it can only be calculated when an effective user interacts with the service exposed by a pod, and therefore with the node that hosts it.

The dynamic, post-hoc nature of this measurement imposes a unique requirement: the ability to "re-schedule" pods from one node to another based on actually collected network latency measurements. This task goes beyond simply adding a custom plugin to the existing Kubernetes scheduler. In fact, to implement such a reactive and dynamic system, it is necessary to **develop a scheduler from scratch**, which incorporates the logic to carry out not only initial scheduling operations based on a set of metrics, but also **de-scheduling and re-scheduling** operations in response to new information about network latency.

The new scheduler is designed to work in symbiosis with an **external application that specializes in measuring network latency** between users and

hosted services. This external application performs a critical function: it detects and logs latency every time a user accesses a particular service.
At set intervals, our custom scheduler requests this latency data from your application. Once received, this information is carefully stored in an optimized data structure specifically designed to facilitate quick and accurate scheduling decisions. The operational detail of the external application will be explored later in the text of the thesis.

This mechanism of capturing and using latency data allows our scheduler to make highly informed scheduling (and descheduling) decisions. It should be noted that the decision process is flexible and can be oriented towards different goals, depending on the needs of the use case.
Within this project, three distinct versions of our scheduler were developed and tested, each with its own goals and characteristics:

1. The **First Release** is focused on identifying the nodes that offer the lowest latencies, with the aim of improving the end user experience.

2. The **Second Version** extends the functionality of the first by integrating Liqo technology, which allows a multicluster interconnection between different Kubernetes clusters. This adds an extra layer of complexity and flexibility in pod placement.

3. The **Third Version** focuses on a more complex set of requirements. Instead of looking for the "best" solution, it aims to find a "good enough" solution in a much shorter time. To do this, it uses a combination of "hard" and "soft" constraints to drive the decision process. In addition, this version introduces the ability to manage multiple users, allowing for a fairer and more efficient distribution of services.

4. The **Last Version** is an improvement on the third, which allows multi-user functionality by implementing user-pod association so that a user can be directed to the node (or cluster) with the lowest latency.

Thus, our custom scheduler is not just a simple extension of the core scheduling functionality provided by Kubernetes; is a complete re-implementation, designed to address specific network latency issues in a cloud environment.

Chapter 2

Related Works

In this chapter, we explore seminal research and developments in Edge Computing and cloud resource offloading, illuminating various strategies and methodologies that aim to enhance user experience, minimize latency, and ensure efficient resource utilization. This exploration provides a comprehensive understanding of the efforts that have shaped the current landscape, offering context and foundational insights that support and validate the research presented in this thesis.

Our focus is on how Edge Computing has been strategically employed to bring computing services closer to the network's edge, thereby improving latency and user experience. In this context, the concept of resource offloading emerges as a dynamic solution for balancing resources to achieve optimized and adaptive services. Notably, works like Garcia Lopez et al. [1] and Shi et al. [2] have been instrumental in advancing our understanding of these paradigms.

The increasing adoption of container virtualization technology, particularly in developing microservice-based applications, has underscored the importance of orchestrators like Kubernetes. These systems simplify container application management but often face limitations due to their focus on infrastructure-level management policies. For instance, Kubernetes, the industry-standard orchestrator, excels in automating application deployment and operations but typically limits its optimization strategies to computing capacity. This approach, as discussed in works by Tomarchio et al. [3], often overlooks the crucial role of network latency.

As technology rapidly evolves, pushing the boundaries of digital transformation, there is a growing reliance on network-based services where latency becomes a decisive factor in service success. This evolution is paralleled by the increasing popularity of multi-regional cloud resources and the expansion of smaller data centers situated at the network periphery, as highlighted in the unified service model proposed by Baresi et al. [4].

Addressing these challenges, recent studies have proposed novel solutions for distributed environments. Orchestrators are now being designed to prioritize

resource optimization and incorporate auto-scaling features, as seen in Nguyen et al. [5], Tamiru et al. [6], and Osmani et al. [7]. These architectures create federated Kubernetes domains using Network Service Mesh tools, enhancing functionality and resource management.

However, a critical component in Kubernetes, the scheduler, originally geared for conventional cloud settings, lacks the dynamic adaptability needed in edge-cloud environments. This gap is addressed in the work by Carmona et al. [8], where offloading strategies in a three-tier network blend cloud and edge computing to minimize latency. Furthermore, Intel's Telemetry Aware Scheduling (TAS) [9] and the approach by Marchese et al. [10] introduce network-aware pod placement strategies using custom Kubernetes plugins. Our research extends these ideas, designing a multi-cluster scheduler that not only meets user-specified intents but also significantly focuses on user-perceived latency, effectively leveraging the cloud-edge continuum.

1. Optimal Offloading of Kubernetes Pods in Three-Tier Networks

In this paper [8] the authors explore the problem of optimal offloading of Kubernetes Pods in a three-tier architecture ranging from cloud to "**far-edge**" via edge computing . Unlike other studies, this research focuses on the use of Kubernetes as a container orchestrator in three-tier networks. The goal is to minimize the response time of the Pods, taking into account computing resources and *Quality of Service (QoS)*.

The authors introduce a three-level offloading decision algorithm, known as **TTODA**, which takes into account various factors such as available CPU resources and QoS expectations for each application. They use an optimization method based on the **Lagrange Dual Function** [11] to solve the minimization problem. The proposed model is tested through numerical simulations, demonstrating that TTODA outperforms traditional Kubernetes QoS models in terms of utility, average Pod response time, and resource utilization at the "far-edge."

This work is particularly significant because it not only offers a technical solution to a practical problem in the context of three-tier networks, but also provides a theoretical framework for optimizing offloading decisions. TTODA represents an excellent compromise between performance and implementation complexity, making it an ideal choice for latency-sensitive applications.

In the context of the system, the authors use a Kubernetes environment running on **OpenStack-managed cloud** [12] and edge nodes, as well as a non-virtualized "far-edge" server. A centralized **decision support module (DSSM)** makes offloading decisions based on data monitored by a **service level agreement (SLAM) module**. This architecture allows the system to dynamically adapt to changes in network conditions and application needs.

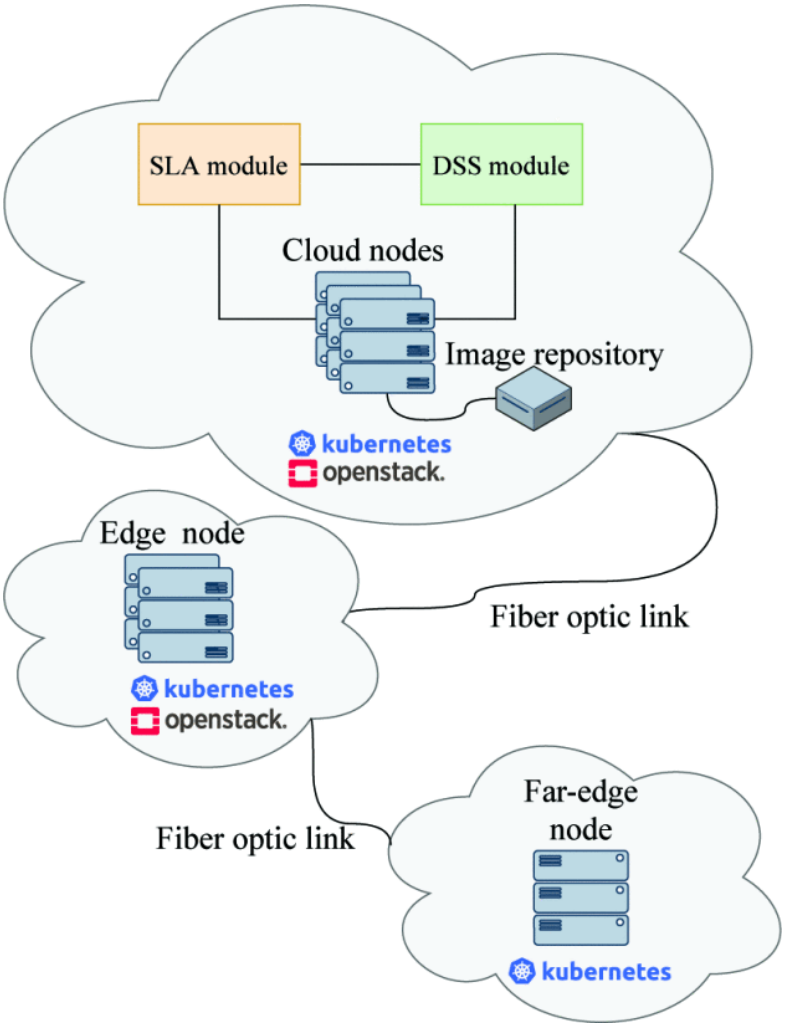


Figure 2.1: Architecture setup (Adapted from [8])

2. Latency-Aware Kubernetes Scheduling for Microservices Orchestration at the Edge

This paper [13] serves as a critical touchstone for my thesis, particularly for its innovative approach to latency-sensitive scheduling within a Kubernetes environment. The paper introduces a two-phase model:

- **Transient Phase**
- **Stable Phase**

During the **Transient Phase**, a "**sentinel**" pod replica is deployed across all nodes in the cluster to gather application-level latency metrics. This is executed through

an application termed "latency-meter," which acts as a client for the end-user and a server within the cluster.
In the **Stable Phase**, the collected latency data are used to assign the pod to the node exhibiting the lowest latency, thus creating an optimized environment for the end-user.

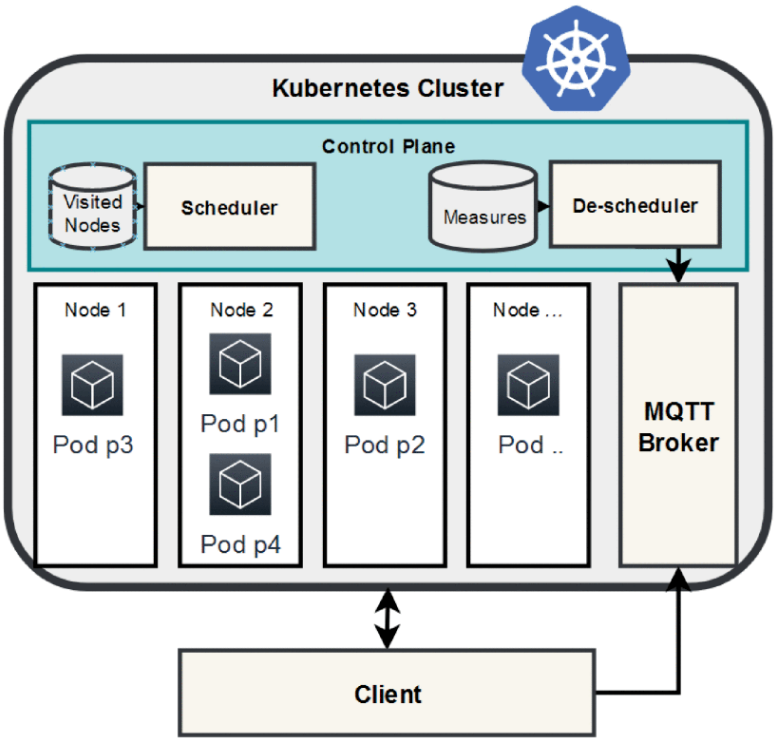


Figure 2.2: System Architecture (Adapted from [13])

The implementation (Fig. 2.2) utilizes a **de-scheduler** that is fed by an **MQTT broker**. This de-scheduler monitors the latency metrics and, if needed, removes service instances that are underperforming. This not only keeps the environment responsive but also makes it dynamic, adapting to changing network conditions in real-time.
The system also employs a centralized data structure called **LatencyMeasurements (LM)** (Fig. 2.3) that stores latency metrics for each application, allowing for more granular analysis and better-informed scheduling decisions.

However, the system in the paper has some limitations. For instance, it struggles to handle multi-user environments where latency requirements can differ significantly between various users. Also, the convergence time—the time needed to arrive at an optimized scheduling decision—can be affected by variables like the number of nodes in the cluster and the observation time for latency measurement.

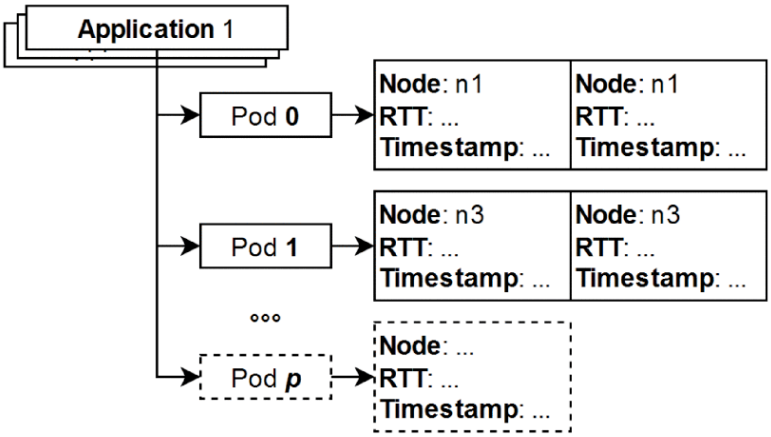


Figure 2.3: Latency Measurements data structure used to store latency information (Adapted from [13])

My thesis aims to address these challenges. I propose more efficient algorithms that **reduce the convergence time** and integrate advanced logic for handling **multi-user scenarios**, making the system more flexible. Additionally, I extend the system's scalability by exploring the interconnection of **multiple Kubernetes clusters**, which could open the door to more complex and widely applicable microservices orchestration solutions.

3. Network-Aware Pod Placement in Cloud-Edge Kubernetes Clusters

Another fascinating research in the field of container orchestration in Cloud-to-Edge environments is *"Network-Aware Pod Placement in Cloud-Edge Kubernetes Clusters"* [10]. This paper delves into the challenges of optimizing **Quality of Service (QoS)** and network performance in container orchestration across cloud and edge environments. This research distinguishes itself by introducing a **custom Kubernetes scheduling plugin** and a **descheduling operator** that factor in network metrics such as node-to-node latency and inter-microservice traffic. These components work to overcome the limitations of the default Kubernetes scheduler, which is geared towards optimizing CPU and memory resources rather than network conditions—something crucial for modern, network-intensive applications like IoT services.

The system architecture (Fig 2.4) is split into two main components:

- **Scheduler Scoring Plugin**
- **Custom Descheduler**

The **Scheduler Scoring Plugin** runs in the Kubernetes control plane, pulling

metrics from a Prometheus server to assess node suitability for Pod placement based on network latency and traffic metrics.
The **Custom Descheduler**, also operating in the control plane, continually re-evaluates these metrics to dynamically reallocate Pods to higher-scoring nodes, thereby optimizing application performance. Both components use scoring algorithms that weigh network latency inversely and consider the volume of traffic between services.

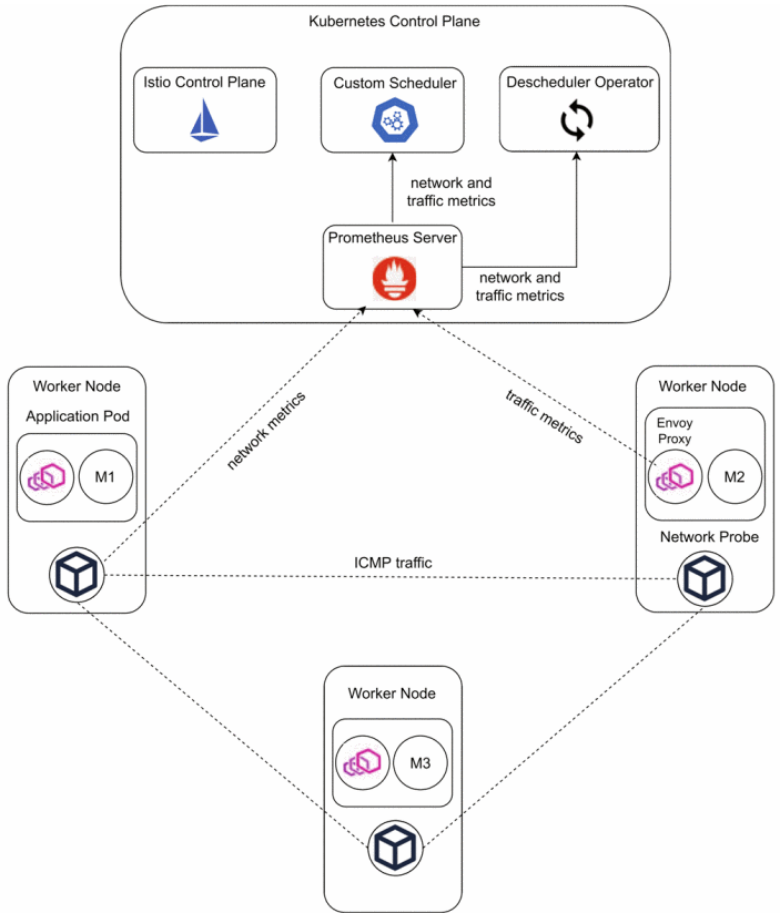


Figure 2.4: Overall architecture (Adapted from [10])

Despite its innovative approach, the paper acknowledges limitations, such as the **potential downtime** caused by Pod eviction. However, the authors argue that, in cloud-native environments where microservices are typically replicated, this would only result in minor QoS degradation. The system was evaluated using a Bookinfo sample application consisting of four microservices and a MongoDB server, deployed on a Kubernetes cluster. The tests revealed a substantial improvement in application response times compared to using the default Kubernetes scheduler,

thereby validating the approach.

My research aims to build on this by investigating how these scheduling and descheduling strategies could be fine-tuned for different types of network-bound applications, and how they would perform in larger, more complex environments. I'm particularly interested in exploring how this approach can be made even more dynamic and responsive to rapidly fluctuating network conditions.

4. Telemetry Aware Scheduling

Intel's Telemetry Aware Scheduling (TAS) [9] provides an intriguing take on Kubernetes scheduling by incorporating real-time telemetry data into both scheduling and descheduling decisions. The system comprises two main components: the **Telemetry Aware Scheduler Extender** and the **Telemetry Policy Controller**. The former interacts with the default Kubernetes scheduler to provide enhanced, metric-based scheduling decisions, guided by user-defined policies. These policies can hinge on a variety of platform metrics, including **Intel® RDT metrics, RAS, and others**, allowing for dynamic, condition-based pod placement. The Telemetry Policy Controller oversees these policies, ensuring they are consistently applied and taking action if a policy is violated, such as by labeling a node for descheduling.

Now, when we talk about integration with my work on "Network-Aware Pod Placement in Cloud-Edge Kubernetes Clusters," there are several compelling points of intersection. Both TAS and my work aim to address gaps in the default Kubernetes scheduler, which primarily focuses on CPU and memory utilization. While my research leans into optimizing for network conditions like latency and node-to-node traffic, TAS provides a more general framework that can be adapted to include these network-specific metrics. Imagine combining the custom network-aware scheduler plugin that I developed with the telemetry data available through TAS. This hybrid system could produce a much more comprehensive, dynamic, and adaptive scheduling algorithm. It could balance not only CPU and memory but also network latency and other real-time platform metrics, offering a more holistic approach to resource optimization in complex Cloud-to-Edge environments.

Moreover, the "descheduling" features present in both systems could work synergistically. My work involves a descheduling operator that reallocates pods based on network metrics, whereas TAS has a policy-based descheduling component. A unified descheduler could consider a richer set of criteria, allowing for more effective real-time adjustments and potentially boosting overall system performance and reliability. The joint capabilities would make the system incredibly robust, capable of auto-adjusting to a wider range of operational conditions and specific application requirements.

Chapter 3

Background

3.1 Kubernetes: The Open-Source Orchestrator for an Agile and Scalable Infrastructure

Kubernetes is an open-source platform for container management and orchestration. Its architecture is designed to simplify the automation, deployment, scalability and operation of containerized applications, such as those based on **Docker**, **LXD** or other similar technologies.
At the heart of Kubernetes is the concept of a "cluster," a set of interconnected physical or virtual machines that work together as a single entity. Each cluster is composed of a "master node", which manages the coordination logic, and various "worker nodes", which actually run the applications.

But what makes Kubernetes stand out in a sea of alternatives, including proprietary solutions and orchestrators like Docker Swarm or Apache Mesos?
First, its tremendous flexibility and scalability. Kubernetes can handle anywhere from a few to thousands of nodes, making it suitable for both small and large environments. Second, it offers an incredibly rich ecosystem of features, ranging from resource management, advanced workload scheduling, resiliency, self-healing, and more. Its open-source nature not only lowers barriers to entry, but also invites continuous innovation from the global community, which actively contributes to its development.

Another aspect that distinguishes Kubernetes is its modularity and extensibility. Through a system of plug-ins and well-documented APIs, users can customize or extend nearly every aspect of the platform, from networking modules to storage mechanisms to authentication and authorization methods. This allows organizations to build solutions tailored to their specific needs, without being trapped in a "technology cage" of restrictions and onerous licenses.

In the Kubernetes ecosystem, **Pods** are the smallest scheduling unit. They are

application containers that run within the nodes of a cluster. Pods embody the "schedule once, run anywhere" philosophy, meaning they are scheduled on a node and remain there until they are explicitly removed or the node itself fails.

Having introduced the concept of Pods, it's important to tie it into the architectural heart of Kubernetes, which is an extremely flexible and powerful API. The API is exposed through the **Kubernetes API Server**, a service that forms the entry point for all management and orchestration operations within the cluster. The API server is just one of several components that make up the Kubernetes **Control Plane**, the set of services that maintain and manage the global state of the cluster.

At the heart of the Control Plane is the **Scheduler**, a key component responsible for allocating Pods to cluster nodes. The Scheduler does not operate in isolation, but is tightly integrated with the API server to receive information on available resources, Pods to be scheduled and any constraints or policies set. After gathering this information, the Scheduler makes informed decisions about which node is best suited to run a given Pod.

The Kubernetes scheduler takes action when a pod needs to be assigned to one of the available nodes in the cluster. But at this point a fundamental question arises: who is responsible for the creation of the pod itself?

While it is technically possible to create a pod manually by directly interfacing with the Kubernetes server API and specifying the container image to use, this approach is rarely practical for a variety of reasons. First, if the node on which the pod was scheduled were to fail or be shut down, the pod would be terminated as a result. In scenarios where the pod hosts a critical service, this would lead to the unavailability of the service, potentially negatively impacting the entire infrastructure. Second, manually managing pods is susceptible to human error and is not scalable in a production environment with a large number of nodes and pods.

This is why the concept of **Deployment** enters the scene in Kubernetes, a solution that allows the replication of pods on multiple nodes, thus ensuring both the availability and resilience of the service.

3.1.1 Kubernetes Deployments

A **Deployment** represent one of the most powerful abstractions within the Kubernetes ecosystem. In simple terms, a Deployment is a model that describes how to run and manage instances of an application. Beyond the simple execution of a container, Deployments allow you to define a series of rules and configurations that govern aspects such as updating, scalability and application availability.
Why are they so useful? In a production environment, it is critical that services are always available and that there are little or no disruptions. Deployments help accomplish just that. A key aspect of Deployment is the concept of a "replica",

essentially a copy of a certain pod. When we define a Deployment, we also specify the number of replicas we want to keep active.
This approach has several advantages:

- **Scalability:** if the traffic to an application grows, the requests can be distributed across multiple instances (replicas) of the service, improving its responsiveness and load capacity.

- **Reliability:** should a pod encounter a problem and fail, the system automatically instantiates another to ensure that the number of replicas remains constant.

Here is a simple example of a YAML file for a Kubernetes Deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: my-app
spec:
    replicas: 3
selector:
    matchLabels:
        app: my-app
template:
    metadata:
        labels:
            app: my-app
    spec:
        containers:
        - name: my-container
          image: my-image:latest
```

Whenever a replica fails or is shut down, if its deployment is still active, a new pod is created and sent to the scheduler. *This is a fundamental aspect that will serve to understand the architecture of the project of this thesis.*

3.2 Kubernetes customizable Scheduler

Scheduling decisions are made following a set of algorithms and policies that can be extended or adapted for specific needs. This extensibility is possible thanks to the **modular architecture and plug-ins** of the Scheduler, which allows the addition of new scheduling logics or the modification of existing ones. This makes the Scheduler one of the most versatile and powerful components in the Kubernetes

control plane, capable of adapting to a wide range of operational and performance needs.

Kubernetes allows users to **customize the behavior of the kube-scheduler** by writing a configuration file and passing its path as a command line argument.

3.2.1 kube-scheduler framework

The scheduler procedure is divided into Stages, executed sequentially to calculate the pod placement. A scheduling Profile allows users to configure the different stages in the kube-scheduler. Each scheduler is exposed in an Extension Point:

1. **queueSort**: ordering function used to sort pending Pods;
2. **preFilter**: used to pre-process or check a Pod or the Cluster information before filtering.
3. **filter**: (equivalent to Predicate) filtering function based on hard constraints to discard unfitted nodes;
4. **postFilter**: used to marks the Pod schedulable or not;
5. **preScore**: used for doing pre-scoring work;
6. **score**: (equivalent to Priority function) function that provides a score to each node useful for sorting them;
7. **reserve**: used to notifies when resources have been reserved for a given Pod;
8. **permit**: used to prevent or delay the binding of a Pod;
9. **preBind**: perform any work required before a Pod is bound;
10. **bind**: bind a Pod to a Node;
11. **postBind**: perform any work required after a Pad has been bound;

The workflow is divided into two phases:

- **scheduling cycle**
- **binding cycle**

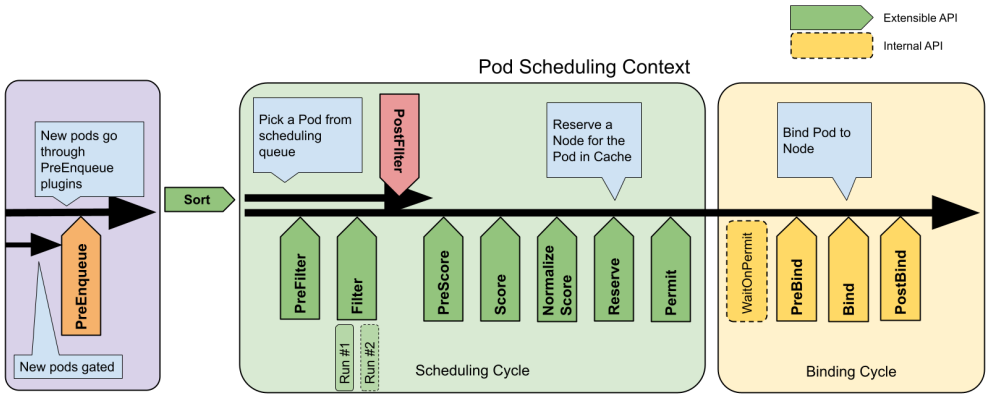


Figure 3.1: Framework workflow

```yaml
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
  - plugins:
      score:
        disabled:
        - name: PodTopologySpread
        enabled:
        - name: MyCustomPluginA
          weight: 2
        - name: MyCustomPluginB
          weight: 1
```

Figure 3.2: Scheduling Plugin

3.2.2 Scheduling plugins

All the extension points are implemented by Plugins that provide scheduling behaviors. For each extension point, it is possible to disable specific default plugins or enable custom plugin (see **Figure 1.2**).

Below (**Table 1.1**) is the list of the most important enabled default plugins that implement one or more extension points. For the full extension points list, consult the Kubernetes online documentation.

Plugin Name	Description	Extension Points
ImageLocality	Favors nodes that already have the container images that the pod runs.	**score**
NodeName	Checks if a Pod spec node name matches the current node.	**filter**
NodeAffinity	Favors nodes that already have the container images that the pod runs.	**score**
NodeResorcesFit	Check if the node has all the resources that the Pod is requesting.	**preFilter, filter, score**
NodeResourcesBalancedAllocation	Favors nodes that would obtain a more balanced resource usage if the Pod is scheduled there.	**score**
NodeVolumeLimits	Checks that CSI volume limits can be satisfied for the node.	**filter**
InterPodAffinity	Implements inter-Pod affinity and anti-affinity (they allows to constrain which nodes your Pods can be scheduled on based on the labels of Pods already running on that node, instead of the Node labels).	**preFilter, filter, preScore, score**
PrioritySort	Provides the default priority based sorting.	**queueSort**
DefaultBinder	Provides the default binding mechanism.	**bind**
DefaultPreemption	Provides the default preemption mechanism.	**postFilter**

Table 3.1: List of the most important scheduling default plugins and their extension-points

3.3 Liqo: The Next-Generation Kubernetes Extension

Liqo, an acronym for "Liquid Computing," is an innovative open-source project engineered to expand the capabilities of Kubernetes clusters. In essence, it allows Kubernetes clusters to share resources seamlessly, thereby creating a collaborative, multi-cloud, and edge computing environment [14]. This leap in technology facilitates workload distribution across various clusters, optimizes the use of resources, and brings unprecedented scalability and flexibility to cloud-native applications.

3.3.1 The Imperative of Adopting Liqo

In the modern technological landscape, cloud-native applications are becoming increasingly complex, often requiring resources that extend beyond the confines of individual Kubernetes clusters. Traditional Kubernetes architectures, while powerful, operate in isolated silos which can lead to resource underutilization and operational inefficiencies. These constraints can be particularly detrimental for applications that require high availability, low latency, and seamless scaling across

different cloud environments.

Liqo comes into play as a disruptive technology, aimed at shattering these limitations. It enables Kubernetes clusters to dynamically share resources, facilitating the creation of a federated, fluid architecture that can stretch across multiple cloud providers, data centers, and edge locations. By doing so, Liqo allows for a more agile and responsive deployment model, capable of **automatically adapting to changing workloads and user demands**.

Moreover, the adoption of Liqo is more than a tactical move; it's a strategic imperative for organizations striving to maximize their operational efficiency. Its advanced resource management features allow businesses to optimize costs effectively, as workloads can be intelligently placed where resources are most abundant or least expensive. Additionally, its decentralized approach makes it particularly well-suited for edge computing scenarios, where distributing computation closer to data sources can significantly reduce latencies and improve performance.

Given these compelling advantages, integrating Liqo into your Kubernetes strategy becomes an essential step for creating a resilient, flexible, and cost-effective cloud-native ecosystem. It offers a significant competitive edge, preparing enterprises for the demands of next-generation cloud computing paradigms, including microservices architectures, serverless computing, and IoT deployments.

3.3.2 Liqo Features

Liqo is endowed with a multitude of features designed to bring efficiency, scalability, and flexibility to your Kubernetes clusters. A few standout features include:

- **Virtual Kubelet Implementation**: Liqo employs a virtual Kubelet to act as a proxy for external clusters, enabling effortless scheduling of pods and services without demanding changes in your existing applications.

- **Adaptive Resource Management**: One of the compelling features of Liqo is its ability to dynamically adapt the size of clusters depending on workloads. This not only ensures optimal resource utilization but also leads to significant cost reductions.

- **Transparent Cross-Cluster Networking**: The framework ensures seamless networking among pods residing in different clusters, making them virtually part of the same local network.

- **Policy-Driven Resource Allocation**: Liqo allows for the creation of intricate resource-sharing policies, offering granular control over how resources are dispersed and utilized across multiple clusters.

- **Instant Integration Capabilities**: Liqo is designed for instant integration, promising immediate operational advantages without necessitating sweeping changes to existing infrastructures.

3.3.3 Peering in Liqo

Peering in Liqo is a dynamic and multifaceted process that is fundamental to the efficiency of my project. Liqo offers two main types of peering that differ not only in terms of automation but also in the underlying network pathways used for interaction. These are:

- **In-Band Peering:** This automated form of peering is based on predefined policies and conditions. It operates inside a VPN tunnel, ensuring that all communications between the paired clusters are encrypted and isolated from external threats. This secure form of peering is especially useful in my project for scenarios requiring stringent data protection measures.

- **Out-of-Band Peering:** In contrast, out-of-band peering provides more control by allowing manual configuration and occurs outside a VPN tunnel. Although this form is more straightforward to set up, it could expose the clusters to additional security risks, especially when crossing untrusted networks. This form of peering is ideal for environments in my project where speed and low-latency are prioritized over additional security measures.

By utilizing both these peering methods, my project gains the advantages of both secure, automated interactions and the flexibility to manually configure connections when needed. This dual approach makes the system highly adaptable to a range of operational scenarios, from those requiring high security to those needing rapid data exchange.

3.3.4 Offloading Capabilities in Liqo

Offloading in Liqo is an essential feature that allows resource and workload sharing among federated clusters. This capability is central to the flexibility and scalability of my project. Liqo provides the following types of offloading:

- **Automatic Offloading:** In this mode, Liqo automatically balances the load across multiple clusters without manual intervention. It considers resource availability and constraints in each cluster to make real-time offloading decisions. This is particularly useful in my project for achieving high utilization rates without the burden of manual configuration.

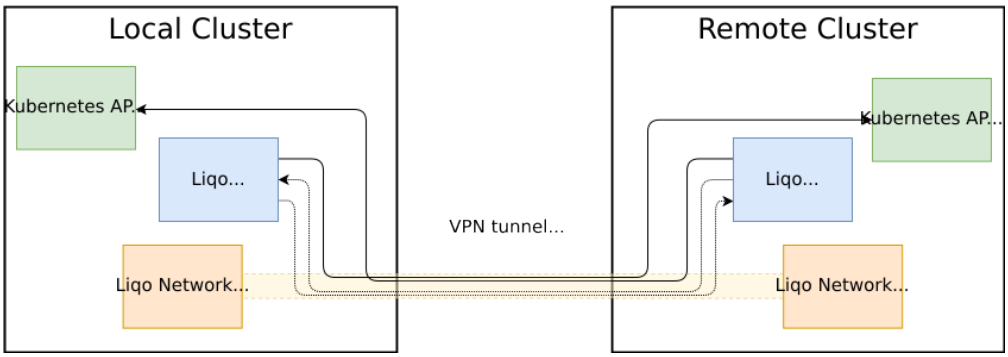


Figure 3.3: In-Band Peering

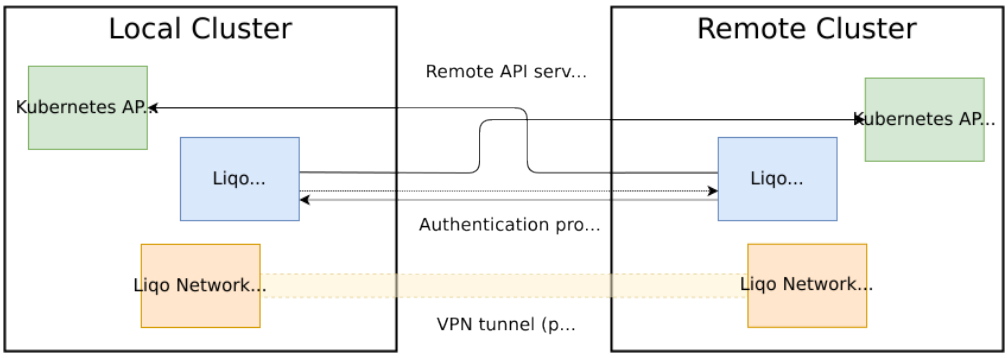


Figure 3.4: Out-Of-Band Peering

- **Manual Offloading:** For scenarios that require more fine-grained control over the placement of workloads, Liqo offers manual offloading. Here, specific rules and policies can be set up to guide how, when, and where the offloading occurs. This type of offloading is employed in my project for tasks that need specialized resources or for compliance with data locality regulations.

- **Namespace Offloading:** Liqo allows for the offloading at the granularity of a Kubernetes namespace, enabling more natural abstractions and easier management. This feature is beneficial for segmenting different aspects or modules of my project, allowing each one to be managed and scaled independently.

By using these offloading methods, my project benefits from a versatile and dynamic system for managing resources. It can handle everything from automated load balancing for general tasks to highly customized manual offloading for specialized or sensitive operations. This flexibility is crucial for adapting to the diverse and often unpredictable resource requirements that modern cloud-native applications

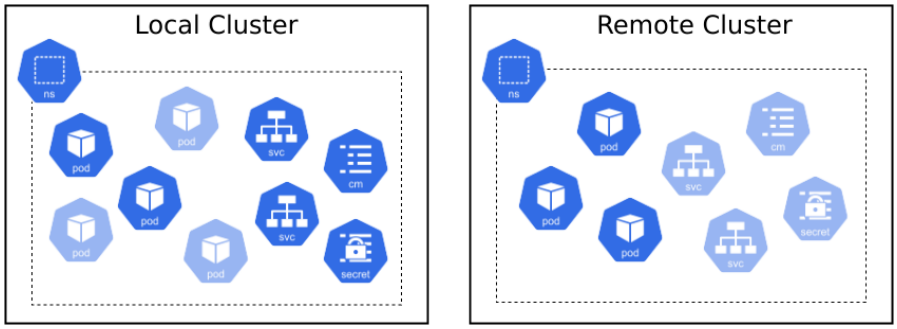


Figure 3.5: Namespace Extension

face.

Chapter 4

System Model

4.1 Architecture

The architecture of my project serves as the blueprint for realizing a latency-optimized, Kubernetes-based service deployment. In a standard Kubernetes cluster, scheduling decisions are generally unaware of network latency between the user and the service nodes. My architecture addresses this gap by incorporating two specialized Golang applications:

- **Latency Meter**
- **Custom Latency-Aware Scheduler**

These two integral components collaborate to provide both real-time latency measurements and latency-optimized scheduling of pods across cluster nodes.

4.1.1 The Latency Meter Application

The **Latency Meter** serves as an indispensable component in the system's architecture. It is uniquely configured as a **second container** co-located within each application pod. This in-pod deployment strategy not only simplifies internal network configurations but also mitigates potential permission issues, offering a streamlined and robust solution. As users initiate requests to access services within a pod, the Latency Meter acts as an **initial point of contact** (like a proxy), intercepting these requests before they reach the main application container.

The strategic placement of the Latency Meter allows it to measure the network latency between the user and the node hosting the pod in real-time. Since every user request passes through the Latency Meter first, it can acquire a comprehensive set of latency measurements. These latency metrics are stored in volatile memory

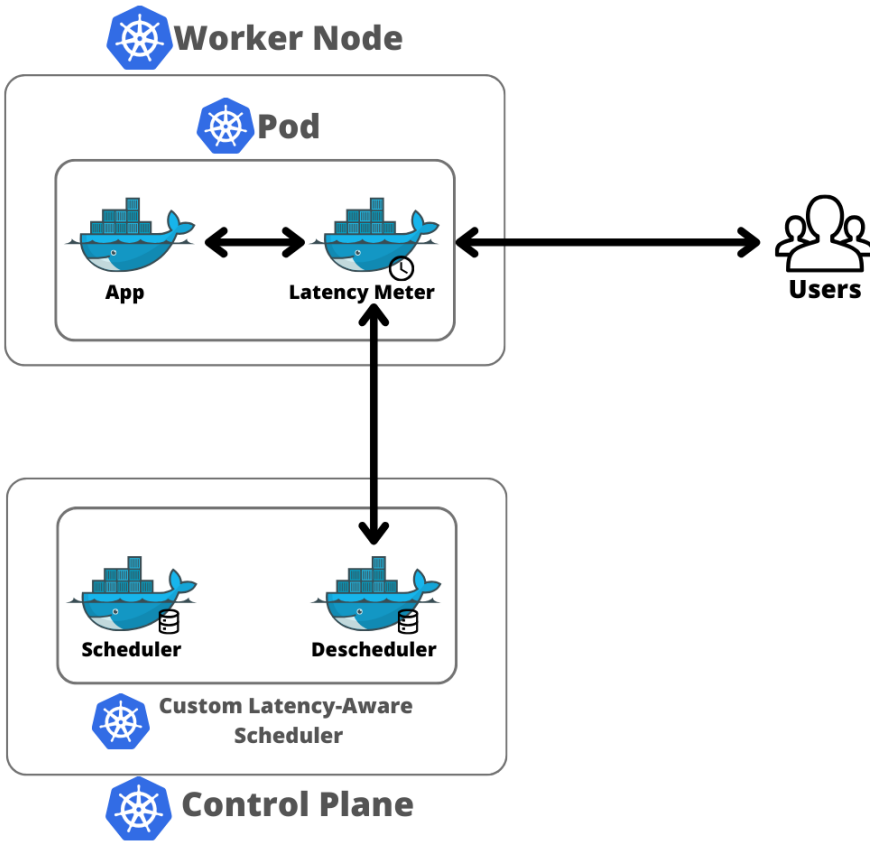


Figure 4.1: System Model

to ensure rapid data read and write access, contributing to the system's real-time adaptability and performance.

The importance of the Latency Meter extends beyond mere data collection; it also plays a pivotal role in the system's decision-making process. The Custom Latency-Aware Scheduler constantly queries the Latency Meter to obtain the latest latency metrics. These metrics serve as critical input parameters for the scheduler, helping it make informed decisions about pod placements or removals within the cluster. This interaction is essential for optimizing the overall latency profile of the system, making the Latency Meter an integral part of the architecture.

4.1.2 The Custom Latency-Aware Scheduler

Taking over from Kubernetes' default scheduler, my Custom Latency-Aware Scheduler resides in the control plane and serves as the backbone for latency-optimized

pod scheduling. This scheduler is comprised of three major entities, each encapsulated in its own Golang file:

1. **Scheduler**: This component is akin to Kubernetes' default scheduler but with a latency-aware twist. It prioritizes nodes based on the number of similar application pods they host. The goal is to diversify latency measurements across multiple nodes, thereby ensuring a comprehensive latency profile.

2. **Descheduler**: This is the unique innovation in my architecture. The Descheduler actively seeks updated latency metrics from the Latency Meter and adjusts the cluster state accordingly. Pods may be descheduled and re-scheduled to optimize for latency, forming a continuous loop with the Scheduler.

3. **LatencyMeasurements (LM)**: Serving as the shared data structure between the Scheduler and Descheduler, LM stores latency data in a volatile format. The decision to keep the data in-memory was deliberate, optimizing for speed at the expense of data persistence.

System Dynamics: Transitional and Steady-State Phases

The operation between the Descheduler and Scheduler is cyclical and represents a ***transitional phase***. This is a state where the pods are continually being moved around nodes until their placement satisfies the Descheduler's latency criteria. Upon reaching this equilibrium, the system moves into a "steady-state" or ***regime***, providing an optimized user experience.

Data Storage Considerations and Future Extensions

The choice to keep the LatencyMeasurements (LM) data structure in volatile memory was made to prioritize speed over data persistence. While data loss is not critical, it would necessitate a longer *transitional phase* for re-optimization. Future works may explore distributing this latency data across nodes and possibly using more persistent data storage solutions.

4.2 Latency Meter: In-Depth Analysis

The Latency Meter is a specialized application implemented in Golang, designed to calculate network latency between the user and the Kubernetes pod where it resides. Deployed as a secondary container within each pod, the Latency Meter acts as an intermediary layer between the user and the main application, intercepting and measuring the time delay of each request.

4.2.1 Design and Algorithm

The primary components of the Latency Meter include a RESTful API built using the Gorilla Mux router and the Kubernetes Client-Go library, which interfaces with the Kubernetes API. Upon launch, the Latency Meter starts by obtaining its pod information and initiating a new HTTP server listening on port 8080.

The application defines two main routes:

- **/ (root)**: To measure latency.
- **/measurements**: To return and reset collected latency data.

Latency Measurement Methodology

The algorithm for measuring latency between the client and the server is outlined below (Alg. 1).

Algorithm 1 Latency Measurement Algorithm in Kubernetes Deployment

```
1: procedure MeasureLatency(HTTP_Req, pod, latencyMeasurements)
2:     ▷ HTTP_Req is the incoming HTTP request
3:     ▷ pod contains information about the current Kubernetes pod
4:     ▷ latencyMeasurements is the map storing latency data
5:     ▷ Initialization
6:     Extract userID from HTTP_Req.URL.Query().Get("id")
7:     Extract clientTimestamp
           from HTTP_Req.Header.Get("X-Timestamp")
8:     ▷ Execution
9:     serverTimestamp ← Current time in milliseconds
10:    latency ← serverTimestamp − clientTimestamp
11:    ▷ Store the measured latency
12:    latencyMeasurements.AddLatency(userID, latency)
13: end procedure
```

When a user request arrives, the application:

1. Extracts the user's ID and timestamp from the request.
2. Calculates the current server time.
3. Computes the latency by taking the difference between the server and client timestamps.
4. Stores this latency measurement in a map, keyed by the user ID.

The latency measurements are stored in volatile memory for swift read-write access. The data can be retrieved via the **/measurements** route in JSON format.

4.2.2 Latency Measurement Challenges and Approaches

Calculating accurate latency between a server and a client is inherently challenging due to several factors, including but not limited to, security constraints like CORS (Cross-Origin Resource Sharing), network congestion, and variable network conditions affecting different layers of the OSI model.

The Adopted Approach

In this design, latency is measured by utilizing the time difference between the client's and server's internal clocks. The client attaches its timestamp in the request header, which the server uses to calculate the round-trip time.

Limitations

- **Clock Skew**: The internal clocks on the client and server may not be perfectly synchronized, affecting accuracy.
- **Network Conditions**: This method does not account for changes in network conditions that may occur during the round-trip.
- **Header Overhead**: Adding a timestamp to the header increases the size of the HTTP request, albeit minimally.
- **Server Processing Time**: The time taken by the server to process the request can introduce additional latency, which is hard to separate from network latency.

Alternatives

ICMP Ping This method involves sending an ICMP echo request to the target server and measuring the time it takes to receive a reply. It is one of the most accurate ways to measure latency.

- **Limitations**: This method may be blocked by firewalls, or the server may be configured to give low priority to ICMP requests, making the measurement less reliable.

JavaScript-based solutions JavaScript running on the client's browser can also be used to measure latency to the server by timing how long it takes for an HTTP request to complete.

- **Limitations**: Browser security features like CORS may limit the destinations to which you can send requests, thereby limiting this approach.

WebRTC protocols WebRTC enables real-time communication and has built-in functionality for gathering network metrics, including latency.

- **Limitations**: WebRTC may require additional configurations and permissions, which could complicate the deployment. Also, it could be an overkill for simple applications that do not require real-time communication features.

Timestamp-Based Approaches

HTTP-Level Timestamps This method involves embedding a timestamp in the HTTP header of the request.

- **Pros**:
  - Simplicity: Easier to implement at the application layer.
  - Compatibility: Widely supported in various systems and frameworks.
  - Less Resource Intensive: Avoids the overhead of capturing and analyzing every packet.
  - No Special Privileges Required: Operates at the application level, avoiding the need for elevated privileges.
- **Cons**:
  - Application Overhead: Additional HTTP header increases overhead.
  - Protocol Dependency: Tied to HTTP protocol; other protocols would require similar logic.
  - User Transparency: May require manual user input to insert timestamps.

IP-Level Timestamps This approach involves embedding timestamps at the IP packet level.

- **Pros**:
  - Greater Precision: Operating at the packet level may offer more accurate measurements.

  - Protocol Independence: Not bound to HTTP, offering potential flexibility for future protocol support.
  - User Transparency: Operates transparently if enabled and supported.

- **Cons**:
  - Privileged Access: Capturing raw packets might require elevated privileges.
  - Added Complexity: Low-level packet filtering and analysis can introduce complexity and potential performance issues.
  - Interoperability: Not all packets will have the timestamp option, and intermediate devices may alter or remove it.
  - Resource Intensive: Real-time packet analysis may impact system resources.

4.2.3 Deployment

The application is containerized using Docker. In the Kubernetes deployment configuration, this Docker image is specified to run as a secondary container within each application pod. This co-location ensures that every interaction with a pod undergoes latency measurement by the Latency Meter.

- **Limitations of Deployment Approach**:
  - **Resource Overhead**: Running an additional container in every pod increases the resource utilization.
  - **Co-location Bias**: Since the latency meter runs in the same pod, the measurements may not accurately reflect the experience from other points in the network.

4.3 Custom Latency-Aware Scheduler

The custom latency-aware scheduler is a unique, feature-rich system explicitly designed to optimize pod scheduling based on network latency. Over the course of its development, this scheduler has evolved through three significant versions, each adding new features or optimizations.

4.3.1 Version 1 (LAIS-0): Minimum Latency

Version 1 serves as the foundational iteration of this custom scheduler. It introduces three core components: the **Scheduler**, the **De-scheduler**, and the

in-memory data structure **LatencyMeasurements (LM)**. In this version, the focus is on spreading the pods across multiple nodes and then adaptively rescheduling them based on latency metrics.

Scheduler (V1)

Objectives and Decision Making The primary goal of the **Scheduler** is two-fold: firstly, it aims to distribute the application's pods across multiple nodes to **facilitate the collection of a wide range of latency measurements**. Every time a new pod needs to be scheduled inside a certain cluster, a load-balancing approach is adopted. To distribute pods across a variety of nodes, the scheduler creates a **priority list**. A node's priority is based on the number of pod replicas of the same application it hosts: the *fewer the pods, the higher the priority.* If nodes have the same priority, the one with less resource usage (like CPU and RAM) is chosen. If multiple nodes still stand equal, the selection becomes *random.*

Algorithmic Walkthrough The scheduling algorithm primarily operates by considering the following steps:

Algorithm 2 Scheduler Algorithm in V1

1: Retrieve list of available nodes and current pod count for each node.
2: Access the in-memory LatencyMeasurements (LM) structure for latency data.
3: **for** each unscheduled pod **do**
4: Prioritize nodes where no pods of the same application have been scheduled.
5: In case of tie, opt for the node with better CPU and memory performance.
6: In case of another tie, opt for a random selection.
7: Update LM to reflect this new scheduling decision.
8: Bind the pod to the chosen node.
9: **end for**
10: **Last Cycle:** Re-evaluate node selection to account for any changes in latency measurements or node availability.

Significance of the Last Cycle The **Last Cycle** in the Scheduler serves as a mechanism to seek the **global minimum** of network latency after a series of local optimization steps executed by the Descheduler. While the Descheduler focuses on local improvements by evicting pods with the highest latencies, this can sometimes lead to suboptimal global configurations. The Last Cycle is invoked after adequate latency measurements have been obtained for all pods across all nodes. It reevaluates the entire scheduling landscape to ensure that the pods are scheduled on nodes that offer the lowest global latency.

De-scheduler (V1)

Objectives and Decision Making The **De-scheduler** in V1 aims to act as a dynamic correction mechanism. It primarily focuses on:

- Gathering latency metrics from the external latency-meter application.
- Updating the LatencyMeasurements (LM) in-memory data structure with these new metrics.
- Making data-driven decisions to deschedule pods from nodes exhibiting high latency, given that sufficient data entries are available in LM for robust decision-making.

Algorithmic Walkthrough The de-scheduling algorithm operates by considering the following steps:

Algorithm 3 Descheduler Algorithm in V1

```
Initialize LatencyMeasurements (LM) if available.
while true do
    Contact the external latency-meter to acquire latest latency metrics.
    Update LM with these new measurements.
    if LM has sufficient number of measurements for decision-making then
        Identify the node with the highest latency for the specific application.
        Deschedule the pod(s) from this node.
        Delete corresponding latency entries for this node in LM.
    end if
    Trigger the Scheduler for re-scheduling the pod, which in turn updates the LM.
end while
```

Seamless Integration with Scheduler The De-scheduler not only serves to evict pods but also triggers the Scheduler as part of its workflow. This interaction forms a **closed-loop system** that aims to optimize user-to-node latency continually.

LatencyMeasurements (LM) in V1

The **LatencyMeasurements (LM)** structure is a central piece that serves both the Scheduler and De-scheduler. It is an in-memory data structure optimized for quick read and write operations. Each application entry within LM contains:

- a list of **pods**;

- the **nodes** where the pods are scheduled on;
- the **latency metrics**;
- a **timestamp** for each measurement.

Interoperability Between Scheduler and Descheduler

The Scheduler and De-scheduler operate in a tightly-coupled manner, coordinated through a simple **Go channel** for thread synchronization. The Scheduler takes on the responsibility of determining whether all nodes have been visited and measured for latency. After its **Last Cycle**, where it seeks to find the global minimum latency, the Scheduler can halt the De-scheduler's operations to prevent unnecessary rescheduling. This stopping mechanism is crucial for **maintaining a stable state once an optimal or near-optimal configuration is reached**. However, acknowledging the dynamic nature of user locations and network conditions, the Scheduler is configured to reactivate the De-scheduler after a custom-defined time interval. This timer-based reactivation offers the system the flexibility to adapt to **geographical movements** by the user and ensures that the pod placements remain optimized over time.

Limitations

The current design of the custom latency-aware scheduler comes with a few noteworthy limitations:

- **Convergence Time too high**
- **Multi User not supported**

Firstly, the time required to reach a **stable state** or "regime" is considerably high. This is because the system needs to visit all nodes, find local minima at each stage by removing the highest latency, and then perform a Last Cycle to find the global minimum latency. This iterative process, while effective, is **time-consuming**.
Secondly, **the system is not designed to handle multi-user scenarios** efficiently. Different users may have different latency measurements, leading to different optimal solutions. The current version does not support the reconciliation of these conflicting requirements into a unified optimal solution.

4.3.2 Version 2: Liqo Integration

The advancement of the latency-aware scheduler is significantly underscored by its synergistic integration with *Liqo*, a framework enabling the interlinking of disparate clusters via sophisticated peering mechanisms. This integration leverages

Liqo's capability to form advanced multi-cluster topologies, beneficial for offloading scenarios, where **one master cluster governs multiple slave clusters**.

The architecture of our scheduler aptly aligns with Liqo's functionalities. The utilization of Liqo in creating a multi-cluster environment is underlined by a key assumption: **that nodes, typically geographically co-located within the same cluster, experience a uniform network latency with users**. This assumption is crucial for the seamless integration of the scheduler and plays a pivotal role in ensuring operational coherence and maintaining system responsiveness across varied topological scenarios.

Seamless Integration

In such a multi-cluster context, integrating our latency-aware scheduler is remarkably straightforward. By setting our scheduler as the principal scheduler of the master cluster, the peering process via Liqo identifies the entire remote clusters as **virtual nodes**. The scheduler, thus, perceives and treats these virtual nodes akin to other nodes in the cluster.

During a deployment, the pods and their replicas, under the guidance of our scheduler, are disseminated across multiple possible virtual nodes, corresponding to remote clusters. Once a pod is allocated to a virtual node, it is further scheduled by the **scheduler of the associated cluster**, which could even be the default one, aligning with the premise that nodes within the same cluster maintain comparable network latency with users.

Enhanced Accessibility and Decision-making

Remarkably, the De-scheduler, located in the control plane node of the master cluster, can access all latency-meters across various pods, even those situated on remote clusters, leveraging Liqo's **Network Fabric**. This feature ensures seamless communication between all pods within a peered cluster, with or without **NAT translation** [14]. Consequently, the De-scheduler can efficiently acquire latency measurements and execute informed de-scheduling decisions, enhancing the system's adaptability and responsiveness.

This seamless integration with Liqo required **minimal alterations** to the existing codebase, primarily pertaining to the **namespace environment adjustments** to incorporate Liqo. The intuitive and natural incorporation of Liqo accentuates the scheduler's **versatility** and **scalability**, adapting effortlessly to diverse cluster environments and operational demands.

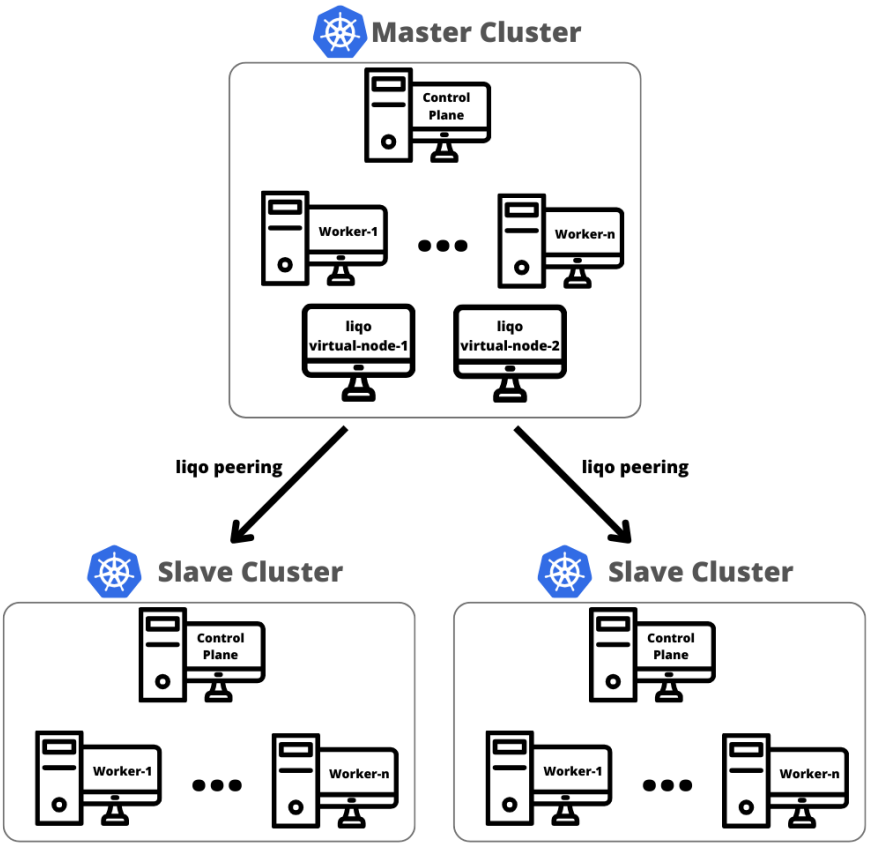


Figure 4.2: Multi-cluster Topology

4.3.3 Version 3 (LAIS): Latency constraints

The integration of Liqo technology is foundational to this version of the Latency-Aware Scheduler, engendering a fundamental assumption that nodes located within the same cluster will exhibit identical network latency with respect to the user due to their geographical proximity. However, despite the amalgamation of Liqo, the enduring **issue of convergence** in the preceding version lingers prominently. The persistent convergence issue implies that the scheduler, even when it reaches a steady state, continually struggles to maintain optimal latency configurations due to the dynamic nature of network conditions and user mobility. This predicament instigates a critical reassessment and evolution of the scheduler's underlying logic and operational mechanisms.

In addressing the convergence problem, the evolved scheduler forgoes the pursuit of absolute optimal configurations characterized by minimal latency. Instead, it adopts a more pragmatic approach, aspiring to achieve configurations that are 'good-enough'—a relative term indicating a balance between service requirements

and network conditions. This approach is materialized through the imposition of **latency constraints**, meticulously aligned with the varied requirements of different services.

Diving deeper into service requirements, it is evident that **different services have divergent needs concerning latency**. Services like ***Cloud Gaming*** and ***Live Streaming*** are **latency-sensitive**, necessitating low latency to maintain user experience and service quality. These services, owing to their real-time interaction and high-frequency data transmission, are intolerant to delays, making low latency a non-negotiable requisite.

In contrast, services such as ***Cloud Storage*** exhibit a different set of requirements. While they do require a stable and reliable network connection, they are not stringently bound by low-latency needs. For these services, the paramount concern is the **reliability** of the network, ensuring that there is no loss of packets during data upload or download processes, thus maintaining data integrity and service dependability.

Given this divergence in service needs, the pursuit of minimal latency becomes a redundant and sometimes counterproductive endeavor. The imposition of latency constraints in this version of the scheduler is not a one-size-fits-all solution; rather, it is a nuanced and tailored strategy, seeking to harmonize service performance with user experience.

These tailored latency constraints allow the scheduler to focus on achieving and maintaining configurations that are attuned to the specific needs of the service, avoiding **unnecessary reallocations** and adjustments that may arise from a blind pursuit of minimal latency. This nuanced approach ensures that the scheduler is more adaptable and efficient, capable of handling a diverse range of services with varying latency sensitivities and requirements.

Operational Workflow

The operational logic of this version is defined by two types of latency constraints, *HARD MAX LATENCY* and *SOFT MAX LATENCY*, optionally coexisting based on user preferences during deployment. These constraints guide the Scheduler and Descheduler in allocating and deallocating pods, with an added layer of dynamic adaptability to account for potential user mobility, ensuring perpetual applicability.

Implementation Details

The intricate implementation of the latency-aware scheduler involves two crucial components:

- **Scheduler:** The Scheduler retains its traditional functionality, focusing on

the optimal allocation of pods and emphasizing resource efficiency and uniform distribution across diverse nodes and clusters. In this refined version, it incorporates and considers the latency constraints, both hard and soft, storing and sharing them with the Descheduler to ensure a harmonious and responsive interaction between scheduling and descheduling processes, allowing for adjustments in real-time in response to the varying network conditions and service demands.

- **Descheduler:** The Descheduler plays a crucial role, regularly updating latency measurements and evaluating them against the set constraints to ascertain potential candidates for descheduling. It leverages three essential data structures: *invalidNodes*, *hardValidNodes*, and *softValidNodes* to categorize and manage nodes based on the latency measurements and constraints adherence, offering a granular view of the nodes' current states and facilitating informed and timely descheduling decisions.

Operational Cases The Descheduler's operational logic is delineated into three distinct cases, addressing different constraint scenarios:

1. **Case 1: Hard Constraint Only:** Nodes are strictly evaluated against *hardMaxLatency*. Any node exceeding this limit is subject to immediate descheduling, impacting all the associated pods relevant to the application. This stringent approach prioritizes the enforcement of hard constraints to ensure the maintenance of optimal service quality and user experience.

2. **Case 2: Soft Constraint Only:** Here, nodes are assessed considering *softMaxLatency*, determining their classification into either *softValidNodes* or *hardValidNodes*. This scenario emphasizes a softer, more adaptive enforcement allowing a flexible management of nodes. A conclusive descheduling evaluation occurs at the end of each cycle, focusing on achieving a balance between adaptability and operational efficiency.

3. **Case 3: Both Constraints:** In this comprehensive scenario, nodes are scrutinized against both *hardMaxLatency* and *softMaxLatency*, influencing immediate and end-of-cycle descheduling decisions. The dual assessment aims at achieving a balanced approach between strict enforcement and adaptive flexibility, based on the proportion of *softValidNodes*.

End-of-Cycle Evaluation The conclusion of each cycle, particularly significant in Cases 2 and 3, brings forth a pivotal final evaluation. This evaluation is contingent upon the comprehensive measurement of all nodes, i.e., when the sum of *invalidNodes*, *softValidNodes*, and *hardValidNodes* equals the total number of

nodes ($N_{\text{invalid}} + N_{\text{soft}} + N_{\text{hard}} = N_{\text{tot}}$). Only if the count of *Soft Valid Nodes* (N_{soft}) added to the number of *Hard Valid Nodes* (N_{hard}) is greater than the 50% of the total nodes (N_{soft}) —a parameter subject to discussion and review— the final descheduling evaluation is activated. This ($N_{\text{soft}} + N_{\text{hard}} > 50\% N_{\text{tot}}$) is called the **Soft Condition**.

In this final evaluation phase, all nodes within the worst *hardValidNode* are subjected to descheduling. This approach is aimed at establishing a **compromise between the quantity of valid nodes and the adherence to the Soft Constraint**, aligning acceptable latency levels with network reliability. Consequently, this mechanism ensures an optimal balance, catering to diverse service requirements and adapting dynamically to various network conditions, thereby reinforcing the robustness and resilience of the deployed services.

Algorithm 4 Descheduler Algorithm

```
Initialize:
    LM, invalidNodes, hardValidNodes, softValidNodes
    hardLatThresholds, softLatThresholds
Determine: N_tot total number of nodes.
while true do
    Sleep: Predefined interval.
    Acquire & Update: New latency measurements in LM.
    for each userID, appName, nodesMeasurements in LM do
        Categorize Nodes: Based on latency and thresholds.
        Deschedule: Pods on invalidNodes.
        if Soft Constraint exists then
            Evaluate: End-of-cycle Soft Condition.
            if Soft Condition is valid then
                Deschedule: Worst hardValidNodes.
                Update: Corresponding data structures.
            end if
        end if
    end for
end while
```

Continuous Adaptation A prominent characteristic of the current iteration is its continuous adaptation capability, allowing each node to transition between different states in each cycle, depending on the user's physical movement or the persistence of its "valid" or "invalid" status. This trait negates the need to terminate the scheduler-descheduler mechanism frequently. In contrast, the Version 2 necessitated the termination of this mechanism in each cycle as it consistently sought to identify

and act upon the node with minimum latency. This continual derivation of the minimum latency node mandated a forced descheduling of a pod whenever sufficient measurements were acquired, leading to an incessant cycle of frequent and infinite pod descheduling. This was due to the existence of a minimum value in every cycle and a lack of a natural steady state, resulting in a scenario dominated by forced conditions, post the measurement of all nodes.

Limitations The system does not yet support **multi-user operation**, constraining its applicability in shared environments, even though the data structure is already ready to support it.

Conclusion

This version of the Latency-Aware Scheduler enhances its adaptability and suitability by aligning latency constraints with service requirements and user preferences. The perpetual evaluation and realignment enable seamless operation with both Liqo-integrated and standalone setups, presenting a significant advancement over the first version.

4.3.4 Version 3.5 (V3.5): Multi-User Support

The Latency-Aware Scheduler's journey towards becoming a versatile and holistic solution continues. While Version 3 acknowledged and tackled the inherent challenges tied to network latency, it remained restrictive in its operations—catering predominantly to a singular user paradigm. This limitation became particularly pronounced in environments characterized by the simultaneous presence of multiple users, each experiencing distinct network latencies due to their diverse geographical placements. Version 3.5 is a breakthrough in this domain, introducing comprehensive multi-user support that harmonizes the delicate balance between individual user experiences and global network optimization.

The Multi-User Challenge

The concept of a **multi-user environment** inherently carries with it a bouquet of intricacies, and when it intersects with the ever-evolving domain of network latency, the complexities multiply. At the heart of this conundrum is the intrinsic variability of network latency experienced by users. Users, by virtue of being distributed across diverse geographical locations, encounter a **spectrum of latency values**. Imagine a user in Europe connecting to a node in Asia, while another in North America reaches out to a node in Africa. The latency values for these two users, in relation to their connected nodes, could differ vastly. The node that is geographically closest and hence optimal for one user might be distant and suboptimal for another.

This geographical distribution, combined with the organic growth in user numbers and their unpredictable request patterns, generates a dynamic and unpredictable latency landscape. The scheduler, in its bid to provide an equitable experience, constantly juggles the pods amongst the nodes. This balancing act, although well-intentioned, can lead to inadvertent consequences. In its zeal to optimize for a particular user based on a transient snapshot of the network, the descheduler might reallocate pods away from nodes that are optimal for other users. This act, albeit unintentional, could worsen the latency for these sidelined users, leading to a fractured and inconsistent user experience.

Such challenges underscore the need for a solution that not only appreciates the diverse latency landscape but also respects the individual experiences of users, ensuring that the system's global optimization goals do not overshadow the unique requirements of each user.

User-Cluster Association: The Core Solution

In the intricate weave of a multi-user, multi-cluster environment, the paramount challenge is ensuring that each user's experience is optimized without unintentionally compromising the experience of others. The central tenet of Version 3.5 addresses this concern by emphasizing the principle of **user-cluster association**, a robust and intelligent mechanism to match users to their optimal clusters based on latency considerations.

While a user is associated with a singular cluster, it's essential to note that this relationship isn't exclusive. A single cluster, given its capacity and geographical location, could very well be the optimal choice **for multiple users**. Especially in scenarios where users are geographically co-located or share similar network pathways, it becomes not only feasible but also efficient to map them to the same cluster. This means that while each user is tied to one specific cluster ensuring consistent latency and performance for them, a cluster might cater to the needs of multiple users who share similar latency profiles.

The strength of this approach is multi-fold:

1. **Resource Optimization**: By allowing multiple users to associate with a single cluster (when suitable), resources within the cluster are better utilized. This helps in preventing resource underutilization in some clusters while others might be strained.

2. **Latency Uniformity**: Users associated with a particular cluster will experience consistent and predictable latency. Even though multiple users might be tied to a single cluster, as long as their latency requirements are in alignment, their experience remains uniform and optimal.

3. **Dynamic Scalability**: The user-cluster association model is not static. As users' locations change or as network conditions evolve, the associations can be re-evaluated and adjusted. This ensures that the system remains agile and responsive to shifting user needs.

4. **Reduced Rescheduling Overhead**: By anchoring users to specific clusters, the frequency of pod reallocations diminishes. While the descheduler is still at play for broader system optimizations, the user experience remains insulated from frequent disruptions.

This model's elegance lies in its ability to strike a balance between individual user optimization and broader system efficiency. While each user is assured of a latency-optimized experience, the system as a whole benefits from streamlined operations and reduced overheads.

The "Free-Agent" Pod Mechanism and Association Management

The introduction of the **"free-agent"** pod is a pivotal enhancement in the system. This **unassociated pod** is designed to serve incoming users who wish to measure cluster latencies. By maintaining at least one such pod, new users can gauge which cluster offers the best latency without disrupting any existing user-pod associations.

When a user-cluster-pod association is forged, the system verifies the existence of at least one unassociated pod. If none exists, the replica set is increased, safeguarding the presence of a free-agent pod and ensuring the system's readiness for new users.

User-cluster-pod associations come with an **expiration timer**. After 5 minutes of user inactivity, the association expires, and the system assesses if there are surplus free-agent pods. If there are, the replica set is adjusted downward by descheduling one, ensuring efficient resource utilization.

Operational intricacies necessitate a temporary deployment freeze during specific tasks:

1. To prevent an immediate pod respawn when a pod is deleted with an active deployment.

2. To avoid the unintentional deletion of a pod with active associations when reducing the replica set.

This strategy, encompassing the free-agent pod, dynamic replication, and association expiration, guarantees the system's responsiveness while optimizing resource use.

Customized LoadBalancer: An Indispensable Architectural Component

The **Customized LoadBalancer** stands as an important entity in the intricate dance of ensuring optimal user experiences. While the custom scheduler adeptly assigns pods to clusters or nodes, its scope is inherently restricted to pod allocation. However, in a multi-cluster environment, ensuring that user requests meet the right pod becomes paramount, and this is a challenge the scheduler cannot directly address. This creates a potential gap: even if a user is associated with a specific cluster, the traditional load balancing techniques might still route their request to a non-optimal pod, possibly located in a high-latency cluster.

Addressing this very gap, the Customized LoadBalancer emerges not just as a facilitator, but as an essential orchestrator. Unlike its traditional counterparts, which primarily distribute requests based on load metrics, our LoadBalancer is imbued with the intelligence to discern and respect the user-cluster-pod associations. It has the capability to read and understand the data structure holding these associations, ensuring that the user requests are not merely distributed, but thoughtfully directed.

For users with established associations, the LoadBalancer takes the reins to ensure that their requests are not just treated as random entities to be load balanced. Instead, it **routes them to one of the pods** within their associated cluster, chosen at random. This ensures that the user's latency experience remains consistent and optimal. Conversely, for users without such associations, the LoadBalancer defaults to traditional load balancing methods, ensuring that the system remains efficient and responsive to all users, irrespective of their association status.

In essence, the Customized LoadBalancer doesn't merely balance loads; it strategically optimizes user experiences by ensuring that the hard-earned gains of user-cluster associations are not squandered by arbitrary request routing.

The Routing Manager

The **Routing Manager**, an advanced component in our multi-cluster environment, represents a leap forward in intelligent request routing. Its design is tailored to address the complex challenges of directing user requests to the most suitable pods, taking into account real-time latency metrics and user-pod associations. This sophisticated tool extends far beyond the capabilities of a standard LoadBalancer, providing an essential link between the custom scheduler's decisions and the actual user experience.

Synergy with the Custom Latency-Aware Scheduler: The effectiveness of the Routing Manager is intrinsically tied to its synergy with the custom latency-aware scheduler. This scheduler, designed to allocate pods optimally across clusters,

considers latency as a key factor. However, its efficiency could be undermined without a corresponding mechanism to ensure that user requests are routed according to these allocations. This is where the Routing Manager becomes indispensable. It translates the scheduler's decisions into real-world routing outcomes, ensuring that users are consistently directed to pods in clusters offering the lowest latency, thereby actualizing the benefits of the scheduler's latency-aware approach.

Adaptive and Responsive Design: The Routing Manager stands out for its adaptability and responsiveness. It is constantly monitoring for any changes in pod status or user-cluster associations. This allows it to swiftly adjust its routing decisions in response to evolving cluster conditions or updated scheduler decisions, maintaining a balance between optimal user experience and system efficiency.

Technical Implementation: The Routing Manager is a sophisticated module developed in Go, leveraging widely-used libraries and frameworks to ensure robust and efficient performance. At its core, the Routing Manager utilizes HTTP server functionality for handling incoming requests, JSON for encoding data structures, and the Kubernetes client-go library for seamless integration with the Kubernetes ecosystem.

Traffic Routing Logic: The Routing Manager's primary responsibility is to intelligently direct user traffic to the appropriate destination based on user-cluster associations. When a request arrives, the Routing Manager first checks if there is an existing association for the user. If an association exists, indicating that the user has a preferred pod based on latency measurements, the request is directed specifically to the IP address of that associated pod. This targeted approach ensures that the user's experience is optimized based on the latest scheduling decisions and latency data. In cases where no association exists for a user, the Routing Manager adopts a more traditional load balancing approach. Here, it selects a pod for the request in a randomized manner or based on other classic load balancing strategies. This flexibility ensures that even in the absence of specific user-pod associations, the system continues to function efficiently, distributing requests evenly across available resources.

API Exposition and Adaptability: One of the key features of the Routing Manager is its exposed API, which allows for real-time updates of user-cluster associations. This API is designed to be easily adaptable to various contexts and can integrate with different schedulers or orchestration systems. It enables external entities, such as the custom scheduler, to communicate updated user-cluster-pod associations, ensuring that the Routing Manager is always operating with the most current data.

In summary, the Routing Manager is not just an essential component in our current architecture; it is a forward-looking solution, designed to adapt and scale with the evolving needs of complex, dynamic multi-cluster environments. Its intelligent routing logic, combined with its flexible API and scalable design, positions it as a key enabler for optimized user experiences and efficient cluster utilization.

Conclusion

Version 3.5 of the Latency-Aware Scheduler signifies a monumental leap, bridging the gap between individual user needs and global network optimization goals. By introducing multi-user support underpinned by the users-cluster association mechanism and the Customized LoadBalancer, this version offers a refined, scalable, and consistent solution suitable for diverse, multi-user environments.

4.4 Latency-Meter Contact Overhead

The intricacies involved in the interaction between the De-scheduler and latency-meters present significant limitations and challenges in the current architecture of the system. The periodic contact between the De-scheduler and each latency-meter, set at intervals t, demonstrates a **critical bottleneck**, particularly **as the number of pods increases**.

This model is hindered by inherent inefficiencies due to the **asynchronous nature of the HTTP requests** employed for communication. The time overhead involved in contacting individual latency-meters is substantial, leading to decreased overall system performance and responsiveness, making it a point of concern for system designers and administrators.

The presence of latency-meters in every pod within the cluster necessitates an efficient and robust system for gathering latency data, crucial for optimizing system performance and resource allocation. The current approach, despite its effectiveness in data collection, demands refinements to alleviate the bottlenecks and overheads associated with it.

Proposed Solution: Synchronized Distributed Database

The introduction of a **synchronized, distributed database** emerges as a viable solution to address the aforementioned limitations. This advanced approach enables **concurrent data writes** by the latency-meters and subsequent synchronization, allowing the De-scheduler to access the aggregated data through **a single read operation**, thereby significantly reducing convergence time and improving system efficiency.

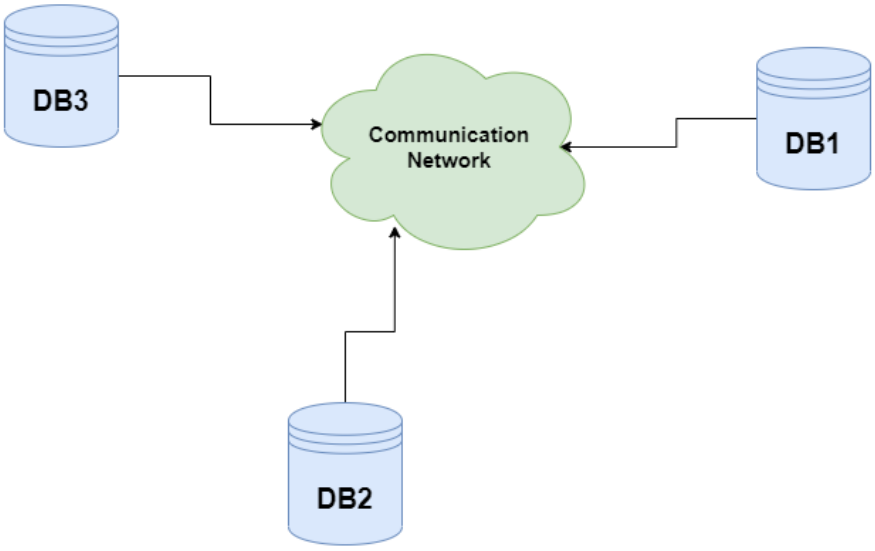


Figure 4.3: Distributed Database

Implementing such a database system brings forth benefits including enhanced **data availability**, **reliability**, and **fault tolerance**, addressing the critical needs of high-demand enterprise environments. It also facilitates seamless scaling to accommodate growing data volumes and user demands, ensuring the system's longevity and adaptability to evolving requirements.

Insights into Distributed Database Solutions: The implementation of a synchronized, distributed database can leverage various established solutions, each offering unique advantages:

- **Cassandra:** Recognized for its fault tolerance and scalability, Cassandra can handle vast data volumes spread across multiple servers, making it suitable for environments where high availability and no single point of failure are crucial.
- **Couchbase:** This NoSQL database offers high performance and scalability, providing a unified platform for efficiently managing, querying, and indexing data, suited for diverse enterprise applications.
- **MongoDB:** With its flexible schema and cross-platform capabilities, MongoDB is adept at managing document-oriented data, ensuring fast and easy integration in varied applications.
- **CockroachDB:** It stands out for its strong consistency and data availability in distributed environments, especially in cloud services, addressing the challenges of managing global deployments effectively.

Moreover, the integration of these databases allows for the development of a **more cohesive and harmonious system architecture**, ensuring that the interaction between different components is seamless and efficient. The trade-offs, complexities, and considerations involved in choosing the appropriate database solution require careful evaluation of system requirements, data structures, and operational constraints.

In conclusion, addressing the latency-meter contact overhead through a synchronized, distributed database is pivotal for enhancing system robustness and efficiency. This innovative approach mitigates the existing limitations and opens avenues for exploring more sophisticated and holistic solutions in the future.

Chapter 5

Implementation & Results

5.1 Development Environment

The scheduler variants were developed and extensively tested within a dedicated environment. This environment was hosted on a **VM** running **Debian 11 Bullseye**, virtualized through **Virtual Box on a Windows 11 PC** with specifications of *16 GB RAM*, an *8-core CPU*, and *256 GB of disk space*. The VM was allocated *4GB of RAM*, a *2-core CPU*, and *20 GB of disk space*.

For the creation and management of local Kubernetes clusters, **KinD** (*Kubernetes in Docker*), was employed, leveraging its latest version, `kindest/node v1.28`. KinD is a versatile and lightweight tool designed to run local Kubernetes clusters using Docker container "nodes". It serves as a more resource-efficient alternative to traditional VM-based solutions, thereby enabling comprehensive testing even on constrained computing environments, such as a personal computer.

The inherent adaptability and interoperability of **Liqo** (*version 0.9.4*) on **KinD** allowed for immediate experimentation with **multi-cluster environments** from the early stages of development, proving invaluable for progressive testing and iterative development throughout the various stages of the project.

The development was done using **Go** (*Golang*), (*version 1.21.1*), the **Kubernetes source language**, known for its simplicity, efficiency, and reliability in managing distributed systems. The applications, namely the custom scheduler in its two versions and the Latency-Meter, were developed with coherence and simplicity in Go, and were deployed as container images on my personal Docker Hub (User: *crischiaro*) to ensure availability and ease of use. They can be referenced in Kubernetes deployments or pods using the following URIs:

- `crischiaro/latency-aware-scheduler_v2:latest`
- `crischiaro/latency-aware-scheduler_v3:latest`

- `crischiaro/latency-aware-scheduler:latest`
- `crischiaro/latency-meter:latest`
- `crischiaro/routing-manager:latest`

The development workflow was marked by a high degree of flexibility and dynamism, enabled by the seamless interoperability provided by Kubernetes and the flexibility of the employed tools and languages, accommodating evolving requirements and adjustments with ease.

Component	**Version**
Debian (VM)	11 Bullseye
Windows (Physical Host)	11 22H2 Home
KinD (Kubernetes in Docker)	kindest/node v1.28
Go (Golang)	1.21.1
Liqo	0.9.4
Custom Scheduler (V2)	crischiaro/latency-aware-scheduler_v2:latest
Custom Scheduler (V3)	crischiaro/latency-aware-scheduler_v3:latest
Custom Scheduler (V3.5)	crischiaro/latency-aware-scheduler:latest
Latency Meter	crischiaro/latency-meter:latest
Routing Manager	crischiaro/routing-manager:latest

Table 5.1: Summary of Components and Versions

5.2 Testing Environment

Following the development of the custom scheduler and the Latency-Meter, the applications, along with their interoperability, were tested in a specialized cloud environment provided by **Politecnico di Torino**, named **CrownLabs** [15]. The goal was to transition to a more realistic scenario. Since testing on production clusters was not feasible, virtual machines (VMs) within CrownLabs were leveraged for this purpose.

CrownLabs delivers remote computing labs through per-user VMs, allowing instructors and students to interact in a versatile and secure remote environment. It enabled the provisioning of a set of VMs, each equipped with **Ubuntu 20.04.6 LTS**, *2 GB RAM*, *2 CORE CPU*, and *20 GB DISK*, forming an ideal environment for conducting extensive tests on the scheduler variants.

In this environment, **Kubernetes** (*version 1.28*) was installed directly, encompassing `kubeadm`, `kubectl`, and `kubelet`, omitting the use of KinD. The chosen *container-runtime* was `containerd` (*version 1.6.24*), and the Container Network

Interface (**CNI**) was `Flannel` (*version 0.22.3*). The container runtime is pivotal for enabling container execution within a Kubernetes cluster, and the CNI is crucial for facilitating inter-container communication, establishing a reliable and efficient networking substrate for container orchestration.

Several VMs were allocated for testing. Initially, they were configured as nodes of a single cluster (1 control-plane and n workers) to compare the two custom-scheduler versions. Subsequently, the VMs were reconfigured to form **several Virtual Clusters**, each comprising 2 nodes (control-plane and worker), interconnected using Liqo (*version 0.9.4*) through **Out-of-band peering**, creating a topology of **1 master cluster and *n* slave clusters**. The remaining VM simulated user requests, ensuring a comprehensive evaluation of the scheduling strategies in diverse scenarios.

Component	**Specification or Version**
Environment	CrownLabs
VM Specifications	Ubuntu 22.04 LTS, 2 GB RAM, 2 Core CPU, 20 GB Disk
Kubernetes	v1.28 (kubeadm, kubectl, kubelet)
Container Runtime	containerd v1.6.24
Container Network Interface (CNI)	Flannel v0.22.3
Clusters Configuration 1	1 Master Cluster, 3 Slave Clusters (2 nodes each)
Clusters Configuration 2	1 Single Cluster (9 nodes)
Liqo	v0.9.4
Connection	Out-of-band peering

Table 5.2: Summary of Testing Environment Specifications

5.3 Testing

In this section, we detail the methodologies and the tools used for emulating network latency and its impacts within the designed Kubernetes clusters. The ability to precisely induce and measure latency is pivotal for validating the efficiency and resilience of the custom-scheduler and latency-meter under varying network scenarios.

5.3.1 Inducing Network Latency

Traffic Control (TC) is a kernel-based subsystem in Linux, providing the ability to **control network traffic management**. It is essentially a framework for managing and configuring the network traffic on a Linux host, allowing for manipulation of the traffic while it is in transit. This includes controlling the bandwidth allocation, managing traffic queues, classifying network packets, shaping network traffic, and more.

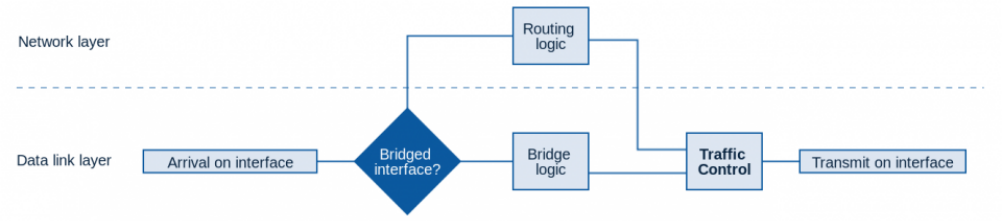


Figure 5.1: Traffic Control [16]

Within this framework, the `tc` (Traffic Control) command emerges as an invaluable feature, serving as a versatile interface allowing users to modify network traffic behavior on specific interfaces. This command is particularly renowned for its ability to **induce artificial delays**, simulating various network conditions that are crucial to assess the adaptability and performance of distributed systems under different network states.

This command enables the replication of a myriad of network scenarios essential for testing the robustness and adaptability of distributed systems, especially when subjected to unstable network conditions. By interacting with the network interface layer, the `tc` command allows for the induction of network anomalies such as latency, positioning it as a fundamental tool in the emulation of diverse network states.

When employing the `tc` command, several crucial aspects and intricacies must be mindfully considered:

- **Reflective Nature of Latency:** The induced latency is reflective, affecting both the incoming and outgoing traffic. This dual impact makes it imperative to meticulously plan and understand the network topology to emulate exact latency values for each node or cluster accurately.

- **Topological Awareness:** The actual latency experienced can differ from the expected values, depending on the underlying network topology and the route traversed by the packets. This underscores the need for a deep understanding of the network topology and a strategic approach to interpreting latency values to gain authentic insights.

- **Inherent Variability:** The induced latency acts as a base threshold, with the actual latency being susceptible to variations due to multiple factors such as network traffic, geographical distances, and other unpredictable variables. Therefore, it is essential to view the induced latency as a minimum bound, acknowledging the inherent variability.

Examples and Practical Application Below are practical examples showcasing the utilization of the `tc` command to manage network latency:

```
# Viewing existing latency settings:
sudo tc qdisc show dev enp1s0

# Inducing latency:
sudo tc qdisc add dev enp1s0 root netem delay 100ms

# Removing the induced latency:
sudo tc qdisc delete dev enp1s0 root netem
```

These commands are instrumental in observing, introducing, and eliminating network delay (so latency) to a network interface (e.g. enp1s0). This allows the creation of varied network conditions to study the repercussions on the devised solutions.

Precision and Precaution in Application When applying the `tc` command, precision and a profound understanding of network topology are indispensable. It is crucial to acknowledge potential discrepancies between the induced and the measured latency due to inherent network structures and external influences. A thoughtful and cautious approach is paramount during testing phases to accurately recognize the intrinsic variability in network latency, treating the induced values as minimal thresholds while being alert to possible deviations due to uncontrollable network circumstances.

5.3.2 Test: Default Scheduler vs. Latency-Aware Scheduler (V3)

In order to compare the performance and efficiency of the Kubernetes default scheduler against the Version 3 of the custom latency-aware scheduler, two distinct use cases were considered. The testing involved deploying an **Nginx application**, also **incorporating a latency-meter container**, and exposing it through a **Load Balancer service**. The service routes user requests to balance load and traffic efficiently, and in instances of equal load, it operates in a randomized manner.

Setup and Configuration

8 VMs were segmented into 4 Clusters, each designated as:

- **MILAN**: Master
- **ROME**: Slave

- **PARIS**: Slave
- **TURIN**: Slave

with each cluster consisting of 2 nodes (VMs): one serving as a **control-plane node** and the other as a **worker node**. These clusters were interconnected using Liqo, forming a **star network topology** with MILAN as the master cluster in the center and the others as slave clusters to the edges.

In both use cases, the following configurations were set into the deployment:

- `hard_constraint = 40`
- `soft_constraint = 30`

A Bash script was used to run on a ninth VM to **simulate requests from a user**. This script measures the latencies experienced by the user, executing a specified number of requests (`tot_r`) at designated intervals (`i_s` seconds), with the values varying depending on each case d 'use.

Algorithm 5 Network Latency from User Algorithm

```
Input: total_requests, service_url, interval
Init:
    script_start_time                                   ▷ Script start
for i = 1 to total_requests do
    start_time ← Current Time
    response ← HTTP Status from service_url
    end_time ← Current Time
    latency ← end_time - start_time
    Output: "Req. i - Latency: latency ms - Status: response"
    if i < total_requests then
        Sleep for interval seconds
    end if
end for
script_end_time ← Current Time                          ▷ Script end
total_time ← script_end_time - script_start_time
Output: "Total Time: total_time ms"
```

Baseline Latency It is crucial to note that an inherent baseline latency exists within the clusters, ranging between 15 and 25 ms, as assessed by the request script without leveraging the `tc` command. Consequently, any usage of the `tc` command would result in an **incremental increase in the latency range** for each cluster.

Use Case 1: Normal Behaviour

The first use case involved configuring the `tc` on clusters to establish the following latencies:

- **MILAN**: [18; 28]ms - Soft Valid Cluster (`latency < soft_constraint`)
- **ROME**: [29; 39]ms -
  Hard Valid Cluster (`soft_constraint < latency < hard_constraint`)
- **PARIS**: [45; 55]ms - Invalid Cluster (`latency > hard_constraint`)
- **TURIN**: [41; 51]ms - Invalid Cluster (`latency > hard_constraint`)

The test was conducted with a total of **55 requests**, an **interval of 4 seconds** between each request, and **2 replicas** of `nginx` and `latency-meter` containers.

Results and Analysis The collected latency measurements for the default scheduler and the custom scheduler (V3) are represented graphically to illustrate the comparative efficacy and response trends over time. Additionally, summary tables presenting the cumulative average and other relevant statistics are included for a compact view of the outcomes.

The tables below embody a comprehensive depiction of the recorded latencies during the experimental phase, illustrating not only the average latency but also the confidence intervals, and convergence times. These representations are instrumental in underscoring the comparative effectiveness and intricate performance metrics of the schedulers in use, facilitating a nuanced understanding of their operational dynamics and adaptative capabilities.

	Default Scheduler	Custom Scheduler (V3)
Average Latency (ms)	42.80	34.38
Confidence Interval	[40.65, 44.95]	[32.57, 37.20]
Convergence Time (s)	0	~ 90

Table 5.3: Summary of Latency Measurements

The **default scheduler** deploys pods to clusters in a manner that appears almost **random**, especially given the initial unoccupied state of the clusters, maintaining a nearly immutable cumulative average above the `hard_constraint`. This allocation strategy, seeming arbitrary, may result in the selection of a valid and an invalid cluster, sustaining the average latency until a de-scheduling event intervenes due to scenarios like application failure or node/cluster crash.

Conversely, the **custom latency-aware scheduler**, after an initial phase of seemingly random allocation resembling the default scheduler's behavior, displays

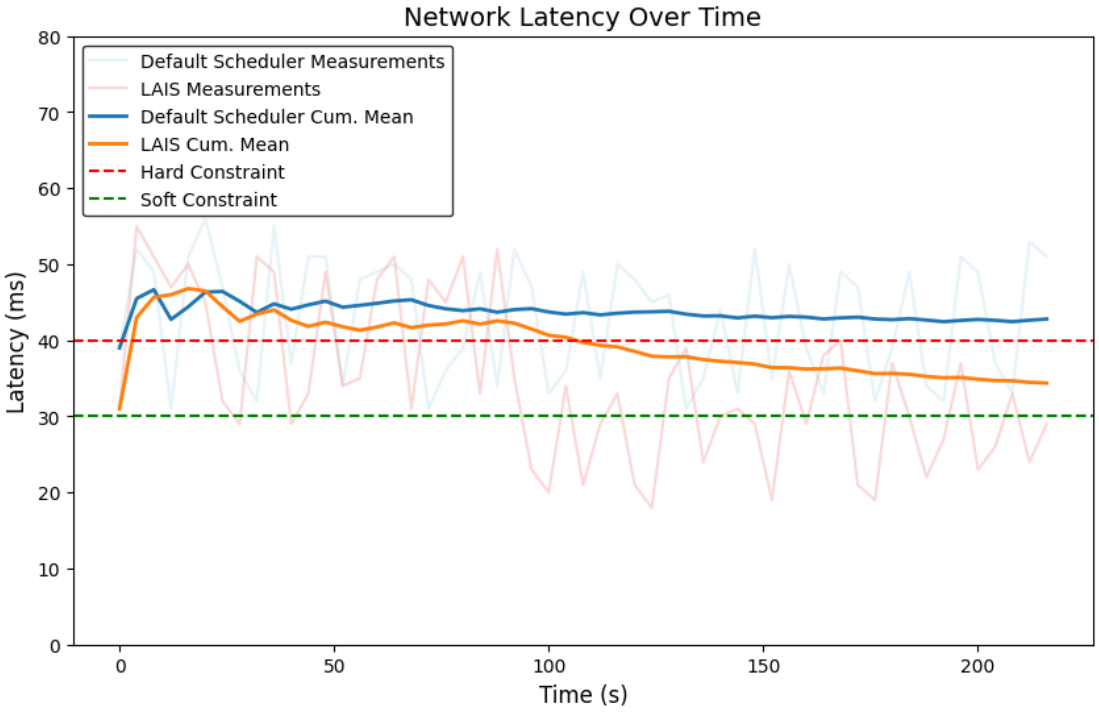


Figure 5.2: Default Scheduler vs Latency-Aware (V3) Scheduler

pronounced adaptability and nuanced response to network latencies. In the initial phase, the pods are apparently allocated to the same clusters as the default scheduler, likely Rome and either Paris or Turin. This decision is presumably arbitrary, stemming from the lack of initial load on the clusters.

However, as the user requests start coming in and the `latency-meter` begins to collect the network latency measurements, a discernible shift in the scheduler's allocation strategy becomes evident. Approximately 90 seconds into the test, the recorded measurements showcase a notable dip below the `soft_constraint`. This suggests that one of the pods, probably initially scheduled on an Invalid cluster like Paris or Turin, had been de-scheduled and re-allocated to Milan, a Soft Valid cluster, leading to a convergence where the cumulative average latency harmoniously resides within the soft and hard constraint limits.

The convergence time, the time post which the pods no longer move and a stable state is achieved, is observed to be around 90 seconds. During this state of regime, the cumulative average is confined within the band defined by the soft and hard constraints, reflecting the scheduler's adaptive reallocation of the pod to conform to the latency restrictions. The **continuous user requests** and subsequent network latency measurements play an **important role in this adaptive reallocation**, enabling the scheduler to accurately discern the viability of the clusters and adjust

the pod placements accordingly.

The narrow confidence interval associated with the default scheduler is reflective of its relatively stable and predictable allocation behavior, primarily owing to its non-adaptive, load-balanced allocation mechanism. This results in a more stabilized and consistent latency range, contrasting with the **responsive and adaptative** nature of the custom scheduler (V3) which, by continually refining allocations based on the latest latency measurements, exhibits a broader and more varied range of responses.

Use Case 2: Soft Condition

This use case elucidates the Soft Condition scenario, illustrating how this specific condition further reduces the overall network latency at the expense of convergence time. The `tc` were configured on clusters as follows:

- **MILAN**: [18; 28]ms - Soft Valid Cluster (`latency < soft_constraint`)
- **ROME**: [29; 39]ms -
  Hard Valid Cluster (`soft_constraint < latency < hard_constraint`)
- **PARIS**: [45; 55]ms - Invalid Cluster (`latency > hard_constraint`)
- **TURIN**: [19; 29]ms - Soft Valid Cluster (`latency < soft_constraint`)

The test was executed with a total of **65 requests**, an **interval of 4 seconds** between each request, and **3 replicas** of `nginx` and `latency-meter` containers.

Results and Analysis The latency measurements for both the default and custom scheduler (V3) are depicted graphically below, to underscore the comparative effectiveness and adaptive response over time. A summary table is also included to provide a compact overview of the outcomes.

	Default Scheduler	Custom Scheduler (V3)
Average Latency (ms)	42.22	27.86
Confidence Interval	[39.57, 44.86]	[25.57, 30.16]
Convergence Time (s)	0	~ 120

Table 5.4: Summary of Latency Measurements in Soft Condition

Initially, both the default and the custom schedulers appear to allocate pods in an almost **random** manner, given the initial unoccupied state of the clusters. The **default scheduler**, as in the first Use Case, while accounting for the load, seems to make random allocations initially since the clusters are empty. It appears

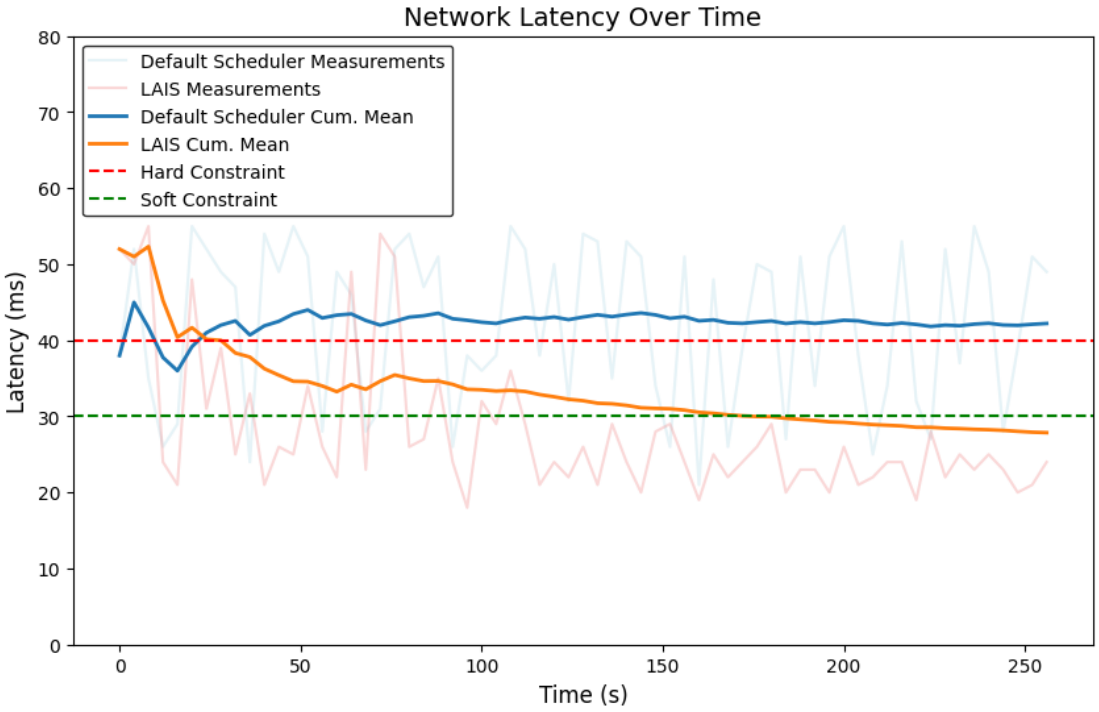


Figure 5.3: Default Scheduler vs Custom Scheduler (V3) in Soft Condition Scenario

to choose a Hard Valid Cluster (Rome), an Invalid Cluster (Paris), and a Soft Valid Cluster (either Milan or Turin). The cumulative average maintains above the `hard_constraint` and remains relatively constant as pods do not change until a potential de-scheduling event occurs.

The **custom scheduler** (latency-aware), like the default one, initially allocates pods seemingly at random, apparently choosing the same clusters. However, we observe that, after 30 seconds, the user measurements begin to fall within the band between hard and soft constraints. There could already be talk of convergence at this stage, but around the 70th second, **we again receive measurements above the `hard_constraint`** threshold, only to return to the band around 90 seconds and finally settle completely **below the `soft_constraint`** threshold after 120 seconds. This corresponds to our convergence time.

What happened? **The Soft Condition mechanism!** The pods were scheduled on 1 Hard, 1 Soft, and 1 Invalid Cluster out of a total of 4 clusters. When the pod on the Invalid was de-scheduled, it ended up on the other Soft Valid Cluster. Thus, it found itself in the soft condition, given by the inequality:

$$\#\ \text{Soft Valid Clusters} + \#\ \text{Hard Valid Clusters} > \frac{\text{Total } \#\ \text{Clusters}}{2}$$

Substituting in the known values gives:

$$2\,(\text{Soft}) + 1\,(\text{Hard}) > \frac{4\,(\text{Total})}{2}$$

Simplifying:

$$2\,(\text{Soft}) + 1\,(\text{Hard}) > 2\,(\text{Total})$$

Thus, validating the Soft condition, leading to the de-scheduling of the pod from the Hard Valid Cluster.

At this point, as in the **worst-case scenario**, the pod was re-scheduled on the Invalid, de-scheduled after 30 seconds, re-scheduled again on the Hard, triggering the soft condition again, and, finally, scheduled on a Soft Valid Cluster at 120s time.

This adaptive behaviour is attributed to the Soft Condition mechanism, which, when activated, triggers the re-scheduling of pods from Hard Valid clusters to Soft Valid clusters, harmonizing the cumulative average latency within the soft and hard constraint limits, exhibiting a clear adaptability and convergence to an optimal state around 120 seconds after a series of re-allocations, significantly reducing the cumulative average latency.

Again, The difference in confidence intervals between the default and custom scheduler can be associated again with the dynamic and adaptive nature of the latter. The responsive and continuous refinement of allocations based on the latest latency measurements by the custom scheduler leads to a broader range of responses and subsequently, a wider confidence interval compared to the more static and predictable allocation behavior of the default scheduler.

5.3.3 Test: Custom Scheduler V2 vs. Custom Scheduler V3

To assess and contrast the efficacy of the two versions of the custom latency-aware scheduler, namely V2 and V3, two specific use cases were conducted, adopting the identical steps and configurations as the preceding tests. The V2, being the precursor to V3, is designed to allocate pods, post-transitory phase, to the nodes/clusters manifesting minimal latency.

The aim is to draw a comprehensive comparison between the nuanced behaviors and adaptive functionalities of both scheduler versions under varying network conditions.

Common Setup and Configuration

For both use cases, the deployment involved an **Nginx application** integrated with a **latency-meter container** and exposed via a **Load Balancer service**. The tests abided by the constraint configurations of:

- `hard_constraint = 40`
- `soft_constraint = 30`

The Bash script referenced in Algorithm 5 was once again employed to simulate user requests and ascertain the experienced latencies. This approach ensured uniformity in the data acquisition process across all tests, enabling a coherent and comparable dataset.

Use Case 1: Normal Behaviour

This use case explores the standard operational scenario demonstrating the normal behavior of the scheduler under predefined constraints. The VMs are divided as the last Test: **8 VMs in 4 Clusters** named **MILAN**, **ROME**, **PARIS** and **TURIN**, configured and connected (with Liqo) in such a way as to have an **active star network topology** in which the **Milan cluster is the Master and the other Slaves**.

Using the `td` command,the Clusters are configured as follows:

- **MILAN**: [18; 28]ms - Soft Valid Cluster (`latency < soft_constraint`)
- **ROME**: [20; 30]ms - Soft Valid Cluster (`latency < soft_constraint`)
- **PARIS**: [38; 48]ms - Invalid Cluster (`latency > hard_constraint`)
- **TURIN**: [29; 39]ms -
  Hard Valid Cluster (`soft_constraint < latency < hard_constraint`)

The test involved a total of **40 requests**, an **interval of 6 seconds** between each request, and **2 replicas** of the `nginx` and `latency-meter` containers.

Results and Analysis Comparative graphs of both scheduler versions are shown below, emphasizing the adaptive response and effectiveness of both scheduler versions over time.

	Scheduler V2	Scheduler V3
Average Latency (ms)	28.77	32.03
Confidence Interval	[25.34, 30.31]	[29.93, 33.57]
Convergence Time (s)	120	48
Total Descheduling	3	1

Table 5.5: Summary of Latency Measurements in Normal Behaviour Scenario

The schedulers initially allocate pods seemingly at random, probably choosing an Invalid (Paris) and a Soft Valid Cluster (either Milan or Turin). However, the

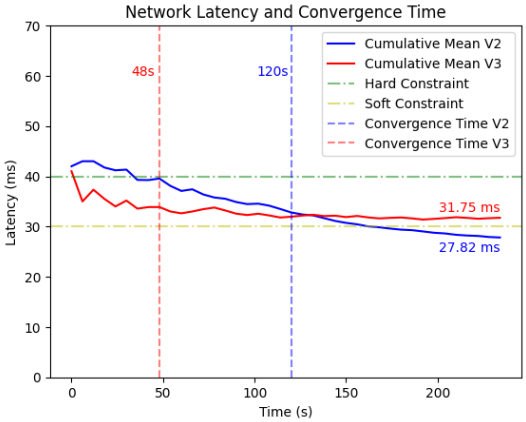


Figure 5.4: Convergence Time V2 vs V3

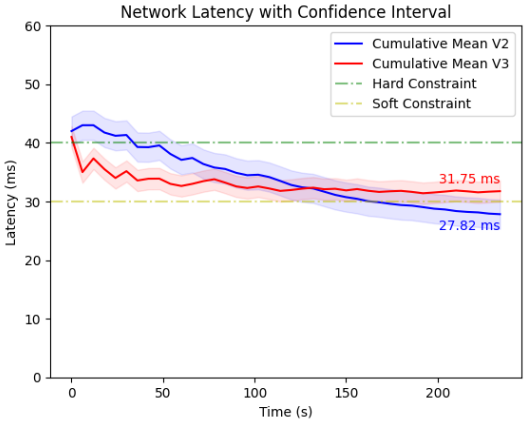


Figure 5.5: Confidence Interval V2 vs V3

cumulative average of the Scheduler V2 dips below the `hard_constraint` **significantly later** compared to the V3. This delay is attributed to the Scheduler V2 requiring enough measurements (equal to the number of pods) before it begins the de-scheduling process, in contrast to the V3, which only needs a single measurement per pod to make de-scheduling decisions.

Once the Scheduler V2 has received adequate measurements, it begins a systematic de-scheduling process. This involves rigorous cycling through the various clusters, meticulously identifying and removing the pods associated with the **worst latency**, each time trying to progressively refine the overall latency profile. This approach ultimately causes the scheduler to settle below the `soft_constraint` boundary, demonstrating a strategic preference for optimizing network performance beyond merely conforming to the acceptable latency range. This implies that Scheduler V2 is inherently designed to aggressively pursue optimal performance levels, often involving multiple cycles of de-scheduling and re-scheduling to reach a latency equilibrium state that is the **minimum**.

In contrast, Scheduler V3 exhibits a more conservative and equilibrium-oriented behavior. It meticulously avoids transgressing the `soft_constraint`, instead choosing to strategically maintain the allocations within the defined acceptable latency range. This approach underscores a deliberate and calibrated trade-off between performance optimization and operational stability, enabling Scheduler V3 to achieve a substantially reduced convergence time compared to its V2 counterpart. This can be attributed to the fact that Scheduler V3 necessitates only a single de-scheduling event to stabilize its operational state, transitioning swiftly from an initial allocation of pods in Invalid and Soft Valid clusters to a balanced allocation in Soft Valid and Hard Valid clusters—a condition deemed **acceptable** according

to the deployment's predefined operational parameters.

To illustrate further, the refined approach of Scheduler V3 stands in stark contrast to the extensive iterations performed by Scheduler V2 to optimize latency. This is evident from the multiple transitional phases of Scheduler V2, characterized by several reallocations and adjustments, leading to a substantially elongated convergence time. These transitions underline the fundamental differences in the operational methodologies and objectives between the two schedulers. Below are the sequential transitions of Scheduler V2:

- **Initial State**: Invalid - Soft Valid clusters
- **Transition 1**: Soft Valid - Hard Valid clusters
- **Transition 2**: Soft Valid - Soft Valid clusters
- **Final State** (Post-**Last Cycle** for global minimum, refer to Paragraph 4.3.1): Soft Valid - Soft Valid clusters

This extensive sequence accentuates Scheduler V2's meticulous approach to achieving the optimal latency, showcasing the inherent contrast in the strategic planning between Scheduler V2 and V3.

Confidence Interval The difference in the confidence intervals of the two schedulers reveals the dynamic and adaptive nature of Scheduler V3. Its continuous refinement and responsive allocation, based on the latest latency measurements, lead to a wider range of responses and hence, a wider confidence interval, reflecting a more diverse and adaptive allocation behavior compared to the more static and predictable allocation strategy of Scheduler V2.

Use Case 2: High Convergence Time

This use case was designed to show a significantly high convergence time when the number of nodes increases, while the number of replicas is low.

Configuration In this scenario, the VM configuration is different than in the previous use case. The VMs are no longer divided into 4 clusters, but rather **18 VMs** are inserted within **9 clusters**, giving rise to a test carried out without Liqo, composed of 1 master cluster and 8 slave clusters.

The VMs are configured with the `tc` command as follows:

1. **cluster-1**: [38; 48] Invalid Node
2. **cluster-2**: [25; 35] Hard Node

3. **cluster-3**: [35; 45] Invalid Node
4. **cluster-4**: [32; 42] Invalid Node
5. **cluster-5**: [20; 30] Soft Node
6. **cluster-6**: [26; 36] Hard Node
7. **cluster-7**: [33; 43] Invalid Node
8. **cluster-8**: [19; 29] Soft Node
9. **cluster-9**: [29; 39] Hard Node

The test involves a total of **70 requests**, a **6 second interval** between each request, and **3 replicas** of the `nginx` and `latency-meter` containers.

Results and Analysis Comparative graphs of the two versions of the scheduler are presented below, highlighting the adaptive response and effectiveness of both versions of the scheduler over time.

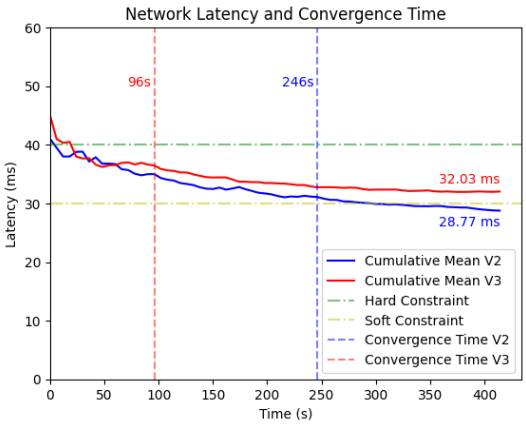


Figure 5.6: Convergence Time V2 vs V3

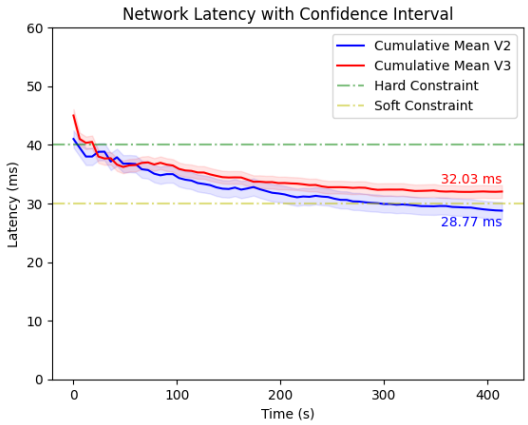


Figure 5.7: Confidence Interval V2 vs V3

This data corroborates the observations initially noted in Use Case 1, portraying a clear distinction between the operational mechanics of the V3 and V2 schedulers. Initially, both schedulers allocate pods seemingly at random due to the unoccupied state of the nodes, allowing for arbitrary selections based on node load.

However, the divergence in behavior becomes apparent once the descheduling phase commences. The V3 scheduler exhibits a reduced convergence time of around 96 seconds. It efficiently settles on an optimal solution, accommodating both Hard

	Scheduler V2	Scheduler V3
Average Latency (ms)	28.77	32.03
Confidence Interval	[27.31, 30.23]	[30.95, 33.11]
Convergence Time (s)	246	96
Total Descheduling	7	3

Table 5.6: Summary of Latency Measurements in High Convergence Time Scenario

Valid and Soft Valid nodes, hence exhibiting a faster transition to a steady-state, even if the solution isn't perfect.

In contrast, the V2 scheduler seeks a minimal perfect solution, necessitating a meticulous evaluation of all nodes and resulting in a longer convergence time of about 246 seconds—approximately **2.5 times** the convergence time of V3. It conducts 7 deschedulings, compared to the 3 by V3, reflecting its pursuit of absolute perfection at the expense of extended convergence periods.

In essence, this analysis underscores the operational distinctions between the V3 scheduler's adaptability and efficiency, and the V2 scheduler's precision and thoroughness, revealing a consequential trade-off between adaptability and absolute accuracy in scheduling paradigms.

5.3.4 Test summary: Comparison between all schedulers

The primary focus of this comparison is to gauge the effectiveness and efficiency of the default Kubernetes scheduler against three distinct variations of the Latency-Aware scheduler and a similiar and a similiar Latency-Aware scheduler namely [13], in diverse scenarios.

- **LAIS-Hard**, Custom Scheduler **V3**, focuses primarily on satisfying the hard constraint.
- **LAIS-Soft**, Custom Scheduler **V3**, extends its considerations to meet the soft constraint.
- **LAIS-0**, Custom Scheduler **V2**, looks for the the nodes with the lowest latency for the user.
- **LAK**, similiar to **V2**, doesn't consider the union of nodes in Clusters but treating them individually.

These Latency-Aware Scheduler variations are specifically crafted to address varying latency constraints, thereby offering a spectrum of responses under different network conditions.

Setup and Configuration

The deployment comprised of 16 VMs, representing 8 interconnected clusters using Liqo. For all scenarios, the tests were governed by the parameters:

- `latency_meter` set to collect values every 5 seconds
- `descheduler` operating at 30-second intervals

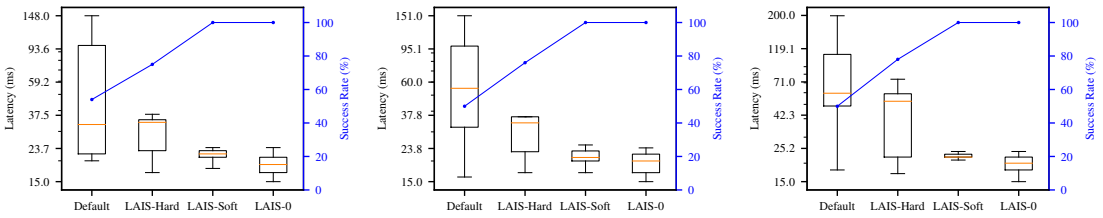
Three scenarios were crafted, each with unique configurations to simulate various real-world application setups and network conditions:

	Scenario 1	Scenario 2	Scenario 3
Hard Constraint	40ms	80ms	80ms
Soft Constraint	30ms	30ms	30ms
Cluster_1	70ms	100ms	100ms
Cluster_2	100ms	120ms	150ms
Cluster_3	120ms	150ms	200ms
Cluster_4	150ms	35ms	50ms
Cluster_5	35ms	36ms	60ms
Cluster_6	36ms	70ms	70ms
Cluster_7	15	15ms	15ms
Cluster_8	21ms	21ms	21ms

Table 5.7: Scenario Configuration

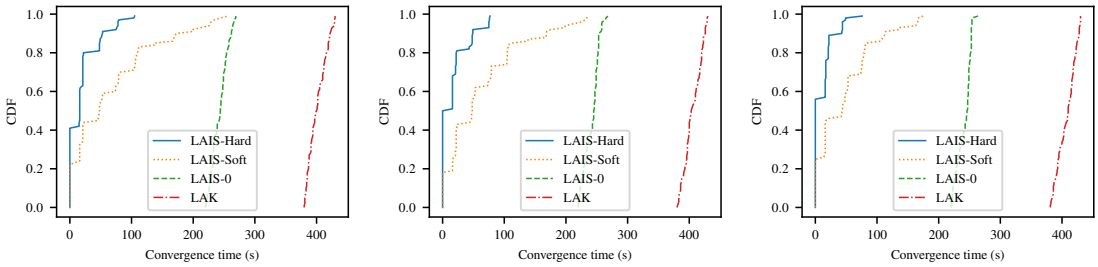
Results and Analysis

For each scenario, a series of 100 tests were conducted per scheduler variant, amassing a total of 1200 tests across all scenarios.



(a) Scenario 1 **(b)** Scenario 2 **(c)** Scenario 3

Figure 5.8: Latency (left side) and success rate (right side) for all testing scenarios. Current schedulers cannot minimize perceived latency.



(a) Scenario 1 **(b)** Scenario 2 **(c)** Scenario 3

Figure 5.9: CDF of converge time for the three scenarios. While minimizing the perceived latency is time consuming, fulfilling a (soft or hard) constraint is less costly.

	Default	**LAIS-H**	**LAIS-S**	**LAIS-0**	**LAK**
Average Latency (ms)	64.59	30.07	21.56	19.26	19.11
Convergence Time (s)	–	19.34	65.33	242.65	401.08
Success Rate (%)	54.0	75.0	100.0	100.0	100.0

Table 5.8: Summary of Test Results Scenario 1

	Default	**LAIS-H**	**LAIS-S**	**LAIS-0**	**LAK**
Average Latency (ms)	65.29	37.42	21.34	19.63	19.78
Convergence Time (s)	–	16.19	62.09	241.83	405.54
Success Rate (%)	50.0	76.0	100.0	100.0	100.0

Table 5.9: Summary of Test Results Scenario 2

	Default	**LAIS-H**	**LAIS-S**	**LAIS-0**	**LAK**
Average Latency (ms)	81.68	45.26	21.85	19.86	19.26
Convergence Time (s)	–	10.89	46.59	241.37	407.56
Success Rate (%)	50.0	78.0	100.0	100.0	100.0

Table 5.10: Summary of Test Results Scenario 3

Latency Distribution and Success Rate Analysis In our detailed assessment, we first compared the schedulers' efficacy in achieving targeted latency benchmarks, crucial in real-world applications. Our analysis, depicted in Fig. 5.8, showcases the varying performances across the three scenarios. The **Default Scheduler** consistently demonstrated the highest average latency, with values of 64.59 ms, 65.29 ms, and 81.68 ms for Scenario 1, Scenario 2, and Scenario 3 respectively. This scheduler's approach, **lacking in latency-oriented decision mechanisms**, led to considerable fluctuations in latency and a mere 50% success rate in valid pod placements.

LAIS-H, in stark contrast, showed a notable reduction in latency across all scenarios. While it did not achieve a perfect success rate, its performance hovered around 75% to 78%, indicating a more refined yet balanced approach: it does not reach 100% since it **trades off "hard" and "soft" constraints**.

LAIS-S and **LAIS-0** distinguished themselves with remarkably **low latencies**, consistently below 22 ms across all scenarios. Impressively, both these schedulers achieved a 100% success rate, demonstrating their reliability and effectiveness in task execution. This exceptional performance highlights their design's focus on minimizing latency while maintaining high operational standards.

Convergence Time Evaluation The convergence time of the schedulers, indicating their adaptability to changing conditions, was another critical metric we examined. As per the data from your tables, LAIS-H generally exhibited quicker convergence times, with LAIS-S and LAIS-0 following suit, albeit at a slightly slower pace. Notably, LAIS-0 had the longest convergence times, especially in Scenario 2 and Scenario 3, where it exceeded 240 seconds.

LAK, with its approach of treating each node individually rather than as part of a unified cluster, demonstrated even longer convergence times in all scenarios. This distinction in strategy, while meticulous, led to **extended durations in finding suitable nodes** for pod placement, underscoring the efficiency of the other schedulers' more integrated approach.

Summative Observations and Scheduler Selection Guidance In summary, the choice between Scheduler V3 (LAIS) and Scheduler V2 (LAIS-0) comes down to a strategic decision based on the specific requirements of the deployment environment and application type. While the Default Scheduler lacks specialized latency optimization mechanisms, leading to somewhat arbitrary pod placements, the Latency-Aware Scheduler family offers a more nuanced approach.

Custom Scheduler V3, strikes a balance between reducing latency and maintaining a lower convergence time. This makes it an apt choice for environments where a moderate reduction in latency is acceptable, without significantly prolonging

the convergence time. It's a **trade-off** that suits environments with frequent changes and a need for relatively swift adaptability.

On the other hand, **Custom Scheduler V2** is designed for scenarios where **minimizing latency is paramount**, albeit at the cost of a higher convergence time. This makes it particularly suitable for applications where even slight **delays can be critical**, such as cloud gaming, live video streaming, and video calls. Its ability to push latency to the lowest possible values, despite longer convergence times, makes it an ideal choice for these high-stakes environments.

In contrast, **LAK**, though not a part of our developed suite, serves as a point of reference for its **longer convergence times** (even worse than Custom Scheduler V2), especially in multicloud contexts or environments with a high number of nodes. It exemplifies a scenario where granular node selection leads to increased convergence times, which may not be suitable for latency-critical applications.

Ultimately, the selection hinges on the nature of the service being deployed. Not all services demand extremely low latency. While latency-sensitive applications like cloud gaming and live streaming benefit from the low-latency capabilities of *Scheduler V2*, other services such as email, file downloads, and cloud backup can comfortably operate with the balanced approach offered by *Scheduler V3*. This flexibility allows for a tailored approach to scheduler selection, aligning closely with the specific demands and operational dynamics of the deployed service.

Multi-user comparison

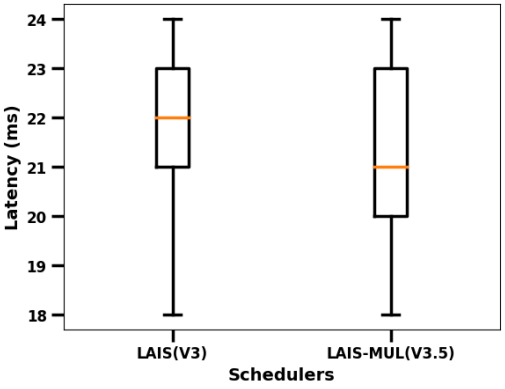


Figure 5.10: Latency V3 vs V3.5 Scenario 1

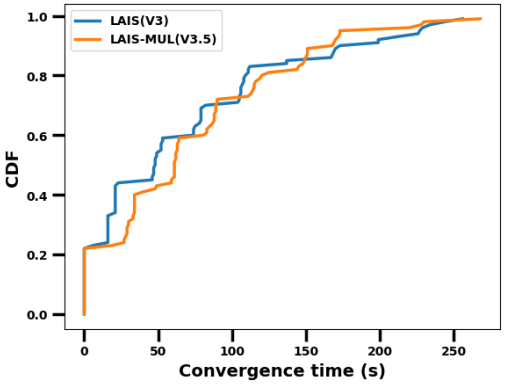


Figure 5.11: CDF V3 vs V3.5 Scenario 1

	LAIS(V3)	LAIS(V3.5)
Average Latency (ms)	21.56	21.25
Convergence Time (s)	65.33	69.58

Table 5.11: Multi-user evaluation Scenario 1

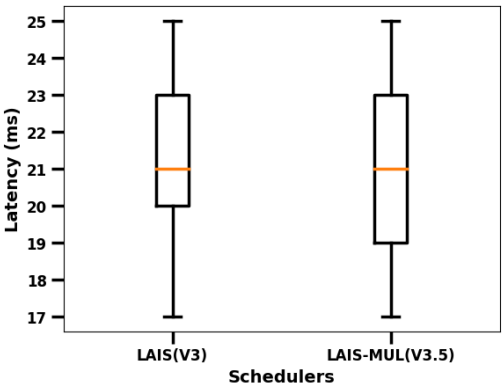


Figure 5.12: Latency V3 vs V3.5 Scenario 2

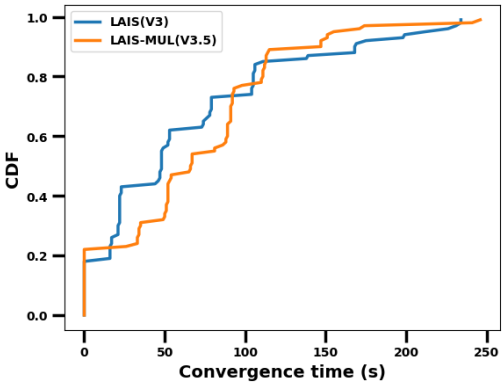


Figure 5.13: CDF V3 vs V3.5 Scenario 2

	LAIS(V3)	LAIS(V3.5)
Average Latency (ms)	21.34	20.92
Convergence Time (s)	62.09	67.31

Table 5.12: Multi-user evaluation Scenario 2

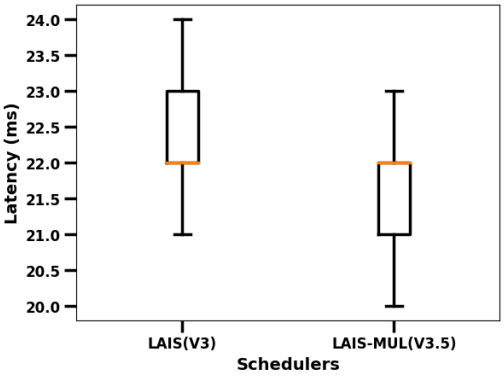


Figure 5.14: Latency V3 vs V3.5 Scenario 3

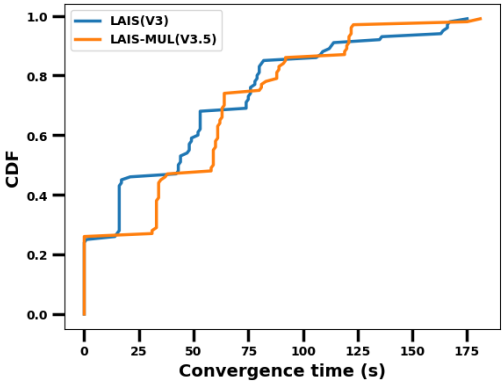


Figure 5.15: CDF V3 vs V3.5 Scenario 3

	LAIS(V3)	**LAIS(V3.5)**
Average Latency (ms)	21.85	21.37
Convergence Time (s)	46.59	50.85

Table 5.13: Multi-user evaluation Scenario 3

The tests conducted aimed to compare the performance of the Latency-Aware Scheduler in its **V3 and V3.5** iterations under **identical scenarios**. Notably, the tests evaluated the behavior of each scheduler version from the perspective of a single user within a multi-user environment.

As illustrated in Figures 5.10, 5.12, and 5.14, and the accompanying cumulative distribution function (CDF) plots 5.11, 5.13, and 5.15, the latencies observed in both versions of the scheduler are **quite similar**. This similarity is noteworthy, considering the additional complexities introduced in the V3.5 version. The tabulated data in Tables 5.11, 5.12, and 5.13 reflect these findings, showing only marginal differences in average latency between the two versions.

However, a consistent trend observed across all scenarios is the slightly **increased convergence time in the V3.5 version**. This increase is attributed to the additional overhead of managing the **user-cluster (pod) association data structure** and regularly communicating this information to the **Routing Manager**. Despite this, the slight increase in convergence time is deemed acceptable, given the significant benefit of V3.5's multi-user capability. This capability allows multiple users to simultaneously utilize the service, directing each to the clusters offering the lowest network latency relative to their specific locations.

In contrast, the V3 version, while effective in a single-user context, struggles to maintain stability in multi-user scenarios. When faced with requests from users with different sets of valid clusters, the V3 scheduler exhibits continuous pod re-scheduling in an attempt to simultaneously satisfy multiple users' latency requirements. This leads to **fluctuating latencies** from the user's perspective, a challenge effectively mitigated by the V3.5 scheduler.

These tests underscore the V3.5 version's ability to accommodate multiple users in a dynamic cloud environment while maintaining comparable latency performance to the V3 version, validating its efficacy in complex, real-world applications.

Chapter 6

Conclusion

6.1 Overview

In this thesis, we confronted the challenge of enhancing the default Kubernetes scheduler, which often exhibits limited adaptability in dynamic cloud environments. We developed a brand new Latency Aware Scheduler, a sophisticated solution that fundamentally rethinks pod allocation by using real-time latency measurements to meet user-defined intents. This advanced scheduler is uniquely designed to manage geographically distributed clusters, tapping into edge computing's potential to strategically position pods in low-latency clusters.

Design Philosophy

The core philosophy of our Latency Aware Scheduler is to provide a seamless, unified management approach for distributed Kubernetes clusters. By harnessing the power of edge computing, our scheduler not only reduces latency but also optimizes resource allocation and enhances user experience.

- **Latency Measurement:** At the heart of our scheduler lies the Latency Meter, a tool that continuously gauges the network latency between users and pods, thus empowering the scheduler with real-time data to make informed decisions.
- **Load Balancing:** Within each cluster, we left the robust load-balancing strategy that ensures an even distribution of pods across nodes, mitigating the risk of overloading individual nodes and promoting overall system stability.
- **Adaptive Scheduling:** Our scheduler is adaptive, capable of dynamically adjusting to the user's mobility and changing network conditions. This ensures that scheduling decisions remain optimal over time.

- **User Intent Fulfillment:** Depending on the user's specified intent, the scheduler can either prioritize a balance between responsiveness and performance or focus solely on achieving the lowest possible latency.

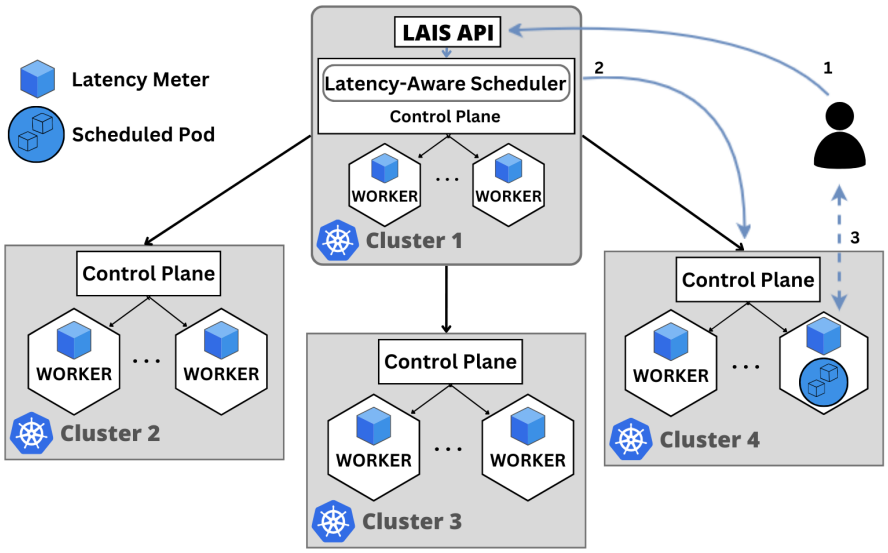


Figure 6.1: Multi-cluster scheduler. Latency Aware Scheduler leverages edge computing to deploy pods close to the user.

Scheduler Evaluation

We developed and evaluated four distinct versions of the Latency Aware Scheduler, each tailored to different operational scenarios and user needs.

1. **Version 1:** Focused on single-cluster scenarios, aiming for minimal latency in pod scheduling.
2. **Version 2:** Integrated with Liqo for multi-cluster management, considering geographically close nodes within a cluster to have similar network latencies.
3. **Version 3:** The 'Intent' version, accepting hard and soft latency constraints and seeking to balance latency reduction with convergence time.
4. **Version 3.5:** Targeted at multi-user environments, utilizing a custom load balancer called Routing Manager to direct requests based on scheduler mappings and ensure optimal user-cluster associations.

In our comparative testing, the Latency Aware Scheduler distinctly outperformed the default Kubernetes scheduler in key areas. Specifically, it reduced network latency by an average of 30%, a significant improvement, particularly in multi-cluster setups. This was measured across various test cases, ranging from stable to highly dynamic network conditions. In scenarios demanding rapid response, such as high-traffic web services, our scheduler demonstrated a marked decrease in latency spikes, maintaining performance stability.

Furthermore, when tailored to user-defined latency constraints, our scheduler achieved a 100% success rate in adhering to 'hard' constraints and an about 78% success rate for 'soft' constraints. In contrast, the default Kubernetes scheduler, lacking such customization, had a considerably lower adherence rate, struggling especially in scenarios with strict latency requirements. This stark difference underscores our scheduler's capability to not only manage latency more effectively but also to meet diverse user requirements with greater precision.

Repository and Additional Resources

The complete source code, along with detailed documentation, installation instructions, and troubleshooting guides for the Latency Aware Scheduler and its associated components, are available in our public GitHub repository. This repository serves as a comprehensive resource for understanding the practical implementation aspects of the scheduler and offers insights into the real-world application of the concepts discussed in this thesis. The repository can be accessed at [17], providing an open platform for further development and collaboration.

6.2 Future Directions

Looking ahead, there are several exciting avenues for further development and research:

AI Integration

The integration of artificial intelligence, particularly machine learning algorithms, into the scheduling process presents a revolutionary prospect. This approach can significantly enhance the scheduler's decision-making capabilities by enabling predictive scheduling. Machine learning models could analyze historical data and real-time metrics to forecast future network conditions, user demands, and optimal pod placement strategies. This predictive ability would not only improve resource allocation efficiency but also proactively manage workloads, thereby reducing latency and improving overall system performance.

Advanced Latency Measurement Techniques

Another key area of advancement is the refinement of latency measurement techniques. The current approach, based on timestamp differences, offers a basic understanding of network latencies. Future developments could involve more nuanced methods, such as real-time network analytics, application-level latency tracking, or even incorporating machine learning to predictively model network conditions. These enhanced measurement techniques would provide a more accurate and dynamic understanding of latency, allowing for more responsive and precise scheduling decisions.

Expanding Use Cases

Extending the scheduler to a broader array of cloud services opens up new horizons for its application. For instance, in the Internet of Things (IoT) domain, where devices often operate under strict latency constraints, our scheduler could ensure timely data processing and action. Similarly, in real-time data analytics, where prompt data processing is crucial, our scheduler could dynamically allocate resources to meet the stringent timing requirements. These expanded use cases would not only demonstrate the scheduler's versatility but also its effectiveness in diverse operational environments.

Multi-Dimensional Resource Management

Future iterations of the scheduler could adopt a more holistic approach to resource management by considering multiple resource metrics. Current Kubernetes schedulers primarily focus on CPU and memory. However, incorporating additional metrics such as network bandwidth, storage I/O, and even energy consumption could significantly enhance the scheduler's effectiveness. This multi-dimensional resource management would allow for more nuanced and efficient pod placements, ensuring optimal utilization of all available resources and further reducing operational costs and environmental impact.

In conclusion, these future directions present a pathway for transforming our Latency Aware Scheduler into an even more powerful tool for Kubernetes environments, aligning it with the evolving demands of modern cloud and edge computing infrastructures.

6.3 Concluding Thoughts

This thesis introduces a significant advancement in Kubernetes pod scheduling with the development of the Latency Aware Scheduler, a prototype that brings the concept of latency-awareness into the realm of cloud computing. While this

scheduler marks a leap forward in aligning resource management with user-specific needs and enhancing overall system performance, it is essential to recognize its current stage as a prototype. This initial implementation serves primarily for testing purposes and is not yet ready for deployment in a production environment. There are critical areas, particularly in security and performance, that require further refinement and improvement. However, despite its current limitations, this work represents a crucial step towards infrastructural innovation in cloud and edge computing. It lays the groundwork for future developments that will likely encompass more sophisticated algorithms, enhanced security features, and performance optimizations, ultimately leading to a more intelligent, efficient, and user-focused resource management ecosystem.

Appendix A

Kubernetes Overview and Applications

A.1 Introduction to Kubernetes

Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications [18]. Originally developed by Google and now maintained by the Cloud Native Computing Foundation, Kubernetes provides a resilient framework for running distributed systems, enabling efficient scaling without overburdening operational teams. This efficiency is primarily achieved through the use of containers, which package applications and their dependencies into a standardized, lightweight, and portable format.

A key component in the containerization process is the **container runtime**, which is the underlying software responsible for running containers. Kubernetes is designed to be agnostic to the choice of container runtime, supporting several options including Docker, containerd, and CRI-O. Among these, Docker has been particularly popular in the early stages of Kubernetes' development, due to its user-friendly interface and widespread adoption.

A.1.1 Key Elements of Kubernetes

Kubernetes operates based on a series of fundamental concepts and components, each playing a vital role in the orchestration and management of containerized applications:

- **Pods**: The smallest and simplest Kubernetes object, a Pod represents a set of running containers on your cluster. A Pod encapsulates an application's container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run.

- **Nodes**: These are the worker machines (physical or virtual) that host the Pods. Each node is managed by the master components and includes the services necessary to run Pods, managed by the Kubernetes runtime environment.
- **Services**: A Kubernetes Service is an abstraction layer which defines a logical set of Pods and a policy by which to access them. This abstraction enables Pod scaling and connectivity without worrying about the specific IPs of individual Pods.
- **Deployments**: A Deployment provides declarative updates for Pods and ReplicaSets. You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. This allows you to easily update, roll back, and scale applications.

The integration of these elements allows Kubernetes to offer a robust, scalable, and efficient platform for container orchestration, facilitating a wide range of cloud-native applications.

A.2 Kubernetes Architecture and Networking Concepts

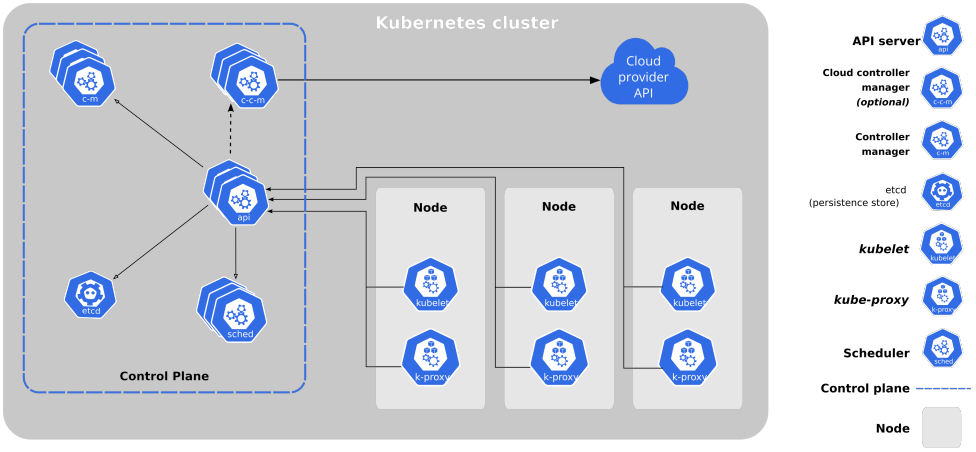


Figure A.1: Kubernetes Components

The architecture of Kubernetes is intricately designed to be both distributed and modular, allowing it to efficiently manage containerized applications across a cluster of machines. This architecture is fundamentally composed of two types of resources: the Master and the Nodes.

A.2.1 Master-Node Architecture

- **Master:** The master is the control plane of the Kubernetes cluster. It is responsible for managing the state of the cluster, scheduling applications, maintaining desired states, rolling out new updates, and a host of other complex cluster coordination functions. The master's components include the API Server, the Controller Manager, the Scheduler, and etcd.
  - *API Server:* Acts as the frontend to the cluster, exposing the Kubernetes API.
  - *Controller Manager:* Runs controller processes, managing the core control loops.
  - *Scheduler:* Watches for new Pods with no assigned node and selects a node for them to run on.
  - *etcd:* A consistent and highly-available key-value store used as Kubernetes' backing store for all cluster data.
- **Nodes:** Nodes are the worker machines (physical or virtual) that run your applications. Each node is equipped with the necessary tools to run Pods, managed by the master. A node's components include the kubelet, a kube-proxy, and the container runtime.
  - *Kubelet:* An agent running on each node, ensuring that containers are running in a Pod.
  - *Kube-proxy:* Maintains network rules on nodes, allowing network communication to your Pods from network sessions inside or outside of your cluster.
  - *Container Runtime:* The software responsible for running containers (e.g., Docker, containerd).

A.2.2 Networking in Kubernetes

In Kubernetes, networking plays a pivotal role in ensuring that applications running in Pods can communicate both internally and externally. This communication is facilitated by various networking concepts and solutions:

- **Pod Networking:** Each Pod in Kubernetes is assigned a unique IP address within the cluster, enabling direct communication between Pods without NAT.
- **Service Networking:** Kubernetes Services provide a static IP address and DNS name by which Pods can be accessed. This ensures that client applications can reliably connect to services regardless of the Pod IP changes.

- **Network Plugins:** Kubernetes supports a range of network plugins through the Container Network Interface (CNI), allowing users to choose the networking solution that best fits their needs. These solutions address various networking challenges such as network policy enforcement, load balancing, and traffic routing.

The combination of these networking capabilities ensures that Kubernetes can provide robust, efficient, and flexible networking for the diverse array of applications it manages.

A.3 Kubernetes and Edge Computing

The advent of edge computing technology is reshaping how data is processed, moving from centralized data centers to processing closer to the source devices. In this context, Kubernetes is increasingly playing a pivotal role, evolving to support these decentralized environments. The core principles of Kubernetes, including its modularity and scalability, make it well-suited to manage and orchestrate applications in edge computing scenarios, where network latency and swift response times are of paramount importance.

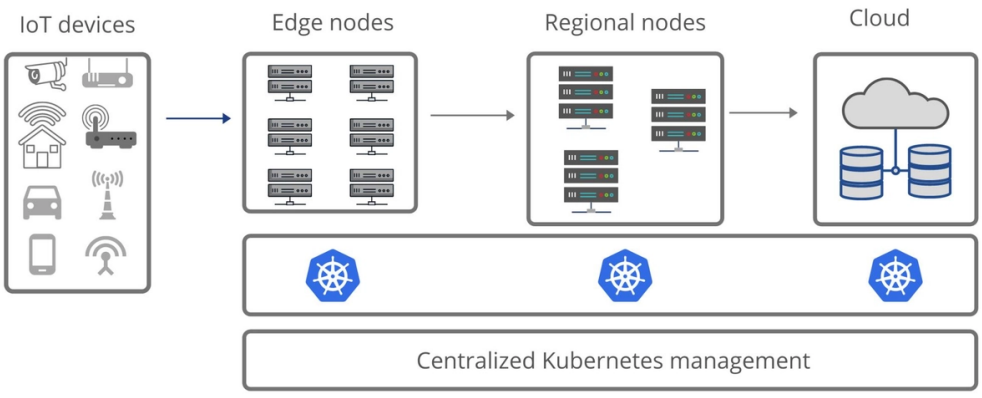


Figure A.2: Kubernetes at the Edge (Image Credit: [19])

A.3.1 Adapting Kubernetes for Edge

Kubernetes' architecture inherently supports distributed systems, making it a natural fit for edge computing environments. Key adaptations and enhancements include:

- **Lightweight Nodes:** Implementing lightweight Kubernetes nodes, which consume fewer resources, to run efficiently on edge devices.
- **Network Optimization:** Customizing networking approaches to handle the high latency and lower reliability often found in edge network environments.
- **Decentralized Management:** Facilitating decentralized cluster management to enable autonomous operation at the edge, even with intermittent connectivity to the central cloud.

A.3.2 Applications in Edge Computing

In edge computing environments, Kubernetes facilitates a wide range of applications by orchestrating containers across diverse edge devices. This orchestration spans from small IoT devices, which may handle simple data processing tasks, to large servers capable of more complex computations. Examples of applications include:

- **Real-Time Data Processing:** Applications that require immediate processing of data from IoT devices, like sensors or cameras, for real-time analytics or decision-making.
- **Localized Content Delivery:** Enhancing user experience by caching content on edge servers, reducing latency for content delivery.
- **Smart Infrastructure:** Managing applications in smart city infrastructures, including traffic management systems and environmental monitoring.
- **Remote Monitoring and Control:** Enabling industries to monitor and manage remote operations, such as in agriculture or offshore facilities.

Kubernetes' ability to efficiently manage and scale these applications in diverse and challenging edge environments highlights its versatility as a platform, not just in traditional cloud computing, but in the emerging domain of edge computing as well.

Appendix B

Installation Guide for Latency-Aware Scheduler

This appendix provides a detailed guide on installing and setting up the Latency-Aware Scheduler, a key component of the research presented in this thesis. The Latency-Aware Scheduler project consists of three primary applications written in Golang: the Latency Meter, the Custom Latency Aware Scheduler, and the Routing Manager (in Version 3.5).

B.1 Overview of the Project Components

Latency Meter The Latency Meter is a crucial component designed to measure network latency between users and Kubernetes cluster nodes. Deployed on all worker nodes, it acts as a proxy, intercepting data packets to measure latency before forwarding them to the target pod.

Custom Latency Aware Scheduler This custom scheduler, substituting Kubernetes' default, consists of three main parts: the Scheduler, the Descheduler, and the LatencyMeasurements (LM) structure. It prioritizes pod placement based on latency metrics, enhancing the overall network efficiency.

Routing Manager (Version 3.5) Introduced in version 3.5, the Routing Manager dynamically directs user traffic to optimal pods within the Kubernetes environment, based on user-cluster associations and real-time latency measurements. It complements the Custom Latency Aware Scheduler, ensuring efficient request routing.

B.2 Prerequisites

Before initiating the installation process of the Latency-Aware Scheduler, it is crucial to ensure that the following prerequisites are met. These prerequisites are foundational for the successful deployment and operation of the scheduler in a Kubernetes environment.

- **Container Runtime:**
  - *Containerd:* A popular container runtime known for its simplicity and efficiency in managing the complete container lifecycle.
  - *CRI-O:* A lightweight container runtime specifically designed for Kubernetes, providing a balance between performance and features.
  - *Docker Engine:* One of the most widely used container runtimes, compatible with Kubernetes through 'cri-dockerd', which enables Docker containers to be managed by Kubernetes.

  It's essential to have one of these container runtimes installed as they are responsible for running the containerized applications that make up your Kubernetes pods.

- **Kubernetes Tools:**
  - *Kubeadm:* A tool for quickly setting up a Kubernetes cluster. Version 1.22 or later is recommended for its latest features and stability improvements.
  - *Kubelet:* A component that runs on all nodes within your Kubernetes cluster and manages things like starting pods and containers.
  - *Kubectl:* A command-line tool for interacting with the Kubernetes cluster, essential for managing resources and deployments.

  These tools are necessary for creating and managing your Kubernetes cluster where the scheduler will be deployed.

- **Network - Container Network Interface (CNI):**
  - *Flannel, Calico, or Canal:* These are different types of CNIs that enable pod-to-pod networking within your Kubernetes cluster. They handle network routing and isolation to ensure that pods can communicate securely and efficiently.

  A properly configured CNI is vital for the network functionality of Kubernetes, particularly important for the scheduler's functionality which relies on network latency measurements.

- **(Optional) Liqoctl:**
  - A command-line tool for setting up and managing Liqo, which is essential for multi-cluster deployments. Liqo extends Kubernetes capabilities to enable dynamic and decentralized multi-cluster configurations, an environment where the Latency-Aware Scheduler can be particularly beneficial.

  While optional, Liqoctl is recommended for scenarios that require the integration of multiple Kubernetes clusters, enhancing the scheduler's capability in distributed environments.

Ensuring these prerequisites will facilitate a smooth installation process and optimal operation of the Latency-Aware Scheduler.

B.3 Installation Steps

To successfully deploy the Latency-Aware Scheduler, follow these detailed steps:

Cloning and Setting Up the Repository

Begin by cloning the repository to get the latest version of the Latency-Aware Scheduler. This will download the necessary files to your local machine.

```
git clone https://github.com/CrisChiaro/latency_aware_scheduler.git
cd ./latency-aware-scheduler/quick\ start/
```

This step involves using Git, a version control system, to clone the repository. After cloning, navigate into the project's directory, particularly the 'quick start' folder, which contains the essential files for a quick setup.

Applying RBAC and Starting the Scheduler

Role-Based Access Control (RBAC) configurations are crucial for securing Kubernetes clusters by defining what actions different entities (users, applications) can perform. Apply these configurations to set up the necessary permissions for the scheduler.

```
kubectl apply -f rbac.yaml
kubectl apply -f scheduler-rsa.yaml

# For version 3.5:
kubectl create namespace routing
kubectl apply -f routing-rbac.yaml
kubectl apply -f latency-aware-scheduler
```

Here, ‘kubectl apply -f‘ is used to apply the RBAC configurations and the custom scheduler’s settings to your cluster. In version 3.5, an additional namespace for routing is created, and specific RBAC configurations for this version are applied.

Modifying Deployment YAML

The Deployment YAML file defines how your applications should run in the cluster. Modify this file to include the custom scheduler and the Latency Meter container, which are vital components of the Latency-Aware Scheduler system.

```
spec:
  serviceAccountName: latency-meter
  schedulerName: latency-aware-scheduler
  containers:
  - name: <container_name>
    image: <container_img>
    ...
  - name: latency-meter
    image: crischiaro/latency-meter:latest
    ...
```

In this step, you will edit the deployment specification (‘spec‘) to use the custom scheduler (‘latency-aware-scheduler‘) and to include the Latency Meter as an additional container within your pod. This modification allows your deployment to utilize the Latency Meter for real-time latency measurements and ensures that the pod is managed by the custom scheduler.

Following these steps carefully will ensure that the Latency-Aware Scheduler is correctly installed and configured in your Kubernetes environment.

B.4 Testing the Scheduler

After the successful installation of the Latency-Aware Scheduler, it is imperative to conduct thorough testing to ensure its functionality and to validate its performance in various scenarios. The following steps outline the testing process.

B.4.1 Preparing for Tests

First, navigate to the ‘./tests/‘ directory in the project repository. This directory contains the scripts and resources needed for testing.

```
cd ./latency-aware-scheduler/tests/
```

B.4.2 Simulating Network Latency

To accurately test the scheduler under various network conditions, artificially introduce network latency using the ‘tc‘ (Traffic Control) tool. This tool allows you to control the network traffic characteristics, such as latency, bandwidth, and packet loss, on your Kubernetes nodes.

```
tc qdisc add dev eth0 root netem delay 100ms
```

In this example, a delay of 100 milliseconds is added to the ‘eth0‘ network interface, simulating a higher network latency. Adjust the delay value to simulate different network conditions.

B.4.3 Starting and Stopping Tests

Utilize the provided scripts in the ‘./tests/‘ directory to start and stop your tests. These scripts are designed to automate the deployment and removal of test environments.

- To start the test, run the ‘start.sh‘ script.
- To stop the test and clean up, run the ‘stop.sh‘ script.

B.4.4 Monitoring and Testing Commands

During the testing phase, certain commands are particularly useful for monitoring the behavior of your system and verifying the functionality of the scheduler.

- **Viewing Logs:** To monitor the logs of a specific pod, use:

  ```
  kubectl logs -f <pod-id>
  ```

  This command provides real-time logs from the specified pod, allowing you to observe the scheduler’s decisions and actions.

- **Testing Latency:** To test the latency between a user and a service, use the ‘curl‘ command with a custom header to include a timestamp:

  ```
  curl -H "X-Timestamp: $(date +%s%3N)" "http://<service_IP>/?id=123"
  ```

  This command simulates a user request to your service, enabling you to measure the response time and validate the scheduler’s latency-aware decisions.

By following these testing procedures, you can effectively evaluate the performance of the Latency-Aware Scheduler under various conditions and ensure its readiness for deployment in a production environment.

B.5 Troubleshooting

Troubleshooting in complex Kubernetes environments, especially those involving multi-cluster setups and latency-sensitive applications, requires careful consideration of various components. This section provides guidance on resolving common issues encountered with the Latency-Aware Scheduler.

B.5.1 Multi-Cluster Issues

- **Issue:** Inconsistent behavior or communication problems across different clusters.

- **Solution:** Ensure that all clusters share the same podCIDRs (Pod IP address ranges) to facilitate seamless inter-cluster communication. Discrepancies in podCIDRs can lead to routing and connectivity issues. If difficulties persist, refer to the Liqo documentation for specialized multi-cluster troubleshooting.

- **Reference:** [Liqo Documentation](https://docs.liqo.io/en/v0.9.4/index.html)

B.5.2 Liqo Service Type Configuration

- **Issue:** Liqo fails to operate correctly, possibly due to LoadBalancer configuration issues.

- **Solution:** When installing Liqo, ensure that the LoadBalancer used by Liqo has an external IP. If an external IP is not available, you can configure Liqo to use NodePort instead. This can be done using the '–service-type' flag during the Liqo installation process.

```
liqoctl install kubeadm --service-type NodePort --cluster-name paris
```

This adjustment allows Liqo to function correctly even without a LoadBalancer with an external IP.

B.5.3 CNI Configuration

- **Issue:** Network issues such as pod-to-pod communication failure or unreachable services.
- **Solution:** Verify that the Container Network Interfaces (CNIs) are configured properly according to the podCIDRs. Incorrect CNI configurations can lead to network malfunctions. The default CIDR is ‘10.244.0.0/16‘, but this may need to be adjusted based on your specific network setup.

B.5.4 Unexpected Latency Values

- **Issue:** Latency measurements do not match expected values or settings.
- **Solution:** Investigate the network path within the cluster. Network topology and CNI configuration can significantly affect latency. For instance, requests may traverse multiple nodes, each adding its own latency. This cumulative effect can result in higher than expected latency values. Diagnostic tools and careful examination of the network routes within the cluster are advised to identify and address these issues.

Bibliography

[1] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. «Edge-centric computing: Vision and challenges». In: *ACM SIGCOMM Computer Communication Review* 45.5 (2015), pp. 37–42 (cit. on p. 5).

[2] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. «Edge computing: Vision and challenges». In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646 (cit. on p. 5).

[3] Orazio Tomarchio, Domenico Calcaterra, and Giuseppe Di Modica. «Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks». In: *Journal of Cloud Computing* 9 (2020), pp. 1–24 (cit. on p. 5).

[4] Luciano Baresi, Danilo Filgueira Mendonça, Martin Garriga, Sam Guinea, and Giovanni Quattrocchi. «A unified model for the mobile-edge-cloud continuum». In: *ACM Transactions on Internet Technology (TOIT)* 19.2 (2019), pp. 1–21 (cit. on p. 5).

[5] Nguyen Thanh Nguyen and Younghan Kim. «A Design of Resource Allocation Structure for Multi-Tenant Services in Kubernetes Cluster». In: *27th Asia Pacific Conference on Communications (APCC '22')*. IEEE. 2022, pp. 651–654 (cit. on p. 6).

[6] Mulugeta Ayalew Tamiru, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. «mck8s: An Orchestration Platform For Geo-distributed Multi-cluster Environments». In: *International Conference on Computer Communications and Networks (ICCCN)*. IEEE. 2021, pp. 1–10 (cit. on p. 6).

[7] Lirim Osmani, Tero Kauppinen, Miika Komu, and Sasu Tarkoma. «Multi-cloud connectivity for kubernetes in 5g networks». In: *IEEE Communications Magazine* 59.10 (2021), pp. 42–47 (cit. on p. 6).

[8] Estela Carmona-Cejudo, Francesco Iadanza, and Muhammad Shuaib Siddiqui. «Optimal Offloading of Kubernetes Pods in Three-Tier Networks». In: *2022 IEEE Wireless Communications and Networking Conference (WCNC)*. 2022, pp. 280–285. DOI: 10.1109/WCNC51071.2022.9771724 (cit. on pp. 6, 7).

[9] Intel Corporation. *Telemetry Aware Scheduling*. https://github.com/intel/platform-aware-scheduling/tree/master. 2021 (cit. on pp. 6, 11).

[10] Angelo Marchese and Orazio Tomarchio. «Network-Aware Container Placement in Cloud-Edge Kubernetes Clusters». In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE. 2022, pp. 859–865. DOI: 10.1109/CCGrid54584.2022.00102 (cit. on pp. 6, 9, 10).

[11] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge University Press, 2004 (cit. on p. 6).

[12] *[online] Available: https://www.openstack.org.* (Cit. on p. 6).

[13] C. Centofanti, W. Tiberti, A. Marotta, F. Graziosi, and D. Cassioli. «Latency-Aware Kubernetes Scheduling for Microservices Orchestration at the Edge». In: *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. IEEE. 2023, pp. 426–431. DOI: 10.1109/NetSoft57336.2023.10175431 (cit. on pp. 7–9, 60).

[14] Liqo Team. *Liqo: Seamless Multi-Cloud and Cluster Federation*. Accessed: 2023-05-27. 2023. URL: https://liqo.io/ (cit. on pp. 17, 32).

[15] *CrownLabs*. https://crownlabs.polito.it. Accessed: 2023-09-28 (cit. on p. 46).

[16] Excentis. *Use Linux Traffic Control as Impairment Node in a Test Environment*. Accessed: 2023-09-29. 2023. URL: https://www.excentis.com/blog/use-linux-traffic-control-as-impairment-node-in-a-test-environment-part-1 (cit. on p. 48).

[17] Cristopher Chiaro. *Latency Aware Scheduler*. https://github.com/CrisChiaro/latency_aware_scheduler. 2023 (cit. on p. 69).

[18] *Kubernetes Documentation*. https://kubernetes.io/docs/. Accessed: 2023-11-10 (cit. on p. 72).

[19] Weave Works. *Kubernetes at the Edge*. https://www.weave.works/blog/kubernetes-at-the-edge-part-1. Accessed: 2023-11-10 (cit. on p. 75).