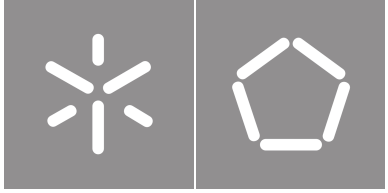
Universidade do Minho
Escola de Engenharia

João Pedro Machado Vilaça

Orchestration and Distribution of Services in Hybrid Cloud/Edge Environments

December, 2021



Universidade do Minho
Escola de Engenharia

João Pedro Machado Vilaça

Orchestration and Distribution of Services in Hybrid Cloud/Edge Environments

Master Thesis
Integrated Master's in Informatics Engineering

Work developed under the supervision of:
Ricardo Manuel Pereira Vilaça
João Tiago Medeiros Paulo

December, 2021

Acknowledgements

First and foremost, I want to thank my supervisors, Dr. João Paulo and Dr. Ricardo Vilaça. Through their support and review, they made all this work possible, steering me in the right direction all along the way. They provided invaluable feedback in all stages of both this thesis and related scientific paper.

I also want to leave my deepest gratitude to my parents and girlfriend. They always encouraged and supported me unconditionally throughout my academic journey and especially during the process of researching and writing this thesis. Without the motivation and hope they gave me, this work would not have been possible.

Finally, a thank you note to all my friends and colleagues. Together we always pushed each other to learn more and be better. Thank you for all the good moments throughout the years, from nights outs, dinners, hackathons, among many others.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

Braga, March 11, 2022

(Place) (Date)

Assinado por: **João Pedro Machado Vilaça**
Num. de Identificação: 15654201
Data: 2022.03.28 22:56:20+01'00'

CHAVE MÓVEL

(João Pedro Machado Vilaça)

"For pure joy, I look at a small painting by Arbit Blatas. An ocean liner is at the center of the composition, perhaps ready to depart. It holds the promise of discovery." (António Damásio)

Abstract

Orchestration and Distribution of Services in Hybrid Cloud/Edge Environments

The Edge Computing paradigm aims at leveraging the computational and storage capabilities of Internet of Things (IoT) devices, while resorting to Cloud Computing services for more demanding processing tasks that cannot be done at commodity devices. However, deploying distributed services across Edge and Cloud nodes raises new challenges that must be addressed. Namely, the choice of what nodes run each service component may be critical for ensuring an efficient service for users. For example, if two critical components, that must frequently exchange data, are placed in different geographic locations, the whole performance of the service will be affected. Therefore, these geographically dispersed environments demand new orchestration and distribution systems for hybrid Cloud and Edge environments, based on geographic location, service demand, business objectives, laws, and regulations.

This thesis proposes Geolocate, a generic scheduler for workload orchestration and distribution across heterogeneous and geographically distant nodes. In more detail, it provides the design and implementation of a scheduling and placement algorithm based on nodes' geographic location and resource availability and a fully functional prototype, integrating Geolocale with KubeEdge, an edge computing orchestration platform based on Kubernetes.

The experimental results show that as the network latency and amount of data being transmitted between nodes increases, so does the response time for applications resorting to these distributed deployments. Our evaluation of an e-commerce application shows that the use of Geolocate can reduce, relative to KubeEdge's default-scheduler, the average response time for requests by about 85%.

Keywords: scheduling, edge computing, containers, kubeedge

Resumo

Orquestração e Distribuição de Serviços em ambientes híbridos Cloud/Edge

O paradigma da Computação na Borda visa alavancar as capacidades computacionais e de armazenamento dos dispositivos Internet of Things (IoT), ao mesmo tempo que recorre aos serviços de Computação em Nuvem para tarefas de processamento mais exigentes que não podem ser feitas em dispositivos comuns. No entanto, a implementação de serviços distribuídos através de nós na Nuvem e na Borda levanta novos desafios que devem ser resolvidos. Nomeadamente, a escolha dos nós que executam cada componente do sistema pode ser fundamental para assegurar um serviço eficiente para os utilizadores. Por exemplo, se dois componentes críticos, que devem frequentemente trocar dados, forem colocados em localizações geográficas diferentes, todo o desempenho do serviço será afectado. Assim sendo, estes ambientes geograficamente dispersos necessitam de novos sistemas de orquestração e distribuição para ambientes híbridos de *Cloud* e *Edge*, com base na localização geográfica, utilização dos serviços, objectivos empresariais, leis, e regulamentos.

Esta tese propõe o sistema Geolocate, um *scheduler* genérico para orquestração e distribuição de cargas de trabalho em nós heterogéneos e geograficamente distantes. Em detalhe, esta tese fornece o design e implementação de um algoritmo de *scheduling* baseado na localização geográfica dos nós e na disponibilidade de recursos, e ainda um protótipo totalmente funcional, integrando Geolocale com KubeEdge, uma plataforma de orquestração computacional de borda baseada em Kubernetes.

Os resultados experimentais mostram que à medida que a latência da rede e a quantidade de dados transmitidos entre nós aumenta, aumenta também o tempo de resposta das aplicações que recorrem a estas implantações distribuídas. A nossa avaliação de uma aplicação de *e-commerce* mostra que a utilização de Geolocate pode reduzir, relativamente ao *scheduler* por defeito de KubeEdge, o tempo médio de resposta aos pedidos em geral em cerca de 85%.

Palavras-chave: escalonamento, computação em borda, containers, kubeedge

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Despite being a reasonably recent trend, microservices are gaining increasing popularity in the industry. In the last few years, with the advance of technological development in small devices with processing ability and in the quality of internet networks, we have observed an increase in popularity and higher market adoption of microservices architectures. Both in terms of new products, entirely built based on a set of small intercommunicating services from the ground up, but also in terms of companies migrating old monolithic applications to this architecture, microservices are an indisputable reality in the current software development landscape [9, 50].

In this context, the microservices architectures provide an environment with the perfect conditions for the Internet of Things (IoT) to proliferate. An IoT system consists of a set of interconnected heterogeneous devices of low-power computing technology to connect and exchange data with other devices and systems over the Internet [56]. Over the past few years, we have witnessed an innovation revolution in IoT. It is predicted that by 2025, in the industrial area, 50% of companies will be using IoT solutions to improve operations by generating and interpreting data in factories, a significant increase from the registered 10% in 2020 [45]. But IoT cuts across numerous areas with a wide range of possible applications. In medicine, the use of health wearables or monitoring chips in organs. In the areas of smart cities and smart agriculture, with the implementation of climate monitoring systems, soil quality, road traffic, among others [53].

However, as the amount of digital data being generated by IoT devices increases, it is imperative to find scalable and resilient services that can store and process such information. Cloud Computing emerges as a natural solution given the wide variety of available services for storing and processing large quantities of data efficiently since most cloud providers offer a wide variety of on-demand services and have virtually unlimited availability of resources [28]. However, transferring data between IoT devices (producers) and cloud services (consumers) has a non-negligible cost and performance impact for data-centric applications [26]. This cost becomes even more significant for scenarios where IoT devices are deployed at remote geographic locations, which are far away from the cloud data centers. The success of this type of service depends on their base systems to ensure geographic and temporal accessibility, with reliability, efficiency, and scalability [6].

The concept of Edge Computing emerged to leverage the capabilities of the Cloud while reducing the

high latencies and security flaws inherent to the usage of cloud solutions [23], but seeking to bring their advantages closer to the users, improving the quality of services, reducing energy and operational costs. In more detail, the Edge Computing paradigm aims at the maximization of the computational and storage capabilities of IoT devices while resorting to Cloud Computing services for more demanding processing tasks that cannot be done at commodity devices. However, deploying distributed services across edge and cloud nodes raises new challenges that must be addressed. Namely, the choice of what nodes run each service component may be critical for ensuring an efficient service for users [59, 43]. For example, if two critical components that must frequently exchange data are placed in different geographic locations, the whole performance of the service will be affected.

Another challenge is the heterogeneity problem. IoT systems are usually formed by a composition of several small heterogeneous devices with limited resources. Virtualization and orchestration technologies such as Kubernetes [30], widely used in industry, are excellent for managing heterogeneous distributed services running at heterogeneous cloud nodes [10]. Kubernetes allows the distribution and management of various workloads across a wide variety of nodes. But there is still a problem, Kubernetes is built for cloud infrastructures, thus not yet being prepared to accommodate other processing units, such as edge devices.

1.1 Problem

KubeEdge was created to extend cloud orchestration solutions and manage containerized applications at the edge, providing infrastructure support for network, application deployment, and metadata synchronization between Cloud and Edge [29]. Based on Kubernetes, KubeEdge implements add-ons on the original cluster to handle edge networking, node management, and data communication. However, KubeEdge provides simple scheduling algorithms that only take into account the available resources of nodes when deploying service components at these. When cloud and edge nodes are running in close geographic locations, the previous scheduling decisions are sufficient. However, when these nodes are far away from each other and must frequently exchange data, the service's latency is significantly affected and, consequently, the experience of users.

We can consider the case of a real-time street data processing service, with sensors that produce large amounts of information. Firstly, in terms of performance, the processing workloads must be close to the data sources, which will decrease latencies and the pressure of data communication over the network, increasing service response times. Other than that, legal compliance is an increasingly important issue, the generated data may fall under specific data protection and privacy laws, such as the General Data Protection Regulation 2016/67 (GDPR) [21], which limits the transfer of personal data outside the European Union and European Economic Area.

1.2 Goal and Contributions

The main goal of this thesis, for these geographically dispersed environments, is to explore, design, and implement new orchestration and distribution systems for hybrid cloud and edge environments, based on geographic location, service demand, business objectives, laws, and regulations. Throughout this work, we aim to achieve substantial improvements in scalability and quality of service levels by taking advantage of Edge's computational resources. In more detail, the proposed solution must be able to handle various heterogeneous software and hardware environments and reliably ensure its performance requirements. In particular, the protocol must be able to establish processing units on cloud or edge nodes, according to the nature of the computation, the type and geographical location of the data.

To address the previous goals, in this work we propose Geolocate, a generic scheduler for workload orchestration and distribution across heterogeneous and geographically distant nodes. Namely, this work presents the design and implementation of an extensible scheduling engine that can be integrated and used with different orchestration or cluster management tools. Also, we provide the implementation of a scheduling algorithm based on the nodes' geographic location and resource availability. As another contribution, Geolocate's scheduling engine is integrated with the KubeEdge orchestration solution to provide a fully functional prototype.

The evaluation results of our fully-fledged prototype show that, under normal cluster conditions, where data-producing workloads are relatively stable, Geolocate performs well in data processing times and service response times. By reducing latency between data-producing and data-processing services, Geolocate is able to decrease overall service response times by up to 85%, as exemplified for a real e-commerce application.

1.3 Results

The work developed during this thesis resulted in the publication of a scientific paper in the workshop on High-Performance and Reliable Big Data (HPBD'21) that was held co-located with the 40th International Symposium on Reliable Distributed Systems (SRDS 2021) and is available at https://dsr-haslab.github.io/repository/vpv21.pdf.

- Vilaça J, Paulo J, Vilaça R. Geolocate: A geolocation-aware scheduling system for Edge Computing. Workshop on High-Performance and Reliable Big Data (HPBD), colocated with SRDS.

This thesis was done in the context of the Large Scale Collaborative Project of CMU Portugal, AIDA (https://aida.inesctec.pt/). The full-fledged prototype, including Geolocate's scheduling engine and integration with KubeEdge, was integrated into the orchestration and deployment component of the project, and is available at https://github.com/geolocate-orchestration.

1.4 Document Structure

Initially, in Chapter 2, we present some background studies related to the Internet of Things and Edge Computing. Next, we explore the state of the art related to the design and architecture of IoT environments. Then we analyze the current scheduling and resource allocation problems in hybrid cloud and edge environments and some of the existing solutions. We then study the most used orchestration and clustering systems in Edge Computing environments.

In Chapter 3, we first illustrate the design principles of Geolocate, presenting the problems it is set to solve. Then we describe the implementation architecture used, showing the internal components of the proposed scheduling system, their role, and their relationship with other elements.

In Chapter 4, we discuss implementation details related to the scheduling system, for example, node and algorithm management. Finally, we discuss how we performed the scheduler's integration with KubeEdge.

In Chapter 5, we demonstrate and evaluate the results of tests to validate the utility and functionality of our prototype. Then we also present and discuss the results of performance tests of a demo online sock shopping application scheduled using Geolocate.

Finally, in conclusion (Chapter 6), we perform a general summary of our work and the objectives achieved, and we discuss the contributions made and results achieved. We also discuss the solution's applicability and future work.

2

State of the Art

This chapter summarizes and presents the findings of our literature review. Initially, it provides a background on computing paradigms, their designs and architectures, evolution, and use cases. Next, we present the findings on research related to resource allocation and scheduling mechanisms in hybrid cloud and edge environments and orchestration systems for these same environments.

2.1 Background

IoT devices are simple sensors and actuators accessible over the Internet with small processing power. They produce large amounts of data that must be sent to servers in remote data centers for processing. Typically, IoT devices do not make intelligent decisions or take autonomous actions. Despite being smaller than cluster servers, edge computing devices have considerable processing power and can have sensors and actuators physically attached. These new devices enable the migration of processing tasks, previously performed in centralized data centers, closer to the edge or in the edge itself, which in turn results in more data examined with faster response times. Decision-making is thus moved closer to where actions need to occur.

2.1.1 Internet of Things

The Internet of Things (IoT) is an architectural paradigm that defines a vast network of small electronic devices capable of communicating with each other. Transversal to all areas such as industry, agriculture and livestock, and the end consumer, IoT has an impact in our daily life.

Three broad categories have emerged to characterize the types of IoT-related applications [35]. First, «Monitoring and Control», related to metrics collection and data from various systems. For example, industrial production lines where applications collect energy consumption and production performance data, or an agricultural field where weather conditions and soil moisture is measured and analyzed.

Secondly, «Big Data and Business Analytics», where applications use IoT devices to measure and analyze business metrics related to user behavior patterns and market conditions. For example, retail

stores collect data on the number of clients at various times of the day, the path of customers in the store, among others. With that data, companies can make optimized management decisions such as product placement to increase revenue.

Finally, «Information Sharing and Collaboration», were applications seek to improve the situation awareness of those involved and reduce delays and distortions in information communication. For example, retail store employees can have status tracker-related devices, which allow the store manager to check who is available at any given time and quickly assign new tasks to those employees.

However, IoT includes a complex architecture. IoT devices in most applications generate a large amount of data that needs to be aggregated and analyzed. Therefore, all this information processing would cause an exponential increase in the response times of traditional centralized processing methods on local servers or in the cloud. Therefore, the data should be processed in a contextualized environment, using distributed algorithms allowing the computations to occur closer to the data source. Nonetheless, these requirements cause some challenges that the current implementations of IoT architectures have to solve.

One of the main concerns is related to the areas of cybersecurity and data privacy. IoT devices generate, process, and exchange large amounts of data. Additionally, it is usual that data contains security-critical or sensitive private information, thereby making it an attractive target for attacks. In recent years, we have observed numerous attacks against industrial IoT systems. These strikes can result in money losses, infrastructure damage and endanger human lives [42].

As another challenge, since IoT systems produce data at an unprecedented level, and the current cloud architecture is not ready to deal with this volume of information, new solutions must assure scalability, with computing resources becoming more geographically distributed to improve response times.

In parallel, we must note the characteristics of the devices and data created. Given the variety of IoT device types, the managing components must ensure system interoperability to face a high heterogeneity of devices and data by abstracting these software and hardware differences.

Finally, the adaptability of the infrastructure is also a concern for IoT systems. Most devices are unreliable and do not guarantee high availability. Even the network conditions in most IoT architectures are inconsistent and suffer constant changes [34].

2.1.2 Edge Computing

In a Cloud Computing approach, devices such as cell phones and sensors traditionally only act as producers/receivers of data. Despite the services being located at the edge of the network, the cloud is responsible for all data processing. Today, the amount of data circulating has increased substantially, making it increasingly difficult for cloud frameworks to support the existing volume of data.

For example, in a social network service like Facebook, it is common for users to upload large images and videos with high quality. These uploads require a large bandwidth to send the information to the cloud and only there applied the computationally heavy compression techniques. This problem increases exponentially with the large number of clients performing similar actions simultaneously. Edge Computing

approaches aim to allow data processing to occur close to the data sources. Consequently, the services for image and video reduction would be available closer to the clients, at the edge of the network, decreasing the pressure on the grid, removing the load from the central servers, and resulting in improved response times [46].

The Edge Computing paradigm has been evolving rapidly in parallel with IoT. The global Edge Computing market size is expected to expand at a compound annual growth rate of 38.4% from 2021 to 2028, with a forecasted revenue of USD 61.14 billion in 2028 [20]. Nevertheless, an IoT architecture would benefit from an integration with the Edge Computing approach, due to its inherent characteristics and necessities. In general, although limited, Edge Computing offers a relatively good computational capacity and some disk space. But this reduction of available resources, when compared to a Cloud Computing approach, is largely compensated by fast response times derived from the smaller latencies. Integrating these two concepts would mean an evolution of the Edge Computing framework making it more distributed and dynamic. Any IoT device with some computational capacity can become an edge node and integrate a network of services for data provisioning or processing [60].

By adopting an Edge Computing architecture, it is possible to achieve considerable improvements in end systems. Researchers have shown that through the simple use of Cloudlets (small scale data centers at the edge of the network), reducing physical distance, it is possible to increase system predictability and simultaneously reduce application response time [44].

A practical case of the benefits of implementing an Edge Computing-based architecture is an online shopping service on mobile devices, an increasingly popular system nowadays. Considering several types of customer requests, such as browsing items, adding an item to the shopping cart, among others, normally all of these are made and logged in the cloud servers. In the cloud-only approach, all these requests are highly dependent on the network speed and load of the central servers. Using edge nodes is possible to cache items closer to the users, drastically reducing the latency of the vast majority of requests and improving the quality of service in these operations. Only a portion of the requests, such as adding an item to the cart, would be made to the cloud servers [46].

2.2 Related Work

The problem of resource allocation and scheduling in heterogeneous and distributed systems is a subject that has been the matter of extended research at various levels [55, 14, 7, 19]. However, when we focus on scheduling mechanisms that consider the geographic location of the nodes and the data to be processed, there is less material available on the subject. And even then, most of the previous works focus only on the proposal of algorithms for calculating the optimal placements, without practical implementations [52, 25]. Others are only implemented at the network level [5, 61, 4] or for Fog Computing approaches [8, 41].

2.2.1 Resource Allocation and Scheduling

There is however some research related to the resource allocation and scheduling problem in hybrid cloud and edge environments. Zenith [59], for example, is a new model for allocating processing workloads on edge computing platforms. Previous work in the scheduling area assumes tight coupling between the edge infrastructure and the service providers. Zenith uses a decoupled model, where the infrastructure is independent of the service provider, arguing that it is possible to improve available resources usage and decrease infrastructure costs. With this tool, the service providers first establish resource-sharing contracts with the infrastructure providers in advance and then use the latency-aware scheduling algorithm to ensure the defined latency requirements and allocate the resources globally and locally, as efficiently and fairly as possible. Nevertheless, Zenith is only a proposal for an algorithm, but no practical implementation or prototype was developed and published. The conducted experimental evaluation uses simulations of the algorithm execution, and Zenith was never deployed or tested in a setup that replicates a production environment.

Dyme [43] is a dynamic scheduling system for Edge Computing that aims to maximize network throughput while providing end-users with the best quality of service. The authors designed Dyme to solve the problems of latency and microservice completion time. The system schedules the available tasks among the optimal microservices. The algorithm seeks to optimize the quality of service, throughput, network delay, and price, considering several parameters. These include the optimal latency, price factors, and the level of satisfaction of requests from both end-users and other microservices. Price factors depend on the importance of different microservice operations. However, this system only provides the mechanism for scheduling distributed processing tasks through replicas of microservices already deployed in the cluster. Since Dyme does not manage the locations of microservices themselves, it can only optimize the parameters previously presented in the currently deployed infrastructure. In a heterogeneous system with several microservices types, the capability to calculate the best location for each of them is an essential requirement. It ensures the distribution of workloads in an efficient and fair way for the end-users.

HYDRA [27] is a decentralized and distributed orchestrator for containerized microservice applications. While it can manage heterogeneous resources across geographical locations and enables the location-aware deployment of microservice applications via containerization, it is a completely new solution implemented from the ground up. Thus, nor HYDRA nor the other presented solutions leverage the additional orchestration features (e.g., pod abstraction, support for different storage drivers and backends, volume management, maintainability, service discovery, load-balancing, and existing user base), as described in Table 1, of mature solutions such as Kubernetes [10] and Nomad [38], the two most complete engines for container orchestration in multi-node clusters [24].

Project	Independent from Service Provider	Location-Aware Task Allocation	Location-Aware Service Allocation	Orchestration
Zenith	✓		✓	×
Dyme		✓		×
Hydra			✓	×

Table 1: Resource Allocation and Scheduling projects

2.2.2 Orchestration System

New scheduling systems for edge environments have to take advantage of existing edge orchestration systems to be usable in real-case scenarios. As mentioned above, Kubernetes and Nomad are the two most complete engines for container orchestration in multi-node clusters. Nonetheless, by being oriented towards cloud environments, both lack infrastructure capabilities to deal with edge environments, especially in terms of network and node management related metadata synchronization. However, unlike in Nomad, several frameworks extend Kubernetes cloud capabilities to the edge. Therefore we will focus on Kubernetes-related solutions.

From these Kubernetes frameworks to edge environments, some stand out, such as MicroK8s [37], a simple low-sized Kubernetes package that allows users to add edge nodes to a cluster and guarantees high availability through a consensus algorithm. Akri [2] is a Kubernetes interface for Edge, and it dynamically discovers nodes and provides an edge-oriented network protocol. However, both these tools, as seen in Table 2, cannot create cluster resources in Edge Devices.

Kubernetes Cluster Federation [33] is an orchestrator of orchestrators, as it is composed of a series of Kubernetes clusters, and introduces the concept of Cross Cluster Service Discovery, enabling developers to deploy a service that is sharded across a federation of clusters spanning different zones, regions or cloud providers. Despite enabling geographic distribution of workloads, the Kubernetes Cluster Federation is, by definition, a multi-cluster, multi-cloud system that cannot operate in an edge computing paradigm since it assumes an underlying high-performance network in each cluster.

Another alternative is KubeEdge [29]. It is a fully transparent abstraction for the Kubernetes API, which allows the management of the cluster with edge nodes with kubectl, the Kubernetes command-line tool. It ensures fault-tolerant and highly available communication between all nodes and provides lightweight agents for the edge.

Given that none of the previous Kubernetes-based solutions provides scheduling algorithms based on geographic locations, the work proposed in this paper builds on top of KubeEdge, which is further detailed below, to provide such a solution. This technology, as shown in Table 2, provides all edge orchestration features needed for a complete, efficient, and easy-to-maintain edge system. Furthermore, it is endorsed by the Cloud Native Computing Foundation [15].

Project	Multiple Geolocations	Manage Edge Devices	Edge Network Protocol	Create Pods, CMs, etc, in Edge Devices
Kubernetes Cluster Federation	✓			
Nomad	✓	✓		
MicroK8s	✓	✓		
Akri	✓	✓	✓	
KubeEdge	✓	✓	✓	✓

Table 2: Orchestration System projects

2.2.2.1 KubeEdge

KubeEdge extends Kubernetes and allows the management of remote edge nodes and edge applications deployments without changing the Kubernetes API. It provides edge controllers for node and workload handling, a custom network protocol, and a distributed metadata storage, to support system faults and offline scenarios where edge nodes are not connected to the cloud [29].

As depicted in Figure 1, the *EdgeController* module is responsible for managing the edge nodes. It extends the Kubernetes default controller with edge capabilities, allowing the *API Server* to integrate the edge nodes in the cluster. The network connection between the cloud and edge nodes is implemented by *EdgeHub* and *CloudHub*. These modules are responsible for assuring a fast and reliable communication interface between the cluster nodes.

In terms of workloads, edge applications and resources are configured and controlled by the *Edged* module, a lightweight agent that packages all the edge node functionality into one process. This module is also responsible for launching and controlling three other modules composing the system, the *EventBus*, the *DeviceTwin*, and the *MetaManager*. These extra modules manage all external edge devices and data handling.

This architecture allows KubeEdge to provide an offline network mode, ensure fault tolerance, and high availability while efficiently supporting cross-platform and heterogeneous software and hardware environments. All of these while allowing a simplified development process, with an SDK for systems and resource management, and maintenance [58].

One of the main components in KubeEdge clusters is the scheduler, a service responsible for attaching pods, the single most basic instances of a running process in the cluster, to nodes in the system. For each newly created pod or unscheduled pod, the scheduler selects a node to attach the application and execute it.

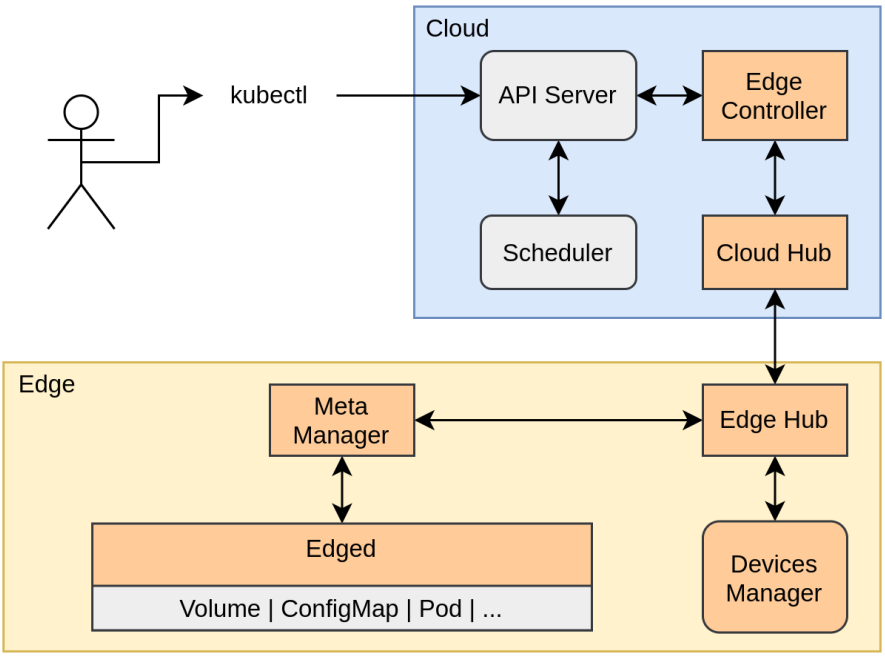


Figure 1: KubeEdge Architecture

However, since one can configure different resource requirements for pods (e.g., to define the minimum CPU and memory necessary for the application to run), existing nodes need to be filtered according to the availability of these resources. Briefly, the scheduler selects a suitable node in a 2-step operation. In the first, the filtering step, it finds the list of nodes where it is possible to attach the pod. If the list is empty, this pod cannot be deployed. When there is more than one possible node, we enter the scoring step, where the scheduler sorts the remaining nodes by available resources to choose the most appropriate for pod attachment.

However, The KubeEdge scheduler is still a very naive dynamic resource-provisioning mechanism which only considers nodes' resource utilization, therefore not being very effective [12]. For example, when considering two cluster nodes in two different zones of the planet and a data-producing workload near the first one, the default scheduler instead of choosing the first node to minimize network latency, will ignore this geographic distance aspect and select any of the nodes.

Therefore, we created a geolocation-aware scheduling system for KubeEdge. With a focus on the geographic location of data-producing workloads, this system can minimize network latencies when deploying consumer workloads and improve service response times.

2.3 Summary

Despite the growing developments and increasing popularity of IoT and Edge Computing systems, there is still a long way to go for them to become more viable and efficient, becoming feasible alternatives to

traditional solutions. In this perspective, one of the most critical problems that need to be solved is the scheduling and orchestration of data processing services. The scheduling solutions should consider the geographical location of the computing nodes and the systems that produce data.

A geolocation-oriented solution will reduce the dependability on the network speed, drastically reducing the latency and, therefore, response times on the vast majority of client requests. By decreasing these response times, systems will be able to handle larger volumes of data and take better advantage of available computing resources.

3

Design and Architecture

In this chapter, we present and detail the design considerations and architecture of Geolocate, a geographically oriented scheduling solution. We describe the main principles for correct and efficient scheduling of applications taking into account the geographical location of the nodes and the data sources. We also illustrate the normal flow of execution of the system, its components, and the relationships between them.

3.1 Design Principles

First of all, the scheduling solution to be devised must be aware of the geographic location of nodes and data sources. In more detail, accounting for the existence of a data-producing workload in a specific location, the scheduler must be able to, given a corresponding data-processing workload, calculate the best fitting node for its deployment. The selected nodes should minimize the distance between data producers and consumers to improve network latency, data processing delay, and service response time.

As shown in Figure 2, the scheduler must be aware of the location of available computing edge nodes at different granularities, namely their city, country and continent. When deploying a data processing workload (Figure 2 - 1), the scheduler allows users to define the location (i.e., city, country, continent) desired for running their processing workloads, for instance, closer to the corresponding data producers.

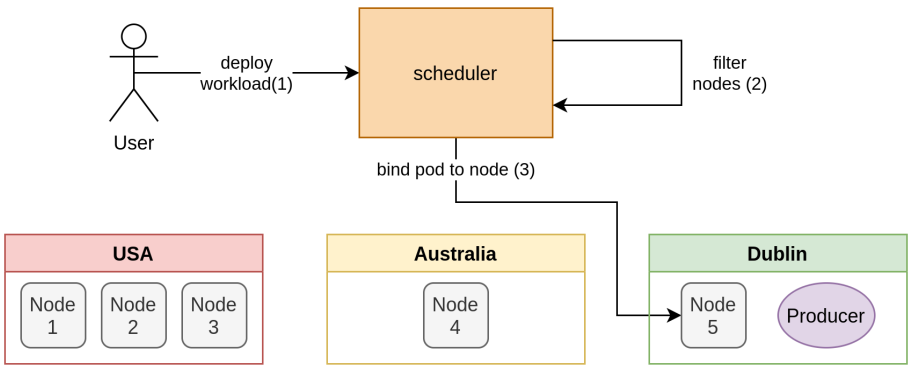


Figure 2: Geolocator's Design Principles

These user-defined locations can be specified as *mandatory*. In this case, if the desired location does not have the necessary computational resources, the scheduler should return an error response, and the workload's deployment should be delayed until enough resources are available. If locations are specified as *preferred* and a node at the desired location is not available, the scheduler selects the closest node with available computing resources. After selecting the best node (Figure 2 - 2), the workload is deployed (Figure 2 - 3).

In the second place, the scheduling solution must be generic to allow for its extensibility, particularly concerning adding new scheduling algorithms. Since conditions are not constant, and each system or application may have different requirements related to calculating the best node for scheduling workloads, the solution must expose a simple interface that all implemented algorithms must follow. Therefore, allowing the development of new scheduling algorithms to be restricted to the actual coding of the logic to calculate the best placement of workloads across nodes.

Lastly, the scheduling solution must be integrable with different orchestration tools. Namely, it should include internal mechanisms for abstracting the management of nodes and workloads from the framework where it is being deployed, which allow it to abstract the specificities of those systems where it is used. Thus, to integrate the scheduler with other systems the developer only needs to implement a connection interface between the target system and the scheduling engine.

3.2 Geolocate

Following the specification mentioned above, we defined several interfaces and modules. The proposed architecture follows the single-responsibility principle [22], encapsulating node and workload information in data structures, facilitating the system's extensibility. In Figure 3, we can observe the various components that make up the Geolocate scheduler.

The *scheduler-core* component allows the creation of generic node selection algorithms for workload placement. In this work, we present an algorithm that takes into account the geographic location of nodes (for example, data-processing and data-producing). This component also offers several data structures for indexing cluster information about nodes and workloads. Finally, the *scheduler-core* provides a public interface with all the necessary methods to add, remove or update the cluster nodes and execute the scheduling algorithm.

The *scheduler-implementation* integrates the *scheduler-core*, through its public interface, with the target orchestration tool, for example, KubeEdge.

The *NodeHandler*, receiving all node changes made to the cluster (Figure 3 - A1), whether adding new ones, editing (including current available resources updates) or removing existing nodes, is responsible for keeping the *scheduler-core* with up-to-date information through the *NodeManager* (Figure 3 - A2). In turn, the *NodeManager* maintains an internal structure with cluster nodes' information.

The *PodHandler* is responsible for consulting/receiving cluster information about the existence of

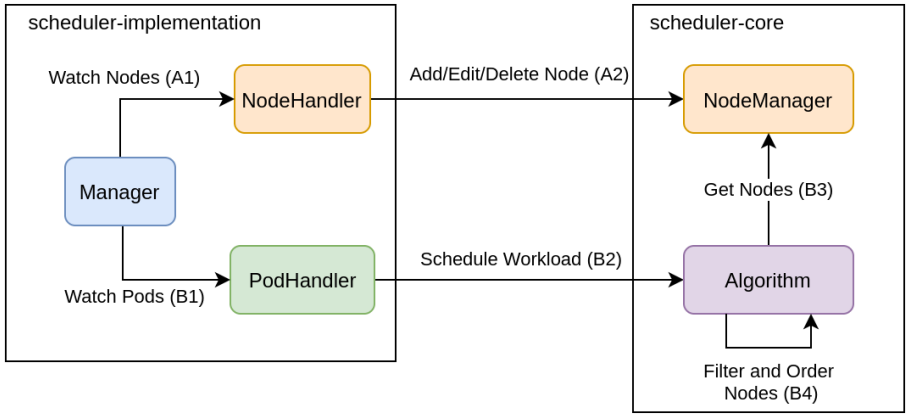


Figure 3: Geolocate Architecture

service workloads associated with the scheduler that are not yet bound to any cluster node (Figure 3 - B1). When the *PodHandler* gets any pending workload, the placement algorithm is executed (Figure 3 - B2). The algorithm fetches from the *NodeManager* available cluster nodes (Figure 3 - B3), excludes those that do not match the workload requirements and that do not have enough available resources to support it (Figure 3 - B4), and if any node remains, one is selected.

3.2.1 Generic Scheduling

Geolocate is a generic scheduling engine capable of distributing workloads on any cluster. Therefore, it provides structures for representing nodes and workloads. We designed these structures to be compatible with any orchestration technology, such as KubeEdge or Nomad. The engine also provides a simple interface that allows four different operations, as shown in the code excerpt 3.1.

Listing 3.1: Geolocate Interface

```
1 interface Scheduler:
2     ScheduleWorkload(workload) => (node, error)
3     AddNode(node)
4     UpdateNode(oldNode, newNode)
5     DeleteNode(node)
```

After the creation of a workload, the scheduler engine should be signaled, through the *ScheduleWorkload*, to proceed with the calculation of the best node for the allocation of the given workload (Figure 4 - 1).

For the internal administration of the nodes, the scheduler exposes an interface for the creation and management of nodes, which is independent from the underlying system (*AddNode*, *UpdateNode*, *DeleteNode*). Node information may include, for example, name, available CPU, memory and storage resources, geographic location, among others.

When calculating the best node for workload allocation, the scheduler first checks the nodes' specifications for the requested requirements, for example, the CPU resources it uses, type of node needed, preferred geographic location, among others (Figure 4 - 2).

Nodes that cannot receive the workload are then filtered out (Figure 4 - 3). For example, nodes that should only run cluster management and configuration workloads or nodes that do not have the necessary resources to receive the workload. Then, when mandatory geographic locations are specified, nodes from different regions are also excluded from the configuration (Figure 4 - 4).

Finally, based on the chosen algorithm, the scheduler calculates the best node for the allocation (Figure 4 - 5). This algorithm can, for example, sort by best geographic location, taking into account the nodes with more available resources. In the end, the scheduler picks the best-rated node and returns a response to inform the cluster in which cluster node it should schedule the workload (Figure 4 - 6).

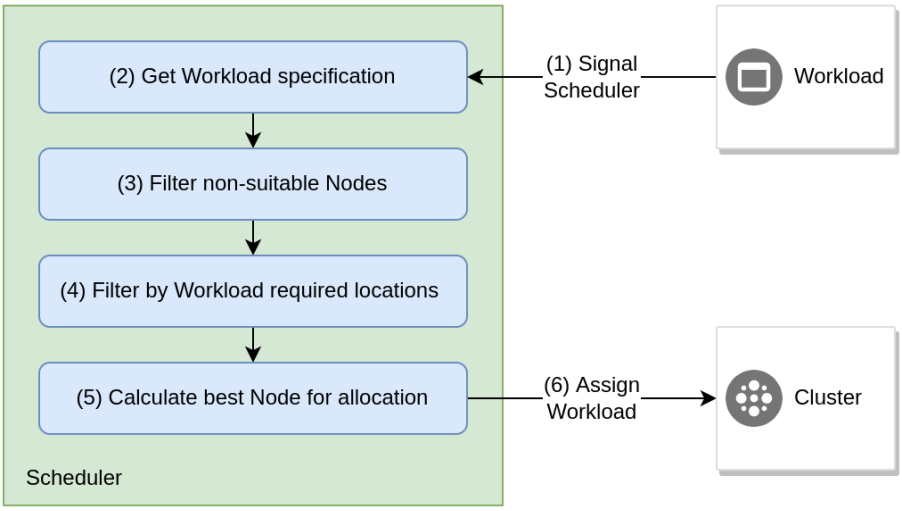


Figure 4: Scheduler Internal Flow

3.2.1.1 Node Management

As discussed earlier, the scheduling engine provides a data structure to abstract node information, as shown in the code excerpt 3.2.

Listing 3.2: Node Structure

```
1 struct Node {
2   Name
3   Labels
4   CPU
5   Memory
6 }
```

When the scheduler-implementation adds a new node to scheduler-core, it is stored in the system and categorized by their labels, for example, the geographic location. The node is associated to the labels' values in the scheduler-core internal structures.

To update a configured node, the system performs an internal search for the node's identifier name. Then it runs a merge patch on its stored configurations according to the new data submitted, whether updating labels or available resources.

When deleting a node, the scheduling engine removes the node system information and its association in each of its labels. The orchestration system should be responsible for deactivating the workloads running on that node and subsequent rescheduling.

3.2.1.2 Workloads

The scheduling engine also provides a structure for workload representation that, at this point, is equal to the structure of the nodes, but instead of representing resource availability, the fields represent resource needs. Workloads can have labels used as preferences and constraints to be taken into account by the algorithms. One constraint may be, for example, a required geographic location. Workloads also contain information about the CPU resources and necessary memory that the nodes have available to guarantee their correct execution.

Listing 3.3: Workload Structure

```
1 struct Workload {
2   Name
3   Labels
4   CPU
5   Memory
6 }
```

3.2.1.3 Algorithms

Given a possible integration with various orchestration systems and the future extensibility of the scheduler, particularly in terms of the scheduling algorithms provided, Geolocate also provides an abstraction for new scheduling approaches. Administrators can extend the engine with more algorithms, later chosen when deploying the scheduler implementation.

As we can see in the excerpt 3.4, new algorithms need to expose two methods. The first one is to query its identifying name, used for management purposes, namely the selection of the algorithm used by the scheduler implementation. Second, the method for executing the algorithm's logic for scheduling a given workload. This method should return the node for allocation or an error message in case of a failure.

Listing 3.4: Algorithm Interface

```
1 interface Algorithm {
2   GetName() => name
3   GetNode(workload) => (node, error)
4 }
```

3.2.2 Location Algorithm

Algorithm 1 Location Scheduling Algorithm

Input: w Workload

Output: r (Node, error)

$allNodes \leftarrow Nodes$
$nodes \leftarrow filterNodesWithoutResources(w, allNodes)$

if $length(nodes)$ is 0 **then**
    return $(nil, error)$
end if

$wLocations \leftarrow getLocations(w)$

if $length(wLocations)$ is 0 **then**
    return $(rand(nodes), nil)$
end if

if $\exists node : getLocation(node) \subseteq wLocations$ **then**
    return $(node, nil)$
else
    if $locationIsRequired(w)$ **then**
        return $(nil, error)$
    else
        $node \leftarrow$
            getNodeInSameCountry(wLocations.Cities)
            $\vee$
            getNodeInSameContinent(wLocations.Countries)
            $\vee$
            rand(nodes)
        return $(node, nil)$
    end if
end if

Geolocate provides a scheduling algorithm, depicted at Algorithm 1, that considers the resources available at the nodes and their geographic location. The first step of the algorithm is to fetch the nodes

registered in the scheduler and filter them, removing those that do not have enough available resources to cover the requests by the workload (lines 1-2). Then, if there are no nodes after the filter is applied, the algorithm returns an error stating that there are no nodes available to allocate this workload (lines 4-6).

The algorithm then processes the geographic location request for the workload (line 8). If there were no geographic location requirements, the algorithm returns a random node from the cluster (lines 10-12). Otherwise, once the algorithm gathers the information, it looks for a node that fulfills the localization claim (line 14). If any node exists, the algorithm finishes its execution by returning that node (line 15). On the other hand, if the algorithm could not find a matching node, there are two possibilities.

First, if the geographic location requested by the workload is mandatory, as it is not possible to satisfy this restriction, the algorithm terminates its execution and returns an error (lines 17-18). If the given geographic locations are preferred, the algorithm looks for a nearby node. Initially, the algorithm checks the cities defined in the workload, looking for nearby nodes in their countries. If it still does not find one, it applies the same method, but this time over countries, looking for nearby nodes in other countries (lines 20-26).

4

Prototype

Previously, we discussed the design principles and the architecture of the presented solution. In this chapter, we present the prototype implementation details. Also, we discuss the tools used, internal operation details, and deployment/usage instructions.

4.1 Golang

A large part of cloud tools like Docker and Kubernetes are written in Golang. Since Geolocate integrates very directly with many of these tools, using the same language greatly eases interoperability between them, in terms of data structures, resource validations, library reutilization, and more.

Furthermore, Goland demonstrates excellent performance regarding parallel processing. KubeEdge is capable of launching hundreds of workloads per second, and the scheduler has to be able to meet the demand as quickly as possible. Golang provides easy-to-use concurrency primitives (goroutines) that are directly mapped to threads in the operating system. Moreover, goroutines have their scheduler associated, which avoids the overhead of system calls and the operating system thread scheduler, optimizing the asynchronous execution of tasks [57].

4.1.1 Geographic Locations information package

For the calculations of the geographic locations, it was necessary to investigate solutions to manage country and subdivision information. Fortunately, there were already projects in Golang that solved these problems, such as the «gountries» package (Countries (ISO-3166-1), Country Subdivisions(ISO-3166-2)) [40].

A tool like this was essential because of configurations such as a node being in the city of Lisbon and a workload to be deployed in Portugal. It was necessary to correlate Lisbon as a city of Portugal.

However, during development, some needs arose for which «gountries» did not yet have a solution. Therefore, we had to contribute to this project by extending its functionality. For example, we created the «FindSubdivisionByName» method, which returns information about a subdivision given its name. And the «FindSubdivisionCountryByName» method, which given the name of a subdivision, returns information

about its country. We also added the concept of a «continent» that did not exist yet and created continent search methods.

4.2 KubeEdge Integration

For development and testing purposes throughout this project, we used KubeEdge as the underlying orchestration system. In terms of management and extensibility, KubeEdge provides one of the best architectures and APIs for the addition of new tools that increase cluster functionality.

Our scheduler-implementation for KubeEdge subscribes to information from KubeEdge. The first subscription is related to information about nodes in the system, in which KubeEdge informs Geolocate whenever a node is added to the cluster, modified, or removed (Figure 6 - a).

The second subscription is related to information about new workloads. When a user creates an *EdgeDeployment*, an extension on KubeEdge's API resources described below, it creates a native Deployment with the configuration of our scheduler and the geographic location defined in the Pod attributes (Figure 6 - 1).

Then, KubeEdge informs the scheduler that a pod to allocate exists (Figure 6 - 2). The scheduler executes its internal flow and informs back KubeEdge of the node where it should place the pod (Figure 6 - 3), which, in turn, allocates it to the node (Figure 6 - 4).

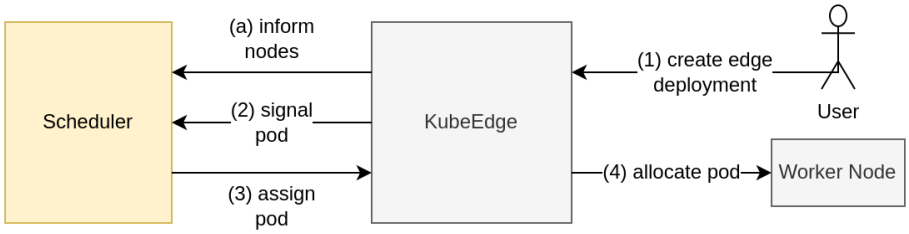


Figure 5: KubeEdge Integration

4.2.1 Operator Pattern

As mentioned earlier, we have expanded the KubeEdge's API by creating a new extension called *EdgeDeployment*. To do this, we used the Operator Pattern [39], a Kubernetes design pattern that further extends Kubernetes functionality. We created the *EdgeDeployment* Custom Resource Definition (CRD), which is then built into the Kubernetes API, to abstract the low-level details of configuring geographic locations from users. We also have an associated Custom Controller that handles instructions from the users to create, modify and delete *EdgeDeployment*s.

4.2.1.1 Custom Resource Definition

The *EdgeDeployment* CRD defines a configuration fully managed and stored by KubeEdge, which provides an OpenAPI v3.0 specification for users to create new *EdgeDeployment* resources [17]. Listing 4.1 shows an example for a configuration of an *EdgeDeployment* resource (line 2). This configuration defines the resource's base metadata settings, such as the name (lines 3-4) and the specific settings associated with an *EdgeDeployment*. These settings include the number of replicas of the service (line 6), the location requirements (lines 7-12), and the execution settings, which follows the KubeEdge's default workloads specification, containing fields such as the image to execute, and the CPU and memory resources required (lines 13-21).

Listing 4.1: EdgeDeployment Example

```
1  apiVersion: edge.geolocate.io/v1
2  kind: EdgeDeployment
3  metadata:
4    name: example
5  spec:
6    replicas: 1
7    preferredLocation: # Or 'requiredLocation'
8      cities:
9        - Braga
10     countries:
11       - Portugal
12       - Spain
13   template:
14     spec:
15       containers:
16         - name: test-app
17           image: nginx:latest
18       resources:
19         requests:
20           memory: 64Mi
21           cpu: 250m
```

4.2.1.2 Custom Controller

As previously mentioned a *Custom Resource* needs to have an associated controller [16] to handle instructions from the users to create, modify and delete *EdgeDeployment*s.

Whenever the user executes an instruction in KubeEdge to create, modify or remove an *EdgeDeployment* resource (Figure 6 - 1), KubeEdge communicates to the *Custom Controller* the new configuration applied so that it can reconcile the state stored in the cluster with what was applied by the user (Figure 6 - 2). The reconciliation phase can include any logic in which the controller can perform actions on the cluster to respond to the changes applied by the user (Figure 6 - 3). In the example shown at Figure 6, this could mean that since the preferred location of the workload was changed, the controller can recreate the workload so that the scheduler can place it again into the correct location.

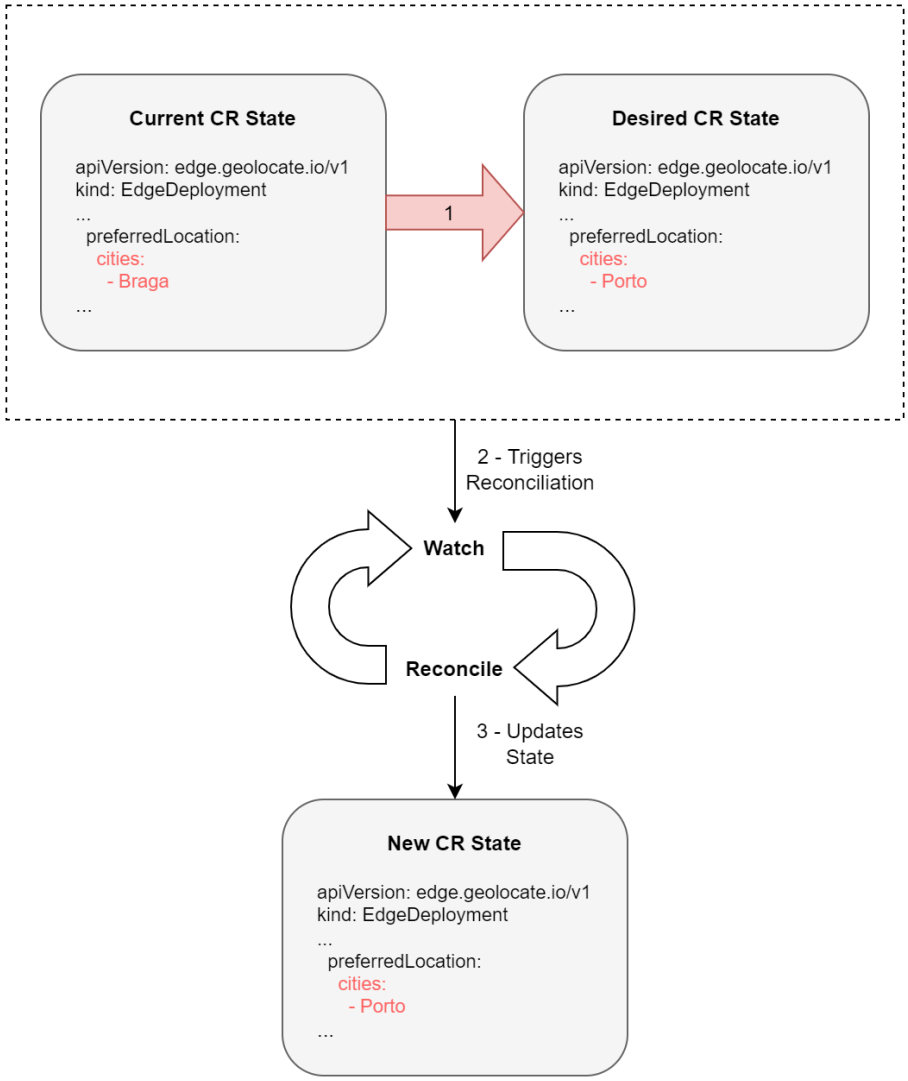


Figure 6: *EdgeDeployment* Controller Flow

4.2.2 Scheduler Integration

The geolocate-scheduler is responsible for calculating and selecting the cluster node where the user's pod (workload) should be placed and inform KubeEdge. The entire process of deploying and managing the

pod on the node is the responsibility of other KubeEdge components (Figure 7 - 6).

In terms of implementation, the geolocate-scheduler subscribes to the KubeEdge's *Node Informer* [31] to receive alerts about all changes to nodes in the cluster (Figure 7 - A), allowing it to stay up to date on what nodes exist, what their configurations are, and what available resources do these have.

Whenever a new node is added to the cluster, or its available computational resources (e.g., CPU, RAM) are updated, geolocate-scheduler reads the total resource capacity of the node and the configurations of the pods running on it. From the pods' information about their resources needs, the scheduler estimates the total used resources on the node at that time. The node geographical location is extracted from the node labels, manually configured by the system administrator, but future work may include the dynamic calculation of node location in this step.

Secondly, geolocate-scheduler also subscribes to the *Pod Informer* [32] to be notified of new pods that are not bound to any cluster node and need to be scheduled (Figure 7 - B). When there is a new pod that needs to be scheduled, the algorithm fetches the location configuration from the pod specification and iterates all nodes trying to get any in the selected location (i.e., given city, country or continent) (Figure 7 - 4).

If the system finds a suitable node, the algorithm finishes and the scheduler instructs KubeEdge to bind the pod to that node (Figure 7 - 5). Otherwise, there are two hypotheses. If the location was specified by the user as *mandatory* the scheduler throws an error, and the pod remains unbounded until a suitable node is available or the *EdgeDeployment* resource is deleted. If the location was specified as *preferred*, the system looks up nodes in a broader area, as close as possible to the desired one. In more detail, if a node is not available at the requested city, then the algorithm searches for a free node at the city's country. If no nodes are available at that country, then the algorithm checks for an available node at the corresponding continent. Note that, for a given workload, a user can specify several cities as viable deployment options. In this case, the previous algorithm is similar but it searches first for nodes at the cities, then at all the countries of specified cities and, finally at the corresponding continents. Since, the location is just a preference, if a suitable node could not be found at this point, a random available node in the cluster is selected. Again, if no nodes are available geolocate-scheduler throws an error, and the workload's deployment is delayed until a node is available or the *EdgeDeployment* resource is deleted.

4.2.3 Scheduler Deployment

To deploy the geolocate-scheduler, we need to build the container image, upload the image to the container registry and then launch it in the cluster. In KubeEdge, we can treat a scheduler like any other application and deploy it using a Deployment configuration. However, the geolocate-scheduler requires extra permissions to function and register itself as a scheduler in the cluster. Therefore we created a *ServiceAccount* for the scheduler, later associated to a *ClusterRoleBinding*, which assigns the «system:kube-scheduler» role.

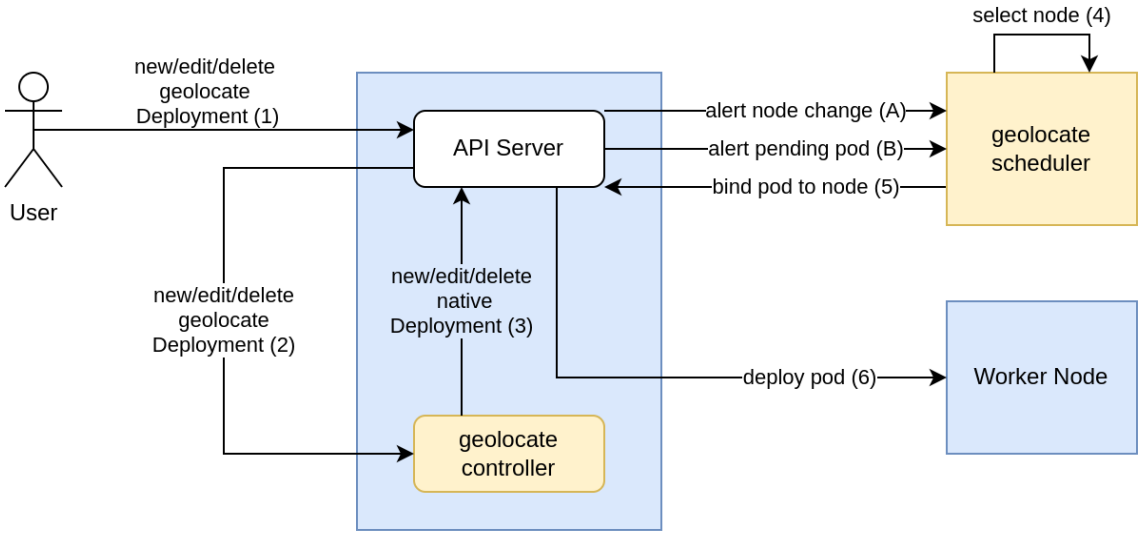


Figure 7: KubeEdge with Geolocate Architecture

5 Evaluation

In this chapter, we present the experiments we performed in order to validate the usefulness and functionality of Geolocate's full fledged prototype. Next, we show an evaluation that assesses the potential gains of a scheduling solution that takes into account the geographical locations of the nodes. Also, we show the performance results for an application representative of a real system, depending on whether its components were scheduled with and without taking into account the geographic location of cluster nodes.

5.1 Setup

For all experiments, we launched a KubeEdge cluster using 6 commodity servers. Each node had an i3 CPU (4 cores at 3.7 GHz), 8 GB of RAM and a 128 GB SSD disk. Hosts were connected over a shared Gigabit Ethernet network.

Since our scheduler proposes an efficient geographical distribution of data processing workloads according to the location of corresponding data producers, it is important to ensure that our cluster correctly simulates a global distribution of nodes. As it was not possible to conveniently arrange and test our system with nodes effectively distributed over wide geographic regions, we locally simulated the latency between them as described in Table 3.

City	Dublin	Lisbon	Sydney	Tokyo
Dublin	-	-	-	-
Lisbon	58	-	-	-
Sydney	274	312	-	-
Tokyo	240	250	160	-

Table 3: Global latency statistics in milliseconds extracted from https://wondernetwork.com/pings.

To apply this latency to the cluster, we used Chaos Mesh [1], a tool that features fault injection methods for complex systems on Kubernetes. Using the NetworkChaos Experiment, we considered each of the

cluster Edge nodes to be in a city mentioned in Table 3. We then configured a *Network Delay action* between them and every other Edge node with the desired latency, causing the expected delays in message sending between pods deployed on those nodes.

5.2 Microbenchmark

Our first experimental goal was to verify whether the use of scheduling algorithms, taking into account the geographical location of the data to be processed, results in substantial improvements in application performance and quality of service. To this end, we measured the variation of response times of a data processing workload as a function of the geographic location of the node where it is scheduled, taking into account that the data producer is located in a fixed node and consequently in a fixed region.

5.2.1 Methodology

In the tests, we simulated an IoT-related data stream processing system. We created two workloads to replicate the processing of energy consumption data collected from an Irish power grid. The grid contained at the time of the dataset publication 6435 sensors. Each of those sensors emitted readings every 30 minutes, and over 500 days of observations were collected in the final file [36].

Some previous work conducted experimental runs measuring the execution times of processing this dataset. In those runs, the final producing rate was 10000 messages per second, and the execution time of a workload to parse the collected XML data was about 3.1ms for each message [49].

For this simulation, we created two applications. First, we developed the application to simulate the data creation. It produces messages at the fixed rate of 10000 messages per second. The workload sends the messages to an MQTT broker, implemented locally on the same node.

Second, we created an application for simulating the parsing of the XML data. This workload pulls messages from the MQTT broker and simulates a data processing task that takes a fixed time of 3.1ms per message.

The latter application allows the configuration of distinct execution parameters. We can set up a variable number of threads processing data in parallel, replicating a more real-life scenario. We can also configure the size for a local message queue, allowing messages to be collected parallelly from the producer and stored in the message queue while no processing threads are available.

In these experiments, we always deployed the data-producing application in Dublin, and we rotate the geographic location of the data processing workload between Dublin, Lisbon, Tokyo, and Sydney.

The test workloads were run on the cluster using two different schedulers and varying the execution parameters. First, we set a fixed message queue size of 0 messages and varied the number of threads between 4, 8, 16, and 32. Subsequently, we set the message queue size to 128 and again varied the number of threads between 4, 8, 16, and 32. For each of these configurations, we ran the experiment 10 times with both the KubeEdge default-scheduler and the geolocate-scheduler.

In this phase, we want to measure the time delay from when the producing workload creates each message until the processing workload finishes the XML parse task for that message. We want to understand the effects of the latency times caused by the different geographical locations of the nodes in the time it takes to process the first 5000 dataset messages.

5.2.2 Results

After running all the tests with the different configurations and collecting the data regarding the processing delays of the issued messages, we performed two types of analysis. Firstly, the comparison of the variation of delay times according to the geographic location of the processing workload. And then the variation of delay times according to the scheduling system used.

5.2.2.1 Comparison per Region

As mentioned above, the first point of analysis was the average message processing delay as a function of the geographic region in which the processing workload is located.

Figure 8 shows that in the configurations considered, namely using 4 and 32 threads with both queue sizes at 0 and 128, the geographic proximity between the producing workload and the processing workload has a considerable impact on the average message processing delay.

Considering the configuration (4 threads, 0-queue), in Figure 8 - (a) we observe that the 5000th message, when the processing workload is in Dublin, is processed with a delay of about 46 seconds. On the other hand, if this workload is in Sydney, the message processing delay grows to about 686 seconds, a 1391% increase.

We observed improvements when considering 4 threads with a 128 message queue size, which allows some messages to be stored locally for later processing. At Figure 8 - (c), and considering the processing workload in Dublin, the processing delay of the 5000th message drops from 46 seconds (with queue size at 0) to 1.66 seconds. However, if the processing workload is in Sydney, the improvement is only 5 seconds in the 5000th message, going from 686 seconds (with queue size at 0) to 681 seconds.

Considering a message queue size of 128, in Figure 8 - (d), we verify some improvements in the message processing delay, mainly in the most distant regions. When the processing workload is in Dublin, the processing delay for the 5000th message goes from 1.66 seconds in the configuration (4 threads, 128-queue) to 1.55 seconds. However, when the processing workload is in Sydney, the processing delay for the 5000th message goes from 682 seconds in the (4 threads, 128-queue) configuration to 82.22 seconds. This difference corresponds to a decrease of about 88%.

Still, the performance differences between the geographic regions chosen for deploying the processing workload are quite visible. For instance, in the best configuration, depicted at Figure 8 - (d) (32 threads, 128-queue), the difference in processing delay for the 5000th message when choosing Dublin or Sydney is 80.67 seconds, which corresponds to a 5205% increase between them.

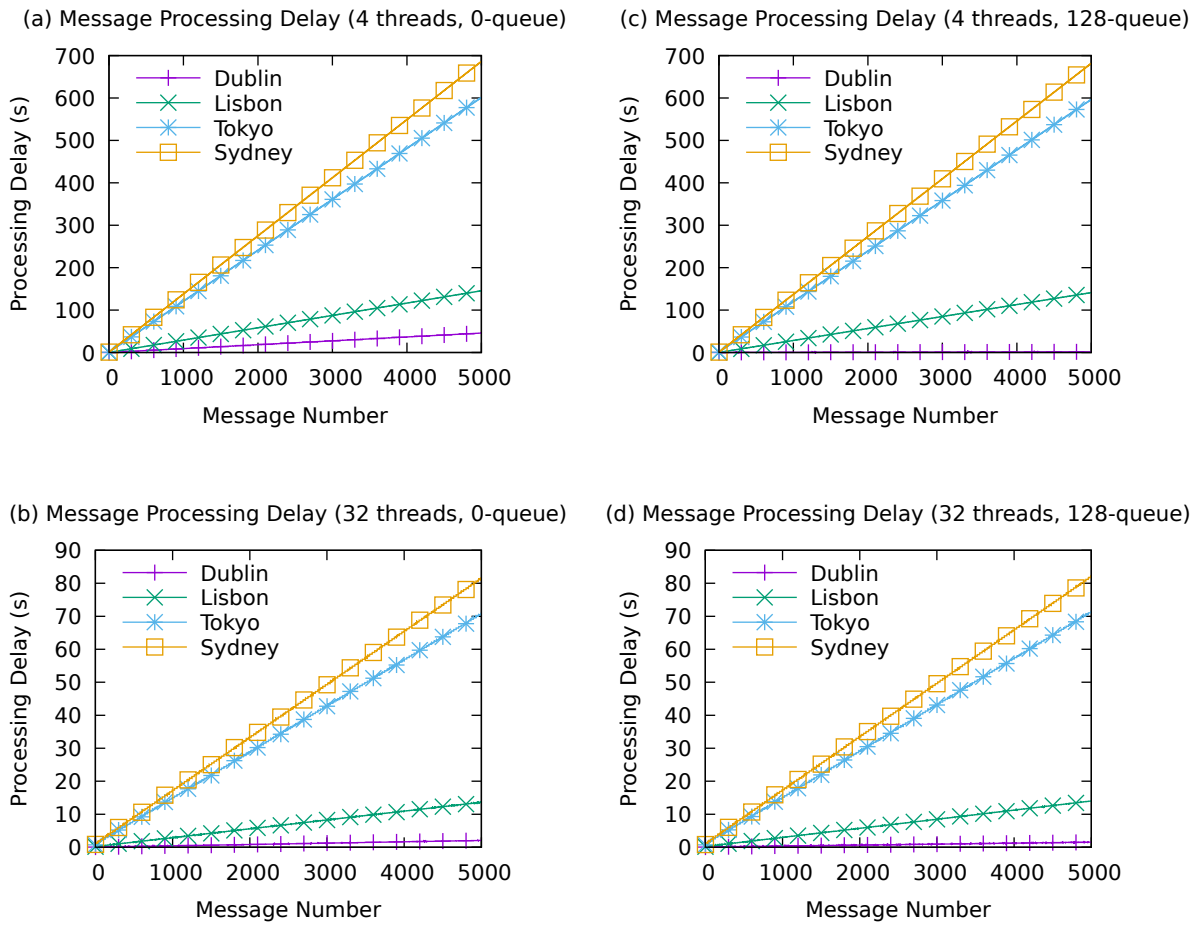


Figure 8: Message Processing Delay between Regions

It is clear the negative influence of high latencies between the systems that produce and consume large amounts of data for different services. Clients using services operating under these conditions observe high response times because of the high latencies between data-producing and data-processing systems.

5.2.2.2 Comparison per Scheduler

Next, we disregarded all the other factors that may condition the placement of workloads besides the geographical location of the data, such as node priorities or balancing their available resources. In these conditions, we know that the default-scheduler places with equal probability the workload on any of the available nodes.

In the best case scenario, with a probability of 25% in this setup, the default default-scheduler places the processing workload in the Dublin node, the one with the lowest latency to the data producing service. The worst case scenario also has a probability of 25% in this setup. In this case, the default-scheduler places the processing workload in Sydney.

Using the geolocate-scheduler, the data processing workload is always scheduled in the region defined

by users. Therefore, if there are enough available resources and the Dublin location is specified by users, the workload is always placed in Dublin. Therefore the low latencies ensured by the geolocate-scheduler always translate into lower message processing delays.

Analyzing the experimental results of these placements, in Figure 9 - (a) we observe that the average message processing delay, using a queue of size 0, is around 424 seconds with 4 threads. Increasing the processing parallelism with the use of 32 threads, we verify that the delay decreased to about 34.64 seconds. Subsequently, raising the message queue size to 128 did not cause considerable variations in the average message processing delay. In Figure 9 - (b) we observe that, with 4 threads, the processing delay decreases from 424 seconds to 412 seconds, and in tests with 32 threads, we observe a value decrease from 34.64 seconds to 34.59 seconds.

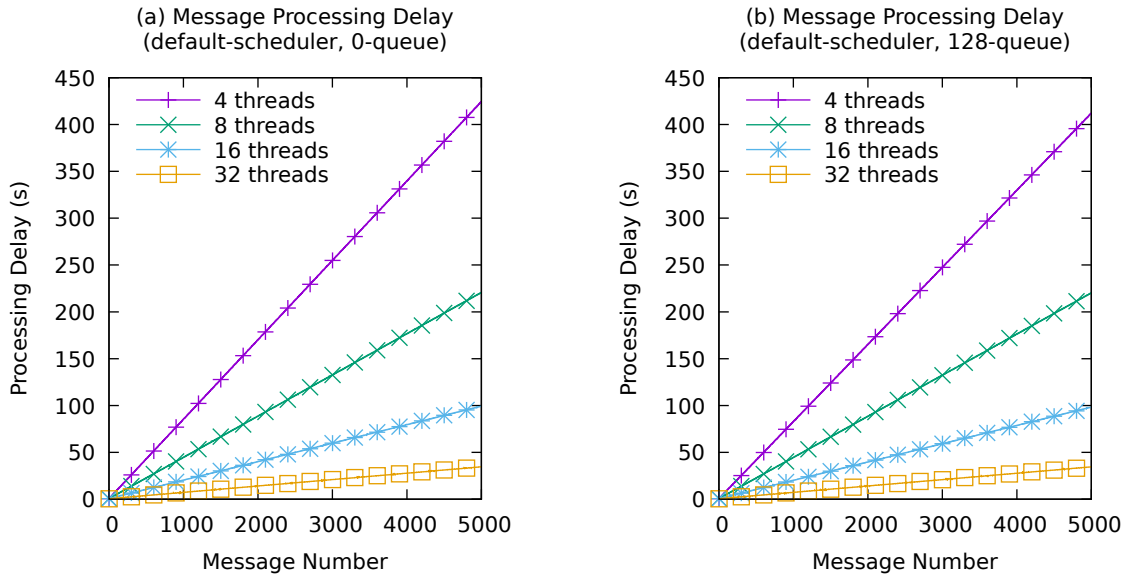


Figure 9: Message Processing Delay using the default-scheduler

On the other hand, using the geolocate-scheduler, in Figure 10 - (a) we observe that with a message queue size of 0, the average delay in processing the 5000th message using 4 threads is 45.82 seconds and, when using 32 threads, the delay is around 2.06 seconds. In the geolocate-scheduler, we see the best impacts of using a message queue. Increasing the queue size from 0 to 128, in Figure 10 - (b) we observe that the 5000th message processing delay using 4 threads drops to 1.66 seconds. And, when using 32 threads, the delay decreases to 1.56 seconds [1].

So directly comparing the best results of each of the schedulers, both with the configuration (32-thread, 128-queue), in Figure 11 we can observe the best average processing delay times for all messages.

[1] Please note the difference in scale on the Y's axis between Figure 9 and Figure 10. In Figure 9, the Y's values range between 0 and 450. In Figure 10, the Y's values range between 0 and 50 on graph (a) and between 0 and 1.8 on the graph (b).

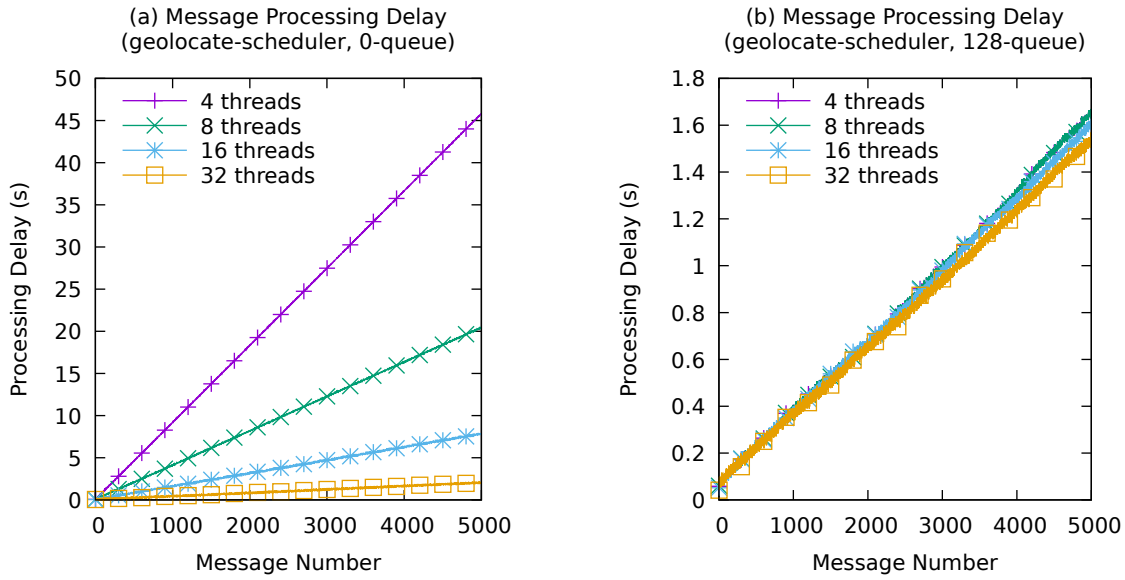


Figure 10: Message Processing Delay using the geolocate-scheduler

On average, in the 2500th message, the processing delay using default-scheduler is about 23.16 seconds while using geolocate-scheduler is 0.81 seconds. This difference of 22.35 seconds corresponds to a reduction of 96.5%.

In the 5000th message, the reduction is even more significant. While using the default-scheduler the delay is around 45.56 seconds. Using the geolocate-scheduler is the delay rounds the 1.55 seconds. A reduction of 44.01 seconds, about 96.6%.

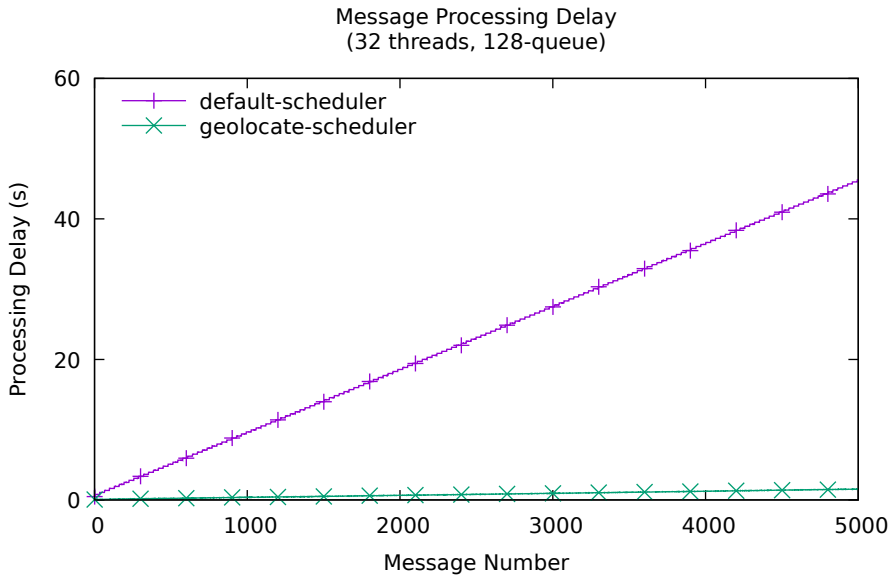


Figure 11: Schedulers direct comparison

5.3 Sock Shop

To test the difference derived from using geolocate-scheduler versus default-scheduler in a situation closer to a real system, we decided to use the Sock Shop application, a microservices demo application that simulates an e-commerce website that sells socks [51].

This application is composed of several distinct components as shown in Figure 12, such as frontend services, backend services, and message brokers. In addition, the Sock Shop microservices are heterogeneous, developed using several different technologies such as Java, Golang, and Javascript. This variety helps closely represent the complexity of today's software systems.

The Sock Shop project already has predefined configurations to deploy the complete application on Kubernetes [48]. These settings are compatible with KubeEdge and have been used to deploy using the default-scheduler. To use geolocate-scheduler, we adapted these configurations by changing the 'apiVersion' to use the controller associated with the geolocate-scheduler and we set the required location in the Deployment spec as desired.

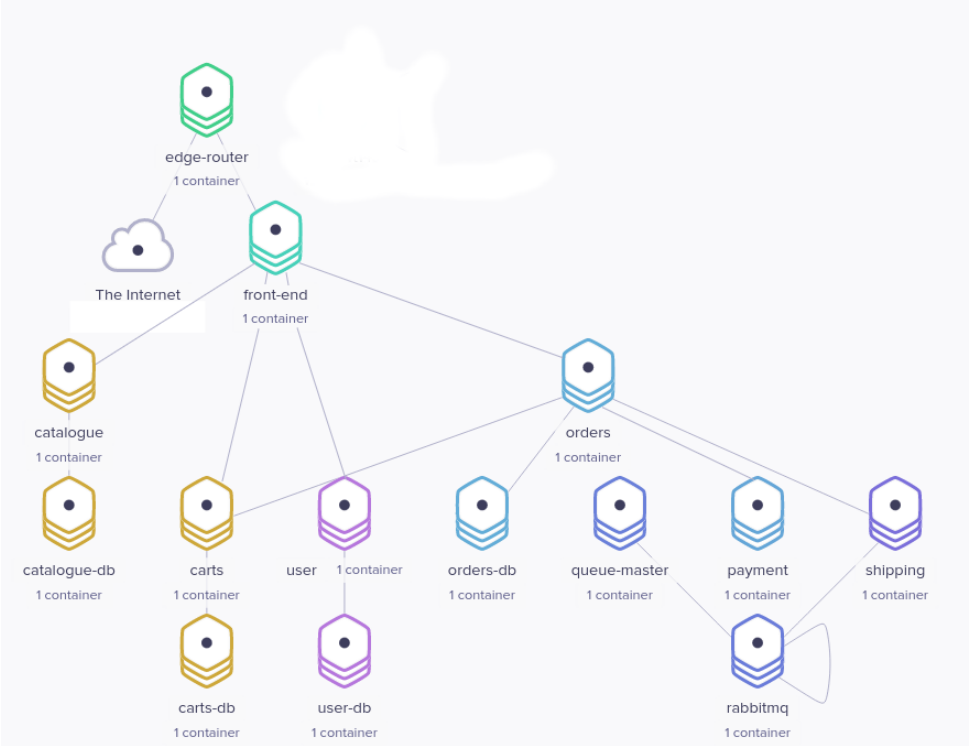


Figure 12: Sock Shop architecture

5.3.1 Methodology

For the experiments, we used a load testing tool provided by the Sock Shop project [47]. This tool uses the Locust framework, which allows the simulation of users in a distributed way by using event-driven

programming, thus enabiling the testing tool to simulate up to thousands of concurrent users using the application [3].

Locust generates at the end of the execution detailed statistics related to requests and response times, both globally and per requests type. Therefore it is possible to recreate real end-usage of the application and to simulate a realistic load on the system.

Initially, we launched a Sock Shop instance locally and executed each of the requests made by the load test. For each request, we traced the communications between each of the microservices to map the connections between them. From that information, we compiled the flow of each request, as shown in the Table 4, where we can observe which services are most accessed by the load test requests. These should be closer to the client to decrease the latency and consequently the average response time of the load test requests.

Request	**Accessed Microservices**
GET /	frontend
GET /basket.html	frontend
DELETE /cart	frontend ->carts ->carts_db
POST /cart	frontend ->catalogue ->catalogue_db frontend ->carts ->carts_db
POST /orders	frontend ->user ->user_db frontend ->orders ->orders_db orders ->user ->user_db orders ->user ->user_db orders ->user ->user_db orders ->carts ->carts_db orders ->payment orders ->shipping ->rabbitmq_server ->queue_master
GET /catalogue	frontend ->catalogue ->catalogue_db
GET /category.html	frontend
GET /detail.html	frontend ->catalogue ->catalogue_db
GET /login	frontend ->user ->user_db frontend ->carts ->carts_db

Table 4: Accessed Microservices for each Sock Shop request

In this test, we considered that in each geographic location, because of resource limits, we can only have 5 Sock Shop application components. Furthermore, we assumed that no component can be placed in Lisbon. There we only deployed a client accessing the application, in this case, the Sock Shop load test service.

Given the most accessed microservices, the best case placement of microservices is as shown in Figure 13. In the region closest to Lisbon, Dublin, we placed the ‘frontend‘, ‘carts‘, ‘carts-db‘, ‘catalogue‘, and ‘catalogue-db‘ microservices. Then, in the second nearest region, Tokyo, we placed the ‘user‘, ‘user-db‘, ‘orders‘, ‘orders-db‘, and ‘payment‘ microservices. Finally, in Sydney, the furthest region from Lisbon,

we placed the ‘shipping‘, ‘rabbitmq-server‘ and ‘queue-master‘ microservices.

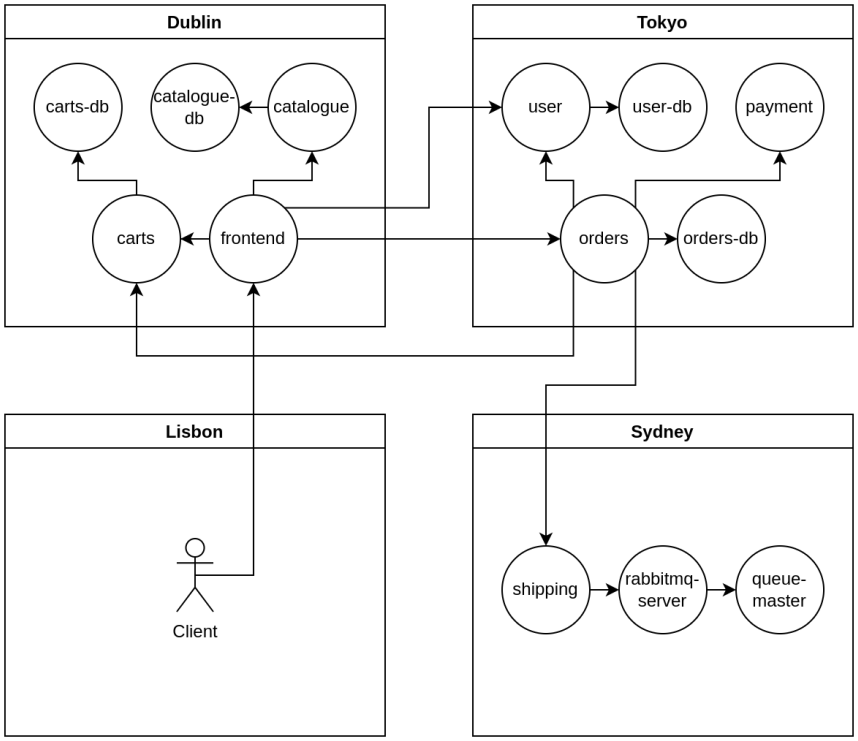


Figure 13: Service Distribution

When using the geolocate-scheduler, we configured the aforementioned geographical distribution for each microservice deployment. The scheduler always tries to deploy each application in the requested region. In the default-scheduler, we use the generic configuration provided by the Sock Shop project.

For each scheduler, we executed five runs of the load-test and collected the geographic locations of each workload and the global and per-request response times. In each run, we configured the load-test to simulate 100 concurrent users. Each user executed 100 requests, 11 executions of each request type.

5.3.2 Results

By analyzing the Figure 14, we can observe that for all types of requests made by the load-test to the application, the response time is always lower when using the geolocate-scheduler instead of the default-scheduler.

The request with the highest average response time is, with both schedulers, the «POST /orders». In this request, with the default-scheduler, the median response time is 4015ms. With the geolocate-scheduler, this value is 723ms. A reduction of about 81%. We can also observe that the performance with geolocate-scheduler is much more consistent than with the default-scheduler. With the default-scheduler, the range of average response time values is 3837ms, with the maximum being 5595ms and the minimum

1758ms. On the other hand, geolocate-scheduler has a range of 254ms, with a maximum of 823ms and a minimum of 569ms.

The request with the lowest response time in both schedulers is «GET /». In this request, the default-scheduler presents the highest median value of 629ms. The geolocate-scheduler shows a value of 121ms, a reduction of about 80%.

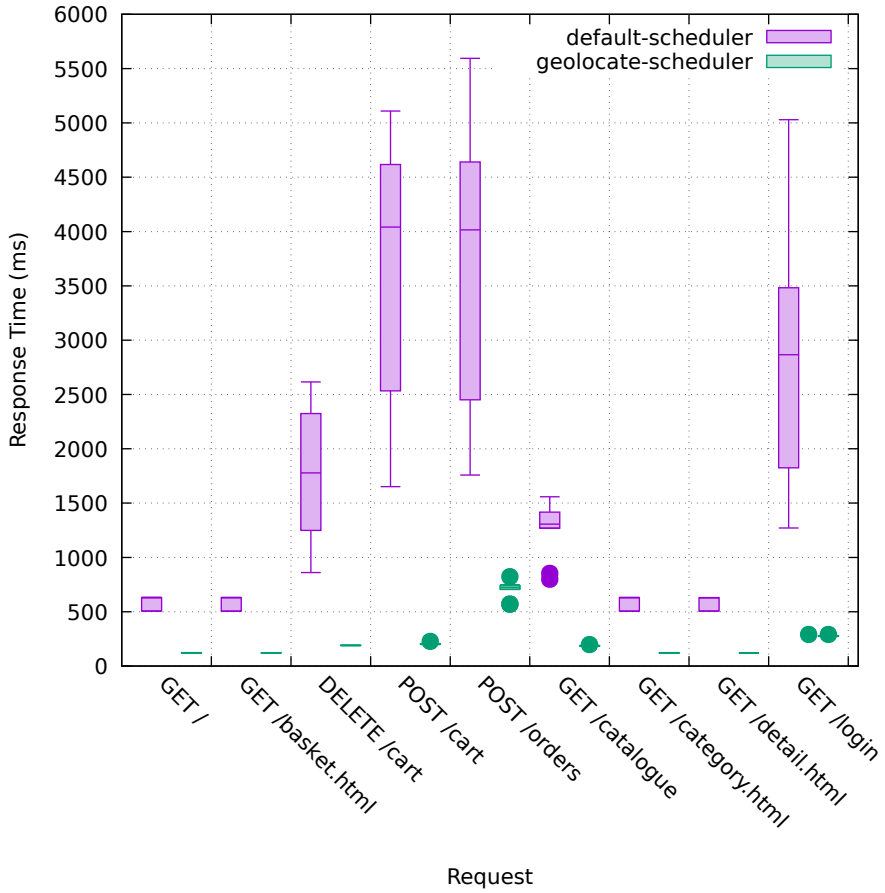


Figure 14: Average Response Time per Request

In Figure 15, we can see a global comparison of the application's response times according to the scheduler used. Using the default-scheduler, the average of response times in the overall requests is 1269ms. The range recorded between the maximum value and the minimum value is 5090ms. Observing the values when we used the geolocate-scheduler, we see that the average response time for requests drops to 189ms. This change represents a decrease of about 85% compared to the default-scheduler. The range recorded between the maximum and minimum average response time values is much lower than with the default-scheduler as well, 703ms.

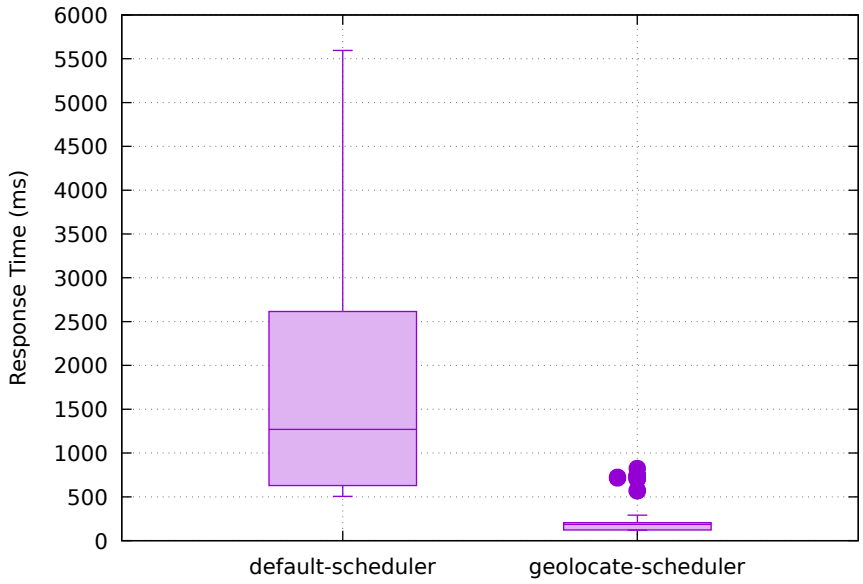


Figure 15: Average Global Response Times

5.4 Discussion

The results obtained show that, as expected, the increase in latency causes an increase in the response times of applications and that, in systems that handle an increasing number of data requests, this impact is cumulative, reaching high values, as demonstrated in Figure 8 graphs. Since Geolocate accounts for and mitigates the geographic location problem, it is able, in certain use cases, to reduce message processing delays by almost 97%, as shown in Figure 11.

In a more common use case, using an e-commerce system in an Edge Computing environment, the results also show that the impact of latency on response times is high and that the use of Geolocate can reduce, when compared to KubeEdge's default-scheduler, these response times by about 85%, as shown in Figure 15. It is also noteworthy that the response times with the use of Geolocate are much more predictable, with low variability of the response time values in each request, reaching almost zero in some cases.

6

Conclusion

Edge computing has emerged as an important paradigm that moves the computation and data storage of distributed services closer to users. While virtualization technologies, such as containers, have eased the task of running distributed services in heterogeneous edge hardware, these are still lacking adequate scheduling and orchestration algorithms that can place computation near data producers.

This thesis presents Geolocate, a new scheduling solution suitable for data-centric workloads being deployed at geographically distributed Edge environments. The proposed scheduler is integrated with KubeEdge, a state of the art orchestration system that is based on Kubernetes and maintains a similar interface for portability and ease of adoption purposes.

We validated the utility and functionality of Geolocate with experiments that validate the impact of latency and geographic distance between production and data processing services on message processing times. We then assessed Geolocate's performance benefits with an application that simulates a real e-commerce application. The results show that the use of Geolocate can reduce, relative to KubeEdge's default-scheduler, the average response time for requests by about 85%.

6.1 Future Work

The work presented at this document opens the way to study various research questions. Currently, Geolocate is dependent on some manual input from the service administrator to set up node geographic location information in nodes metadata, a process that can be automated through IP geolocation estimation techniques [18, 13] adapted to Edge systems. Geolocate can also be enhanced to collect additional historical information about the amount and type of data exchanged between the different services in conjunction with monitoring tools. Geolocate algorithms could then make better scheduling decisions based on that knowledge through machine learning techniques already used in systems such as underlay networks [11] or in cloud-only environments [54]. Secondly, it is essential to examine the scalability and fault tolerance of Geolocate by understanding and testing its behavior on large-scale systems with a high number of nodes and service replicas. Finally, given the genericity of Geolocate, its integration with other orchestration systems such as Nomad can be studied and implemented.

Bibliography

[1] *A Powerful Chaos Engineering Platform for Kubernetes: Chaos Mesh*. url: https://chaos-mesh.org/ (cit. on p. 26).

[2] *Akri*. July 2021. url: https://github.com/deislabs/akri (cit. on p. 9).

[3] *An open source load testing tool*. Accessed on 2021-12-09. url: https://locust.io/ (cit. on p. 33).

[4] Z. Arnjad et al. "Latency Reduction for Narrowband LTE with Semi-Persistent Scheduling". In: *2018 IEEE 4th International Symposium on Wireless Systems within the International Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS)*. 2018, pp. 196–198. doi: 10.1109/IDAACS-SWS.2018.8525713 (cit. on p. 7).

[5] D. Baghyalakshmi, J. Ebenezer, and S. Satyamurty. "Low latency and energy efficient routing protocols for wireless sensor networks". In: *2010 International Conference on Wireless Communication and Sensor Computing (ICWCSC)*. 2010, pp. 1–6. doi: 10.1109/ICWCSC.2010.5415892 (cit. on p. 7).

[6] A. R. Biswas and R. Giaffreda. "IoT and cloud convergence: Opportunities and challenges". In: *2014 IEEE World Forum on Internet of Things (WF-IoT)*. 2014, pp. 375–376. doi: 10.1109/WF-IoT.2014.6803194 (cit. on p. 1).

[7] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira. "DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm". In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. 2010, pp. 27–34. doi: 10.1109/PDP.2010.56 (cit. on p. 7).

[8] L. F. Bittencourt et al. "Mobility-Aware Application Scheduling in Fog Computing". In: *IEEE Cloud Computing* 4.2 (2017), pp. 26–35. doi: 10.1109/MCC.2017.27 (cit. on p. 7).

[9] A. Bucchiarone et al. "From Monolithic to Microservices: An Experience Report from the Banking Domain". In: *IEEE Software* 35.3 (2018), pp. 50–55. doi: 10.1109/MS.2018.2141026 (cit. on p. 1).

[10] B. Burns et al. “Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade”. In: *Queue* 14.1 (Jan. 2016), pp. 70–93. issn: 1542-7730. doi: 10.1145/2898442.2898444. url: https://doi.org/10.1145/2898442.2898444 (cit. on pp. 2, 8).

[11] X. Cao et al. “A Machine Learning-Based Algorithm for Joint Scheduling and Power Control in Wireless Networks”. In: *IEEE Internet of Things Journal* 5.6 (2018), pp. 4308–4318. doi: 10.1109/JIOT.2018.2853661 (cit. on p. 37).

[12] C. Chang et al. “A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning”. In: *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. 2017, pp. 1–6. doi: 10.1109/GLOCOM.2017.8254046 (cit. on p. 11).

[13] J.-n. Chen et al. “Towards IP location estimation using the nearest common router”. In: *Journal of Internet Technology* 19.7 (2018), pp. 2097–2110 (cit. on p. 37).

[14] B. Cirou and E. Jeannot. “Triplet: A clustering scheduling algorithm for heterogeneous systems”. In: *Proceedings International Conference on Parallel Processing Workshops*. 2001, pp. 231–236. doi: 10.1109/ICPPW.2001.951956 (cit. on p. 7).

[15] *Cloud Native Computing Foundation*. July 2021. url: https://www.cncf.io (cit. on p. 9).

[16] *Custom Resource Controller | Kubernetes*. Nov. 2021. url: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#custom-controllers (cit. on p. 22).

[17] *Custom Resource Definition | Kubernetes*. Nov. 2021. url: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#custom-resources (cit. on p. 22).

[18] O. Dan, V. Parikh, and B. D. Davison. “IP Geolocation Using Traceroute Location Propagation and IP Range Location Interpolation”. In: *Companion Proceedings of the Web Conference 2021*. 2021, pp. 332–338 (cit. on p. 37).

[19] M. Daoud and N. Kharma. “A high performance algorithm for static task scheduling in heterogeneous distributed computing system”. In: *Journal of Parallel and Distributed Computing* 68 (Apr. 2008), pp. 399–409. doi: 10.1016/j.jpdc.2007.05.015 (cit. on p. 7).

[20] *Edge Computing Market Share & Trends Report, 2021-2028*. May 2021. url: https://www.grandviewresearch.com/industry-analysis/edge-computing-market (cit. on p. 7).

[21] *General Data Protection Regulation - Document 02016R0679-20160504*. May 2016. url: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:02016R0679-20160504&qid=1532348683434 (cit. on p. 2).

[22] W. Haoyu and Z. Haili. “Basic Design Principles in Software Engineering”. In: *2012 Fourth International Conference on Computational and Information Sciences*. 2012, pp. 1251–1254. doi: 10.1109/ICCIS.2012.91 (cit. on p. 14).

[23] N. Hassan, K. A. Yau, and C. Wu. “Edge Computing in 5G: A Review”. In: *IEEE Access* 7 (2019), pp. 127276–127289. doi: 10.1109/ACCESS.2019.2938534 (cit. on p. 2).

[24] M. Hausenblas. *Container Networking*. O’Reilly Media, Incorporated, 2018 (cit. on p. 8).

[25] S. Heemstra de Groot and O. Herrmann. “Rate-optimal scheduling of recursive DSP algorithms based on the scheduling-range chart”. In: *1990 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1990, 1805–1808 vol.3. doi: 10.1109/ISCAS.1990.111986 (cit. on p. 7).

[26] J. Hiller et al. “Secure Low Latency Communication for Constrained Industrial IoT Scenarios”. In: *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*. 2018, pp. 614–622. doi: 10.1109/LCN.2018.8638027 (cit. on p. 1).

[27] L. L. Jimenez and O. Schelen. “HYDRA: Decentralized Location-aware Orchestration of Containerized Applications”. In: *IEEE Transactions on Cloud Computing* (2020), pp. 1–1. doi: 10.1109/TCC.2020.3041465 (cit. on p. 8).

[28] H. Khodkari, S. Maghrebi, and R. Branch. “Necessity of the integration Internet of Things and cloud services with quality of service assurance approach”. In: *Bulletin de la Société Royale des Sciences de Liège* 85.1 (2016), pp. 434–445 (cit. on p. 1).

[29] KubeEdge, K. Wang, and F. Xu. *KubeEdge*. Oct. 2017. url: https://kubeedge.io/en/ (cit. on pp. 2, 9, 10).

[30] *Kubernetes*. July 2021. url: https://kubernetes.io/ (cit. on p. 2).

[31] *Kubernetes - client-go Node Informer*. Accessed on 2021-12-11. url: https://pkg.go.dev/k8s.io/client-go/informers/core/v1#NodeInformer (cit. on p. 24).

[32] *Kubernetes - client-go Pod Informer*. Accessed on 2021-12-11. url: https://pkg.go.dev/k8s.io/client-go/informers/core/v1#PodInformer (cit. on p. 24).

[33] *Kubernetes Cluster Federation (KubeFed)*. July 2021. url: https://kubernetes.io/blog/2018/12/12/kubernetes-federation-evolution (cit. on p. 9).

[34] G. M. Lee and Jeong Yun Kim. “The Internet of Things — A problem statement”. In: *2010 International Conference on Information and Communication Technology Convergence (ICTC)*. 2010, pp. 517–518. doi: 10.1109/ICTC.2010.5674788 (cit. on p. 6).

[35] I. Lee and K. Lee. “The Internet of Things (IoT): Applications, investments, and challenges for enterprises”. In: *Business Horizons* 58.4 (2015), pp. 431–440. issn: 0007-6813. doi: https://doi.org/10.1016/j.bushor.2015.03.008. url: https://www.sciencedirect.com/science/article/pii/S0007681315000373 (cit. on p. 5).

[36] T. R. P. Ltd. *Cer smart metering project*. Tech. rep. Commission for Energy Regulation, Ireland (cit. on p. 27).

[37] *MicroK8s*. July 2021. url: https://microk8s.io (cit. on p. 9).

[38] *Nomad by HashiCorp*. url: https://www.nomadproject.io/ (cit. on p. 8).

[39] *Operator pattern | Kubernetes*. Nov. 2021. url: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ (cit. on p. 21).

[40] *pariz/gountries: Gountries provides: Countries (ISO-3166-1), Country Subdivisions(ISO-3166-2), Currencies (ISO 4217), Geo Coordinates(ISO-6709) as well as translations, country borders and other stuff exposed as struct data.* (Cit. on p. 20).

[41] X.-Q. Pham and E.-N. Huh. "Towards task scheduling in a cloud-fog computing system". In: *2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. 2016, pp. 1–4. doi: 10.1109/APNOMS.2016.7737240 (cit. on p. 7).

[42] A.-R. Sadeghi, C. Wachsmann, and M. Waidner. "Security and privacy challenges in industrial Internet of Things". In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6. doi: 10.1145/2744769.2747942. (cit. on p. 6).

[43] A. Samanta and J. Tang. "Dyme: Dynamic Microservice Scheduling in Edge Computing Enabled IoT". In: *IEEE Internet of Things Journal* 7.7 (2020), pp. 6164–6174. doi: 10.1109/JIOT.2020.2981958 (cit. on pp. 2, 8).

[44] M. Satyanarayanan et al. "The Case for VM-Based Cloudlets in Mobile Computing". In: *IEEE Pervasive Computing* 8.4 (2009), pp. 14–23. doi: 10.1109/MPRV.2009.82 (cit. on p. 7).

[45] V. Shah. "IoT Predictions That Could Shape Manufacturing In 2021". In: *Forbes* (Jan. 29, 2021). url: https://www.forbes.com/sites/forbesbusinesscouncil/2021/01/29/iot-predictions-that-could-shape-manufacturing-in-2021/ (visited on 12/22/2021) (cit. on p. 1).

[46] W. Shi and S. Dustdar. "The Promise of Edge Computing". In: *Computer* 49.5 (2016), pp. 78–81. doi: 10.1109/MC.2016.145 (cit. on p. 7).

[47] S. Shop. *microservices-demo/load-test: A load-test script container for Sock Shop*. Accessed on 2021-12-09. url: https://github.com/microservices-demo/load-test (cit. on p. 32).

[48] S. Shop. *microservices-demo/microservices-demo: Deployment scripts config for Sock Shop*. Accessed on 2021-12-11. url: https://github.com/microservices-demo/microservices-demo (cit. on p. 32).

[49] A. Shukla, S. Chaturvedi, and Y. Simmhan. "RIoTBench: An IoT benchmark for distributed stream processing systems". In: *Concurrency and Computation: Practice and Experience* 29 (Nov. 2017). doi: 10.1002/cpe.4257 (cit. on p. 27).

[50] V. Singh and S. K. Peddoju. “Container-based microservice architecture for cloud applications”. In: *2017 International Conference on Computing, Communication and Automation (ICCCA)*. 2017, pp. 847–852. doi: 10.1109/CCAA.2017.8229914 (cit. on p. 1).

[51] *Sock Shop*. Accessed on 2021-12-09. url: https://microservices-demo.github.io/ (cit. on p. 32).

[52] D. Stiliadis and A. Varma. “A general methodology for designing efficient traffic scheduling and shaping algorithms”. In: *Proceedings of INFOCOM '97*. Vol. 1. 1997, 326–335 vol.1. doi: 10.1109/INFCOM.1997.635151 (cit. on p. 7).

[53] B. Subramanian, K. Nathani, and S. Rathnasamy. “IoT Technology, Applications and Challenges: A Contemporary Survey”. In: *Wireless Personal Communications* 108 (Sept. 2019). doi: 10.1007/s11277-019-06407-w (cit. on p. 1).

[54] Z. Tong et al. “QL-HEFT: a novel machine learning scheduling scheme base on cloud computing environment.” In: *Neural Computing & Applications* 32.10 (2020) (cit. on p. 37).

[55] H. Topcuoglu, S. Hariri, and M.-Y. Wu. “Performance-effective and low-complexity task scheduling for heterogeneous computing”. In: *IEEE Transactions on Parallel and Distributed Systems* 13.3 (2002), pp. 260–274. doi: 10.1109/71.993206 (cit. on p. 7).

[56] *What is the Internet of Things (IoT)?* url: https://www.oracle.com/internet-of-things/what-is-iot/ (cit. on p. 1).

[57] J. Whitney, C. Gifford, and M. Pantoja. “Distributed execution of communicating sequential process-style concurrency: Golang case study”. In: *The Journal of Supercomputing* 75.3 (2019), pp. 1396–1409 (cit. on p. 20).

[58] Y. Xiong et al. “Extend Cloud to Edge with KubeEdge”. In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 2018, pp. 373–377. doi: 10.1109/SEC.2018.00048 (cit. on p. 10).

[59] J. Xu et al. “Zenith: Utility-Aware Resource Allocation for Edge Computing”. In: *2017 IEEE International Conference on Edge Computing (EDGE)*. 2017, pp. 47–54. doi: 10.1109/IEEE.EDGE.2017.15 (cit. on pp. 2, 8).

[60] W. Yu et al. “A Survey on the Edge Computing for the Internet of Things”. In: *IEEE Access* 6 (2018), pp. 6900–6919. doi: 10.1109/ACCESS.2017.2778504 (cit. on p. 7).

[61] M. Zorzi and R. Rao. “Geographic random forwarding (GeRaF) for ad hoc and sensor networks: energy and latency performance”. In: *IEEE Transactions on Mobile Computing* 2.4 (2003), pp. 349–365. doi: 10.1109/TMC.2003.1255650 (cit. on p. 7).