DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Service scheduling for the EdgeIO platform

Oliver Haluszczynski

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Service scheduling for the EdgeIO platform

Service Scheduling für die EdgeIO Plattform

Author:	Oliver Haluszczynski
Supervisor:	Prof. Jörg Ott
Advisor:	Dr. Nitinder Mohan
Submission Date:	January 15th, 2022

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, January 15th, 2022 Oliver Haluszczynski

Acknowledgments

First of all, I would like to thank the Chair of Connected Mobility at the Technical University of Munich and especially the chairholder, Prof. Dr. Jörg Ott, for the opportunity to write my master thesis in the emerging and exciting field of edge computing. Furthermore, I want to express a huge thank you to my advisor, Dr. Nitinder Mohan, for the numerous discussions and helpful weekly meetings. You always answered all my questions patiently, were able to give me extremely helpful input on my ideas, and gave me the freedom to pursue and implement those ideas.
I would also like to express my gratitude to all the fellow students I met during my bachelor and master studies at TUM for the exciting discussions.
Another thank you goes to my colleagues at Msg systems, who, over the past few years, have taught me the technical skills and their practical application, that were indispensable for the successful completion of the Master's program.
Many thanks also to my friends for the countless discussions about various topics and conversations about everything and anything.
Last but not east, I would like to thank my parents and siblings from the bottom of my heart for their endless support and motivation over the past 27 years.

Abstract

With the proliferation of Internet of Things (IoT) and data-intensive and latency-sensitive applications, there was a call for the cloud computing paradigm to evolve so that the processing and collection of data happens closer to the edge or end users, where that data is often generated. This crystallized the idea of cloud computing into a new paradigm called edge computing, which aims to create a continuum from the cloud to the edge of the network to achieve low latency by being closer to users, thus providing a better user experience. In order to leverage the potential benefits that edge computing brings, a sophisticated and efficient strategy for placing services within the edge network is immensely important. Service scheduling is a well-known NP-hard problem that becomes more complex as the number of variables increases. For this purpose, an algorithm has been designed within the scope of this Master's thesis that, based on the available information about the edge network, finds suitable nodes that can deploy a service with several different requirements on e.g., computational capacity, geographical location and maximum latencies. In this work, an algorithm, namely **c**onstraint-**a**ware **s**cheduling **a**lgorithm (CASA) is proposed, that finds suitable nodes using information about computational capacity, geographic position, and latencies to users and other nodes in the network. To minimize the additional overhead caused by active network measurements, the algorithm uses a network coordinate system (NCS) that maps the nodes in the network into a 2-dimensional coordinate system in a way, that the Euclidean distance of two nodes approximates the round trip time between them. Based on this, multilateration can be used to determine the position of users that are not an active part of the network coordinate system (NCS) in order to estimate the latencies of edge devices to user groups.
Furthermore, a prototype implementation of this algorithm and integration into EdgeIO, a flexible multi-cluster edge orchestration platform, is described. In addition, existing service placement solutions in cloud and edge environments are analyzed and the results of the newly developed scheduling algorithm are evaluated.

Kurzfassung

Mit der Ausbreitung von Internet of Things (IoT) und datenintensiven und latenzsensitiven Applikationen wurde der Ruf nach einer Weiterentwicklung des Cloudcomputingparadigmas laut, sodass die Verarbeitung und Sammlung von Daten näher am Rand bzw. an den Endnutzern geschieht, wo diese Daten oft auch erzeugt werden. Dadurch entstand aus der Idee des Cloud Computings ein neues Paradigma namens Edge Computing, welches das Ziel verfolgt, ein Kontinuum von der Cloud bis zum Rand des Netzwerks zu schaffen, um durch die Nähe zu den Nutzern niedrige Latenzen zu erreichen und damit für ein besseres Benutzererlebnis zu sorgen. Um die potentiellen Vorteile des Edge Computings bestmöglichst zu nutzen, ist eine ausgeklügelte und effiziente Strategie zur Platzierung von Services innerhalb des Edge Netzwerks immens wichtig. Service Scheduling ist ein bekanntes NP-hartes Problem, das mit steigender Anzahl von Variablen immer komplexer wird. Dafür wurde im Rahmen dieser Masterarbeit ein Algorithmus entworfen, der basierend auf den vorhandenen Informationen über das Edge-Netzwerk geeignete Knoten findet, die einen Service mit mehreren unterschiedlichen Anforderungen an bspw. Rechenkapazität, geographische Position und maximalen Latenzen deployen können. In dieser Arbeit wird ein solcher Algorithmus namens **c**onstraint-**a**ware **s**cheduling **a**lgorithm (CASA) vorgestellt, der mit Hilfe von Informationen über Rechenkapazität, geographischer Position und Latenzen zu Nutzern und anderen Knoten im Netzwerk geeignete Knoten findet. Um den durch aktive Netzwerkmessungen verursachten Mehraufwand gering zu halten, greift der Algorithmus auf ein Netzwerkkoordinatensystem zurück, das die Knoten im Netzwerk so in ein $2-$dimensionales Koordinatensystem einordnet, dass die Euklidische Distanz zweier Knoten die Round Trip Time zwischen ihnen approximiert. Basierend darauf kann mittels Multilateration die Position von Nutzern bestimmt werden, die kein aktiver Teil des Netzwerkkoordinatensystems sind, um so die Latenzen von Edge-Geräten zu Nutzergruppen zu schätzen.
Im Folgenden wird eine prototypische Implementierung dieses Algorithmus und Integration in EdgeIO, einer flexiblen multi-cluster Edge-Orchestrierungsplattform, beschrieben. Zudem werden existierende Service-Platzierungslösungen in Cloud und Edge-Umgebungen analysiert und die Ergebnisse des neu entwickelten Schedulingalgorithmus evaluiert.

Contents

1. Introduction

The rise of Internet of Things (IoT) applications has revitalized the popularity of wearables, smart cities, e-health, and many more important fields. As the number of users and performance requirements increases, so does the number of connected devices. Cisco estimates that there will be 29.3 billion connected devices by 2023 [1], such as self-driving cars, smart homes, cities and factories, etc. As a result, a gigantic amount of data, also known as *Big Data* will be collected from IoT sensors, which in turn will be stored in cloud data centers and then analyzed and processed. While data processing speeds have increased rapidly, the bandwidth at which data is transferred to and from data centers has not evolved at the same rate. Those requirement increments were initially satisfied by integrating IoT and cloud environments [2]. However, for the reason of being far away from client devices, cloud computing has its own limitations especially for time and resource critical applications [3]. On the one hand, due to the immense volume of data and the geographical distribution of devices, it is becoming increasingly difficult to support the transfer of data to and from billions of IoT devices in the IoT and cloud scenario. On the other hand, the requirement to reduce latency and to collect and store data closer to the place where the data is generated is getting louder. All this calls for extending the cloud to the edge of the network, i.e., to the IoT devices. Computing nodes closer to the edge, and thus closer to the users and data sources, can act both as a kind of filter that filters the amount of data sent to the cloud as well as a mini data center that processes the data closer to where it is generated or used.
To tackle the aforementioned challenges, the concept of edge computing emerged to cover the limitations by leveraging end devices for data congestion and processing locally in a distributed and decentralized way and it opened a broad range of renewed challenges in topics such as security, reliability, sustainability, scaling, or resource management [4].
Edge computing is an emerging technology that aims to integrate latency-sensitive and data-intensive applications into the cloud ecosystem by placing compute resources at the edge of the network. The resulting proximity to data producers and consumers enables significant improvements in terms of latency and bandwidth. Since edge resources are by definition very limited in computational capacity and heterogeneous in e.g. operating system or the instruction set, compared to their cloud counterparts, a trade-off between deploying a service close to the users and avoiding overuse of resources is inevitable [5]. To effectively take advantage of an edge infrastructure, services must be placed on a node that meets all requirements, especially contextual knowledge such as application and network related information [6]. Optimal service placement not only increases user satisfaction by reducing end-to-end latency, resulting in a delay-free experience, but also reduces the load on the network by reducing bandwidth usage and associated costs, and can even improve energy efficiency by allowing tasks to be completed on less energy-intensive edge devices rather

than offloading them to the energy-intensive cloud. Most of the rather few context aware scheduling solutions for edge computing focus only on response times between services rather than end-to-end latencies experienced by users [7].
The scheduling approaches developed for similar paradigms, especially for cloud and content delivery networks (CDNs), are not suitable for computing the optimal placement of latency sensitive services due to the infrastructural limitations. In cloud environments, there are huge distances between the data centers and the users, which inevitably leads to higher latencies and hence poorer user experiences. In addition, cloud solutions run the risk of congesting the network and wasting bandwidth due to potentially large amount of data. CDN paradigm is similar to edge computing in the sense that the resources are distributed and close to users [8]. However, CDNs are designed for data-intensive services rather than computation-intensive ones. Hence, computational tasks might have to be offloaded to the cloud and the cache servers placed near users are only used for the distribution of the output data [9]. However, this results in the servers still being several network hops away from users, and therefore does not reduce network latency in case of computationally heavy tasks [5].

1.1. Problem Statement

The current implementation of EdgeIO's service scheduling consists of two steps. Both root orchestrator (RO) and cluster orchestrators (COs) have a local scheduling component that is responsible for solving a subset of the service placement problem within their respective domains. Let $A = \{a_1, a_2, \ldots, a_{|A|}\}$ denote the set of applications requested to be deployed by the developers at EdgeIO's RO. Each application $a_p \in A$ can consist of n different microservices (with $n \geq 1$), i.e., $a_p = \{ms_{p,1}, ms_{p,2}, \ldots, ms_{p,n}\}$. Each microservice $ms_{p,i}$ of the $p-th$ application in turn has its own computational requirements (central processing unit (CPU) and memory usage), denoted by $Q_{ms_{p,i}}$. These requirements and other information about the microservices to be deployed can be specified by the developer in a deployment descriptor. Figure 1.1 shows the structure of such a deployment descriptor.

```
service_name: service
service_ns: test
virtualization: docker
image: path/to/image
memory: 500
vcpus: 1
```

Figure 1.1.: An example deployment descriptor.

In the first step the root scheduler has to find a suitable cluster for the application deployment. The resources of the i-th cluster with m workers are denoted by $R^i = \{R^i_1, R^i_2, \ldots, R^i_m\}$. Each worker j periodically sends its current resource usage to the CO such that it is aware of the currently available resources. The root scheduler then matches $Q_{ms_{p,i}}$ to $\bigcup(R^i)$ for each cluster and calculates a priority list of best-fit clusters. This step filters out all clusters that are not suitable for the task. Whether a cluster is suitable is determined by finding out if it has enough resource availability, supports the desired virtualization technology, etc. Once the list of best-fit clusters is found, the RO offloads the deployment request to the CO with the highest priority.
In the second step, the CO calculates the optimal Worker in terms of memory and CPU availability to deploy the service. By default, EdgeIO uses a best-fit policy where the Worker node that has the most available CPU and memory is chosen. Once the optimal Worker is found, the deployment request is offloaded to the respective node where the service is subsequently deployed.
Figure 1.1 shows the native version of the service level agreement (SLA), where the developer can only specify constraints on computational resources like required CPU, memory and virtualization technology. Taking these requirements into account, the Root and cluster schedulers find a suitable target cluster and worker, respectively.

```
customerID: 1
applications:
- applicationID: 1
  app_name: hello
  app_ns: test
  microservices:
  - microserviceID: 1
    service_name: world
    service_ns: test
    virtualization: docker
    code: path/to/image
    memory: 1000
    vcpus: 1
    constraints:
    - type: geo
      location: 49.5,11.5
      threshold: 100
    connectivity:
    - target_microservice_id: 1
      type: latency
      threshold: 100
  - microserviceID: 2
    service_name: service
    service_ns: test
    virtualization: docker
    code: path/to/image2
    memory: 500
    vcpus: 1
    constraints:
    - type: latency
      area: Munich
      threshold: 100
    connectivity:
    - target_microservice_id: 2
      type: latency
      threshold: 100ms
```

Figure 1.2.: An example deployment descriptor.

Since one of the main goals of edge computing is to provide low latencies to users by making use of the geographically distributed infrastructure, developers should be able to define additional constraints on both the geographical area where the service should be deployed, and the maximum latency to users requesting a service from within a specified area. That allows a developer to deploy, e.g., latency-sensitive applications consisting of multiple microservices, each with latency thresholds for specified areas or related microservices. In case any of the constraints defined in the SLA is violated, the service can be migrated to a worker that does not violate the SLA ensuring that users keep experiencing low round trip times (RTTs). Additionally, those constraints should not only be possible to be defined for service-to-user (S2U), but also service-to-service (S2S) cases.
To allow the specification of such constraints in the SLA, the scheduling process of EdgeIO was extended and an updated version of the SLA (see 1.2) is accepted, which offers the possibility to specify the aforementioned constraints in addition to the already existing computational resource requirements. Furthermore, with the new SLA, developers can define multiple applications, each in turn consisting of several microservices in a single deployment descriptor.
Therefore, during the first step of the scheduling process, the RO also has to take into account the area covered by the workers within clusters. To achieve this, the COs periodically

send information about the location of their workers to the RO. Based on the geographical information about the clusters, the RO filters out those that do not have any workers in the specified area. For the remaining clusters the list of best fits for the service deployment is calculated as before.
Currently, there are two possible constraint types for the service-to-user (S2U) and service-to-service (S2S) communication. The first one is *latency*. In the S2S case the developer specifies the maximum latency this service can have to the service defined in the SLA. For the S2U case on the other hand, the threshold specifies the maximum allowed latency to users in the specified area. The second type, *geo*, specifies, for the S2S case, the maximum distance to the worker on which the service specified in the SLA is deployed. For the S2U connections, this constraint limits the maximum distance to the location defined in the SLA.
As a result, the scheduling process at the cluster level has to be adapted such that it takes the new constraints into consideration. The process for the *geo* constraint is rather straightforward. Since the CO is aware of its worker's positions, the distance to both, the area specified in the SLA for the S2U case, and the distance to the worker that runs the service for the S2S case are known. The scheduling process for the *latency* constraint however, is more sophisticated. The CO needs to be aware of the intra-cluster latencies to place the services in a way that they comply the SLA. To avoid a large and costly overhead generated by letting each Worker ping each other worker in the cluster, a network coordinate system (NCS) is deployed within each cluster to estimate the intra-cluster latencies with far less network measurements. The NCS embeds the CO and its Workers into a 2-dimensional coordinate system such that the Euclidean distance between the Workers approximate their RTTs. Then multilateration is used to predict the user position in the network coordinate system (NCS) such that the latency between the workers and the users can be estimated. The new scheduling process including the use of the Vivaldi NCS and multilateration is further explained in the implementation section.

1.2. Related Work

Edge technology's right to exist has already been proved in many use cases, however there are still open research challenges such as service scheduling on edge nodes [6]. Several works such as from [10], [11], and [12] tackle the problem of deciding whether to schedule a task on a mobile device or a local/internet cloud, also known as edge computation offloading problem. In [7] a scheduling approach in fog computing for the service module placement on fog nodes is described. However, they are solely focusing on the optimization of response time among components, without considering the end-to-end service time. A resource estimation and pricing model for IoT applications that estimates the amount of allocated resources for a given device is proposed in [13] . Again, their approach does not solve the problem of selecting a suitable node for the service deployment.
In the cloud environment, however, the scheduling of latency-sensitive services has been widely studied. virtual machine (VM) placement solutions for distributed cloud environments have been developed in [14]. These approaches try to find an optimal virtual machine

(VM) placement on nodes that minimizes the network latency among them. However, these mapping methodologies rely only on service requirements and physical infrastructure knowledge without considering user related information, such as geolocation, which is an important feature for the scheduling process at the edge. [15] describes a service allocation methodology that integrates user information. Their proposed provisioning algorithm is based on queuing theory to identify the number of VMs to be deployed in order to minimize the user waiting time. Unfortunately, this approach, which is intended for the use in infrastructure as a service (IaaS) clouds, only defines the number of VMs required to cope with the incoming load but is still missing VM placement policies which would lead to sub-optimal results in case of edge computing. In [6], the authors propose a score-based latency, bandwidth, and resource aware edge scheduling framework for latency-sensitive services. The algorithm schedules each service instance on the VM whose computing and network capabilities can optimize the service response time experienced by the end users. Similar to EdgeIO's scheduling process, the proposed scheduler first evaluates the eligibility of available VMs on edge nodes based on network and resource capabilities. Second, the services are scheduled in the most suitable VMs according to an eligibility score that combines a connectivity score which assesses the connectivity quality of a VM by evaluating the quality of the network routes connecting user groups to the nodes running the VM, a bandwidth score that assesses the available bandwidth of the node, and a resource score that measures the service load a given VM can handle. Based on this eligibility score the services instances are scheduled to maximize the overall quality of the selected VMs. This optimization problem is then mode led as a binary integer linear programming problem. The authors only consider latency-sensitive services which diminishes the applicability of the proposed scheduler in an edge environment, since latency-sensitivity is a very important criterion, but not the only one.
Another interesting approach is described in [16]. The authors propose a scheduling algorithm that can be applied to VMs and lightweight containers. For each service to be deployed, the presented algorithm calculates the VMs that minimize the response time for end users based on their compute and network capacities. For this, an eligibility score based on network latencies, bandwidth, processing power, and reliability is calculated. Then the service is deployed to the most suitable target. To evaluate the latencies to potential users, first the users are clustered by assigning each user to its geographically closest edge node. For the network evaluation, the authors assume that the edge nodes are connected to each other, and that users are connected to the nearest Base Station. Then, to evaluate latency, the edge network is first modelled as a weighted graph, where each node is an edge data center. The edges connecting two nodes define the connections between data centers and the edge weight describes the estimated latency. Furthermore, for each user group g, the path with the lowest latency from node n to g is calculated using Dijkstra's algorithm. However, it is not described in detail how the latencies are measured in order to calculate estimates, but only referred to the fact that the required latency data for Dijkstra's algorithm can be obtained with the help of active measurements by e.g. ping measurements or passive network monitoring and aggregation of connection-relevant information.

1.3. Contribution

To tackle the aforementioned issues the scheduling process both at the root and cluster level was extended, a network coordinate system (NCS) to estimate intra-cluster node-to-node latencies was deployed, and a monitoring component was added to the Worker nodes which monitors all services that are deployed on the nodes and triggers an alarm to the cluster orchestrator (CO) in case of service level agreement (SLA) violations.
The adapted scheduling at the root level now also considers the new constraint types *latency* and *geo* that can be specified in the SLA as service-to-user (S2U) and service-to-service (S2S) constraints. As a results, the root scheduler not only filters out clusters that can not provide the required computational resources, but also filters cluster that, in the case of a S2U constraint, do not have any worker nodes located in the specified area and in the S2S case, do not have workers located close to the specified target worker. Similarly, the cluster scheduler not only verifies, whether the requested service can be deployed on the potential Worker node, but also checks which node is located within the area defined in the S2U constraint. However, it is not possible to evaluate a priori whether a node fulfills the latency constraint in a certain area because at that time no information about S2U connections are available. To improve the chances of achieving rather low latencies the initial service placement is based on geographical distance to the area specified in the SLA. In the S2S case, for a *geo* constraint, the cluster scheduler can find Workers that are in range of the target worker. In case of a *latency* constraint the scheduler can use the Vivaldi network to find workers that provide low enough latencies to the target workers. This does not require any user related information.
Once a service was deployed to a node, the service is monitored by the nodes monitoring component. This component regularly checks whether the node's resources are within the specified requirements and the node is still fulfilling the service constraints. If constraint violations occur, the monitoring component triggers an alarm containing the corresponding violating measurements to the cluster orchestrator (CO) to initiate a service replication or migration, depending on which SLA violation handling strategy the developer specified. In case of a S2U *latency* violation, the CO receives the information from the violating worker about the latencies between the node running the service and user groups to use during the re-scheduling of the service. These latency information, amongst others, are used to find a node that offers a lower latency based on the deployed NCS that allows to estimate latencies without requiring all nodes to ping each other.
Thus, the new **c**onstraint-**a**ware **s**cheduling **a**lgorithm (CASA) finds a suitable worker that satisfies all defined S2U and S2S constraints, taking into account contextual knowledge such as user and network related data. If SLA violations occur during the lifetime of the deployed service, the monitoring component ensures that the service is automatically deployed to a new worker that meets the SLA again. This allows EdgeIO to keep fulfilling the SLA and hence ensuring good user experience.

1.4. Thesis Structure

The first part of this master's thesis is an introduction to the problem at hand, related work, and the contributions made to solve the described scheduling challenges with respect to contextual knowledge. Subsequently, information about the background of cloud and edge computing, network coordinate systems (NCSs) are presented. The major part of this work describes the implementation and integration of the newly developed **c**onstraint-**a**ware **s**cheduling **a**lgorithm (CASA) in EdgeIO. The modifications of the scheduling process on root and cluster level, as well as the new component for monitoring the service level agreement (SLA) are presented. Following this, CASA is evaluated and the results are compared to EdgeIO's native, resources-only scheduling algorithm. Finally, the work is concluded with some final remarks and limitations, and possible future work to improve the accuracy of the updated scheduling process.

2. Background

The problem of scheduling exists in numerous areas and it continues to evolve over time along with industry and technology [17]. With the development of computers, much attention has been paid to scheduling in computer processors. The most common objective is the minimization of task completion times, also known as *makespan* [18]. Task scheduling is a known non-deterministic polynomial-time hard (NP-hard) problem [19]. Therefore, heuristic methods have to be used, which are basically algorithms that find an approximate optimal solution in fewer then polynomial time [20]. Beside some peculiarities, the basic principles remained the same as in scheduling activities among machines in production. In supercomputers, multiprocessor scheduling considers several parallel processors with the same capacity. Additionally, the source of data is considered to be centralized and connected by high speed channels, in a way that activities can exchange messages quickly [21]. Progress in the area of computer networks allowed clusters of homogeneous computers to act as a multiprocessor computer with distributed data sources. However, when compared to supercomputers, clusters initially had a slow communication channel between processors resulting in more expensive data exchange. As a result, the scheduling of jobs in computing clusters led to another branch of research, the scheduling in distributed systems [22]. With improvements in the aforementioned progress with computer networks, the connection among computing nodes in clusters became much faster. On the other hand, simultaneous with advances in networking, new applications demanded more and more bandwidth, storing and exchange of massive volumes of data. In the late 90's grid computing emerged as a heterogeneous collaborative distributed system that evolved from homogeneous distributed computing platforms [23]. They are shared systems that enclose potentially any computing device connected to a network and hence possibly the edge.

2.1. Edge Computing

Cloud computing, since its introduction in 2005, has significantly changed the way we live and work [24]. Applications such as Facebook, Netflix, and Google Apps have long been firmly integrated into our daily lives. Internet of Things (IoT) was first mentioned in 1999, and the term started its life as the tile of a presentation linking the then-novel idea of radio-frequency identification (RFID) to supply chain management [25]. Before the time of IoT, virtually all computers, and therefore the Internet, depended on people for information and data. Most of the available data on the Internet was therefore generated by people. At that time, the term Internet of Things (IoT) was used to describe the idea of enabling computers to independently

collect and process data with the help of for example sensors. With the immense proliferation of Internet of Things (IoT) and data-intensive and latency-sensitive applications in the recent years [26], a large quantity of data will be generated by things firmly integrated in our daily life. Simply put, as more and more data is produced at the edge of the network, it is more effective to process it at the edge as well. For example, an autonomous car generates 40 terabytes of data every hour [27]. If all cars had to upload the collected data to the cloud for processing, this would congest the core network and lead to high response times, and hence to bad user experiences or potentially fatal situations in scenarios with very strict latency requirements such as e-health.
Consequently, the need to move the data collection and processing closer to the edge of the network, and therefore closer to the end users, has become increasingly important. An important characteristic and at the same time challenge of edge computing is the immense heterogeneity of the edge devices, which have limited computing capacities. The cloud-edge continuum that edge computing envisions to create is visualized in figure 2.1. At the top is still the cloud environment. Further down at the edge of the network and thus much closer to the users than the cloud counterpart, is the edge layer. Within it are many different small data centers, the Edge devices, possibly exchanging data with the cloud or with other edge devices. Finally, there are the users, who access the services that have been deployed in the edge layer. These users can be a variety of consumers such as autonomous cars, smart grids and wearables. A major challenge posed by the extreme heterogeneity is to find suitable targets on which to deploy the required applications so that all user requirements are satisfied. Since neither the users nor the edge data centers are necessarily stationary, service placement must always ensure that changes in the topology can be detected quickly such that the quality of service can be maintained throughout the lifetime of the service.

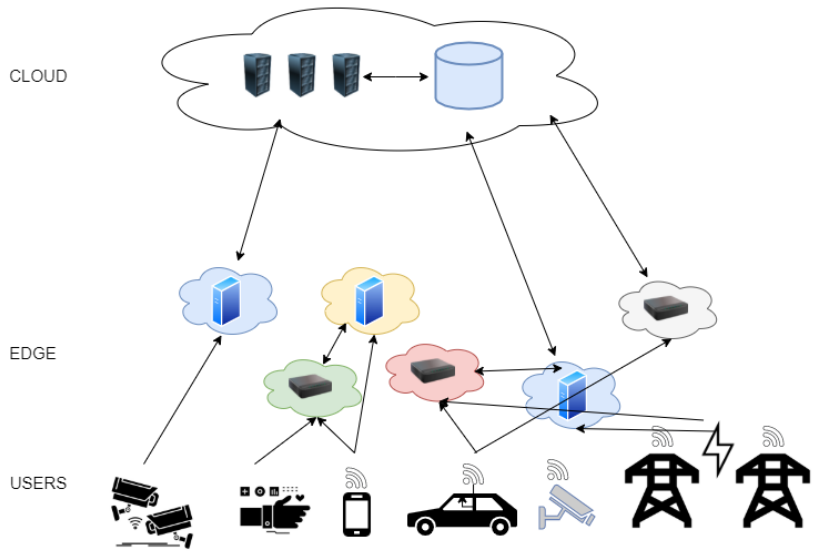


Figure 2.1.: Cloud-to-edge continuum.

2.2. Service Scheduling in the Cloud

Cloud computing is currently the predominant hosting solution for internet services due to the economical and infrastructural advantages it provides. The massive pool of resources characterizing cloud data centers benefits from significantly lower marginal costs due to economies of scale and guarantees high level of reliability and dynamism, which allows the providers to scale up/down the allocated resources based on current needs [24]. It offers virtualized computing resources as services to the users, while hiding technical aspects regarding the management of said resources and allows users to access computing resources as services remotely through the Internet. Cloud computing is distinguished from traditional computing paradigms by its scalability, adjustable costs, accessibility, reliability, and on-demand pay-as-you-go services. As clouds serve millions of users simultaneously, it must have the ability to meet all users' requests with high performance and guarantee of quality of service (QoS) [28]. Consequently, clusters and grids can be part of data centers in the cloud computing infrastructure, demanding new optimization objectives. Cloud computing is broadly defined by Salot as an elastic execution environment of resources involving multiple stakeholders and providing a metered service at multiple granularity for a specific level of quality of service (QoS). In a more specific view, a cloud is a platform or infrastructure that enables execution of code e.g., services, applications, etc. in a *managed* and *elastic* fashion. *Managed* means the reliability according to pre-defined quality parameters is automatically ensured and *elastic* implies that the resources are put to use according to actual current requirements observing overarching requirement definitions. Additionally, elasticity implicitly includes both up and downward scalability of resources and data, but also load-balancing of data throughput. Especially in the cloud there is a high communication cost that prevents well known task schedulers to be applied in large scale distributed environments. Job scheduling is the most important task in cloud computing environments, because users have to pay for resources used based upon time. Hence efficient utilization of resources must be important and for that scheduling plays a vital role to get the maximum benefit from the resources [30]. In general, the goal of scheduling algorithms in distributed systems is spreading the load on processors and maximizing their utilization while minimizing the total task execution time. But, task scheduling cannot be done just based on a single criterion, and rather takes into account a set of rules that can be considered as an agreement between users and provider of a cloud. This agreement is actually nothing more than the service level agreement (SLA) that the user makes with the provider. In order to be able to offer the users good quality of service, the tasks must be mapped to the available resources in such a way that the desired quality can be achieved [31].
Scheduling algorithms, which are based on heuristics to find a good solution for the NP-hard task scheduling problem, can be classified into cluster scheduling and list scheduling. Cluster scheduling tries to reduce the makespan of a task by mapping the subtasks to a large number of available processors that do the work. In list scheduling algorithms, first each subtask gets scored based on certain criteria and then gets appended to a list to wait for their turn from highest to lowest priority.

2.3. Service Scheduling at the Edge

As a significant challenge to the centralized paradigm of cloud computing, the rapid progress in smart devices and network technologies has enabled new categories of inter-based services with strict end-to-end latency requirements and a vast amount of data to process [5]. Since centralized deployment solutions fail to satisfy strict latency requirements and network architectures will soon become incapable of handling the massive amount of data communication, more geographically distributed approaches are necessary. Edge computing proposes to place computation and storage capabilities at the edge of the network in the form of micro scale data centers, which is a highly promising solution for the aforementioned latency-sensitive services since it allows the deployment of services in close proximity of their end users to meet response time requirements [16].
Latency-sensitive and data-intensive applications, such as IoT, banking, and health [26], are leveraged by edge computing, which extends the cloud ecosystem with distributed computational resources in proximity to data providers and consumers. This brings significant benefits in terms of lower latency and higher bandwidth. However, edge computing has, by definition, very limited resources with respect to the cloud counterparts. Thus, there exists a tradeoff between proximity to users and resource utilization. Moreover, service availability is a significant concern at the edge of the network, where extensive support systems as in cloud data centers are not usually present [16].
Unlike in a cloud environment, edge devices can be geographically extremely distributed. To find a suitable target for service deployment, a scheduling algorithm must take into account user- and network-related information in addition to computational resources. Without knowing latencies to specific user regions or geographic coordinates of edge devices, no target can be computed that satisfies all possible requirements for maximum latencies or position. This increased number of variables compared to the cloud scenario is the reason why service placement at the edge remains so complicated. The heterogeneity of available information has to be used to find the sweet spot to achieve user proximity and efficient resource utlization.

2.4. Orchestration Platforms

During the traditional deployment era, organizations ran their applications on physical servers. Since there was no way to set resource constraints, this often led to resource allocation problems. For example, if multiple applications are running on a single server, one application may take up the majority of resources, causing other applications to underperform. A possible solution to this would be to deploy each application to its own server, but this would lead to unused resources and high costs due to a potentially large number of servers. To overcome this problem, virtualization, which allows multiple virtual machines (VMs) to run on a single server, was introduced [32]. This limits resources and allows applications to be isolated between VMs, which brings a certain level of security as information from an application cannot be easily accessed. This better server resource management also improves scalability, because additional applications can be added easily, and reduces hardware costs, because multiple VMs can be placed on the same server. Containerization was then introduced to meet further requirements. Containers are similar to VMs, but have looser properties to e.g., share the operating system between applications and bring other advantages such as being lightweight, continuous integration and continuous deployment, and much more [33]. Both, in the cloud and in the emerging edge computing environment, developers use orchestration platforms to transparently deploy their applications on a hardware infrastructure. The idea behind these platforms is to abstract the layer of individual compute nodes and associated hardware maintenance so that the developer only has to worry about business requirements. They can specify their requirements in a service level agreement (SLA) and only need to provide the source code of the application for which the deployment is handled by the orchestration platform.
This section describes some well-known frameworks and platforms and covers their service scheduling in particular.

2.4.1. Kubernetes

The official website [34] describes *Kubernetes*, also known as *K8s*, as a portable, extensible, open-source system for automating, scaling, and managing containerized workloads and services. The platform automates many of the manual processes involved in deployment and scaling and helps to cluster a group of hosts running Linux containers to manage them easily and efficiently and is the de-facto standard for container orchestration.
Such a cluster has at least one worker node that deploys the pods, which are the smallest unit of work in Kubernetes, and a certain number of nodes that make up the control plane, which manages the workers and deploys the pods. This control component has a global view of all activities within a cluster and can make high-level decisions about the state of workers and their deployments.
The *kube-scheduler* is the default scheduler for Kubernetes and runs as part of the control plane. The scheduler watches for newly created pods that have no node assigned. For every pod that the scheduler discovers, the scheduler becomes responsible for finding the best node for that pod to run on. Each container in Pods has different resource requirements and each

pod also has different requirements. Therefore, available nodes must be filtered with these requirements in mind. Similar to EdgeIO, Kubernetes considers a node feasible, if it meets the requirements of the pod. In the event that a cluster has no nodes that meet the requirements, the pod remains unscheduled until the cluster has a feasible node. The scheduler evaluates the feasible nodes using various scoring functions and ultimately selects the node with the highest score to assign the pod to, or a random node in case multiple nodes have the same score. The criteria by which the filtering and scoring phases are performed can be defined in a scheduler configuration. Various scheduling plugins are available for the configuration, such as for the filter phase e.g., *NodeName*, which checks if the pod node name matches the current node, or *VolumeRestrictions* which checks that the volumes mounted in the node satisfy the pod's restrictions, and for the scoring phase for example, *ImageLocaility*, which favors nodes that already have the container images the pod runs. However, the kube scheduler cannot be configured to consider contextual knowledge, which makes it unsuitable for the use in an edge computing environment.

2.4.2. KubeEdge

KubeEdge is an open source system with the goal to create a continuum from the cloud to the edge, by extending native containerized application orchestration capabilities to hosts at the edge. It builds upon the functionality of Kubernetes and extends and modifies it for use cases on edge nodes. KubeEdge itself is divided into different components. The central cloud component, which is required for the management of the edge nodes, communicates with the non-deterministic polynomial-time hard NP-hard (API) server of a Kubernetes cluster. The necessary Kubernet cluster can also serve as a host for the KubeEdge cloud component. Due to the strong integration of Kubernetes with KubeEdge, users can use the common Kubernetes commands. However, KubeEdge does not have its own scheduling component, but instead relies on Kubernetes' *kube-scheduler* [35].

2.4.3. ioFog

ioFog is an exciting project from the *Edge Native* working group of the Eclipse foundation. It is a complete edge computing platform that provides capabilities to build and run applications at the edge of the network at enterprise scale. It brings cloud native architectures like *Kubernetes* to the edge to allow developers to easily deploy, orchestrate and manage microservices and any edge device. The basic principle of the *ioFog* architecture is the so-called *edge computing network (ECN)*, which in turn consists of the following three components:

Controller: Orchestration, lifecycle management and deployment of distributed microservice applications.

Agents: Lightweight universal container runtimes installed on edge devices that manages the device's computational resources and microservice lifecycles.

Service Mesh: Facilitates communication between controller, agents, and distributed microservice applications.

Unfortunately, like *KubeEdge*, *ioFog* does not have its own scheduling component and can be used either without a scheduler or with a custom *Kubernetes* scheduler [36].

2.5. Latency Monitoring

Many applications like e.g. in the area of banking and health are often very latency sensitive [37]. They often have several distributed components e.g., frontend, backend and database, which need to communicate over low latency connections to keep response times low. Therefore network operators need to continuously monitor the quality of the network, especially latency, in order to avoid critical situations by quickly routing traffic over low-latency paths and avoiding high-delay segments [38]. This can be done by actively conducting ping measurements. However, there are some factors complicating this. First, some data centers restrict access to customer servers. Second, with end-to-end ping measurements one cannot acquire information on path segments [39], making it hard to calculate low latency paths consisting of low delay segments. Alternatively, Kompella, Levchenko, Snoeren, and Varghese proposed to instrument switches with a hash-based primitive that records packet timestamps and measures latencies. However, these kinds of methods required hardware modifications that firstly, may not be available in regular switches and secondly, can not be expected from the operators of the components of EdgeIO.
Yang, Cai, and Xu propose a link layer discovery protocol (LLDP) that is utilized by controllers to quickly discover changes in the underlying network topology. They add timestamps into an optional field in link layer discovery protocol (LLDP), so that the controller can estimate the latency on a single path segment. The approach does not use active network measurements and relies on passive measurement using the inserted timestamps. This allows high accuracy to be achieved even in larger networks with more complex conditions. The collected link latencies are then stored in a matrix to infer the aggregated path latency. In [42] the authors propose a software-defined latency monitoring (SLAM) framework between any two network switches that does not require specialized hardware, but rather uses software-defined networks (SDNs). SLAM is deployed on the network controller and dynamically sends specific probe packets to trigger control messages from the first and last switches of a path to a centralized controller. Based on the arrival timestamps of said control messages SLAM then estimates the path latencies. The estimation happens in three steps. First, rules are installed on every switch along the path. These rules instruct every switch to forward the matched packets to the next switch on the path. Furthermore, the first and last switch also generate notifications to the controller. Next, SLAM sends probes that are built such that they only match to the aforementioned monitoring rules and hence traverse only the monitored path. Finally, the path latency is estimated based on the timestamps at which the notification messages from the first and last switch are received. The advantages of SLAM over other latency monitoring techniques are, first, that by exploiting control packets that are inherent to SDN, SLAM does not require hardware modifications or access to end hosts. However, these SDN based approaches aim to gather latency information in data center networks and the potential user groups are not part of this SDN, and hence S2U latencies cannot efficiently be evaluated.

2.6. Network Coordinate Systems

Due to the rapid development of internet technology, the scale of networks has grown enormously and many large-scale distributed services have emerged. As a natural consequence the users' requirements for the quality of such network services are also constantly growing. Inter-node latency is one of the key parameters for measuring network performance, and many internet applications relay on accurate measurements of such a parameter, such as peer-to-peer file sharing [43] and online games [44].
The most straightforward way of latency measurement is to perform end-to-end measurements between the network nodes. While this approach is the most accurate one, it is only suitable for small-scale networks. In a *n-node* network, there are $n(n-1)$ end-to-end links which would result in a measurement overhead of $\mathcal{O}(n^2)$.
Therefore, explicit network measurements are often unpractical, because the measurement costs can easily outweigh the benefits especially for large networks. Many network applications can benefit from the possibility to predict round-trip times to other hosts within the network without having to perform direct network measurements. Network Coordinate Systems (NCS) can be used to predict the inter-node delays by embedding the network space into a certain measurement space. The measured latency is converted into the calculation of the distance between nodes in the space, which greatly reduces the overhead caused by explicit network measurements.
Most Network Coordinate Systems compute synthetic network coordinates in some coordinate space such that the distance between the synthetic coordinates between two hosts predicts the round-trip time between them in the Internet. Thus, if a host x learns the coordinates of a host y, x does not have to perform an explicit measurement to obtain the RTT to y. Instead, the distance between x and y in the coordinate space is an accurate predictor of the RTT [45]. Most NCSs use a model based on Euclidean distance. In this calculation model, the network coordinate system maps n nodes into a *d-dimensional* space, and defines the network coordinates of any node i as $x_i = r_{i,1} + r_{i,2} + \cdots + r_{i,d}$. The predicted network latency can then be obtained by calculating the distance of any two nodes i and j as $D(i,j) = ||x_i - x_j||$ [46].
The accuracy of NCSs that approximate latencies based on distances, is largely shaped by the number of triangle inequality violations (TIVs). The triangle inequality states that the sum of the length of any two sides of a triangle must be greater than or equal to the length of the remaining side. Figure 2.2 visualizes this violation. In this case, the path from host **A** to C via B would be shorter than from **A** directly to **C**, since $RTT(A,B) = 32ms$, $RTT(B,C) = 24ms$, $RTT(A,D) = 65ms$ and thus $RTT(A,B) + RTT(B,C) < RTT(A,C)$, which is a violation of the triangle inequality. These violations are very common in the internet [47], which results in the nodes not fitting into the Euclidean space in such a way that the distances perfectly represent the latencies, since the Euclidean space does not allow TIVs.

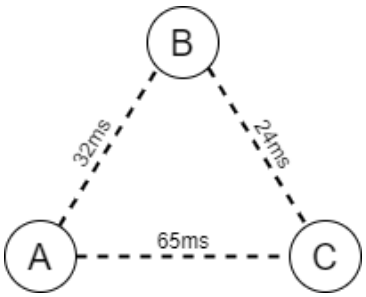


Figure 2.2.: Triangle Inequality Violation.

Furthermore, there are various types of network coordinate system (NCS). Those that rely on distance measurements, can be divided into two types: NCSs with stationary reference points and without. NCSs that require stationary reference points first calculate the coordinates of these, which do not change afterwards. Then the other common nodes in the network calculate their position based on the known and fixed coordinates of the reference points and the measured distance to them. However, this is less suitable for use in edge computing environments, where it cannot be assumed that the reference points are stationary. One could, of course, define stationary nodes, but this would result in more restrictions on the infrastructure, which would not make sense in view of the fact that solutions without reference points also exist. Besides these NCSs there are also some based on completely new approaches like matrix decomposition [48], singular value decomposition [49], and non-negative matrix factorization [50], that have the advantage of being less influenced by TIVs, but bring other problems and requirements. For use in the edge environment and integration with EdgeIO, the Vivaldi NCS has emerged as useful because it is very lightweight, scalable, and decentralized and does not rely on predetermined fixed reference points.

3. EdgeIO Service Scheduling

3.1. Architecture

Figure 3.1 shows the system architecture of EdgeIO. EdgeIO is an orchestration framework designed for edge computing applications that organizes the edge resources into distinct clusters (see cluster 1 and *k*). The platform consists of a centralized controlplane, the root orchestrator (RO) and several cluster orchestrators (COs) each with its own worker nodes. A cluster in EdgeIO is a very abstract entity, since it can be used by an operator to deploy multiple clusters to logically segregated resources e.g., geographical regions or logical domains. EdgeIO consists of three distinct components, the RO, the CO, and the workers, that are described in the following sections.

3.1.1. Root Orchestrator

The root orchestrator (RO) is the centralized control plane of the platform and has the responsibility to manage the participating clusters. Furthermore, the root is also the touch point for developers. When a developer wants to deploy a service or an application consisting of multiple microservices, he sends a file containing the service level agreements (SLAs) for the RO via and API call to the root's system manager. The system manager registers the service in the local database and then contacts the root scheduler ①, that in turn calculates a priority list of clusters suitable for the deployment of the requested service or services. Once the priority list was computed, the scheduler picks the cluster with the highest priority and sends its result back to the system manager. Consequently, the system manager executes an API call to the target cluster orchestrators (COs) deployment endpoint ②. The service manager monitors the state of all operational services in EdgeIO such as service addressing, virtual subnetworks, external service discovery requests, and inter-cluster service-to-service (S2S) communication[1].

[1] A detailed description of the service manager is beyond the scope of this thesis, as the focus is on the scheduling process.

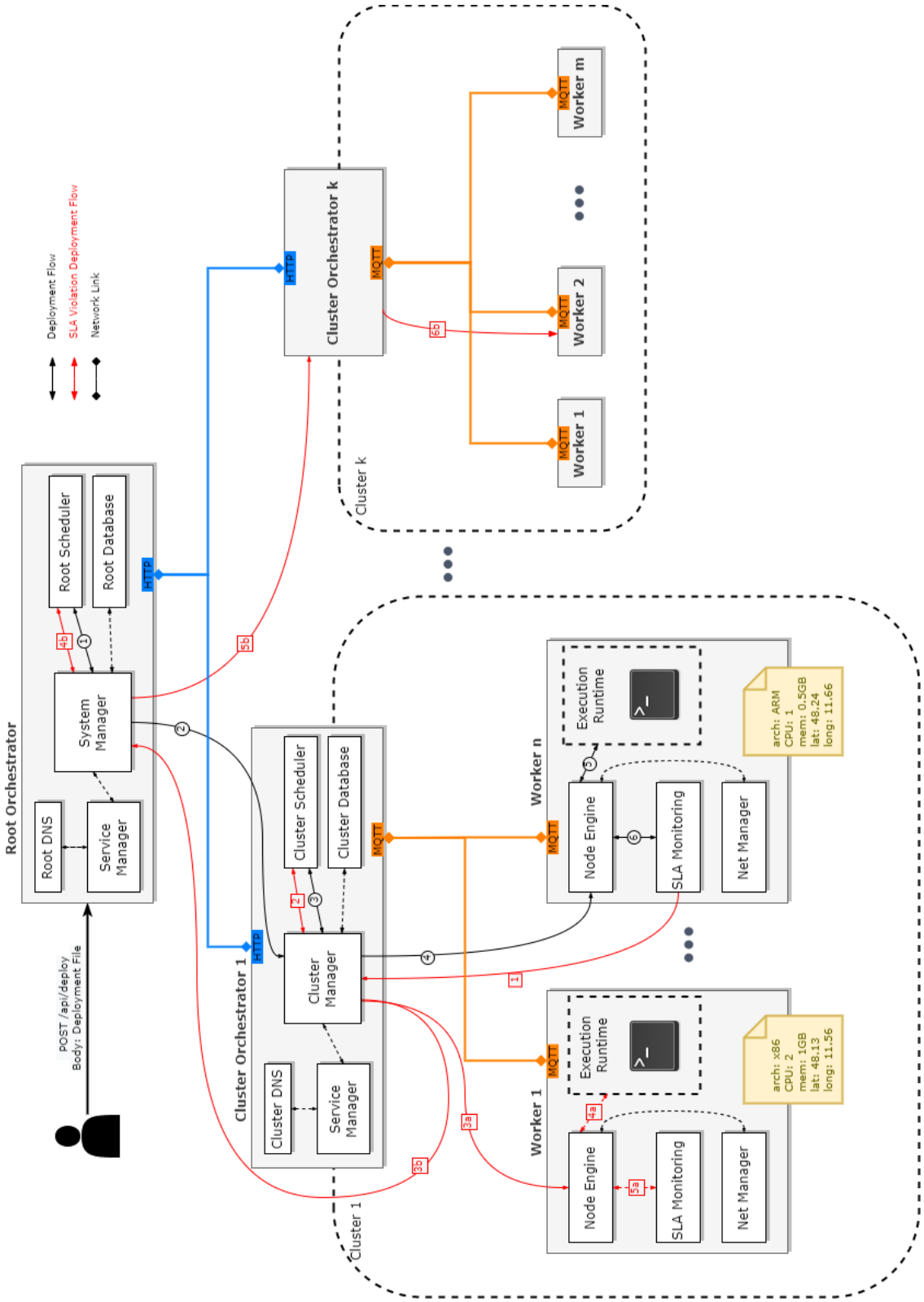


Figure 3.1.: EdgeIO Architecture overview including the user initiated deployment and SLA violation induced deployment flow.

3.1.2. Cluster Orchestrator

The cluster orchestrator (CO) contains similar components as the RO since its responsibility is to manage the edge resources belonging to this cluster. When a CO initially joins EdgeIO, it registers itself in the root and establishes a bi-directional hypertext transfer protocol secure (HTTPS) connection.
Once the deployment endpoint of a cluster manager was called the cluster manager registers the job in the database and contacts its local scheduler ③. The scheduler computes suitable workers based on the requirements and constraints contained in the SLA file sent by the developer. Once a suitable node was found the result is sent back to the cluster manager. The cluster manager then tells the resulting node ④ via message queue telemetry transport (MQTT) to deploy the service.

3.1.3. Worker

When a worker node joins a cluster it contacts the cluster orchestrator (CO), that in turn registers the resources of the node in the local database. Typical for the edge environment, worker nodes have very heterogeneous capacities and capabilities, such as CPU, GPU support, virtualization runtimes, etc., which they send to the cluster manager at the time of registration. To allow information exchange within a cluster, the CO establishes a MQTT message broker that is subscribed by the Worker nodes registered to that cluster. The same communication channel is used by each Worker to periodically send information about current resource utilization, geographic location, etc.
When a certain node receives a message on the deployment topic it subscribes to, depending on the virtualization technology specified by the developer, the node deploys the service ⑤ and registers it to the *net manager*[2] and *SLA Monitoring* component ⑥. The deployed service is then monitored by the workers monitoring component. As soon as the service violates one of the provided constraints, the monitoring component triggers an alarm to the CO via MQTT 1, which then contacts its Scheduler 2 to find a new Worker that fulfills the SLA again. If the scheduler finds a suitable worker, the CO forwards the deployment job via MQTT to the target worker 3a, who then deploys the service 4a and registers it with its monitoring component 5a. In case the Scheduler cannot find any suitable Worker in the same cluster, the deployment request is forwarded to the root orchestrator (RO) 3b. Following that, the RO contacts its scheduler 4b to find a suitable cluster 5b, which in turn finds a Worker on which the service can be deployed 6b.

[2] A detailed description of the network manager is beyond the scope of this thesis, as the focus is on the scheduling process.

3.2. Vivaldi Network Coordinate System

For latency-based scheduling, the cluster scheduler needs information about the latencies between the Workers of a cluster and to user groups. To avoid the overhead generated by explicit network measurements, the Vivaldi network coordinate system (NCS), as proposed in [45], is used to predict round-trip times (RTTs) between hosts without end-to-end *ping* measurements. Vivaldi is a light-weight, distributed algorithm that embeds nodes of a network in a synthetic coordinate system such that the distance between two synthetic coordinates accurately predicts the RTT between the two hosts.
Since the hosts in a Vivaldi network need only little information from other hosts to update the position, the required information can piggy-back on already existing communication channels, thus generating hardly any extra overhead. The connection properties of the Internet have the greatest influence on the accuracy of the Vivaldi predictions. For example, if there is hardly any delay, and the Internet is well-enough connected so that there is an almost direct path between each pair of nodes, then synthetic coordinates can predict latencies very well. However, these circumstances are not given because, for example, packets are often sent over non-optimal routes, because only a few pairs of nodes are directly connected, and because transmission time and processing in the routers cause delays. The resulting distorted latencies make it impossible to embed the nodes in a 2-dimensional coordinate system in such a way that the latencies can be predicted perfectly. Therefore, the NCS needs a strategy to minimize the prediction errors of the coordinates. No low-dimensional coordinate space would allow Vivaldi to predict the round-trip times between hosts exactly, because, for example, Internet latencies violate the triangle inequality, i.e., that the sum of the lengths of any two sides must be greater than or equal to the length of the reaming side [47]. Therefore, the algorithm tries to find coordinates that minimize the prediction error.

3.2.1. Prediction Error

Let L_{ij} be the actual measured round-trip time between nodes i and j, and x_i and x_j be the coordinates assigned to node i and j, respectively. The errors in the coordinates can be characterized by using a squared-error function:

$$E = \sum_i \sum_j (L_{ij} - ||x_i - x_j||)^2 \tag{3.1}$$

with $||x_i - x_j||$ being the distance between the coordinates of nodes i and j. The authors chose the *squared error function*, because it has an analogue to the displacement in a physical mass-spring system: minimizing the energy in a spring network is equivalent to minimizing the *squaerd-error function* [45]. For the implementation of the Vivaldi network in EdgeIO 2-dimensional coordinates with the standard *Euclidean* distance function are used. Mapped to the *spring relaxation problem*, the algorithm places a spring between each pair of nodes (i, j) with a rest length equal to the measured round-trip time L_{ij}. Then, the current length of a spring is considered to be the distance between the nodes in the coordinate system. The potential energy of such a spring $U_{el} = \frac{1}{2}kx^2$ is directly proportional to the square of the

change in length and the spring constant, meaning that the sum over the potential energies in the entire spring system is exactly the chosen error function. Since the error function is equivalent to spring energy, it can be minimized by simulating movements of nodes under the spring forces. The minimum energy configuration of the spring system corresponds to the minimum error coordinate assignment in the NCS. From Hooke's law (3.2) it can be shown that the force vector is as shown in equation 3.3, where $(L_{ij} - ||x_i - x_j||)$ is equivalent to the displacement of the spring from rest $\Delta x = x - x_0$ from 3.2, and with x being the length of the spring in relaxed state, i.e., perfectly representing the round-trip time between the nodes connected by the spring, and x_0 being the length of the spring in either a compressed or relaxed state representing the Vivaldi prediction for the RTT of the two nodes connected by the spring. This quantity gives the magnitude of the force exerted by the spring on i and j (ignoring the spring constant). On one side, when the spring is stretched beyond its equilibrium, the force is pulling the attached node i back towards the anchor node j, i.e., the force vector, too, points towards the anchor node. Otherwise, if the spring is compressed, the force pushes it away from the anchor. The unit vector $u(x_i - x_j)$ gives the direction of the force on node i

$$F = d \times \Delta x = d \times (x - x_0) \tag{3.2}$$

$$F_{ij} = (L_{ij} - ||x_i - x_j||) \times u(x_i - x_j) \tag{3.3}$$

The net force on i (F_i) is the sum of forces exerted on node i from other nodes:

$$F_i = \sum_{j \neq i} F_{ij} \tag{3.4}$$

To simulate the evolution of the spring network small intervals of time are considered. At each interval, the algorithm moves each node (x_i) a small distance in the coordinate system towards the direction of F_i and then recomputes all the forces. Hence, the coordinates at the end of a time interval are:

$$x_i = x_i + F_x \times t \tag{3.5}$$

where t is the length of the interval. The magnitude of the node's movement at each interval is determined by the size of t. All nodes start at the origin of the coordinate system. Therefore, in the beginning, the unit vector $u(x_i - x_j) = u(0)$ and thus has zero length. To separate them initially, $u(0)$ is defined to be a unit vector with a random direction.

3.2.2. Adaptive Timestep

The main challenge of the Vivaldi network is the convergence of the system to coordinates that predict the round-trip times accurately. Similar to machine learning, where the learning rate determines the rate of convergence, in the Vivaldi network the timestep δ does the job. Large values of δ result in large coordinate adjustments. Therefore, if all nodes use large timesteps, the network participants tend to oscillate and hence do not converge to useful coordinates. A small δ on the other hand will result in a slow convergence of the network coordinates. Optimally, Vivaldi should provide both quick convergence and no oscillation. To achieve that, the authors of Vivaldi propose to use an adaptive timestep, by varying δ depending on how certain a node is about its coordinates. This certainty is evaluated by each node by comparing the new measured RTT with the predicted RTT and maintaining a moving average of recent relative errors. Consequently, if a node first joins the network ,large values of δ will help the node to move quickly to a more accurate location. Once the node reached that position, the δ values become smaller, so that the node can refine its position to reach better precision. This adaptive timestep is implemented in the Vivaldi network as follows:

$$\delta = c_c \times \frac{e_i}{e_i + e_j} \tag{3.6}$$

Using this δ allows efficient adaption in the following three cases: First, an accurate node, i.e., low local error e_i that samples an inaccurate node, will not move much, second, an inaccurate node sampling an accurate one will move a lot, thirdly, two nodes of similar accuracy split the difference. This implementation of the adaptive timestep provides the desired properties for the Vivaldi network: fast convergence, low oscillation, and resilience against erroneous nodes.

3.2.3. The Vivaldi Algorithm

Each node that is part of the Vivaldi network, simulates its own movement in the spring system and maintains its own current coordinates, starting at the origin. Every time a node communicates with another participant, it measures the round-trip time to that node and also learns that node's coordinates. During an update, a node is pushed for a short time step by the corresponding spring. Each of those movements reduce the node's error with respect to one other node in the system. As the nodes of the Vivaldi network periodically communicate to other nodes, they converge to coordinates that predict all RTTs in the network well.
Figure 3.2 shows the pseudocode for Vivaldi. The update algorithm requires the measured round-trip time between node i and j (L_{ij}), the coordinates (x_j) and error (e_j) of node j as input. The coordinates and error estimate of the node that updates its position are x_i and c_i, respectively. First, the update procedure determines the RTT estimate by calculating the distance from node i to node j (line 12). Note, that this implementation of the Vivaldi update process also works with the proposed 2-dimensional euclidean coordinates augmented with

a height vector. Hence, the distance between two coordinates is computed as follows:

$$\begin{aligned} ||[x, x_h]|| &= ||x|| + x_h \\ [x, x_h] - [y, y_h] &= [(x - y), x_h + y_h] \\ ||[x, x_h] - [y, y_h]|| &= ||[(x - y), x_h + y_h]|| \\ &= ||x - y|| + x_h + y_h \end{aligned} \tag{3.7}$$

Then, the weight based on local and remote error is computed (line 15), followed by the local relative error (line 17). Consequently, the weighted moving average of the local error is updated (line 19). Following this, the aforementioned adaptive timestep is updated (line 21) and finally, the force magnitude is computed (line 23) and applied to the coordinates of node i (x_i) in order to finish the update process. Since Vivaldi is fully distributed, efficient and reactive, i.e., if the underlying topology changes, the nodes in the Vivaldi network update their coordinates accordingly, it is very suitable for the use in the dynamic edge computing environment.

```
1  def update_vivaldi(L_ij, x_j, e_j):
2      """
3      This node's coordinates and error estimate are x_i and e_i.
4      c_e and c_c are tuning parameters.
5
6      Input:
7          - L_ij: Measured RTT from this node i to remote j
8          - x_j: Coordinates of node j
9          - e_j: Prediction error estimate of node j
10     """
11     # Get RTT estimate by calculating the distance to node j
12     d = ||x_i - x_j|| + x_{i,h} + x_{j,h}
13     # Sample weight balances local and remote error to push
14     # node i in proportion to the error
15     w = e_i / (e_i + e_j)
16     # Compute relative error of this node
17     e_rel = |d - L_ij| / L_ij
18     # Update weighted moving average of the local error
19     e_i = e_rel × c_e × w + e_i × (1 - c_e × w)
20     # Update adaptive timestep
21     δ = c_c × w
22     # Apply force exerted on i and update coordinates
23     F = δ × (L_ij - dist)
24     x_i = x_i + F × u(x_i - x_j)
```

Figure 3.2.: The Vivaldi update algorithm.

3.2.4. Integration in EdgeIO

To integrate the Vivaldi network coordinate system (NCS) into EdgeIO, the implementation of the workers and the cluster orchestrator (CO) was extended. Each CO together with its associated Workers forms a separate and independent Vivaldi network in which, in the converged state, the *intra*-cluster distances approximate the latencies between pairs of nodes. Consequently, statements about the *inter*-cluster latencies of two Workers have no significance because they belong to two different Vivaldi networks.
As soon as a new worker joins a cluster, it starts in the origin of the coordinate system. The worker adds its Vivaldi coordinates to the worker's computational resource information, which it already sends to the CO every 8 seconds. Thus, there is no overhead due to an additional communication channel. The CO stores the coordinates it receives in a database along with other data about the workers. This means that only the CO has an accurate overview of the Vivaldi network. To provide the workers with the information required for the update process of their coordinates, the CO selects a fixed number of random nodes from the Vivaldi network and sends their Vivaldi information via MQTT to the workers. This information includes, among other things, the IP addresses of another workers to which a worker updates its coordinates. Since the Vivaldi information piggy-back on the already existing MQTT channel that is used for the periodic exchange of information about the workers computational capabilities every 8 seconds, the Vivaldi coordinates are also only updated every 8 seconds.
Once worker receives the IP addresses of the random workers selected by the CO, it starts to *ping* these workers. Based on each *ping* measurement, the worker updaets its position in the Vivaldi network with respect to the *pinged* neighbors. Due to the fact, that the CO picks random workers, each worker updates its position towards each other worker in the cluster over the lifetime of the Vivaldi network. After the update process, the workers send their new coordinates to the CO such that in the updated coordinates can be used during the next update phase.
An evaluation of the Vivaldi network with respect to accuracy, rate of convergence, etc. is done in Section 4.2.

3.3. Multilateration

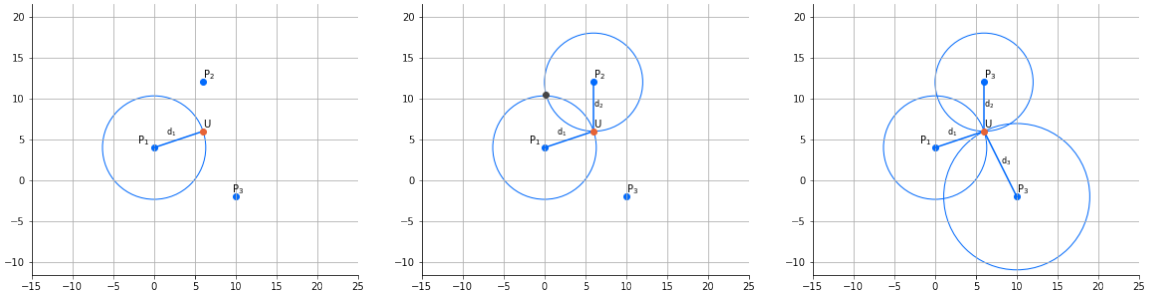
True-range multilateration is a method to determine the position of a static or moving point in space using multiple distance measurements between the target point and the known positions of the reference points, or *beacons* [51].
To understand the geometric interpretation, let us imagine that we want to determine the position of a user U in a 2-dimensional space. Figure 3.3a shows the first attempt, where we have only one reference point P_1, whose coordinates are known. P_1 cannot locate U directly, but can estimate its relative distance d_1. In this scenario with only one reference point, from the point of view of P_1, the user U is located somewhere on the circumference of the circle around P_1, with radius equal to the measured distance d_1. This situation can be improved by using not only one, but two reference points P_1 and P_2 (see figure 3.3b). From the point of view of P_2, just as for P_1, U must be located somewhere on the circumference of the circle around P_2. Since this criterion must be fulfilled for both P_1 and P_2, the user must be located at one of the intersections of the two circles. With just one additional reference point, the position of the user U can be reduced to two possibilities. To get a unique solution, we include one more reference point P_3. Figure 3.3c shows the final solution of the multilateration with three reference points in the 2-dimensional space. In such a scenario we speak of trilateration. The user must now be located on the unique intersection of the three circles and its position is known only using the three distance measurements of our known reference points, CO and two workers.
From a mathematical point of view, a point (x, y) on the Cartesian plane lies on a circle of radius r centred at c_x, c_y if and only if a solution to the following equation exists:

$$(x - c_x)^2 + (y - c_y)^2 = d_1^2 \tag{3.8}$$

With the same reasoning, we can derive equations for the circles generated by the reference points. Let $P = \{P_1, P_2, \ldots, P_n + 1\}$ be set set of reference points. Each beacon has its own position $(x_i, y_i) \forall i \in [1, \ldots, |P|]$, expressed in a specified coordinate system. The problem of trilateration, i.e. $P = \{P_1, P_2, P_3\}$, is solved mathematically by finding the point $P = (x, y)$ that simultaneously satisfies the equations of the three circles:

$$(x - x_1)^2 + (y - y_1)^2 = d_1^2 \tag{3.9}$$
$$(x - x_2)^2 + (y - y_2)^2 = d_2^2 \tag{3.10}$$
$$(x - x_3)^2 + (y - y_3)^2 = d_3^2 \tag{3.11}$$

Although trilateration can be viewed and solved as a geometric problem, it is often impractical. However, if one relies on mathematical modelling, very precise measurements are required. The worst case scenario would be, if the circles do not intersect at a single point, and hence the equations have no solution, which is often the case in the internet due to the TIVs as explained in section 2.6. Even if absolute precise measurements are available, the mathematical approach does not scale well, when we increase the number of reference points. To overcome these challenges, the trilateration in EdgeIO uses optimization. Ignoring circles and intersections, we want to know which point $\tilde{U} = (x, y)$ best approximates the correct



(a) Trilateration - One reference (b) Trilateration - Two references (c) Trilateration - Three references

Figure 3.3.: Visualisation of the trilateration process.

point U. Assuming we have a point $\tilde{U}$, we can estimate, how well that point replaces the correct point. This can be done quite easily by measuring the distance from each beacon to the estimate $\tilde{U}$. If these distances are equal to the distances d_i, $\tilde{U}$ is indeed U. Under these assumptions, trilateration can be modelled as an optimization problem. We want to find the point $\tilde{U}$ that minimizes a certain error function. Here, there is one source of error, for each reference point, and thus in our 3-reference-point scenario, the following errors:

$$e_1 = d_1 - \text{dist}(\tilde{U}, P_1) \tag{3.12}$$

$$e_2 = d_2 - \text{dist}(\tilde{U}, P_2) \tag{3.13}$$

$$e_3 = d_3 - \text{dist}(\tilde{U}, P_3) \tag{3.14}$$

A popular way to merge the error contributions is to average their squares. This also prevents negative and positive error from cancelling each other out, since squares are always positive. The obtained quantity is the called *mean squared error*:

$$\frac{\sum[d_i - \text{dist}(\tilde{U}, P_i)]^2}{|P|} \tag{3.15}$$

Unlike the mathematical approach, the optimisation approach scales very well and can be used with an arbitrary number of reference points.

3.4. Scheduling Workflow

The new scheduling process, proposed in this thesis, required some modifications of the root orchestrator (RO), cluster orchestrator (CO), and of the workers. In the following, we will take a closer look at the implementation for the three components.

3.4.1. Root Orchestrator

The EdgeIO scheduling workflow starts at the root orchestrator (RO) as soon as a developer calls the root's deployment endpoint. Figure 1.2 shows an example deployment descriptor. Such a deployment descriptor can contain multiple applications, which in turn can consist of several microservices. Once the deployment endpoint of the RO has been called, the deployment file is processed. Since the goal is to deploy a service to a suitable worker, the microservices belonging to the application, contained in the deployment file, are sent individually to the root scheduler. With this design decision, the scheduler only deals with finding a suitable node for a service and does not need to know the relationship to the associated applications and other related services. The task of the root scheduler is then to find a suitable cluster. Suitable means that the potential cluster has nodes, which can provide the computational resources required by the microservice. Since the cluster orchestrators periodically send aggregated information about the computational resources and other characteristics e.g., the locations, of its workers to the RO, the RO can evaluate the suitability of the clusters by checking, whether the provided resources cover the requirements and fulfill any constraints.

(a) Worker locations (b) Clustered Workers (c) Clustered Workers with buffer

Figure 3.4.: Locations of workers spread across Germany. (a) shows the position of each worker. (b) shows the clustered workers, as well as the *buffer* in grey. (c) shows the clustered worker groups without positions of the corresponding workers.

The developer has the possibility to specify a desired cluster location in the deployment descriptor by defining the location of the cluster, where the service should be deployed to. The operator of a cluster configures the CO by defining amongst others the environment variable *CLUSTER_LOCATION*. If the deployment request to the cloud scheduler contains a desired cluster location, the scheduler queries the database to find a cluster with the requested location. If a cluster was found, the scheduler checks, whether the cluster provides enough resources to run the service. Finally, if the resource check was successful, too, the scheduler sends the scheduling result to the system manager.
In case the developer did not specify a specific location, the cloud scheduler first filters out clusters that cannot provide enough resources to deploy the requested service. Then the scheduler checks, if any constraints are specified. If that is the case, it filters out all clusters that do not have any nodes in the areas or locations specified in the *latency* or *geo* constraints, respectively. The positional information, required by the root scheduler to determine, whether a cluster has nodes within specified areas of effect, are delivered by the CO along with the aggregated information that are periodically sent to the RO. In order to hide the exact locations of the Workers belonging to a cluster, the CO obfuscates the information that are send to the RO. This is achieved by first collecting the locations of the workers. Figure 3.4a shows example locations across Germany of workers that belong to the same cluster. These workers are then grouped using the density-based spatial clustering of applications with noise (DBSCAN) algorithm. Although k-means algorithm is one of the most common clustering algorithms, DBSCAN is far superior for spatial data. $k-$means groups N observations into k clusters, however, it is not very suitable for latitude-longitude based spatial data, because it minimizes variance and not geodetic distance. Due to substantial distortion at latitudes far from the equator, the k-means algorithm would still work but deliver poor results. The DBSCAN algorithm requires two parameters: a maximum physical distance from each point to still be considered to be in the same cluster, and a minimum cluster size, which is set to one in EdgeIO, because there might be Worker nodes that are located in very separated locations. The resulting clusters can be seen in figure 3.4b. If the CO would send the boundaries of the resulting worker groups, the RO could still infer to the exact location of Workers that are located at a corner of a cluster boundary. To avoid that, the CO adds a *buffer* to the cluster boundaries, such that all Workers are completely encapsulated by the respective cluster meaning that there are no Workers located directly on the boundary, hence making it impossible to anyone, to infer exact locations of workers (see figure 3.4c).
Either way, the root scheduler sends the resulting target cluster back to the system manager. Subsequently, the system manager propagates the deployment command to the target cluster.

```
def constraint_aware_scheduling(workers, constraints):
    # Filter out workers with insufficient capacity
    # Filter out workers that do not fulfill S2U constraints
    s2u_filtered_workers = service_to_user_constraint_scheduling(workers,
        constraints["S2U"])
    # Filter out workers that do not fulfill S2S constraints
    s2s_and_s2u_filtered_workers = service_to_service_constraint_scheduling(
        s2u_filtered_workers, constrains["S2S"])
    # Return first of suitable Workers
    return s2s_and_s2u_filtered_workers[0]

def service_to_user_constraint_scheduling(nodes, s2u_constraints,
    is_sla_violation):
    for constraint in s2u_constraints:
        if constraint["type"] = "latency":
            # If the scheduling process was triggered due to a SLA violation,
                find new worker based on network measurements
            if is_sla_violation:
                nodes = sla_alarm_latency_constraint_scheduling(nodes,
                    s2u_constraints)
            # Otherwise, do initial service placement using area defined in SLA
            else:
                nodes = initial_latency_constraint_scheduling(nodes,
                    s2u_constraints)
        elif constraint["type"] = "geo":
            # Find workers that are located within constraint area:
            nodes = find_worker_in_geo_area(area)

    return nodes

def service_to_service_constraint_scheduling(nodes, s2s_constraints, target_id)
    :
    for constraint in s2s_constraints:
        if constraint["type"] == "latency":
            # Find all workers that are located in range to target worker in
                Vivaldi network
            nodes = find_worker_in_vivaldi_range(nodes, target_id, threshold)
        elif constraint["type"] == "geo":
            # Find all workers that are located within range to target worker
            nodes = find_worker_in_geo_range(nodes, target_id, threshold)
    return nodes
```

Figure 3.5.: Pseudocode for the latency and location constraint aware scheduling algorithm CASA.

3.4.2. Cluster Orchestrator

When a cluster orchestrator (CO) receives a deployment command from the system manager it sends the deployment request to the cluster scheduler. Similar to the possibility of defining a cluster location that is considered during the scheduling process at the root level, the developer can define a specific node, to which the service should be deployed. If a node was specified in the deployment descriptor, the cluster scheduler checks, if the node exists in the first place, if the node provides the required resources and can fulfill the defined constraints. If all checks are successful, the scheduler returns the positive scheduling result to the cluster manager. In case no target node was specified, the cluster scheduler first filters nodes that cannot provide the required computational resources to reduce the search space and then finds suitable nodes by checking which nodes can fulfill the constraints. The pseudocode of the new constraint-aware scheduling algorithm is shown in listing 3.5 and one can see, that the scheduler differs between two types of constraints that can be specified both for service-to-user (S2U) and service-to-service (S2S) connections (lines 3 and 5).
Geo: The scheduling process for this constraint is rather straightforward. For S2U constraints, the scheduler simply checks, which nodes are located within or close to the specified area (line 20). The S2S looks similar and only differs in the start and end point between which the distance is calculated. Here, the scheduler checks, which orkers are located within the range to the target orker specified in the SLA (line 31). This is simple, because the CO has the location information about its workers stored in the local database and the constraint areas are defined in the SLA.
Latency: For the S2U latency constraint we have to consider two cases:

- *Initial service deployment or developer initiated migration/replication:* Since at the time of the initial deployment of the requested service no information about user latencies can be known, the initial placement is done similar to the *geo* constraint by selecting the node, located within the specified area (line 17). By choosing a node, based on geographical proximity, chances are good that the node provides a rather good latency towards the users accessing the service from that area.
- *Service migration/replication due to SLA violation*: In case of a *latency* constraint service level agreement (SLA) violation, the service has to be migrated or replicated to a new node that can fulfill the SLA. To do this, latency information about violating requests are required to find a node that is closer to the request location (line 14). This is done based on the Vivaldi network and trilateration as described in this section.

The S2S latency case, however, does not require any additional information that have to be collected using explicit *ping* measurements and trilateration. Since the Vivaldi coordinates of all workers are known to the cluster scheduler, it simply can calculate their distances to check, which nodes fulfill the constraint.
Once the cluster scheduler found the node that covers all constraints, it sends its result to the cluster manager, which in turn sends a deployment command via MQTT to the resulting worker node.

3.4.3. Worker Node

When a worker receives a deployment command, it deploys the service and registers it at the node's networking and monitoring component. The monitoring component periodically checks whether the node fulfills the SLA, until the service is stopped. In case of any SLA violations, the monitoring components sends an alarm to the CO to take further steps necessary to resolve the violation. After the deployment and service registration, the node informs the CO about whether the deployment was successful such that the cluster manager can update the corresponding entry in the database.
In more mathematical terms, the suitability of workers is evaluated as follows:
Filtering based on computational capacity proceeds as in the native approach. The available resources of the i-th cluster with m workers is $R^i = \{R^i_1, R^i_2, \dots, R^i_m\}$. $Q_{ms_{p,i}}$ describes the computational requirements of the i-th microservice of the p-th application. In addition, let $C = \{c^1_t, c^2_t, c^3_t, \dots\}$ be the set of constraints, where $t \in \{geo, lat\}$ describes the type of a constraint. The geographic longitude and latitude coordinates of the k-th worker are $P_k = (\phi_k, \lambda_k)$. $V_k = (x_k, y_k)$ defines the Vivaldi coordinates of each worker. Furthermore, let the fulfillment of the geo and latency constraints be defines as follows:

$$geo(P_k, P_{c_{geo}}) = \begin{cases} 1, & \text{if } great\_circle\_dist(P_k, P_{c_{geo}}) \leq tol_{c_{geo}} \\ 0, & \text{otherwise} \end{cases}$$

With $P_{c_{geo}}$ representing a geographic location in the case of a S2U constraint and in event of S2S constraints, $P_{c_{geo}}$ is the geolocation of the target worker.

$$lat(V_k, V_{c_{lat}}) = \begin{cases} 1, & \text{if } euclidean\_dist(V_k, V_{c_{lat}}) \leq tol_{c_{lat}} \\ 0, & \text{otherwise} \end{cases}$$

In the S2U, $V_{c_{lat}}$ represents the approximated user position in the Vivaldi network obtained by multilateration. For the S2S case it represents the Vivaldi location of the target worker. Furthermore, *great_circle_dist* calculates the shortest distance between to points located on the surface of a sphere using the Haversine formula [52], and *euclidean_dist* simply computes the euclidean distance between two points in the Vivaldi coordinate system. Finally, the suitable workers are those that satisfy the following constraints:

$$\forall c_{lat}, c_{geo} \in C$$

$$R^i_k \geq Q_{ms_{p,i}} \wedge$$

$$\sum_{c \in C} geo(P_k, P_{c_{geo}}) + lat(V_k, V_{c_{lat}}) = |C|$$

3.5. SLA Monitoring

The service level agreement (SLA) monitoring happens inside the new SLA monitoring component and is responsible to continuously check over the lifetime of a deployed service, whether it violates any resource requirements or constraints. For each registered service a *celery* task is started that checks, whether the SLAs are fulfilled, by periodically evaluating the following metrics.

CPU and ***Memory***: The *CPU* and *Memory* requirements are specified at the microservice level in the SLA, as seen in figure 1.2. Accordingly, the *CPU* and *Memory* usage data must be evaluated for each registered service and compared with the defined requirements. If, for example, *docker* was the chosen virtualization technology for a deployed service, the container's resource usage statistics are read (*docker stats*), the required data regarding CPU and memory usage is extracted and then compared with the requirements from the SLA.

Geo: This constraint refers to the geographic position of the worker and is also defined for each microservice of an application. Unlike constraint types *CPU* and *Memory*, the *Geo* constraint can be defined both as a service-to-user (S2U) and service-to-service (S2S) constraint. For S2U, a position is defined in the form of latitude and longitude coordinates, and a maximum distance, within which a potential worker must be located. In the S2S case, on the other hand, the constraint refers to the maximum distance, a candidate may be located from another worker, on which the reference service has been deployed earlier. The periodic check whether this constraint is still fulfilled, makes it possible to detect position changes of a worker and, if it leaves the constraint area, to take the necessary actions to migrate the service to another worker that is located in the defined area. The GPS information of a worker can either be configured by the worker's operator or, if available, received from a GPS module.

Latency: With this constraint, developers can set maximum service-to-user (S2U) and service-to-service (S2S) latencies for each microservice of an application, similar to the *geo* constraint. For the S2U case, an area is selected from a predefined list of urban areas, for which the measured latencies to the source IP addresses of the requests, coming from that area, may be at most as high as the specified threshold. To measure these round-trip times between the worker and users, accessing the service, the monitoring task listens on the port, exposed by the service. This is achieved using *pyshark*, a Python wrapper for *tshark* [53], which is a network protocol analyzer allowing to capture packet data from a live network. When an incoming request is registered, the location, where the request originates from has to be determined in order to decide, whether request comes from an area, for which a maximum latency was defined and hence must be monitored, or if it originates from a different area and therefore can be ignored. A worker determines request IP addresses, by calling the cluster manager's corresponding geolocation endpoint. The cluster manager then looks up the requested IP addresses in a *GeoLite2* database, which is a free IP geolocation database from MaxMind that is available as a complete database *snapshot* [54]. These *snapshots* are currently updated on a weekly basis and contain variable length IP prefixes, each with an associated geolocation. If an IP address was found, the corresponding latitude and longitude information are extracted and send back to the requesting worker. To avoid redundant API calls, the Worker first checks if the IP was already looked up before and hence is available in

the Workers cache. To optimize the lookup, the data from the GeoLite2 database was loaded into a *pandas* [55] dataframe. Pandas is an efficient and powerful open-source tool for data analysis and data manipulation. To avoid having to iterate through the entire dataframe for each lookup, the index of the IP address that has the same first octet as the lookup IP, is computed. This way the search starts near the IP address, for which the geoposition is to be determined, which allows a much faster lookup. Then the monitoring component can simply check, if the retrieved location is within the specified area. If that is the case, *ping* measurements to the requesting IP addresses are performed and the resulting round-trip times are compared to the maximum allowed latency. The S2S case is much simpler to handle, as it does not require monitoring of incoming network traffic and geolocation of IP addresses. In this case, the identifier of a service, deployed on another worker in EdgeIO and the maximum allowed latency, are defined in the deployment job. Using said identifier, a worker can easily request its cluster orchestrator (CO) and get the corresponding IP address of the worker, for which a latency threshold is defined.
If constraint violations are detected during monitoring, the corresponding entry in a counter is incremented. As soon as a constraint was violated a certain number of times, the workers triggers an alarm to the CO via MQTT. The published message contains the original deployment job, the type of the violation that triggered the alarm, and, in case the violation at hand is a latency violation, the violating IP addresses with the corresponding *ping* results. The ping result is later required by the cluster scheduler to approximate user locations via multilateration.
As soon as the CO receives the alarm, triggered by one of its nodes, the necessary steps are initiated to migrate or replicate the service to a more suitable Worker. This process is based on the implemented Vivaldi network coordinate system (NCS). Since all worker nodes periodically publish information about them including the current free memory, CPU cores and information about its position in the Vivaldi system to the CO, the CO knows the location of all workers in the NCS spanning across the whole cluster with the CO as a passive member at the origin. The CO is a passive member, because it does not actively measure network latency to other nodes in the cluster, but rather is part of the node's update process. That way all nodes update their location with respect to a selected number of other workers and to the CO, while always keeping the CO in the center of the network. Furthermore, the CO acts as a gravitational force pulling the nodes towards the center, thus preventing them from drifting away during the lifetime of the system. In order to approximate the user position true-range multilateration is used, which is a technique to determine a position based on distance measurements between the unknown position and multiple spatially-separated known reference locations. Since we are trying to get the position in the NCS, the measured distance approximates the latency and our known locations are the Vivaldi coordinates of the Workers and the CO. In a n-dimensional space $n + 1$ measurements from known locations are required for the trilateration to give useful results.
As described in [56] for this work, a 2-dimensional Vivaldi coordinate with an additional non-euclidean height term was used. Therefore, three measurements are required to find the user position via trilateration. Since the node that triggered the alarm, sends the violating

latency values in the alarm message, only the CO and another random node has to ping the given IP address. Subsequently, the CO sends a HTTPS request containing the IP addresses of the users that experienced too high latencies to a random Worker's endpoint. When the node pinged the IP address, it immediately sends answers with an response that contains the latency to the CO. To obtain the third and last required measurement, the CO itself pings the IP. Once, the necessary measurement data are collected, the cluster scheduler can approximate the user location by trilateration, as described

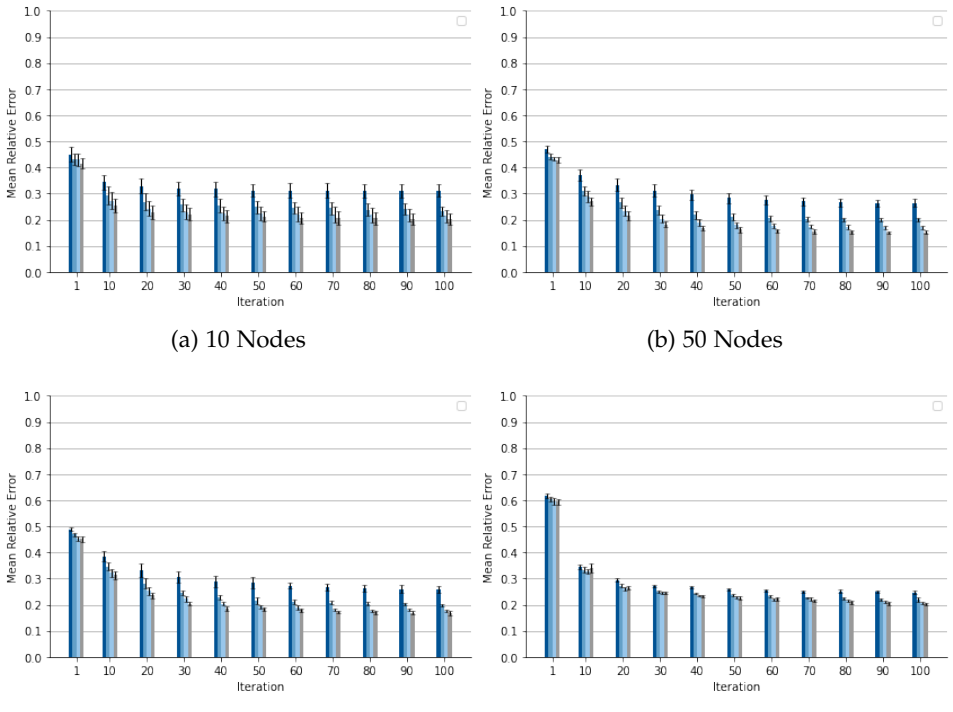
4. EdgeIO Service Scheduling Evaluation

4.1. Experimental Setup

To evaluate the effectiveness of the new **c**onstraint-**a**ware **s**cheduling **a**lgorithm (CASA), two different test environments were created. The first one is a simulated edge environment, where workers are assigned random values for CPU and memory capacity. Randomly generated points within Germany are used for the geographical position of the workers. The latency information required by the Vivaldi network coordinate system (NCS) is based on the King data set [57], that contains end-to-end latency measurements of 1740 domain name system (DNS) server. In the simulator, for each experiment a new Vivaldi network is generated that goes through 100 update phases to obtain a converged state. Furthermore, it is thus possible to have a different initial state for each scheduling test. In order to evaluate both service-to-user (S2U) and service-to-service (S2S) latencies and distances, respectively, the native and constraint-aware scheduling processes were run in each scheduling test with the same initial state, i.e., same Vivaldi network, worker positions and computational capacities. Since no constraints can be set for the native scheduling process, the results are compared to the constraints, defined for the scheduling with CASA. Thus, for each test execution, the result of the native approach can be compared to that of CASA in terms of latency, distance and runtime. Furthermore, the simulation was conducted on a desktop computer with 32 GB memory and 6 CPUs.
The second test environment is a real EdgeIO system with a root orchestrator (RO), cluster orchestrator (CO) and 10 Workers, each deployed on a virtual machine (VM) in the *Future SOC Lab* cluster [58], which is a place for scientific exchange in the area of service-oriented computing, provided by the Hasso Plattner Institute of the University of Potsdam. The RO and CO were deployed on L machines (4 GB memory, 4 CPU), Worker 1 and 2 on S machines (1 GB memory, 1 CPU), and the remaining workers on M machines (2 GB memory, 2 CPU). Furthermore, the latencies between workers, also based on the King data set, were simulated using the *tc* [59] and *netem* [60] commands. These commands can be used to add arbitrary network delays to traffic. Furthermore, filters can be defined such that traffic received from and sent to specific IP addresses can be individually delayed. That allows to simulate realistic latencies based on the measured RTTs between the DNS servers in the King data set.

4.2. Vivaldi Network Coordinate System

In the event of a service level agreement (SLA) violations the cluster scheduler has to find a suitable worker to migrate the service to, such that the SLA is satisfied again. When a latency constraint violation occurs, the SLA alarm message, sent to the cluster orchestrator (CO), contains the IP addresses of the users, to which the measurement results exceed the allowed round-trip times along with the corresponding measurement results. The CO in turn contacts the cluster scheduler to calculate the new deployment target closer to the users experiencing excessive latencies.
As described in section 3.2, EdgeIO uses a Vivaldi network coordinate system (NCS) that embeds the workers of a cluster into a synthetic coordinate system such that the distance between any two workers approximates their round trip time (RTT). To get the most accurate information about the RTTs between the potential target workers and users, the Vivaldi network must be as accurate as possible. As mentioned in 2.6, the nodes are mapped into a d-dimensional euclidean space. However, there is a large number of TIVs on the Internet [47], which means that the nodes cannot be perfectly embedded in the NCS and thus discrepancies exist between the measured end-to-end pings and the Vivaldi estimate. To evaluate the effectiveness of the Vivaldi network in EdgeIO, several experiments were conducted, which will be described in the following.



(a) 10 Nodes

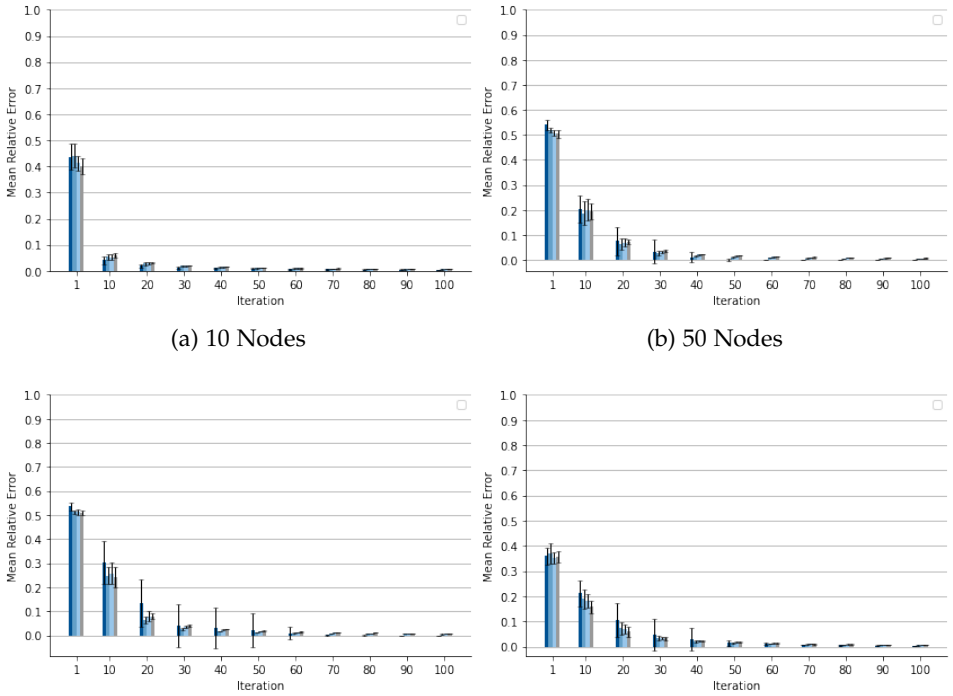
(b) 50 Nodes

(c) 100 Nodes

(d) 500 Nodes

Figure 4.1.: Vivaldi network mean relative error (MRE) evaluations with increasing number of nodes, 100 iterations, and different coordinate dimensions.

Figure 4.1 shows the progression of the MRE in four Vivaldi networks with a different number of nodes. All four experiments were performed with 100 iterations, i.e., each node updates its position in the network 100 times by pinging 6 neighbor nodes. In addition, each plot shows the progression for Vivaldi networks with dimension from 2 to 5. For these four experiments the latency data, required by the Vivaldi nodes to update its positions, is based on the King data set [57], which contains measurements of the latencies between a set of DNS servers and was also used by the authors of the Vivaldi NCS in [45]. From the plot it is easy to see that on the one hand the rate of convergence decreases with increasing number of nodes and on the other hand the MRE is reduced with higher dimensions. Due to the aforementioned TIVs, the best accuracy achieved in the implemented Vivaldi NCS, is about 20%.
To visualize the impact of TIVs the same experiments were conducted but for a data set that contains latencies corresponding to an arrangement of nodes in a 2D grid in a converged Vivaldi network. Figure 4.2 shows these results. Since there are no TIVs in a 2-dimensional grid in the Euclidean space, the Vivaldi estimates represent the inter-node latencies almost perfectly. Again, the rate of convergence decreases with increasing number of nodes in the network. However, all experiments with 2- to 5-dimensional coordinates achieved a similar low MRE.



(a) 10 Nodes

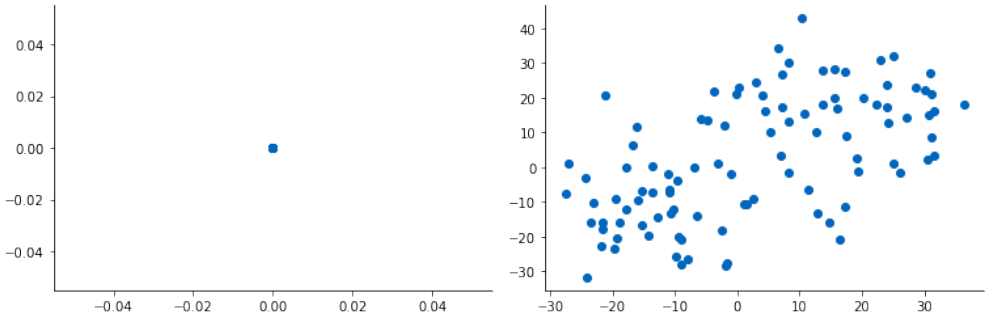
(b) 50 Nodes

(c) 100 Nodes

(d) 500 Nodes

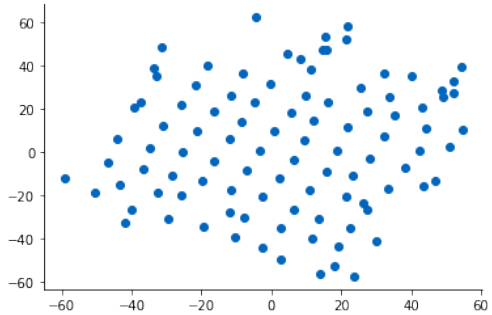
Figure 4.2.: Vivaldi network MRE evaluations with increasing number of nodes, 100 iterations, and different coordinate dimensions for a mesh data set.

By visualizing the evolution of the Vivaldi network for the latency data set based on the 2D grid, as seen in figure 4.3, it is easy to see, how the individual Vivaldi coordinates - starting in the origimemoryn - update their position with respect to their neighbors, until the converged state, that perfectly represents the grid layout, is reached. However, the Vivaldi coordinates do not coincide with the coordinates of the 2D grid, because a rotation in the xy-plane of the coordinate system is observed due to the random choice of update partners. Due to the absence of triangle inequality violations, Vivaldi can embed the nodes such that the original distances are restored. This is also the reason, why the results in this experiment are far better than the ones using real latency data.

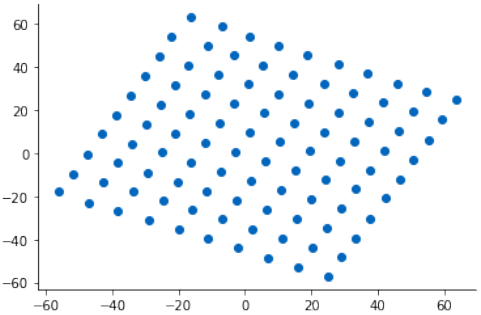


(a) Iteration 0

(b) Iteration 10



(c) Iteration 20



(d) Iteration 30

Figure 4.3.: Development of Vivaldi coordinates with latency data based on a mesh data set.

4.3. Multilateration

In EdgeIO trilateration is used to approximate the location of users within the Vivaldi network, where the distances between pairs of nodes approximate their latency in order to migrate services that do not meet all service level agreements SLAs to a worker that does. As described in section 3.2, there are many triangle inequality violations in the Internet. Consequently, nodes cannot be perfectly embedded in the Euclidean space such that distances would perfectly approximate latencies. As a result, there is usually not a unique point, where the three circles of the reference points intersect, which means that the mathematical approach as described above cannot be used, because the equations would have no solution.

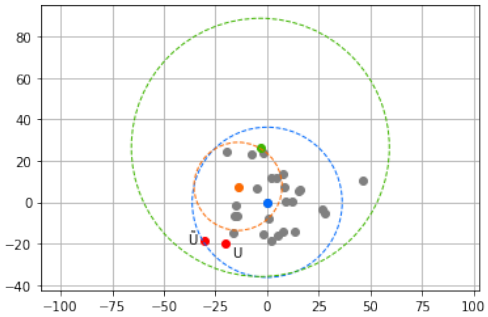


Figure 4.4.: Trilateration in a Vivaldi network coordinate system of a EdgeIO cluster with 25 workers.

Figure 4.4 shows the trilateration process in EdgeIO within the Vivaldi network coordinate system (NCS) of a cluster with 25 Workers. Three members of the network are required to perform an explicit latency measurement to user U. Two of the three members are always the cluster orchestrator (CO), who, as a passive member, always remains in the origin, and the worker that triggered the SLA alarm, which initiated the rescheduling of the service. The CO remaining in the coordinate system's origin additionally ensures that the Vivaldi nodes do not drift away from the center as the network ages due to the dynamic update processes, A random worker is selected as the third reference point. Each beacon now performs an explicit latency measurement by *pinging* the user. Once the results of the latency measurements are available, the trilateration process based on the optimization problem described above begins. The position of the user U in figure 4.4 is not known to the workers of the cluster and the CO, and is shown here only for visualization and evaluation of the multilateration. The approximated user $\tilde{U}$, on the other hand, is the result of trilateration. In the following, during scheduling, all Workers are evaluated, whether they are within the specified latency, i.e. distance in the Vivaldi network, and thus can satisfy the service level agreement (SLA). From the set of suitable Workers, one is then sent to the cluster manager as a result, so that the migration of the violating service can be completed.

4.4. Scheduling Process

In this section, the new constraint-aware scheduling algorithm is evaluated at both the root and cluster levels. The evaluation at the root level consists of comparing the duration of the native computational resource-only algorithm and the newly developed constraint-aware algorithm. The cluster-level evaluation includes a comparison of the computation times, as well as a comparison of the resulting RTTs and geographical distances between Workers and user groups, and between two workers to evaluate, how well the implemented algorithm ensures the service level agreement (SLA) fulfillment with respect to *latency* and *geo* constraints.
In order to compare the results of the two scheduling approaches, both the native and constraint-aware algorithms were run in each test execution. This means that both were run with the same cluster and worker configuration, especially in terms of computational capacity, geographic location and position in the Vivaldi network. Thus, the results of the two algorithms were calculated based on the same data, which allows a good comparison of the results.

4.4.1. Root Scheduler

As described in section 3, the task of the root scheduler is to find a suitable cluster for the service deployment. In the native approach of EdgeIO, the root scheduler only checks whether the resources required by the service can be provided by one of the active clusters. With CASA, the root scheduler is now also able to take into account constraints regarding latencies and geographical location.

4.4.1.1. Simulated EdgeIO System

Runtime Evaluation
Figure 4.5 shows the duration of the computations of the root scheduler in *EdgeIO* with 10, 50, 100, and 500 clusters. Each of the four experiments shows the computation duration of the native resource-only algorithm and the new constraint-aware algorithm. The latter was evaluated with both service-to-user (S2U) and service-to-service (S2S) constraints. The plots show that the runtime of the native approach and **c**onstraint-**a**ware **s**cheduling **a**lgorithm (CASA) grows with the number of clusters, due to the increasing search space. In the native approach, the Root Scheduler iterates through all available clusters and checks if they meet the requirements by comparing the current capacity with the requirements of the microservices to be deployed. Moreover, from the diagram it can be seen that the new scheduling algorithm has a significantly higher execution time due to its more sophisticated nature, including more database queries as well as geolocation and Vivaldi network coordinate system (NCS) related computations.

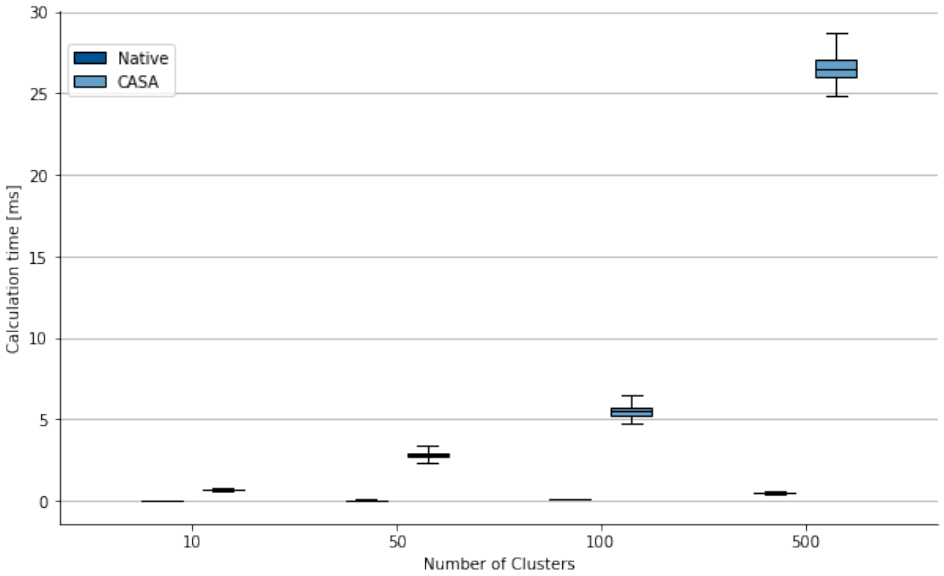


Figure 4.5.: Calculation runtimes for native and constraint-aware scheduling at the root level with rising number of clusters.

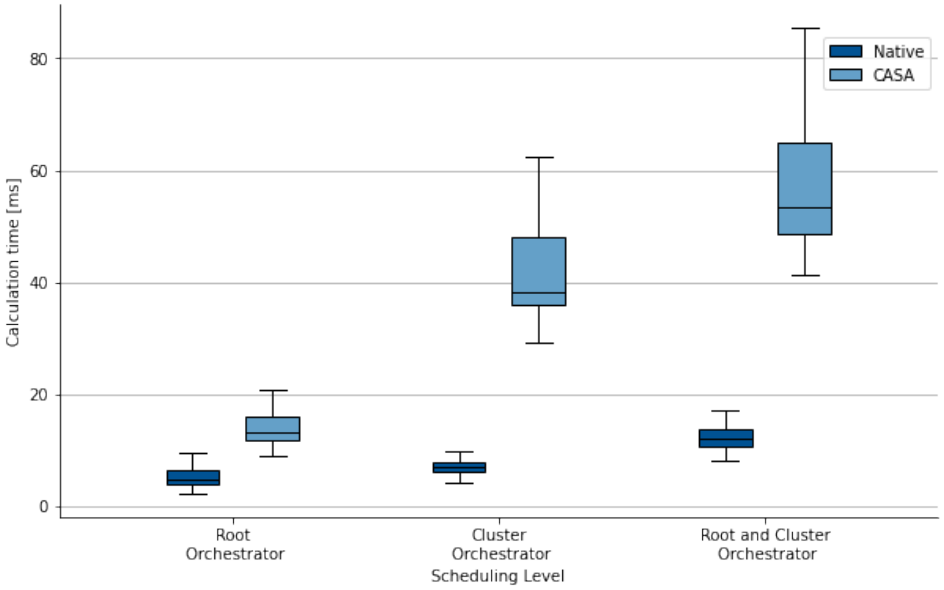


Figure 4.6.: Calculation runtimes for native and constraint-aware scheduling on Future SOC cluster.

4.4.1.2. Live EdgeIO System

Runtime Evaluation

Figure 4.6 shows the calculation runtimes at the root and cluster level, and the sum of both. Compared to the runtimes in the simulated edge environment, the calculations of the root scheduler, which was deployed on a virtual machine (VM) in the Future SOC cluster, take significantly longer. This is due to the fact that the specs of the computer, on which the simulation was conducted, is more performant and that the database queries are executed on a real database and are not - as in the simulation - mocked. The variance in the runtimes of both algorithms can be caused on the one hand by potential fluctuations in the network quality, which affects e.g., the response time of database queries, and one the other hand by varying effort for geolocation and Vivaldi network related computations.

The total time it takes for a job to be scheduled, starting from the job submission and ending when the requested service was successfully deployed, was between 1 and 4 seconds during the conducted experiments. Therefore, the higher calculation time of CASA, compared to the native approach, has a negligible influence on the overall deployment time. This is rather influence by the response times of HTTPS requests between the RO and the scheduler, and by the amount of time the MQTT deployment message requires to reach the destination worker. Docker was chosen as the virtualization technology for the microservice deployments during the experiments. Hence, also the docker deployment time has a major impact on the total deployment duration.

4.4.2. Cluster Scheduler

The most interesting part of the evaluation takes a deeper look into the scheduling process at the cluster level, since the scheduler has to take much more information into account than the scheduler at the root level and ultimately the result of the cluster scheduler determines the QoS. Just like the root scheduler, the cluster scheduler in EdgeIO's native scheduling approach only considers the computational capacities of the workers and hence is not aware of any contextual knowledge like network or position related information. The new constraint-aware scheduling algorithm additionally considers the geographic position and Vivaldi based round trip time (RTT) estimate of the workers with respect to a specified constraint area or other workers in EdgeIO.
In the case of service-to-user (S2U), maximum latencies and geographic distances can be specified for areas, for which the specified constraints have to be fulfilled. The monitoring component of the workers regularly checks, if these constraints are still fulfilled, i.e. whether the Worker moved out of the constraint area, or if users experience threshold exceeding latencies based on explicit *ping* measurements. In the service-to-service (S2S) case, on the other hand, maximum distances and latencies are defined with respect to a target worker instead of an area. Similarly, the monitoring component checks, if the worker is still close enough to the target worker, and the Vivaldi latency estimate is not exceeding the threshold.

4.4.2.1. Simulated EdgeIO System

Runtime Evaluation
Figure 4.7 shows the scheduling runtimes of the native and constraint-aware algorithm at the cluster level with 10, 50, 100, and 500 workers. The constraint-aware algorithm was evaluated with both service-to-user (S2U) and service-to-service (S2S) constraints. It can be seen that the runtimes grow with the number of workers, due to the increasing search space. Again, the cluster scheduler exhibits a significantly higher runtime for **c**onstraint-**a**ware **s**cheduling **a**lgorithm (CASA) than for the native approach. This can also be attributed to the significantly more complex nature of CASA. For the S2S constraints, the cluster scheduler must take into account geoposition and Vivaldi information of the workers to find suitable targets that do not exceed the specified thresholds. The S2U is even more complex, because, based on explicit ping measurements, also a multilateration is performed to approximate the user positions. Based on this and with the help of the Vivladi network and position information, suitable Workers are found. The duration of the ping measurements were ignored for the evaluation, since the ping duration is three seconds by default and is even increased depending on the connection quality. If these times were also included in the evaluation, the runtimes of both scheduling algorithms can be compared poorly, since the actual calculation for CASA is in the millisecond range, which means little additional overhead compared to the ping times in seconds.

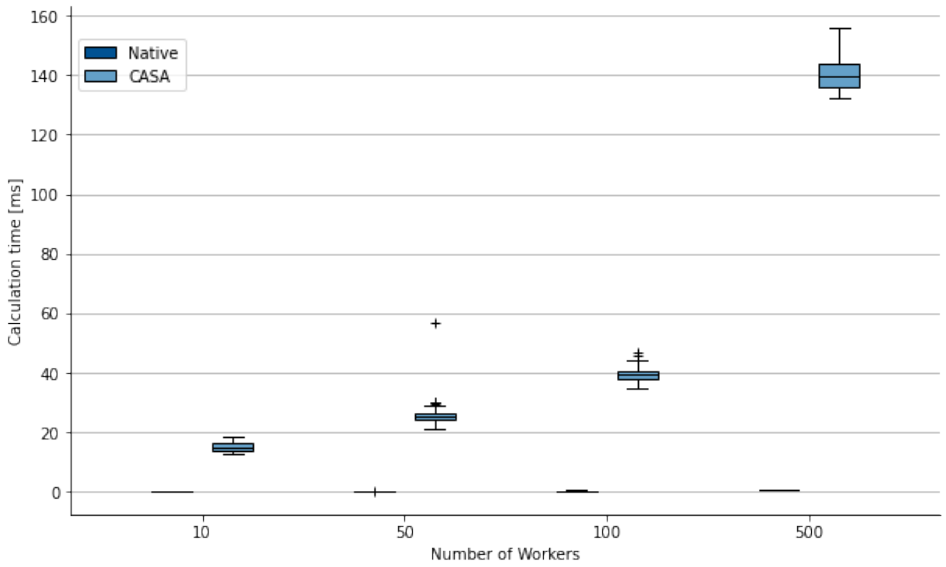
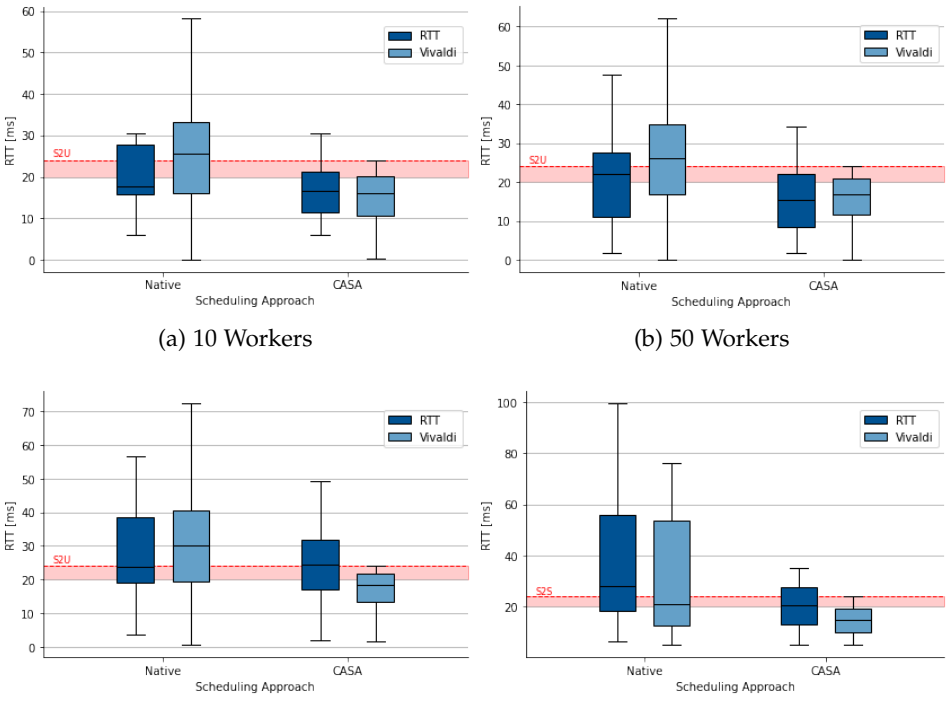


Figure 4.7.: Calculation runtimes for native and constraint-aware scheduling at the cluster level with increasing number of workers.



(a) 10 Workers

(b) 50 Workers

(c) 100 Workers

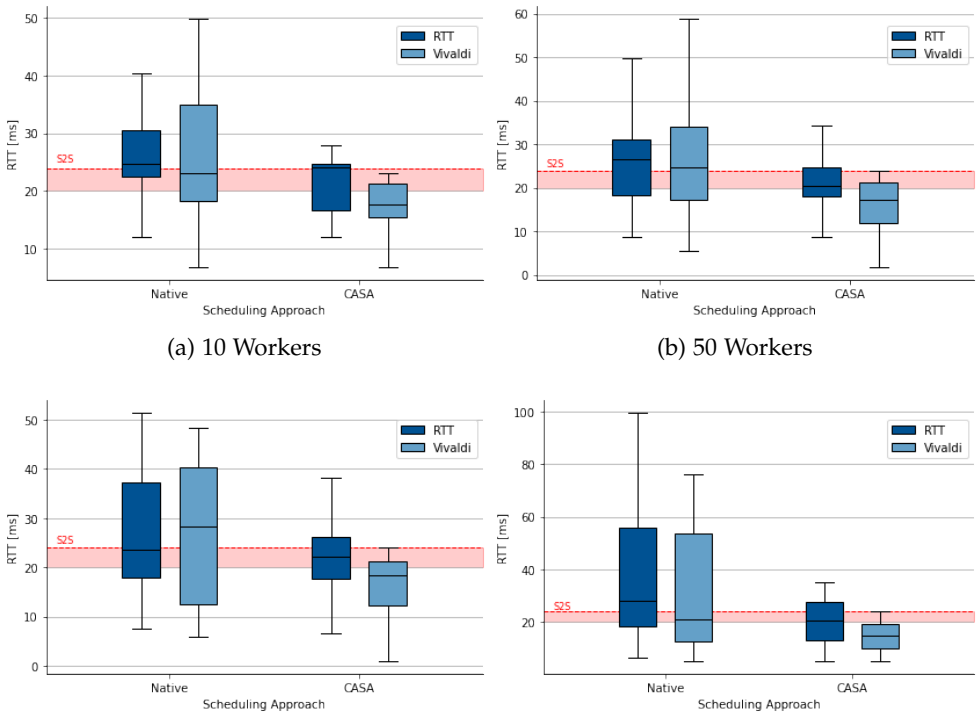
(d) 500 Workers

Figure 4.8.: S2U latencies for native and constraint-aware scheduling with increasing number of workers, and 2-dimensional Vivaldi coordinates.

Latency Constraint Evaluation

Workers in the test cluster were configured such that approximately 10% of them were filtered by the Scheduler due to insufficient computational resources or their geographic location. The intra-cluster latencies are based on the King data set [57], which contains latency measurements between 1740 DNS servers and was used by the authors of the Vivaldi NCS [45] for their evaluation. The resulting latencies and Vivaldi estimates with respect to the S2U and S2S constraints are shown in figures 4.8 and 4.9, respectively.

Figure 4.8 shows the resulting S2U latencies (RTT) and (Vivaldi) estimates for the native, resource-only scheduling approach and the new constraint-aware scheduling algorithm with a Cluster with 10 (4.8a), 50 (4.8b), 100 (4.8c), and 500 (4.8d) workers. The results show on the one hand the S2U latencies resulting from the scheduling and on the other hand the Vivaldi distances to the positions of the users, which are approximated during the constraint-aware scheduling using multilateration. The area highlighted in red shows the S2U latency constraint. In the experiments, this value was set to 20ms. With a tolerance of 20%, this results in a mixmum allowed latency, i.e. Vivaldi distance of 24ms. From the diagrams it is easy to see that this limit is a hard limit for the selection process of CASA, since the Vivaldi distances to the approximate user positions of none of the resulting workers exceeds the threshold, when CASA was used. Based on the same threshold, the monitoring component checks, if a service exceeds the maximum latencies and triggers an alert to the cluster orchestrator (CO) if necessary. The experiments also show that the resulting S2U latencies of the native scheduling process on the one hand partially achieve similar value ranges as CASA, but on the other hand the majority is at or above the maximum allowed latency. CASA, on the other hand, achieves lower latencies, most of which are in the allowed area. Here, too, there are some outliers, especially as the number of Workers increases, which can be attributed to the number of TIVs, which also increases, as the number of workers does, and leads to inaccuracies in the Vivaldi network coordinate system (NCS), which in turn leads to deviations from the actual latencies and thus sometimes erroneous scheduling results.



(a) 10 Workers

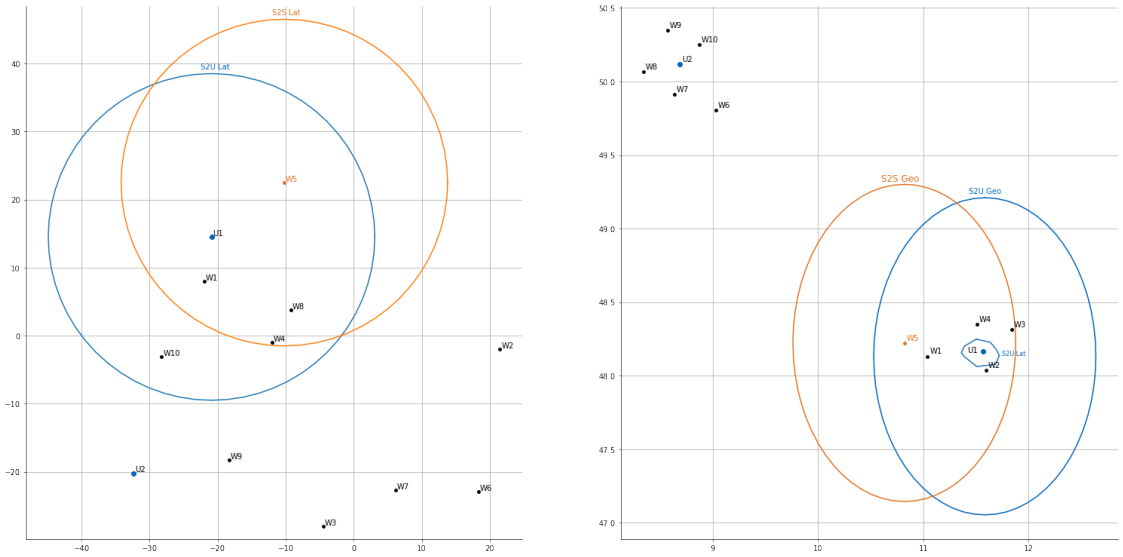
(b) 50 Workers

(c) 100 Workers

(d) 500 Workers

Figure 4.9.: S2S latencies for native and constraint-aware scheduling with increasing number of workers, and 2-dimensional Vivaldi coordinates.

Figure 4.9 shows the achieved S2S latencies of the scheduling simulations. Similar to the S2U case, the majority of the latencies of the resulting workers of the native scheduling exceed the specified maximum value to a target Worker. For CASA, on the other hand, the majority of achieved latencies are below the threshold and one can see, as in the S2U case, that the threshold is again a hard limit for selecting suitable nodes. The range of values reached in the native case increases as the number of Workers increases, since the network of potential Workers grows.
From the plots it can be concluded that the new **c**onstraint-**a**ware **s**cheduling **a**lgorithm (CASA) provides good results in the majority of cases, so that the specified SLAs are met, while the results of the native approach fails to satisfy the constraints in most cases. With fewer outliers that show high latencies, users experience a better response time and fewer alarms are triggered to reschedule the violating service.



(a) Vivaldi Network

(b) Geographic Locations

Figure 4.10.: Vivaldi network and geographical locations of the workers belonging to the same cluster.

Location Constraint Evaluation

Unlike finding workers that must satisfy specified latencies based on the Vivaldi network coordinate system (NCS), finding some that are within a specified distance to a geographic point, does not rely on any real approximations or requires additional technologies like NCS and multilateration. The precision of the Vivaldi estimates, and ultimately the quality of the scheduling result, is significantly affected by the topology of the Internet. In the case of geographic constraints, on the other hand, the distances to the constraint locations can be calculated accurately. The coordinates of the workers can be specified by the edge device operators. Since the workers periodically send their coordinates to the cluster orchestrator (CO), as described in section 3, the cluster scheduler can filter out unsuitable, i.e., too distant workers based on these information during the scheduling process.

Figure 4.11 shows the resulting distances to the service-to-user (S2U) constraint regions of the native and constraint-aware scheduling process. The workers were configured so that their coordinates correspond to a random location in Germany. Just as in the latency evaluation, experiments were conducted with 10, 50, 100, and 500 workers. For a cluster with a small number of workers, the probability that CASA will find very few workers that satisfy the constraint, is relatively high. The native scheduling algorithm, on the other hand, only considers the computational capacity of the workers which is why the distances to the

target vary greatly. The more workers a cluster has, the more evenly they are distributed. Consequently, there are more workers that satisfy the constraints and thus are found by CASA. From the figures, it can be seen very well that the geographic distance threshold is a hard limit for CASA's scheduling process. All Workers found suitable by CASA satisfy the constraints. The results in the native case, on the other hand, are very scattered with most of distances lying above the threshold and therefore do not satisfy the geographic constraint.

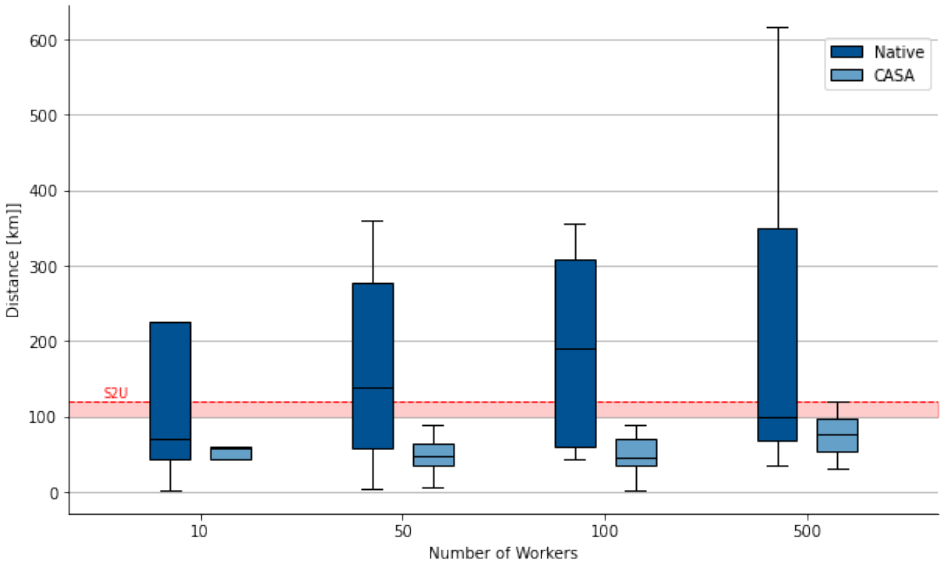


Figure 4.11.: S2U distances for native and constraint-aware scheduling with rising number of workers and 2-dimensional Vivaldi coordinates.

For the evaluation of the location constraints, the S2U and S2S cases are very similar. In both cases, the required coordinates for the calculations of the distance between Worker and user group and distances between workers are given, because the constraint location is defined in the SLA and the Worker's coordinates are known to the CO. Figure 4.12 shows the results of the S2S constraint evaluation. Just as in the S2U case, CASA finds all workers that do not exceed the distance, and the results of the native approach cover all possible distances. Consequently, the results of the native scheduler violate the SLA most of the time, while the constraint-aware algorithm only finds workers that fulfill the requirement.

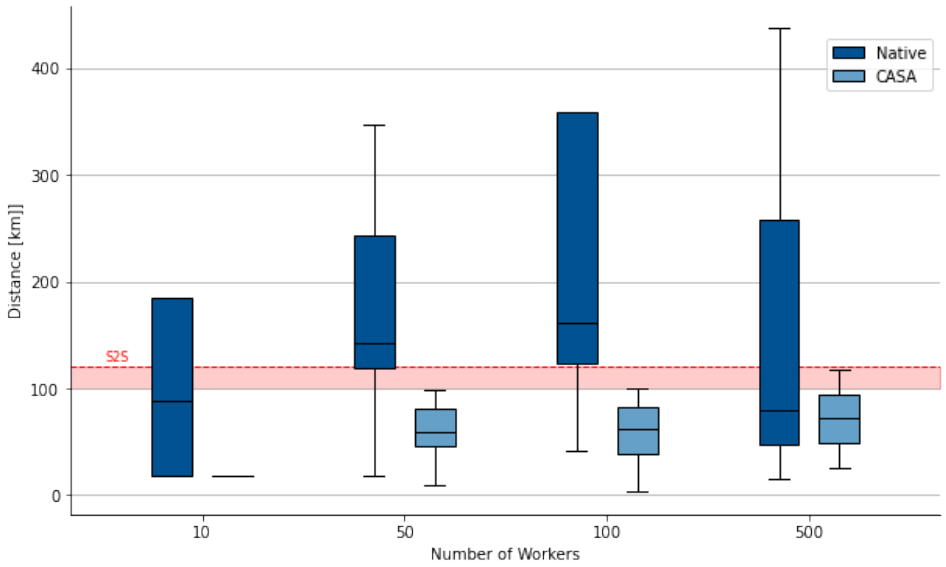


Figure 4.12.: S2S distances for native and constraint-aware scheduling with rising number of workers and 2-dimensional Vivaldi coordinates.

4.4.2.2. Live EdgeIO System

Runtime Evaluation

The evaluation of the live system again shows higher runtimes for the native approach and CASA at the cluster level (see 4.6. As with the root scheduler, this is due to the fact that on the one hand the performance of the VMs of the cluster is worse, and on the other hand the database queries are not mocked. Furthermore, any network delays can mean additional overhead for database queries and HTTPS requests during the scheduling.

The difference in the calculation times between the native and constraint-aware scheduler is higher than at the root level. That is because the cluster scheduler has to take much more information into account then the root scheduler. At the cluster level there are more computational heavy tasks such as the multilateration and calculations based on the Vivaldi NCS.

But as mentioned in the runtime evaluation of the root scheduler, the calculations in the millisecond range show no significant impact on the total deployment time.

```
customerID: 1
applications:
- applicationID: 1
  microservices:
  - microserviceID: 1
    memory: 100
    vcpus: 1
    constraints:
    - type: latency # S2U latency constraint
      area: munich
      threshold: 20
    - type: geo
      location: 49.5,11.5 # S2U geo constraint
      threshold: 100
    connectivity:
    - target_microservice_id: 1
      constraints:
      - type: latency # S2S latency constraint
        threshold: 20
      - type: geo # S2S geo constraint
        threshold: 100
```

Figure 4.13.: Deployment file representing job send to scheduler in the live system evaluation.

Latency Constraint Evaluation

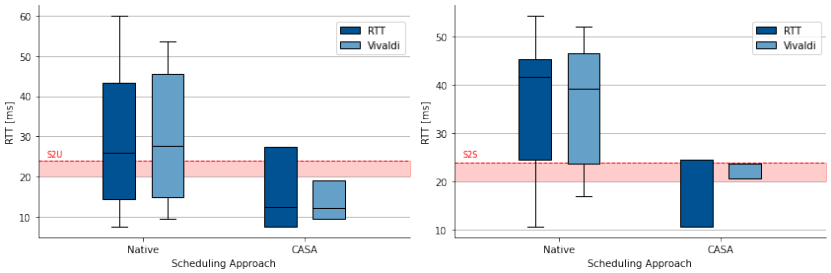
For the evaluation of the live EdgeIO system deployed on a Future SOC cluster, the workers were configured as follows: To avoid randomly affecting the achievable latencies by having workers with insufficient computational capacity and therefore not surviving the scheduling process, the memory and CPU requirements of the service to be deployed were chosen so that all workers met them. The intra-cluster round-trip times are based on the King data set, but were skewed to allow a greater latency variance for a cluster with only 10 Workers. The Vivaldi network resulting from the RTTs is shown in figure 4.10a. To achieve geographic distribution, the Workers were placed in or around Frankfurt (50.11, 8.70) and Munich (48.13, 11.58) (see figure 4.10b). The two plots also show the S2U and S2S constraints chosen in this scenario as seen in listing 4.13.

In case of the S2U latency constraint, this means that for all user requests to the deployed service coming from Munich, the latency may reach a maximum of $24ms$ ($20ms + 20\%$ tolerance). For requests that do not origin from Munich, no constraint was set, hence these requests are ignored. Diagram 4.10b shows a small region, annotated with S2U Lat, that represents the area of effect Munich for the latency constraint. The geographic position of User U1 is looked up during the scheduling by IP geolocation using the GeoLite2 database, and turns out to be located in Munich, so potential workers must satisfy the latency constraint.

Diagram 4.10a shows the user's location - the result of trilateration during scheduling - in the Vivaldi network. The circle with the User U1 as center has a radius corresponding to the maximum allowed latency. All workers that lie within this circle satisfy the latency constraint, given that the Vivaldi estimates approximate the real RTT well enough.
The S2U Geo constraint requires that the worker on which the service is to be deployed be no farther away than $120km$ ($100km + 20\%$ tolerance). In diagram 4.10b, this region within which potential Workers must be located, is represented by the circle annotated with S2U Geo.
In order to test the S2S constraints, a simple microservice with ID 1 belonging to the same application as the service that needs to be scheduled, was previously deployed on worker W5. In the deployment descriptor, the S2S constraints for a target microservice (here deployed on W5) are defined as a connectivity constraint. The maximum latency that the worker, on which the W5-dependent service should be deployed, may have to W5 was set to 24ms. Furthermore, the scheduling was adjusted so that the service is not deployed to the same Worker, otherwise the results of CASA would have almost no latency, because it would be deployed on W5 as well. This would make the comparison of the native and constraint-aware approach difficult, since it cannot be seen, how well the thresholds act as a hard limit for CASA. In figure 4.10a, the S2S latency constraint is represented as an orange circle around W5. Again, all workers that are located within the circle, satisfy the latency constraint towards that Worker. Similar to the S2U case, $120km$ was chosen as the maxmimum geograhic distance that the resulting worker can be located from W5 (see circle annotated with S2S Geo in 4.10b).
From the diagrams it is easy to see that the Workers satisfy the constraints with $S2U_{geo}$, $S2U_{lat}$, $S2S_{geo}$, $S2S_{lat}$ being the set of Workers satisfying the respective constraint, as follows:

$$\begin{aligned} S2U_{geo} &= \{W1, W2, W3, W4, W5\} \\ S2U_{lat} &= \{W1, W4, W5, W8, W10\} \\ S2S_{geo} &= \{W1, W2, W3, W4, W5\} \\ S2S_{lat} &= \{W1, W4, W5, W8\} \end{aligned}$$

Hence, the only Workers that fullfill all constraints and therefore are suitable for the deployment is the intersection of all four sets $R = S2U_{geo} \cap S2U_{lat} \cap S2S_{geo} \cap S2S_{lat} = \{W4, W5\}$



(a) S2U Round-Trip Times (b) S2S Round-Trip Times

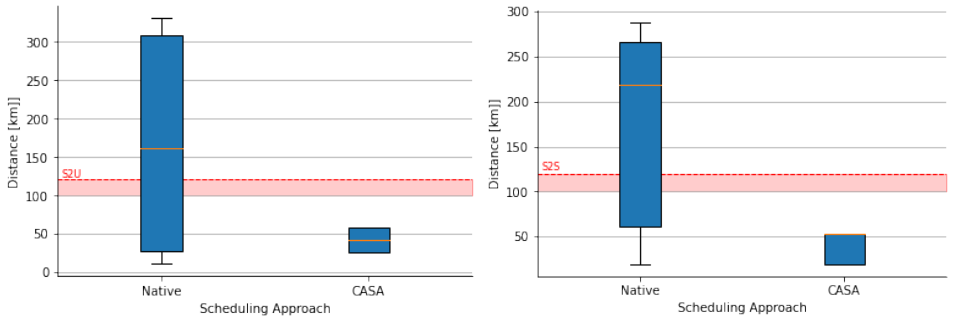
Figure 4.14.: S2U and S2S *latency* constraints for native and constraint-aware scheduling on Future SOC cluster

Figure 4.14 shows the results of both scheduling algorithms with respect to S2U and S2S latency constraints. The only two workers that satisfy all constraints are, as shown above, $W4$ and $W5$. The results from the constraint-aware scheduling algorithm have very good values, most of which are below the threshold of $24ms$. For the native algorithm, on the other hand, which only considers computational capacity, all workers are suitable. As a result, each worker is at least once the result of the scheduler, hence the complete range of possible latencies is covered. Consequently, the deployed service cannot satisfy the constraints for the most part, which means that the native approach requires significantly more service migrations than **c**onstraint-**a**ware **s**cheduling **a**lgorithm (CASA) does and leads to higher response times and therefore lower user experience.
From these results it can be concluded, that the newly developed scheduling algorithm successfully finds workers in a live EdgeIO system, for that the majority satisfies S2U and S2S latency constraints. In combination with the SLA monitoring component this allows EdgeIO to continuously evaluate intra-cluster and end-to-end latencies and migrate the service in case of high latencies thus maintaining fast response times at the edge.

Location Constraint Evaluation

As described in the evaluation of the live EdgeIO system at the cluster level, the only two suitable workers that satisfy all constraints are *W*4 and *W*5. As in the evaluation of the location constraints in the simulated edge environment, CASA only finds workers in the live system that fulfill the location constraints. The native algorithm calculates each of the 10 workers at least once as a target for the microservice that should be deployed, so that the complete range of possible distances from worker-to-user or worker-to-worker are covered here as well.

Considering these results it can be said, that the newly developed scheduling algorithm successfully finds workers in a live EdgeIO system, that satisfy S2U and S2S constraints with respect to geographic locations. In combination with the SLA monitoring component this allows EdgeIO to continuously satisfy strict location constraints by quickly reacting to dynamic movements of the workers.



(a) S2U Distances (b) S2S Distances

Figure 4.15.: S2U and S2S *geo* constraints for native and constraint-aware scheduling on Future SOC cluster

5. Conclusion

5.1. Final Remarks

The constraint-aware scheduling algorithm (CASA) presented in this work extends EdgeIO's native resource-only scheduling approach. Using Vivaldi, multilateration, DBSCAN clustering, etc., the new scheduling algorithm finds workers that satisfy potential service-to-user (S2U) and service-to-service (S2S) constraints and thus the SLAs. Furthermore, the monitoring component also presented, ensures that the constraints that a worker must satisfy, are indeed satisfied and as soon as a constraint is violated, the monitoring component triggers an alarm so that the cluster orchestrator can deploy the service to a new worker, which in turn satisfies all constraints. This means that a developer does not have to monitor the quality of services on his own, but can leave the task to EdgeIO.
The evaluation of the new scheduling algorithm and comparison with EdgeIO's native solution has shown that possible constraints regarding latency and geographical location are taken into account so that the results calculated by CASA can meet the specified SLAs. Furthermore, the monitoring component integrated in EdgeIO, ensures that in case of constraint violations, service rescheduling is automatically initiated to maintain QoS. In addition, the technologies used, such as the Vivaldi NCS, are well suited for use in the edge computing scenario, because they are lightweight and are little affected by the heterogeneity of the edge.
In summary, the scheduling algorithm presented here is well suited for the scheduling of latency-sensitive applications by taking contextual information into account. Unlike existing approaches, not only is approximate optimal service placement computed so that communication between two edge devices experiences low latencies, but also to possible user groups, which is one of the key characteristics and challenges of edge computing.

5.2. Limitations

The most complex part of the new scheduling algorithm and monitoring component are the calculations and approximations of latencies. The heart of these calculations is the Vivaldi NCS, the accuracy of which is significantly affected by occurrences of TIVs, which are very common in the topology of the internet. In the current implemention of Vivaldi in EdgeIO, two dimensions have been selected for the coordinates. Since three reference points are required for the trilateration, and the CO and the alerting worker are always two of the three points, another uninvolved worker must be used. However, considering the principle of separation of concerns, any uninvolved worker should stay uninvolved. By increasing the dimension of the Vivaldi coordinates, the MRE of the Vivaldi network can be reduced, but

this leads to even more uninvolved workers having to ping the IP addresses, for which ping measurements have detected too high values.

5.3. Future Work

In future implementations, the precision of the latency estimates could be increased by optimizing the Vivaldi network. There are various optimization possibilities such as adding a non-euclidean height term to the coordinates, that represents the time it takes a packet to travel the access link from the node to the core. This would lead in more precise estimations by considering any potential queuing delays [56]. Another possibility is to use update filters that compensate large coordinate updates due to strong network fluctuations.
Another idea would be to optimize the scheduling process. To make more precise statements about S2U latencies, the Vivaldi nodes could update themselves not only with respect to other nodes in the cluster, but also considering user positions. This would result in smaller deviations of the approximated position of the multilateration process to the actual user position, which in turn would allow more accurate statements about the latencies from workers to users. Furthermore, Pharos [61] could be used instead of Vivaldi. Pharos is based on Vivaldi and improves the accuracy of network distance predictions by assigning several different network coordinates to each node. One set of coordinates contains the coordinates of the global scope of the workers of a cluster and the other sets represent smaller sub-clusters. These different sized independent Vivaldi networks can then be used to make more accurate statements about latencies using differently sized scopes. However, TIVs are also a problem for Pharos, which means that perfect prediction is not possible. Besides network coordinate systems (NCSs) based on Euclidean distances, there are also systems based on matrix decomposition [46]. The first NCS that uses matrix decomposition is IDES [48]. IDES decomposes the $n \times n$ distance matrix D of a network with n nodes into two $n \times d$ matrices X and Y such that $D \approx X \times Y^T$. The coordinates of a node then consist of two d-dimensional vectors, the incoming and outgoing connection vectors. The predicted distance from node i to node j is then the dot product of the outgoing vector of i and the incoming vector of j. The great advantage of using matrix decomposition is that TIVs do not affect the prediction. However, IDES has two other problems: First, it cannot be guaranteed that the predicted network distance is non-negative. Since the real latencies are always positive, a negative estimate can have a very large impact during the update process and hence on the accuracy of the network. In addition, IDES scales rather poorly, because it requires fixed nodes for the initial computation of the distance matrix D.

A. Figures

A.1. Root and Cluster Orchestrator Scheduling Flow Diagrams

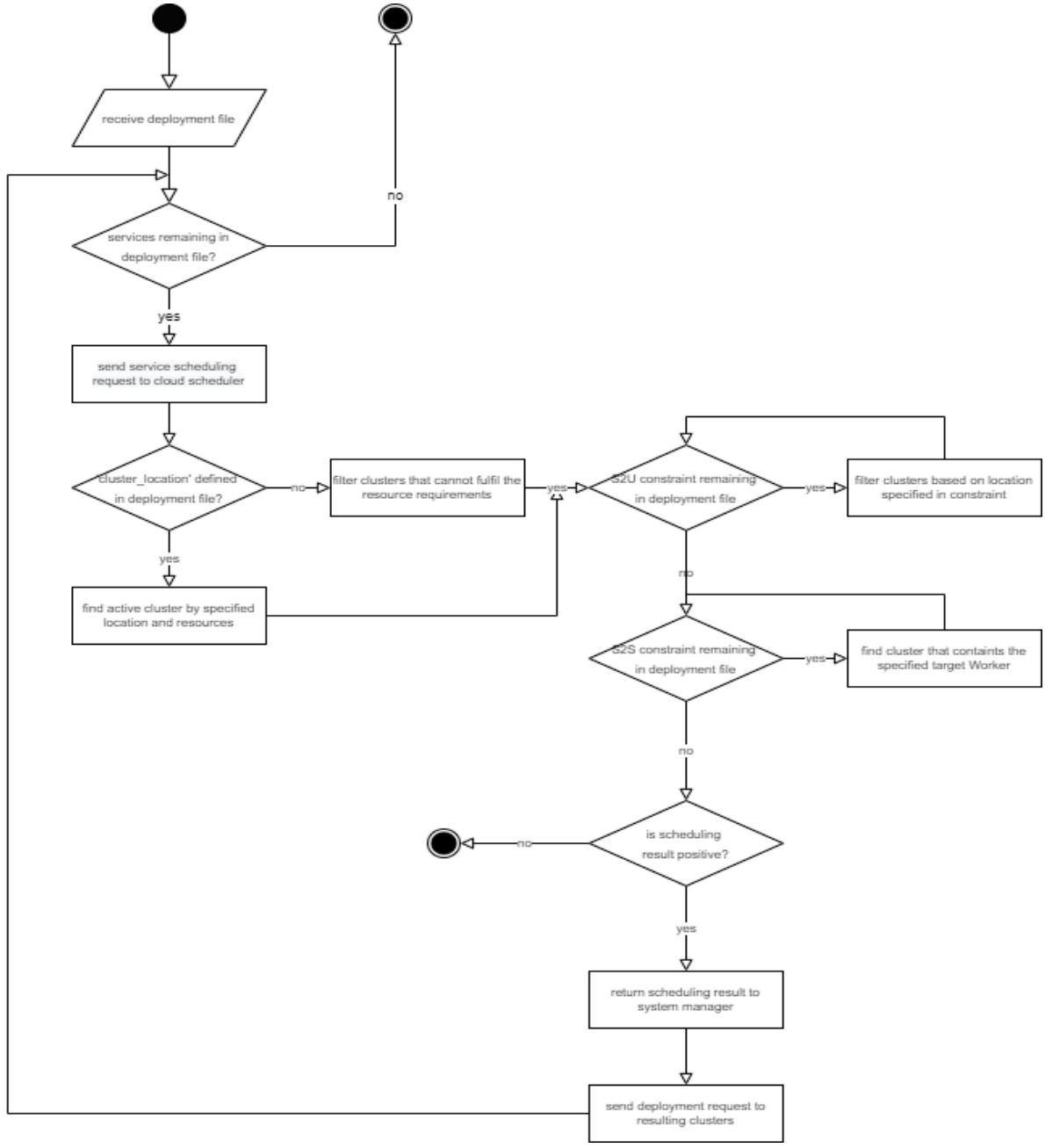


Figure A.1.: Flow diagram for the scheduling process at the root orchestrator level.

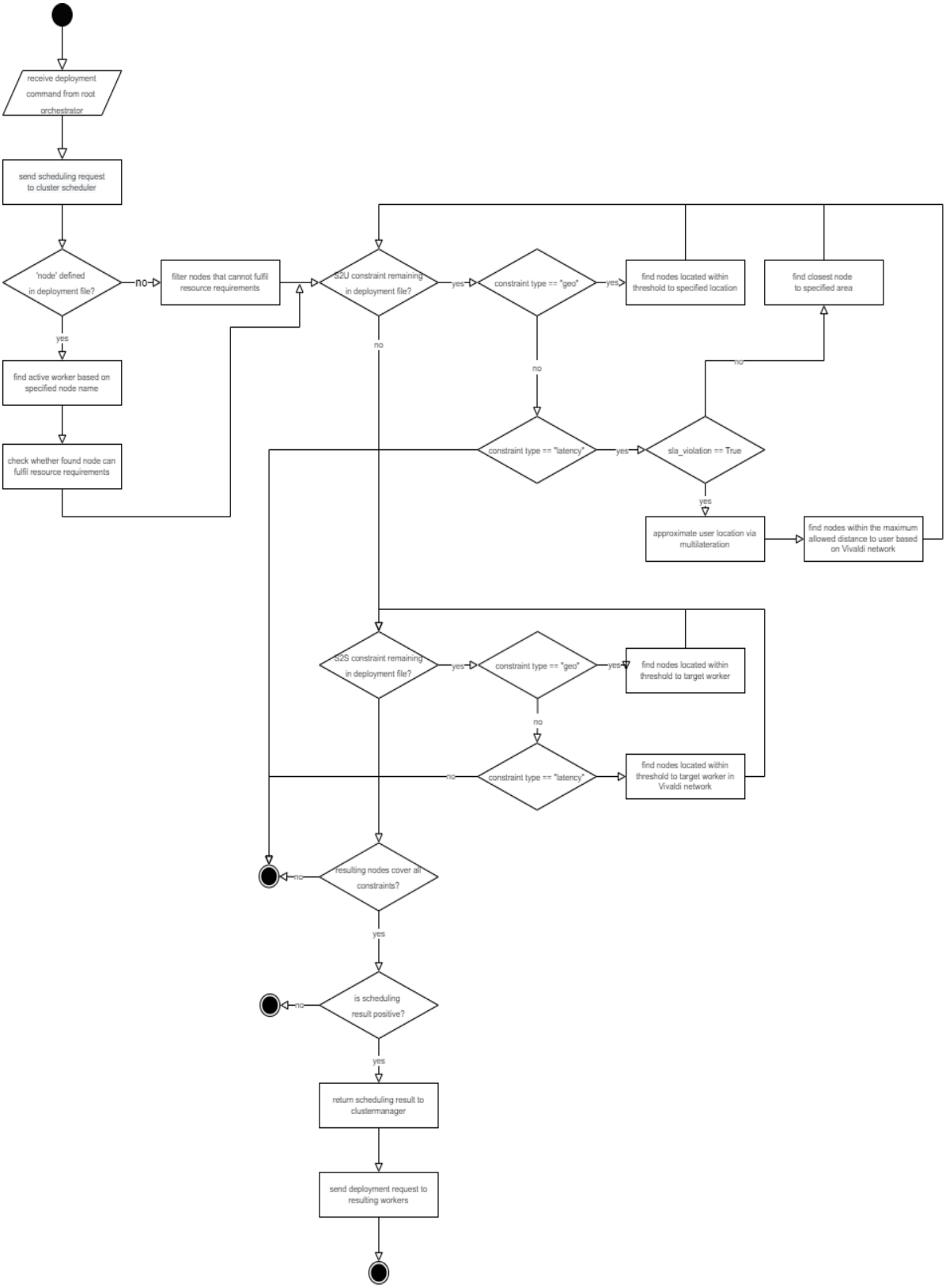


Figure A.2.: Flow diagram for the scheduling process at the cluster orchestrator level.

List of Figures

Acronyms

API non-deterministic polynomial-time hard NP-hard. 14, 19, 34

CASA **c**onstraint-**a**ware **s**cheduling **a**lgorithm. iv, v, 7, 8, 37, 42, 44, 45, 47–51, 53–56

CDN content delivery network. 2

CO cluster orchestrator. 3–5, 7, 19, 21, 26, 27, 29, 30, 32, 33, 35–38, 41, 47, 49, 50, 56

CPU central processing unit. 3, 37, 52

DBSCAN density-based spatial clustering of applications with noise. 30, 56

DNS domain name system. 37, 39, 47

HTTPS hypertext transfer protocol secure. 21, 36, 44, 51

IaaS infrastructure as a service. 6

IoT Internet of Things. iv, v, 1, 5, 9, 12

LLDP link layer discovery protocol. 16

MQTT message queue telemetry transport. 21, 26, 32, 35, 44

MRE mean relative error. 38, 39, 56, 61

NCS network coordinate system. iv, 5, 7, 17, 18, 22, 23, 26, 35, 37–39, 41, 42, 47, 49, 51, 56, 57

NP-hard non-deterministic polynomial-time hard. 9, 11

QoS quality of service. 11, 45, 56

RO root orchestrator. 3–5, 19, 21, 29, 30, 37, 44

RTT round trip time. 4, 5, 17, 22–24, 37, 38, 42, 45, 52, 53

S2S service-to-service. 4, 5, 7, 19, 32–35, 37, 42, 45, 47, 48, 50–56, 61, 62

S2U service-to-user. 4, 5, 7, 16, 32–34, 37, 42, 45–50, 52–57, 61, 62

SDN software-defined network. 16

SLA service level agreement. 3–5, 7, 8, 11, 13, 21, 32–34, 38, 41, 42, 48, 50, 54–56

SLAM software-defined latency monitoring. 16

TIV triangle inequality violation. 17, 18, 27, 38, 39, 47, 56, 57

VM virtual machine. 5, 6, 13, 37, 44, 51

Bibliography

[1] Cisco. *Cisco Annual Internet Report (2018–2023)*. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf. Accessed: 2021-12-29.

[2] C. Guerrero, I. Lera, and C. Juiz. "A lightweight decentralized service placement policy for performance optimization in fog computing". In: *Journal of Ambient Intelligence and Humanized Computing* 10.6 (2019), pp. 2435–2452.

[3] Q. Shaheen, M. Shiraz, M. U. Hashmi, D. Mahmood, R. Akhtar, et al. "A lightweight location-aware fog framework (LAFF) for QoS in Internet of Things paradigm". In: *Mobile Information Systems* 2020 (2020).

[4] A. Brogi and S. Forti. "QoS-aware deployment of IoT applications through the fog". In: *IEEE Internet of Things Journal* 4.5 (2017), pp. 1185–1192.

[5] V. Scoca, A. Aral, I. Brandic, R. De Nicola, and R. B. Uriarte. "Scheduling Latency-Sensitive Applications in Edge Computing." In: *Closer*. 2018, pp. 158–168.

[6] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang. "A survey on mobile edge networks: Convergence of computing, caching and communications". In: *Ieee Access* 5 (2017), pp. 6757–6779.

[7] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar. "Towards qos-aware fog service placement". In: *2017 IEEE 1st international conference on Fog and Edge Computing (ICFEC)*. IEEE. 2017, pp. 89–96.

[8] J. Shamsi, M. A. Khojaye, and M. A. Qasmi. "Data-intensive cloud computing: requirements, expectations, challenges, and solutions". In: *Journal of grid computing* 11.2 (2013), pp. 281–310.

[9] K. Bilal and A. Erbad. "Edge computing for interactive media and video streaming". In: *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE. 2017, pp. 68–73.

[10] T. Zhao, S. Zhou, X. Guo, Y. Zhao, and Z. Niu. "A cooperative scheduling scheme of local cloud and internet cloud for delay-aware mobile cloud computing". In: *2015 IEEE Globecom Workshops (GC Wkshps)*. IEEE. 2015, pp. 1–6.

[11] X. Guo, R. Singh, T. Zhao, and Z. Niu. "An index based task assignment policy for achieving optimal power-delay tradeoff in edge cloud systems". In: *2016 IEEE International Conference on Communications (ICC)*. IEEE. 2016, pp. 1–7.

[12] Y. Mao, J. Zhang, and K. B. Letaief. "Dynamic computation offloading for mobile-edge computing with energy harvesting devices". In: *IEEE Journal on Selected Areas in Communications* 34.12 (2016), pp. 3590–3605.

[13] M. Aazam and E.-N. Huh. "Fog computing micro datacenter based dynamic resource estimation and pricing model for IoT". In: *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*. IEEE. 2015, pp. 687–694.

[14] C. Papagianni, A. Leivadeas, S. Papavassiliou, V. Maglaris, C. Cervello-Pastor, and A. Monje. "On the optimal allocation of virtual resources in cloud computing networks". In: *IEEE Transactions on Computers* 62.6 (2013), pp. 1060–1071.

[15] T. N. B. Duong, X. Li, R. S. M. Goh, X. Tang, and W. Cai. "QoS-aware revenue-cost optimization for latency-sensitive services in IaaS clouds". In: *2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*. IEEE. 2012, pp. 11–18.

[16] A. Aral, I. Brandic, R. B. Uriarte, R. De Nicola, and V. Scoca. "Addressing application latency requirements through edge scheduling". In: *Journal of Grid Computing* 17.4 (2019), pp. 677–698.

[17] M. Pinedo and K. Hadavi. "Scheduling: theory, algorithms and systems development". In: *Operations Research Proceedings 1991*. Springer, 1992, pp. 35–42.

[18] R. Raju, R. Babukarthik, D. Chandramohan, P. Dhavachelvan, and T. Vengattaraman. "Minimizing the makespan using Hybrid algorithm for cloud computing". In: *2013 3rd IEEE International Advance Computing Conference (IACC)*. IEEE. 2013, pp. 957–962.

[19] M. R. Garey and D. S. Johnson. *Computers and intractability*. Vol. 174. freeman San Francisco, 1979.

[20] A. Keivani and J.-R. Tapamo. "Task scheduling in cloud computing: A review". In: *2019 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD)*. IEEE. 2019, pp. 1–6.

[21] C. D. Polychronopoulos and D. J. Kuck. "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers". In: *Ieee transactions on computers* 100.12 (1987), pp. 1425–1439.

[22] L. F. Bittencourt, A. Goldman, E. R. Madeira, N. L. da Fonseca, and R. Sakellariou. "Scheduling in distributed systems: A cloud computing perspective". In: *Computer science review* 30 (2018), pp. 31–54.

[23] F. Dong and S. G. Akl. "Scheduling algorithms for grid computing: State of the art and open problems". In: *School of Computing, Queen's University, Kingston, Ontario* (2006), pp. 1–55.

[24] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[25] K. Ashton et al. "That 'internet of things' thing". In: *RFID journal* 22.7 (2009), pp. 97–114.

[26] Y. Chen, R. Mahajan, B. Sridharan, and Z.-L. Zhang. "A provider-side view of web search response time". In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 243–254.

[27] *Driverless Cars' Need for Data is Sparking a New Space Race*. `https://www.bloomberg.com/news/articles/2021-09-17/carmakers-look-to-satellites-for-future-of-self-driving-vehiclesl`. Accessed: 2021-1-7.

[28] R. Righi. *Scheduling Problems: New Applications and Trends*. BoD–Books on Demand, 2020.

[29] P. Salot. "A survey of various scheduling algorithm in cloud computing environment". In: *International Journal of Research in Engineering and Technology* 2.2 (2013), pp. 131–135.

[30] R. D. Lakshmi and N. Srinivasu. "A dynamic approach to task scheduling in cloud computing using genetic algorithm." In: *Journal of Theoretical & Applied Information Technology* 85.2 (2016).

[31] R. M. Singh, S. Paul, and A. Kumar. "Task scheduling in cloud computing". In: *International Journal of Computer Science and Information Technologies* 5.6 (2014), pp. 7940–7944.

[32] J. Daniels. "Server virtualization architecture and implementation". In: *XRDS: Crossroads, The ACM Magazine for Students* 16.1 (2009), pp. 8–12.

[33] C. Pahl. "Containerization and the paas cloud". In: *IEEE Cloud Computing* 2.3 (2015), pp. 24–31.

[34] *Kubernetes*. `https://kubernetes.io/`. Accessed: 2021-12-29.

[35] *KubeEdge*. `https://kubeedge.io/en/`. Accessed: 2021-12-29.

[36] *ioFog*. `https://projects.eclipse.org/projects/iot.iofog`. Accessed: 2021-12-29.

[37] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. "Reducing web latency: the virtue of gentle aggression". In: *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. 2013, pp. 159–170.

[38] A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and C. Yu. "Transparent and flexible network management for big data processing in the cloud". In: *5th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 13)*. 2013.

[39] M. Moshref, M. Yu, A. Sharma, and R. Govindan. "Scalable rule management for data centers". In: *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 2013, pp. 157–170.

[40] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. "Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator". In: *ACM SIGCOMM Computer Communication Review* 39.4 (2009), pp. 255–266.

[41] L. Yang, Z.-P. Cai, and H. Xu. "LLMP: exploiting LLDP for latency measurement in software-defined data center networks". In: *Journal of Computer Science and Technology* 33.2 (2018), pp. 277–285.

[42] C. Yu, C. Lumezanu, A. Sharma, Q. Xu, G. Jiang, and H. V. Madhyastha. "Software-defined latency monitoring in data center networks". In: *International Conference on Passive and Active Network Measurement*. Springer. 2015, pp. 360–372.

[43] A. Gupta, B. Liskov, and R. Rodrigues. "Efficient Routing for Peer-to-Peer Overlays." In: *NSDI*. Vol. 4. 2004, pp. 9–9.

[44] T. Iimura, H. Hazeyama, and Y. Kadobayashi. "Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games". In: *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. 2004, pp. 116–120.

[45] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. "Vivaldi: A decentralized network coordinate system". In: *ACM SIGCOMM Computer Communication Review* 34.4 (2004), pp. 15–26.

[46] R. Cheng and Y. Wang. "A Survey on Network Coordinate Systems". In: *MATEC Web of Conferences*. Vol. 232. EDP Sciences. 2018, p. 01037.

[47] C. Lumezanu, R. Baden, N. Spring, and B. Bhattacharjee. "Triangle inequality and routing policy violations in the Internet". In: *International Conference on Passive and Active Network Measurement*. Springer. 2009, pp. 45–54.

[48] Y. Mao, L. K. Saul, and J. M. Smith. "Ides: An internet distance estimation service for large networks". In: *IEEE Journal on Selected Areas in Communications* 24.12 (2006), pp. 2273–2284.

[49] C. Boutsidis and E. Gallopoulos. "SVD based initialization: A head start for nonnegative matrix factorization". In: *Pattern recognition* 41.4 (2008), pp. 1350–1362.

[50] D. D. Lee and H. S. Seung. "Learning the parts of objects by non-negative matrix factorization". In: *Nature* 401.6755 (1999), pp. 788–791.

[51] C. L. Sang, M. Adams, T. Korthals, T. Hörmann, M. Hesse, and U. Rückert. "A bidirectional object tracking and navigation system using a true-range multilateration method". In: *2019 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*. IEEE. 2019, pp. 1–8.

[52] N. R. Chopde and M. Nichat. "Landmark based shortest path detection by using A* and Haversine formula". In: *International Journal of Innovative Research in Computer and Communication Engineering* 1.2 (2013), pp. 298–302.

[53] *Tshark - Python Network Analyzer*. `https://www.wireshark.org/docs/man-pages/tshark.html`. Accessed: 2021-1-6.

[54] *MaxMin GeoLite2 Ip Geolocation Database*. `https://dev.maxmind.com/geoip/geolite2-free-geolocation-data?lang=en`. Accessed: 2021-12-10.

[55] *Pandas*. `https://pandas.pydata.org/`. Accessed: 2021-1-6.

[56] J. Ledlie, P. Gardner, and M. I. Seltzer. "Network Coordinates in the Wild." In: *NSDI*. Vol. 7. 2007, pp. 299–311.

[57] *King Data Set*. `https://pdos.csail.mit.edu/archive/p2psim/kingdata/`. Accessed: 2021-1-5.

[58] *Future SOC Lab*. `https://hpi.de/forschung/future-soc-lab.html`. Accessed: 2021-1-3.

[59] *tc - Traffic Control*. `https://linux.die.net/man/8/tc`. Accessed: 2021-1-3.

[60] *netem - Network Emulator*. `https://man7.org/linux/man-pages/man8/tc-netem.8.html`. Accessed: 2021-1-3.

[61] Y. Chen, Y. Xiong, X. Shi, B. Deng, and X. Li. "Pharos: A decentralized and hierarchical network coordinate system for internet distance prediction". In: *IEEE GLOBECOM 2007-IEEE Global Telecommunications Conference*. IEEE. 2007, pp. 421–426.