DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Realizing Self-organizing Platforms for Edge Service Deployments

Ralf Baun

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Realizing Self-organizing Platforms for Edge Service Deployments

Realisierung von selbstorganisierenden Plattformen für die Bereitstellung von Edge-Services

Author:	Ralf Baun
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Dr. Nitinder Mohan
Submission Date:	15.07.2021

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

Munich, 15.07.2021 Ralf Baun

Abstract

Edge computing provides computational resources in proximity to their users instead of—or in combination with—resources from remote clouds. Their locality can enable latency-critical use cases like Virtual Reality (VR) and autonomous vehicles.

Operating services in Edge environments is challenging for several reasons. Edge services can have strict latency or data locality requirements and benefit greatly from runtime adaptions of service placements that account for dynamism in hardware state and user demand. Computing units at the Edge can be heterogeneous and geographically distributed, requiring a platform management system that accounts for these variations. Service cluster sizes in Edge environments can span multiple geographic regions and can be orders of magnitude larger than clusters in the cloud. Currently, there is no service operating system that is able to address all of these issues at once.

In this thesis, we developed VineIO, a system which realizes self-organizing platforms for Edge services. By monitoring hardware load and user latency via eBPF, the platforms can account for heterogeneous, dynamic Edge environments. They communicate within a tree-structured overlay network, sharing capacity data and executing service orchestration tasks cooperatively. To adapt to their operating environments in real-time, they migrate users, replicate and terminate services while considering hardware load as well as volume and origin of user traffic. This way, organically built service trees which do not have a centralized control plane are created, removing the need for manual intervention and micromanagement of service clusters.

Our evaluation highlights the benefits of our decentralized, bottom-up approach to service orchestration by showing that the overhead of user migrations does not depend on service cluster sizes. Furthermore, VineIO was able to unify two heterogeneous and geographically distributed platforms within a service cluster. In these settings, it can adapt service placement according to user demand in real-time across federated infrastructure sets.

Based on our results, we suggest that more research efforts should be spent on distributed, QoS-aware Edge service platforms.

Contents

1 Introduction

Cloud computing became a popular way to operate Internet services and is still adopted increasingly. While in 2018, 24% of all EU enterprises relied on the cloud, it were already 36% in 2020 [1]. Datacenters that are spread across major regions of the world [2] contain numerous machines that are efficiently interconnected with each other [3]. They can efficiently operate and manage services of many customers on shared hardware sets using technologies to isolate encapsulated applications and virtualize network topologies [4] [5]. Especially lightweight virtualization technologies like containers became popular in the cloud [4]. They decrease operational overhead, while bringing benefits like portability and a higher deployment reproducibility for software developers at the same time [6] [7]. Ecosystems around containers emerged, including service cluster platforms like Kubernetes [8], which manage clusters of containers automatically and fulfill a large set of operational responsibilities like horizontal scaling and service maintenance. These tools made Internet service operation more effective for all stakeholders, letting even new software engineering paradigms like DevOps emerge [4].

However, in some cases, there are reasons to operate services outside of datacenters. Precisely, operating services near their users can be beneficial in several ways. This computing paradigm is typically called Edge computing—since the computing devices are located at the Edge of the Internet topology [9]—and has been a popular research topic for industry and academia [10]. Companies like Cloudflare utilize Edge resources to bring their CDN closer to the customer, thereby reducing the network latency to them [9]. Emerging IoT use cases in several domains likewise require low latencies in order to provide real-time interaction modalities to their users [10]. For instance, Edge computing can enable VR and AR devices to offload their processing of visual content without impairing the user experience [11]. Also, mobile robotic devices in industrial environments can be coordinated in real-time on Edge devices, resulting in quick decision-making routines [11]. To give another example, real-time video analytics systems can benefit from Edge devices that are located near the source of the video feed, circumventing the need to share sensitive material to remote cloud datacenters while increasing the effectivity of the video processing pipeline [12]. Edge computing can enable several use cases, acting as a platform for concepts like smart cities [10].

1.1 Problem Statement and Contribution

Although work has been spent in researching this computing paradigm from several parties, there are open issues that limit the usability of the Edge. For one, Edge environments can be heterogeneous, containing computational platforms of different sizes and architecture, which needs to be considered during service placement [13]. Otherwise, incompatibilities between services and Edge devices will result in operational failures. Next, Edge services can have strict latency and data locality requirements in order to operate effectively, which means that their proximity to the user must be managed during runtime in order to keep latency bounds [13]. VR users that access too-far VR backends and therefore experience network delays longer than 20 ms can experience motion sickness [2], which must be prevented. The issues of Edge service requirements and heterogeneous device landscapes become more severe due to the dynamism that is typical present in Edge environments [13]. Edge devices can move, appear or disappear over time and fluctuations in available computing capacity are possible [13]. Therefore, Edge devices can become suboptimal for their hosted service instances. The dynamism also includes user traffic, which can vary over demand in volume and origin [14]. Furthermore, the scale of Edge service clusters can be problematic. While cloud service clusters are usually not larger than thousands of nodes, Edge service clusters can be orders of magnitude bigger, spanning larger geographical areas [15] [14]. Existing solutions that are tailored towards cloud environments do not support these dimensions in operation by default [14] [16]. Although all of these aspects are addressed in several research works, no Edge service platform that addresses all of these issues could be found during our research.

Therefore, the question arises whether it is possible to build such a service platform. Specifically, we ask the following research questions:

RQ1. How can we design platforms that can schedule services in heterogeneous Edge environments?

RQ2. Can these platforms be aware of dynamism on the Edge and operate accordingly?

To create a service platform for Edge environments, it must be aware of the underlying Edge device and respect its hardware settings. Important factors like CPU architecture and available main memory are crucial knowledge for determining the compatibility with potentials services. Furthermore, it must be able to sense changes in available hardware capacity in order to operate its services in a frictionless manner. Due to the mentioned dynamic nature of Edge environments, a service scheduler must have the possibility to horizontally scale services in real time. This way, the effect of

resource shortages, which can be caused by an increased user demand for example can be mitigated.

RQ3. Can these platforms be organized in a way that is scalable and minimizes external intervention?

Due to the size of Edge environments, scheduling platforms must organize themselves efficiently. Means to remove the centralized control plane as a bottleneck of existing platforms like Kubernetes [16] must be found. Since the scale of some operational use cases correspondingly increases the administrational overhead, a replacement for such a control plane should also remove the need for micromanagement [17]. Effective communication structures in combination with corresponding data exchange mechanisms are required to realize this [17].

RQ4. Are these platforms able to respect QoS requirements by adapting to changes in volume and origin of user demand?

As described, the dynamism of Edge infrastructures also includes their user traffic, which can disqualify service placement choices over time. Therefore, matchings between service requirements and Edge platforms as well as matchings between service instances and their users must be reconsidered periodically. To accomplish this, service schedulers should be provided with data about the fulfillment of QoS for users, together with their positional information. It should then not only consider the volume of user demand during its operation, but also the quality of the respective user sessions. This enhances service placement and load balancing decisions which can preserve the QoS by considering these metrics.

In order to answer these questions, we built VineIO, which is a system of self-organizing Edge service platforms.

1.2 Thesis Structure

This thesis is structured as follows. First, we add more context to the topic of cloud and Edge computing while providing an overview about related works in Chapter 2. Then, the concept of self-organizing platforms is proposed and described in detail in Chapter 3, also providing insights into the inner workings of our implementation of the concept. To evaluate VineIO from different perspectives, we discuss the quality of its implementation and experimentally analyze VineIO's behavior at different scales as well as its usability in heterogeneous environments in Chapter 4. Finally, we conclude the thesis with a judgement on the presented work, highlighting opportunities to evolve the concept in Chapter 5.

2 Background

We start the topic of Edge service orchestration by first providing an overview about cloud computing and Edge computing in general, highlighting related works that are relevant for our thesis goals.

2.1 Cloud Computing

Although the concept of cloud computing existed earlier [18], it became popular to Internet-related businesses in the early 2000s. Back then, companies like AWS intended to monetize their expertise about efficient and flexible Internet service operation by offering hosting solutions to third parties [19]. This lead to the launch of cloud hosting products like the AWS Elastic Compute Cloud (EC2) in 2006 [18].

Nowadays, cloud computing has become a primary paradigm for modern Internet service operation [18]. The National Institute of Standards and Technology (NIST) defines cloud computing as a method of enabling "ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources" [20]. Typically, access to these resources is offered by a simple API, via which service operators can make changes to their rented cloud infrastructure within seconds without much interaction. This enables the operation of elastic services which can adapt their capacity quickly to meet their current demand. Although the cloud typically offers traditional hardware abstractions like virtual machines via Infrastructure-as-a-Service products like AWS EC2, alternative operation models that leverage the advantages of cloud systems emerged. In Platform-as-a-Service (PaaS) offerings, the operating system of machines is hidden and only a language or container-specific runtime is provided to the customer [10] [18]. This brings a more flexible resource allocation and a lower operational overhead to the cloud provider while cloud computing customers have an easier operation methodology for their services.

A key enabler of cloud offerings are virtualization technologies, as they provide a convenient deployment format upon which service and cloud operators agree while also separating and, to varying degree, even isolating running services. However, the traditional virtual machine that was already used before cloud computing became popular has a high overhead in terms of image size as well as operational costs [6]. The need for application portability and isolation in combination with a good operation

efficiency triggered the establishment of lightweight virtualization technologies [6] [21]. Nowadays, container technologies like Docker which only virtualize the user space of an application while sharing a common system kernel among running containers are the de-facto standard deployment format for the cloud. Alone in Q4 2020, more than 30 billion pulls were done on the Docker Hub container image repository [22]. The container format introduced by Docker was even standardized via the Open container Initiative [23], which is an institution under the umbrella of the Linux Foundation. Alternative specification-compliant implementations of different part of the container ecosystem exist, like runc [24] and crun [25] container runtimes. Due to the versatility, interoperability, and light weight of containers, they are typically used in all phases of software development. However, since the grade of isolation between containers is considered too weak, approaches to enhance the security of containers exist. For instance, there are wrapper technologies for containers like Firecracker [26] and gVisor [27]. But there are also approaches that explore alternative concepts to containers. Originating from academia with works like MirageOS [28], IncludeOS [29] and RustyHermit [30], Unikernels [21] emerged several years ago and are the central part of industry products like NanoVMs [31]. They aim to consolidate the best of VMs and containers, providing strong isolation between instances at Hypervisor-level while having a high operation efficiency due to a low amount of abstractions between application and hardware. Most of these Unikernels are based on a language-specific OS library, upon which developers need to develop their services, which poses an obstacle to Unikernel adoption. For this reason, research on Unikernels that are able to execute Linux binaries is conducted too, resulting in works like HermiTux [32].

Since cloud users potentially operate numerous virtualized service deployments, an automated way to maintain and control these is necessary [33]. Especially the mentioned large-scale service clusters of big technology companies like Google triggered the development of service orchestration tools [33]. Nowadays, products like Kubernetes [8], Apache Mesos [34] and Docker Swarm [35] are used even by significantly smaller companies, since they automate and take over many typical service operation responsibilities [6]. For instance, they help in efficient resource allocation, lifecycle management, service discovery and autoscaling of deployments [8]. By having a declarative cluster specification, which describes the desired state of the orchestrated service mesh in contrast to imperative deployment scripts that are fragile and inflexible, the operation of a cloud service can be automated effectively [8]. Together with modern operational paradigms like Infrastructure-as-Code and other technologies, orchestration enabled the working-cultural movement of DevOps [4], where employees occupy responsibilities of both development and operation in web development.

To summarize, the flexible operation modes, easy and standardized provisioning of applications and the effective formalization of deployments make the cloud an attractive

operating environment for the majority of Internet-based services. The current volume of the cloud market reflects this fact, since it surpassed 41 billion US dollars in the first quarter of 2021 [36].

2.2 Edge Computing

Nowadays, the cloud is spread all over the world. Countries in North America, Europe, and Oceania are covered well by respective datacenters [2]. Nevertheless, they are not ubiquitous, especially on the countryside or in developing countries. In general, networking conditions to cloud services are worse the farther away the consumer is from the cloud due to the increasing influence of the network path on the data transmission. In addition to regular Internet users that experience long or failed loads of web pages, the Internet of Things (IoT) which is established in various domains and consists out of inter-connected smart devices [37] [15] is affected by this situation. There is a mismatch between datacenter positioning and devices like networked sensors, since these can potentially be located everywhere [37] [11] [38]. This is not an issue for the majority of use cases, since they do not require a particularly reliable network in order to operate and serve the user normally. They are to a certain extent resilient against varying networking conditions or even generally subpar networking conditions. However, while many consumer-oriented IoT-devices can therefore be operated by a traditional cloud Backend, some IoT use cases are not compatible with the cloud [39]:

- *VR/AR*: To prevent issues like motion sickness of the user, VR and AR applications need to work within Motion-to-Photon latency bounds of under 20 ms [2].
- *Industrial IoT*: In the area of Industrial IoT (IIoT), large clusters of robotic devices need to be coordinated in real-time [11]. Sometimes, these robotic devices operate in areas with poor connectivity, like autonomous agricultural vehicles [39].
- *Autonomous Driving*: Similarly to IIoT, emerging autonomous vehicles also need mechanisms to coordinate themselves in traffic, requiring fast data analysis and decision-making processes [11].
- *Real-Time Video Analytics*: To be able to process large volumes of data quickly in real-time and sometimes even to fulfill legal constraints that affect the analyzed material, video analytics use cases might need to be conducted near the origin of the material [12].
- *Smart Cities*: Smart cities incorporate a combination of the mentioned use cases and add more in various domains. Once the number of IoT devices increases

in smart cities of the future, a dependency to external computing resources in a remote data center might be undesirable.

In these scenarios, tight requirements on latency, data locality and privacy cannot be reliably fulfilled by remote datacenters [10] [4] [39]. Furthermore, these issues add to the other relevant challenges that are present in IoT and make serving smart devices more complicated, like device mobility or subpar networking capabilities of IoT devices themselves [37]. One approach to circumvent the mentioned issues of the cloud in the context of IoT is to provide decentralized computing resources which are located in proximity to affected IoT devices. These can assist in all tasks that were intended to be outsourced to the cloud, like entity coordination and high-volume/real-time data processing [10]. Network connectivity to devices in proximity usually beats the connectivity to remote clouds due to the shorter paths significantly, with possible latency improvements up to 30 percent and more [40]. This way, latency-critical use cases like distributed AR/VR computing can be enabled in situations where a cloud connection would not be sufficient. Trivially, data locality requirements also can be fulfilled more easily with devices that are close to their users compared to servers in remote datacenters that might be in a different country.

2.2.1 Concepts of Edge Computing

Motivated by the described issues of the cloud, the concept of having additional computing resources in proximity that supplement or even replace the cloud has been a topic of academia and industry recently. This way of computing is usually called Edge computing, with the respective local devices called Edge devices [10]. The purpose of Edge computing is to assist nearby users in their computational tasks, for example because they have only a constrained set of computing power available. To a user device, which can be for example a smartphone or an IoT device, these assistance devices can be seen as a nearby distributed data center. The term “Edge” therefore describes the location of Edge devices in relation to the Internet topology, since they are typically near the access points of their users. They may be located at cellular stations [41] or at Internet exchange points (IXPs), which is often the case for Edge servers of CDNs like Cloudflare [9].

In the course of Edge computing research, several related computing models have been proposed and partly even standardized. One of these is Fog computing [42]. Similarly to Edge computing, Fog computing refers to computational resources that are located between a user and the cloud. An important distinction to regular Edge computing is that these resources do not need to be located directly at the Edge of the network. Instead, they may be located anywhere along the path between a cloud

datacenter and their users [42] and can be seen as a continuous extension of the cloud. Although Edge computing is typically used as a group term for computational concepts that encompass additional local resources, Fog computing can actually be seen as a superset of Edge computing [10].

Another proposed model is about so-called Cloudlets [43]. They are small helper devices at the Edge that form a local datacenter to supplement nearby consumer devices, which mostly resembles the concept of Edge computing. However, their use case is more specific, since they are targeted towards mobile devices. Also, Cloudlets are intended to be highly flexible in their usage: They are able to receive virtualized tasks that are offloaded by these mobile devices.

Similarly, Multi-access Edge Computing (MEC) [41] can be seen as a subset of Edge computing [10]. In this concept, Edge devices are placed in the Radio Network Access infrastructure to serve users that are connected to the local network. Originally intended to be used by mobile devices—it was formerly called Mobile Edge Computing [10]—the concept was broadened to include more use cases that can operate on wireless connectivity, like real-time video analytics [10]. In general, MEC devices can enable applications that require a low latency or high bandwidth due to their proximity to their users and wireless connectivity standards like 5G [41] [10].

A more extreme form of Edge computing is Mist computing [42]. In Mist computing, there is no need for additional Edge devices, since IoT devices themselves form a dynamic overlay network in order to share resources and cooperative solve tasks. These overlay networks may form ad-hoc, similarly to mobile ad-hoc networks (MANETs) [44], or they may be pre-configured, depending on the concrete use case [10]. The resulting networks are highly resilient and suitable for use cases like a self-aware health monitor system which is distributed over a heterogeneous set of dedicated IoT devices like sensors [45].

The mentioned concepts are only a few beside a variety of others [10]. Between them, there is no clear consensus between the affected corporate and academic entities about which of these concepts shall be implemented in which situations and where the respective computing units should be placed [40]. However, since all of these concepts have their focus on computational resources in proximity of their users, we set the general context of this thesis to exactly this paradigm. From now on, we use Edge computing as a group term to describe this way of computing and Edge environment as a term for infrastructure which includes Edge devices and their part of the Internet topology.

2.2.2 Operational Challenges in Edge Environments

Independent of the way Edge computing infrastructures are implemented, a key interest of service operators is the way their applications are provisioned and managed. Due to the described revolutionary way of managing Internet services in the cloud using service orchestration tools and lightweight virtualization technologies, work has been done in adopting similar ways of service operation to the Edge. Like in cloud computing, service operators would like to deploy and administrate services at the Edge conveniently and automatically following modern DevOps workflows, while the complexity of the underlying infrastructure is hidden below an abstraction layer. In the meantime, Edge environment providers want to manage and utilize their Edge resources effectively while saving operational costs [46] [4] [47]. An Edge service platform that adopts these cloud paradigms would be desirable for these parties.

However, there are major challenges in Edge environments that are not present in the cloud. For one, Edge environments are in several regards more complex than cloud environments. In contrast to uniform and carefully calibrated datacenter hardware, Edge devices come in various forms [13]. They can have varying hardware properties, including computing capacity, CPU architecture and hardware components like a dedicated GPU. This increases the complexity of service placement in Edge environments. The fact that the computational resources of some Edge devices are significantly smaller than a typical computing unit in a datacenter makes this issue more severe [13] [37] [38]. Furthermore, these devices are not located in datacenter buildings, where they can cooperate efficiently, but instead are distributed across the Edge of the Internet. Most notably, they are both physically and logically distributed. This means that units of an Edge device cluster might be not part of the same LAN and they may be even spread over large geographic areas [14]. This hinders access and management of these devices as well as cooperation between the devices [15] [48]. An additional burden are the inconsistent and varying networking conditions between Edge devices [13]. Although wired Internet connectivity can be part of several Edge computing use cases, alternative scenarios have to deal with inferior connection qualities. For instance, some devices could be connected to the Internet via mobile cellular connections [15] or Bluetooth [13]. However, depending on the use case, the connection modality at the Edge could be a satellite connection with only a fraction of the bandwidth of a datacenter backbone, while having a latency that is orders of magnitude higher [49]. In contrast to that, cloud datacenters use reliable, high-performing networking hardware and linked according to optimized network topologies like a Fat Tree [3]. Apart from the hardware and networking related issues, Edge service topologies can dynamic [13]. Devices are not necessarily running all the time and might therefore appear and disappear at certain times. Moreover, they may move physically and logically, which can result in

connectivity issues and an additional management overhead [38].

On top of that, Edge services that are suited towards these environments typically have properties that can make their operation challenging. For instance, some service clusters can be huge, having "hundreds of thousands of devices" [15]. Cluster sizes that reach this order of magnitude can cause issues for service orchestration systems. Furthermore, the requirements of Edge services are often greatly restricting the modes of possible operation. As described above, several Edge use cases require a reliable network that provides low latencies to an Edge backend [13] [14]. This restricts the range of possible service placements greatly. Data locality requirements due to privacy policies or simply the impracticality of remote data processing have a similar effect [10]. Additionally, services might have special requirements on their underlying hardware. For some, a GPU might be essential, e.g. in real-time video analytics use cases that involve object detection algorithms [12]. These can be costly to run on CPUs. Such requirements need attention during service operation due to the heterogeneous nature of Edge environments [14]. Ultimately, all these possible properties of Edge services make Edge platforms non-fungible in contrast to datacenters where the physical identity and properties of a specific computing platform usually does not play a role. Due to the dynamism of Edge environments and varying user demands, Edge service platforms need mechanisms that account for these issues at runtime [14] [47].

2.3 Related Work

Academia and industry try to find ways to apply operational concepts in cloud computing to Edge environments while respecting their requirements and unique properties [4]. However, established tools that implement service platforms in cloud environments were originally not intended to be used in Edge environments. In the following, existing approaches to use established tools as well as tools that were newly developed for Edge environments are presented. Afterwards, we highlight solutions for the individual operational challenges of Edge environments.

2.3.1 Service Platforms in Edge Environments

For cloud environments, the service orchestration systems *Kubernetes* [8], *Apache Mesos* [34] and Docker in Swarm mode (also known as *Docker Swarm*) [35] are among the popular ones. In general, Kubernetes performs better than others, including the mentioned alternatives Mesos and Swarm [14]. However, Mesos scales better in terms of cluster sizes: Kubernetes is designed for sizes under 5000 cluster nodes [8], while Mesos is advertised with having linear scalability and cluster sizes of 10000s of nodes [34]. Regardless of these properties, all three orchestration tools turn out to be insufficient

for the Edge [4] [47], lacking configuration options suited for Edge deployments as well as mechanisms for edge-related orchestration tasks [50]. Among other drawbacks, they do neither consider network delays [14], nor do they consider the heterogeneity of Edge environments [47] in their scheduling behavior. In contrast to that, systems that are modified or even built from the ground up to respect the discussed properties of Edge environments promise to be more effective.

One approach to adopt existing popular tools towards distributed infrastructure setups is *KubeFed* [51], which is a multi-cluster extension of Kubernetes. It is intended to make the management of multiple related sub-clusters easier by unifying them into a hosting cluster over which global operations can be conducted through a single API. This solution explicitly aims to support "multi-geo applications" [51]. However, it does not add any Edge-aware intelligent monitoring and scheduler mechanisms. Therefore, although it enhances the scalability of Kubernetes-managed deployments via the concept of multi-clusters and enables geo-distributed applications, the extension does not alter the original cloud-specialization of Kubernetes. For instance, it does neither respect the heterogeneity of hardware infrastructures, nor does it account for runtime dynamism of Edge Environments. Consequently, it is feature-wise not comparable with multi-cluster service platforms that specifically target Edge environments. It should be noted that this extension of Kubernetes is not finished, but in beta-quality.

With a similar, but more complete and more intrusive approach, the Open Infrastructure Foundation [52] tries to make service orchestration viable in Edge environments. The project is called *StarlingX* [53] and tries to create a fully-featured IaaS and PaaS system that is optimized for Edge environments. It is built on top of Kubernetes and OpenStack [54], which is a framework to host cloud platforms and is used to realize private on-premise clouds. However, it used these technologies with the intention to "reconfigure proven cloud technologies for Edge Compute" [55]. By supporting geo-distributed multi-cluster deployments that have an own control plane each, it conforms to the Distributed Control Plane reference architecture of the OpenStack project[1]. Nevertheless, StarlingX inherits the cloud-centeredness of the tools it relies on. For instance, due to their use of Kubernetes, they do not have any support for dynamic, QoS-aware service topologies which can be especially necessary in Edge use cases.

To solve the issue of cloud-orientation that many existing tools have, other projects attempt to build an Edge-aware service platform from scratch. One of these tools is Eclipse *fog05*, which is a IaaS abstraction layer for Fog environments [39] [56]. It is therefore conceptually comparable to OpenStack and StarlingX. A main goal of fog05 is the unification of Edge and Fog computing in terms of networking and storage

[1] `https://wiki.openstack.org/wiki/Edge_computing_Group/Edge_Reference_Architectures#Distributed_Control_Plane_Scenario`

infrastructure, thereby solving the heterogeneity issue of Edge environments. Also, it tries to respect connectivity and security issues of IIoT applications. To accomplish this, it has 2 Layers: FIM and FOrcE. FIM is the infrastructure virtualization layer which covers all kinds of computing resources that are present in Edge environments. It provides interfaces to uniformly manage infrastructure units and their deployments to the control plane of the system. Notably, It works in a distributed manner and uses Eclipse Zenoh [57] as a decentralized database backend to store and share the system's state. On top of FIM lies the FOrcE, which is the orchestration layer of fog05. Although it also works in a decentralized manner and uses Zenoh, it represents a centralized control plane and controls deployments from the outside. These deployments can lie on multiple separated FIMs as well as Kubernetes clusters, supporting cluster federation this way. Fog05 supports a wide range of different deployment formats. The so-called Fog Deployment Units (FDU) can be Binaries, Unikernels, containers or VMs. Considering all these facts, fog05 has several measures in place to adopt the concept of cloud service platforms to the Edge. However, it has no focus on QoS-aware service topologies, which can be a disadvantage in some Edge use cases.

Apart from industry-driven approaches, researchers in academia also try to analyze and solve the problem of service orchestration in Edge environments. One mentionable work is *PiCasso*, a lightweight Edge orchestration system from Lertsinsrubtavee et al. [50]. Unlike the already presented solutions, PiCasso aims to solve the challenge of latency-aware orchestration on Edge devices while respecting the unique capabilities of each device. The prototype which was developed in Python monitors the hardware load of its nodes and sends the information to a dedicated, centralized orchestration engine. The scheduler of this engine triggers the creation of a replica, if necessary. One goal of their work was to compare two different replication strategies. In one strategy, a replica is spawned on the same hardware node in order to horizontally scale an application on the same geographic position. In the other strategy, they spawned a replica on another hardware node in order to achieve better response times. During their evaluation, they determined that replicating delay-sensitive services to other remote platforms can greatly improve the Quality of Experience (QoE) of the affected users. For this reason, they suggest developing an intelligent scheduling algorithm that respects hardware load, networking conditions and service requirements in order to improve the system.

Similarly, Rossi et al. pursued the idea of QoS aware service orchestration with *ge-kube*, which enables geo-distributed deployments via Kubernetes [14]. They extended Kubernetes with scheduling and resource management components that take the position of deployed service instances and the observed network latency between their worker nodes into account. Since they correlate the QoS of an application with latency between its worker nodes, ge-kube optimizes the application placement according to the observed network delay of worker nodes. To implement such a mechanism, they

utilized reinforcement learning to process monitored information and make scaling decisions at the control plane, involving the custom Kubernetes components that were created in the course of this research project. Since the results of their technique were promising, improving e.g. the performance of Redis significantly in their evaluation experiment, they suggest extending this approach further. For instance, considering stateful migrations and multi-service deployments are ways to increase the usefulness of their solution.

2.3.2 Proximity-Aware Serving: Load Balancing and Service Placement

It can already be seen from the presented related works that creating a service orchestration system for the Edge has various angles that need to be considered. One of these is the topic of proximity-aware serving. Since some Edge services require tight latency bounds and a reliable network in order to fulfill their QoS, the mechanics of service placement and load balancing are of high importance. Ultimately, placing services in proximity to their users and correctly assigning users to these service instances are often necessities in order to fulfill latency bounds in the range of a few milliseconds.

Already in 2005, the issue of proximity-aware load balancing in the context of peer-to-peer (P2P) systems was analyzed by Zhu et al. [58]. They proposed data sharing between the peers via an overlay network that has a tree structure and is spanned over the distributed hash table (DHT) of the P2P system. The nodes of this network tree represent virtual servers and are assigned to real nodes of the P2P system. One real node can manage multiple virtual servers at once. Since in this concept, each tree node is responsible for its subtree, each P2P node is responsible for all assigned virtual server nodes and their individual subtrees. Nodes then advertise their hardware load, which can be above or below the average load, upwards the overlay tree. When an over-average and a below-average advertisement meet at a common parent node, the respective parent node becomes a rendezvous point for both servers and takes care to balance the load between these two. By utilizing landmark clustering for Network Distance Prediction and Hilbert Curves to create proximity-aware key mappings, load balancing happens primarily between nodes that are close to each other in the Internet topology. In their evaluation, this made the process of load balancing more regional and therefore more efficient compared to a system without a proximity-presevering key space.

Respecting network distance to optimize serving has also been considered in Edge computing research. For instance, in 2017 Yin et al. proposed *Tentacle*, a decision support framework for service orchestration [59]. By using network distance prediction, they intended to optimize Edge service placements. However, due to the flawed quantitative accuracy of proposed network distance prediction systems like GNP [60],

they do not rely directly on predicted latencies. Instead, they used the resulting distance ranking as a guide since it proves to be more consistent. Clustering Edge users according to their distance ranking using farthest point clustering, their system calculates the ideal service placement location for each cluster and maps these ideal locations to real, feasible placement locations. Using this approach, they could improve the QoE of users significantly in their evaluation.

Following a similar motivation, Goethals et al. create a service scheduling algorithm called *Swirly*, which is suited to manage at least 300,000 devices in real-time [15]. The algorithm optimizes service placement over whole fog topologies, respecting edge-to-fog latency, heterogeneous infrastructure sets as well as service operation costs. Swirly does not predict network distances between nodes, but uses regular pings to measure distance between Edge and Fog nodes accurately for its service placement optimization. According to them, the resulting networking overhead is not "unacceptably high" [15]. On top of that, they aimed to make decisions in real-time even on large fog topologies and constructed their heuristics-based decision-making algorithm with that in mind. Using Swirly, Edge-to-Fog distances could be lowered by 33-55% compared to a random node selection. It should be noted however that Swirly is intended to be used in controlled environments, where Edge nodes are intelligent and aware of Swirly, providing it with positional and latency information. Moreover, the approach has a number of drawbacks. For instance, it does not reassign Edge nodes to other fog nodes in case of resource bottlenecks. Furthermore, the author recognize that the memory consumption of the algorithm depends on the product $|fog\_nodes| * |edge\_nodes|$. Due to its centralized design, this can be a bottleneck in large Edge environments. Therefore, they suggest a distributed approach in which placement decisions are made locally.

2.3.3 Autonomous Clusters

Analyzing related works, the potentially huge volume of Edge environments and the number of entities to orchestrate is an often mentioned challenge, as well as their geographical distribution. Like Goethals et al. mentioned in their works, one possibility to solve scalability issues of orchestration systems is to have a system design that involves no centralized control plane which poses a potential bottleneck. However, most service platforms, rely on a central control plane in order to make decisions based on a global view of the orchestrated system. Casadei et al. describe a decentralized alternative to manage entities in Edge environments. They propose an architectural pattern called *Self-organising Coordinated Regions* (SCR) [61]. It is intended to be reused and adapted in cases where management component of large-scale systems require a scalable distributed decision-making concept. The key idea of SCR is to partition

a set of nodes into independent clusters. Each has their own independent Monitor-Analyse-Plan-Execute (MAPE) loop [62] running which is responsible to manage the cluster. The cluster elects one of its nodes that fulfills this controlling role. A clustering of all nodes is performed regularly in order to account for service topology changes that are typical in Edge environments. The authors of this pattern suggest different implementation variants in order to achieve additional properties. For instance, this pattern could be applied hierarchically, where independent sub-clusters are themselves managed by a leader. A higher-order leader could then operate on information that is aggregated per cluster. This way, global management decisions can be made based on a global, although reduced view to complement local management decisions which were made on a regional, but more accurate view.

Instead of relying on a hierarchy that consists of flexibly formed clusters, one could also transform the proposed SCR pattern to work on an organic tree structure. This reduces the complexity of organizing the nodes, since changes to the tree structure are not globally executed operations. This is a clear advantage over systems where the whole service topology is reclustered regularly. Zavodovski et al. proposed a concept that resembles this idea, called *Intelligent Container Overlays* (ICON) [17]. Inspired by the self-adaptive aspects of the Internet, ICONs intend to solve the common scalability problem of Edge service orchestration by providing a concept for self-organizing containers. In order to achieve a scalability that suits large-scale or even global settings, it works without any centralized control plane. Instead, containers control themselves autonomously using information gathered from a tree-structured overlay network, resulting in a bottom-up service orchestration as described in [61]. This approach is fundamentally different compared to top-down service orchestration by systems like Kubernetes. In ICONs, declarative high-level orchestration objectives like QoS and cost efficiency can be monitored and fulfilled automatically in a regional manner. Such a mechanism effectively replaces the need for detailed, imperative scaling policies and manual micromanagement. Operational data of overlay tree nodes, like capacity information or observed user latency, is aggregated and transmitted upwards the overlay tree. Scaling decisions like replication, migration, or termination are then made locally at any node in the overlay tree, based on the own available information and received information. This can lead to clusters that can have a size from only a few nodes to millions of nodes without bottlenecks, since data overflows are prevented due to the successive aggregation along the tree height. Due to the fact that the orchestration control plane is distributed evenly among all nodes in the overlay tree, the system becomes not only scalable but also resilient. This is a clear advantage over systems that rely on an external controller node like an elected leader for scheduling decisions. The overlay network structure of such a system is determined organically through successive intervention of individual nodes, which may replicate, move or terminate.

Therefore, periodic node clustering and partitioning operations as described by Casadei et al. [61] are not necessary.

2.3.4 Other Related Work

Besides the mentioned topics, there are many more challenges that emerge once the topic is analyzed to a certain depth. For instance, Edge environments can consist out of devices which do not have a common uniform CPU architecture. Some devices might be x86 devices, while others might be based on ARM. In such a situation, it would be convenient to be able to statefully migrate or replicate running containers from one architecture to another during runtime. Barbalace et al. tries to make this possible with *H-Containers*, which enables stateful migration of containers across Instruction Set Architectures (ISA) [63]. The system builds on top of CRIU, which already enables the stateful migration of Docker containers, although only within the same ISA [64]. It requires no source code modifications or changed compiler toolchains, is compatible with Linux software and has only a small additional operational overhead of a few milliseconds compared to original CRIU migrations. They open-sourced and published their work [65].

Another sub-challenge of service orchestration at the Edge is Network Distance Prediction (NDP), which was an integral part of some of the presented related works in this thesis. NDP brings great potential to service orchestration systems, since it can be used to optimize service placement or assignments between user and Edge devices, thereby optimizing QoE. This is especially useful in Edge environments where users of Edge services are unaware of the underlying service platform and can therefore not choose their ideal service instance on their own. Many approaches try to accomplish NPD by modelling the Internet using a coordinate system (GNP [60], DMFSGD [66], NPS [67]) or using path-fitting (iPlane [68], Netvigator [69]). However, as already mentioned, solutions for network distance predictions are usually not reliable [70] [59]. Furthermore, they are often complex and incur a non-negligible operational overhead [70], which might not be ideal for resource-constrained Edge environments. In general, most approaches suffer from inherent problems of the Internet topology [70]. For instance, edges in the Internet graph can violate the triangle inequality principle, since longer network paths are in practice not necessarily slower than shorter ones. Also, edges might have different network delays depending on the direction they are traversed. Furthermore, the dynamism of Internet topology as well as the variations in Internet traffic make it impossible to give delay guarantees. However, as described in Huang et al.'s NPD survey [70], data-driven approaches that utilize techniques like machine learning may improve the situation as they could account better for the irregularities of the Internet [70].

3 VineIO: Self-organizing Platforms

To answer the research questions defined in Chapter 1, the goal of this thesis is to create a service platform. It should be able to operate through the complexities of Edge environments that were described in Section 2.2, and it should improve on the drawbacks and insights of related works which were described in Section 2.3. For this purpose, we developed a concept for self-organizing platforms in Edge environments, which is explained in the first half of this chapter. Additionally, a prototype of this concept called VineIO was developed. It is explained in the second half of this chapter.

3.1 Self-organizing Service Platforms at the Edge

Our vision of self-organizing platforms has several properties. For one, it solves the scalability issue (cf. RQ3) of service clusters [46] by having a control plane that does not pose a bottleneck to the system. Next, the problem of hardware heterogeneity in Edge environments (cf. RQ1) is approached by having a deployment system that is aware of the infrastructure requirements of a service and respects these requirements during scheduling. On top of that, the orchestration system which the self-organizing platforms form takes typical QoS and data locality requirements of services into account (cf. RQ4) [13]. These requirements should also be fulfilled during runtime of the system, even in the light of dynamic Edge environments with varying connectivity qualities, user demand, computing capacities, energy supply as well as infrastructure mobility (cf. RQ2). Therefore, the system implements adaptive mechanisms that control service placement and elasticity [13] [17]. Finally, users of Edge services should not need to be aware of the self-organizing platforms. Instead, the platforms should be able to operate effectively without active cooperation of the users like preemptively sharing latency statistics.

To fulfill these requirements, we build our platforms with a decentralized network organization in mind.

3.1.1 Decentralized Network Organization

Our goal is to have a service platform system that has no centralized control plane, but instead works in a distributed and decentralized manner. This results in a bottom-up

approach to service orchestration which we explained in Section 2.3.3, where nodes in a service cluster manage themselves and their local service neighborhood autonomously. In order to still have an intelligent service orchestration system which does not only operate on a host-local scope, a cooperation mechanism between participating hosts is necessary. The way these hosts cooperate is significant for the effectivity of the system and therefore, the choice of a suitable concept is critical. We decided to use a tree-based approach to create a decentralized overlay network between the platforms, leaning on discussed related work like Zavodovski et al. [17] and Zhu et al. [58].

For our design, we make several assumptions:

- For a user, all instances in a service tree work equally well, as long as their QoE is good enough and the QoS is fulfilled. This means that services instances are stateless or have a common state that is managed outside the orchestrated Edge cluster, for example in the cloud.
- The QoE of a user can be predicted solely by using the distance between a user and the service instance they access, since distance often correlates with latency. Also, shorter network paths leave less room for interference and therefore tend to be more reliable. Notably, this is not always the case [71]. Nevertheless, we make this simplification in the frame of this thesis and consider it good enough to our purposes.
- Consequently, QoE is represented solely by latency, although jitter, network reliability and bandwidth would in practice also play an important role for many use cases.
- Related to the previous assumption, the position of all infrastructure units that are part of the orchestrated Edge environment are required to be known.

With these assumptions and our design goals in mind, we define our decentralized overlay network as follows. At the core of our concept lies the service tree, which is a self-organizing cluster of all service instances of a deployment. Each platform that hosts these service instances runs the software which embodies our concept. A platform can host instances of different services, where each service has its own service tree. Our software then fulfills the responsibilities of a tree node for each of these service trees.

Regular service tree nodes fulfill several functions. They monitor the infrastructure unit on which it operates as well as the managed service instances. Hardware load and user connection metrics are gathered and analyzed. Some of this data is also shared with other nodes. With the locally available information, a node can trigger and execute orchestration actions like user migration, service replications and terminations. Besides that, nodes have mechanisms to maintain and, if necessary, recover their connections

to the service tree. Furthermore, they have APIs to receive administrative commands from service operators.

Naturally, each service tree has a root, which is also represented by a node. Besides the described regular functions, the root node has additional responsibilities. It helps in certain tree management tasks by determining node placements and enables node recoveries in case of failures. Moreover, root nodes act as an access point to users, providing them with addresses to desired services. To fulfill these responsibilities, each root node maintains an index over all nodes in a service tree.

The main purpose of the service tree is to facilitate local orchestration scheduling and execution by monitoring and successive data exchange between tree neighbors. In general, there are two interesting categories of data: Positional information—which can be used for QoS-aware serving—and capacity information which is useful for resource management tasks. Nodes regularly share these kinds of data with their neighborhood, which encompasses their direct parent, sibling, and child nodes in the service tree. Due to this mechanism, each node has access to a local overview of the capacity available in their local region as well as their geographic distribution.

The captured information can be used to maintain the effectiveness of the orchestrated service instances. Each node has their own metrics assessment and scheduling components, which enables them to make orchestration decisions locally and autonomously. As a result, users can be moved to service instances that promise a better QoE for them, or the service topology might scale up or down by adding or removing nodes in the service tree. In some cases, the execution of these decisions does not only involve the node that made the decision, but also other nodes that are nearby in the service tree. This can be necessary in order to propagate orchestration decisions to other nodes, upon which they may react, or to cooperatively help in the execution of some action, for instance by traversing the service tree in order to find a possible user migration target.

In case the service tree grows as a consequence of an orchestration action, new nodes attach themselves to the tree according to the geographic position of the node's underlying infrastructure unit. This leads to a proximity-based grouping of nodes. Consequently, nodes that are geographically near to each other tend to cooperate in the service tree in the course of data exchanging and the execution of orchestration actions. This is beneficial for locality-aware service management and improves the resilience of the system, since cooperational processes that affect a certain region tend to also stay in that region.

Our concept has several advantages. It prevents scalability issues [46] by distributing the control plane among all platforms. At the same time, resource and QoS-aware scheduling is possible through regional cooperation mechanisms. The maintenance of the overlay tree structure has a limited complexity, since its structure is determined

organically when nodes are added or removed according to their position. Also, the lack of controller nodes increases the resilience of the system, since each node is able to operate autonomously on a best-effort basis.

3.1.2 Operation of Self-organizing Platforms

By default, platforms run idle on each infrastructure unit that is part of the system, managing no deployments. Therefore, in order to make use of the platforms, service operator need to have an interface over which they can issue new deployments as well as manage existing deployments. Therefore, each platform provides an appropriate API to service operators. They can create a new deployment on a specific platform by sending a deployment specification. After a successful launch of a deployment, the system returns relevant information like addresses and ports of the created service instances. In case the specification indicates that the deployment marks the root of a service tree, a new service tree is created, consisting only of the root node which is run at the chosen infrastructure unit. However, administrators can also create deployments that belong to an existing service tree, effectively creating a new replica manually. In that case, the newly launched service instance will contact the specified root node and registers for the tree. The root node will find a parent for the new replica that is geographically nearby and assign it. The new replica will then attach itself to the parent directly, thereby becoming part of the service tree.

Besides the creation of new deployments, existing deployments can be removed from an infrastructure unit via the same API. For this, a service operator needs to issue the respective command to the system, providing it with the deployment ID that was part of the initial deployment specification.

Monitoring

A main purpose of an orchestration system is to maintain the health and effectiveness of the managed deployments. In order to accomplish this, it must collect data about its environment, including the deployments, their underlying platform and infrastructure, and their users. In the concept of a "Monitor, Analyze, Plan, Execute" (MAPE) control loop [62], which is also the general model of service orchestration, monitoring represents the first part of the loop. It enables the intelligence which is needed to fulfill operational requirements. Based on that, the system can determine when interventions are necessary to fulfill these goals.

For an Edge-aware orchestration system that fulfills our specified goals, there are two important aspects to monitor. One of them is the quality of service operation. By determining it, the system can determine whether the infrastructure unit on which a

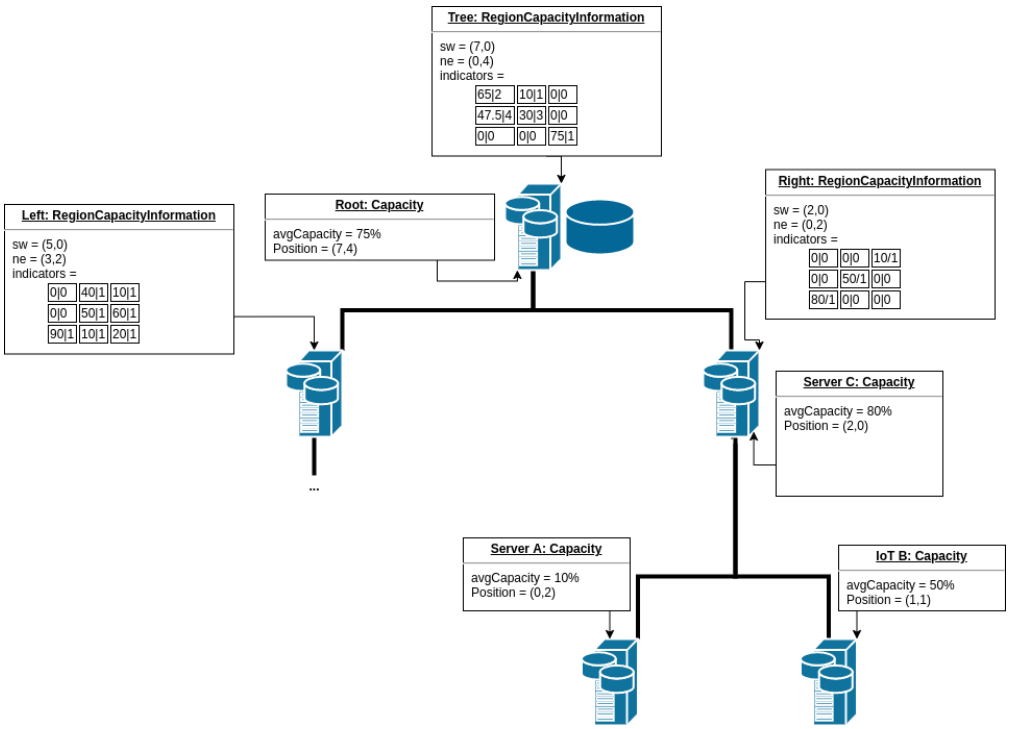

Figure 3.1: Network flow visualization of successive data aggregation
(Cell format: *$Capacity* | *$NumberOfNodes*)

platform runs has enough computational resources in order to operate all deployed service instances normally. The quality of service operation can be inferred by monitoring certain hardware capacity metrics, like CPU, RAM, and storage load. In case the infrastructure unit has additional hardware components like a dedicated GPU, they are also object of monitoring. The other aspect is quality of user experience (QoE). For QoS-aware service operation, it is important to know whether a service is able to serve its users well enough to fulfill the original QoS-requirements of this service. Since this depends on the QoE of the respective users, we can measure factors that influence it, like network latency, bandwidth, jitter, and packet loss. According to our assumptions that we specified in Section 3.1.1, we restrict our monitoring to the latency of a user connection. A node extracts latency statistics from its user connections constantly and frequently, since significant changes in the measured values can happen at any time.

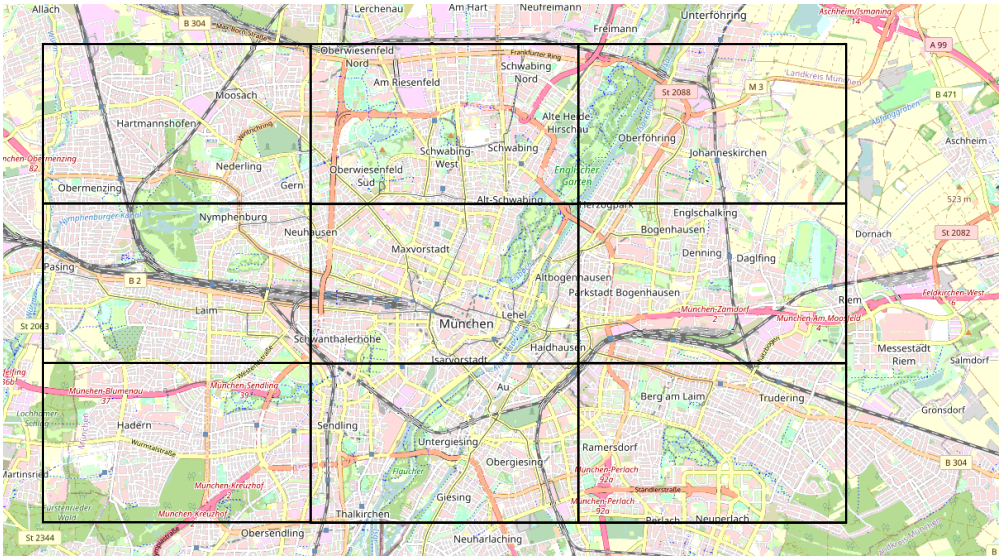

Figure 3.2: Graphical representation of a capacity grid that spans over Munich

Data Exchange

As already described, nodes assign some of their monitored capacity metrics with positional information and propagate them in order to facilitate service orchestration tasks. A recipient of these data packets would then geographically localize free capacity, which is essential for QoS-aware service placement and load balancing. The goal is to provide enough information to each node so it can operate effectively, while not overloading the node itself or the network due to large data volumes. Therefore, we leverage the tree structure of our Edge service clusters and transmit data only to its parent and children. However, since the direct neighborhood of a node consists of only a few other nodes, the view of each node of the service cluster would be greatly restricted, covering only the capacity and positions of its direct neighbors. Therefore, to enhance the effectiveness of data sharing without significantly increasing the network overhead and resource consumption of service tree nodes, nodes enrich the shared data. Precisely, each node receives an aggregated view of the capacity of the subtree of each child, alongside the region each subtree spans. These views are called capacity grids. A node then further enriches these capacity grids by merging them together and inserting its own positional and capacity information, essentially creating a new aggregated capacity grid. After this aggregation process, capacity grids are further propagated to parent nodes, where the same process happens again along the service tree height, until the grids arrive at the root node. This can be seen in Figure 3.1. Besides

aggregating capacity grids and propagating the result upwards, each node shares its received capacity grids to its children, so each child node has the capacity information of its siblings at hand. This information about siblings is not further propagated.

Each capacity grid consists of a matrix with $n * n$ cells and associated metadata that indicates the rectangular geographic area that this matrix spans. This can be seen visually in Figure 3.2, where a $3 * 3$ sized capacity grid is depicted. The precise value of n can be an arbitrarily chosen natural number. Notably, a greater n increases the effectiveness of the aggregation algorithm, but also increases the resource consumption. Therefore, a sane value needs to be chosen according to the properties of the relevant Edge environment. Each cell in the matrix represents its own sub-region, which is $1/n^2$ of the geographic size of the enclosing capacity grid. The content of each cell is an aggregated capacity indicator for each relevant computing resource type that is available in the region of the cell. For instance, the average available RAM that is available in the cell is part indicator, but also the average free GPU capacity. Therefore, each cell indicates the aggregated available capacity that is available in the respective sub-region, together with the number of nodes that are incorporated in the capacity metrics. Such a capacity grid could be visualized as a heatmap.

The aggregation algorithm that produces these capacity grids needs to be able to merge multiple input capacity grids. It also needs to handle singular capacity indicators, since some nodes in the service tree do not have children. Also, the position and capacity information of the current node need to be inserted and aggregated. Our algorithm works as follows: First, it needs to convert the geographic coordinates of input grids and single capacities from spherical coordinates—i.e. in the (lat,lon) format—to planar coordinates that can be interpreted as (x,y) coordinates on a flat surface. This step is necessary because otherwise, the spherical shape of Earth would distort the proportions of different regions, making the aggregation of them more complicated. Once this is done, the algorithm calculates a common cell denominator and uses it to scale up the grids so that the grid cells among all grids span a geographical region of the same size. This usually results in a growing number of matrix cells of each grid. Afterwards, a new, empty matrix with the same cell size is created. It spans all input grids and has exactly the size needed to receive all input capacity matrices and single capacities and align them according to their coordinates. In a following step, the upscaled input matrices are inserted accordingly into the new matrix. Additionally, all single capacity indicators are inserted into the new matrix according to their planar coordinates, including the capacity of the current infrastructure unit and its position. Once this process is done, the result is scaled down to the original $n * n$ aspect ratio. During downscaling, cells of the matrix need to be collapsed and merged by aggregating the capacity information they contain, respecting the number of nodes of each input cell proportionally. The result is a new $n * n$ matrix that contains the aggregated capacity of all input regions

and positions, together with their new node count. The geographic region of the new matrix is a rectangle that encompasses all input regions. For further reference, please refer to the precise implementation of this algorithm in the code repository of VineIO[1].

Due to this mechanism, each node has access to an overview of the capacity available in regions within its subtree and also around its subtree.

Scaling Actions

According to our service operation goals, we need to intervene when:

1. The quality of service operation is not good enough anymore.
2. The quality of user experience is not good enough anymore.

Therefore, as part of the Analysis phase in the MAPE control loop [62] of our orchestration system, monitored data is not only shared with neighbor nodes, but also fed into an assessment component. There, the data is searched for anomalies and, if there are any, their properties are determined. Based on this assessment, appropriate actions to normalize the system state are then executed. A map of anomalies and how to handle them is provided in Table 3.1.

To analyze the quality of service operation, we need to process the demand rate of deployed services on the platform as well as current hardware statistics. In case a service was not accessed by any user for a certain period of time, it is terminated. In case a high system load is noticed, i.e. metrics like CPU load or RAM consumption exceed a certain defined threshold value, the load on the infrastructure unit needs to be reduced. To achieve this, we increase the available amount of capacity at the current geographic position by creating a replica at an infrastructure unit that is as near to the current one as possible. Afterwards, we offload users to the newly created replica successively cluster-wise until the hardware statistics are in a normal level again, not

[1] `https://gitlab.lrz.de/cm/2020-ralf-masterthesis/-/blob/master/Code/sop_orchestrator/sop_orchestrator/overlay/treedata.py`

Table 3.1: Scaling actions and their cause

Anomaly	Scaling action
Resource shortage	Replication and user migration, offloading user clusters to a new replica until the shortage is mitigated
Bad QoE	User cluster migration to a potentially better replica, or creation of a new replica if no suitable existing alternative was found
No user demand	Termination of the affected service instance

exceeding any configured thresholds. At first, users where a bad QoE was already observed are offloaded. Only if there are not any users where this applies or if this measure turns out to be insufficient, we determine the users to migrate arbitrarily.

The assessment of users' QoE is done using the measured latency of all user connections to orchestrated services. Notably, dealing with every user separately would increase the operational overhead of QoE maintenance significantly. Therefore, to make the process of user migrations more efficient, the system keeps track of users that currently have a bad QoE by clustering them according to their position. So, in case the median latency of a user in a moving time window is above the specified upper delay limit of a deployment, it is added to a suitable cluster that is in proximity. If no cluster exists yet or if all existing clusters are too far away from the user, a new cluster is created and initialized with the respective user. For each cluster, an overall QoE score is maintained together with an average position of all users in a cluster. Once the QoE score surpasses a certain predefined threshold, the process of user migration is started for the respective cluster. Should this process fail, e.g. because no suitable migration target was found in the service tree, a replication is initiated. The replica is spawned as near to the average position of the user cluster as possible in order to fulfill the QoS for these users more effectively. Then, the user cluster is migrated to this new replica.

In total, there are three possible intervention methods in order to normalize the state again:

1. Migrate users to another service instance in order to improve their QoE or to free computing capacities on their current service instance via offloading.

2. Replicate a service instance to a new infrastructure unit in order to create a possible migration target.

3. Terminate a service instance in order to save operational costs in case of longer idling.

Our method for user migrations is based on the assumption that nodes that are physically close to each other are also closely located in the service tree. As already explained in Section 3.1.2, each node has access to capacity grids of its children and its siblings, on top of the own monitored and positional data. By analyzing this dataset, a node can determine whether it makes sense to trigger a migration process. If a child node looks like a promising target due to the distance between its region and the cluster position, and enough capacity appears to be available, a migration request is sent to this child. If sibling node or the parent node appear to be promising for the same reasons, the node sends a migration request to the parent. The parent then chooses itself as a migration target or relays the request to the best-fitting sibling node. In case all siblings do not seem ideal from the gathered data, the request is forwarded further

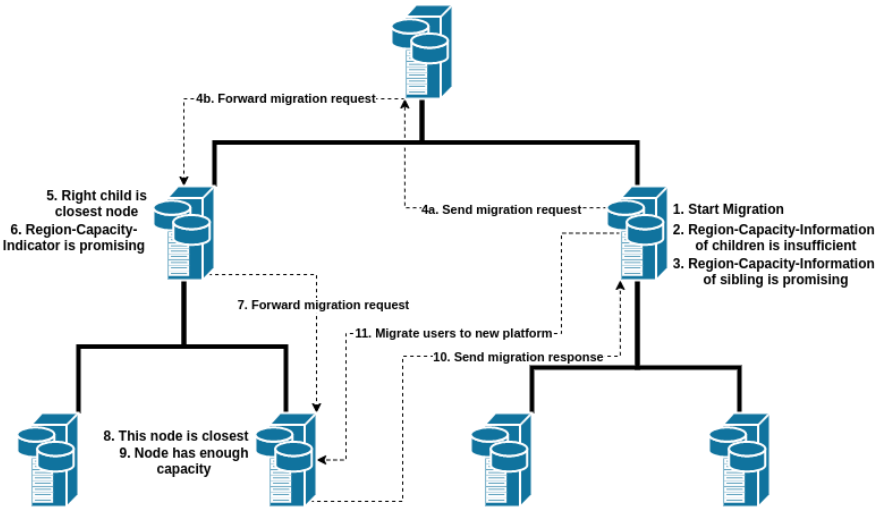

Figure 3.3: Network flow visualization of a user migration procedure

upwards the service tree to its own parent, which is the grandparent of the request origin. The step of checking itself for suitability and forwarding the request is repeated successively until a suitable node is found or no better alternative exists, in which case the process of target finding aborts. If a suitable host is found during the process, this host will notify the origin of the migration request about this fact, providing its address and position. The requesting node will then migrate its users over by sending a user redirection request to the root node. The root node will make a note of the successful user migration and will provide the new service instance address to these users. In case no suitable host is found within a specific time window, the migration process aborts without solving the anomaly. Notably, we do not allow for backtracking during the search for a migration target. Therefore, a migration process is always guaranteed to end, since no circles occur during the traversal of the tree. An example process of a user migration is visualized in Figure 3.3.

In case a migration request fails without solving the anomaly, an alternative measure is necessary. For instance, a new migration target could be created via replication. In general, a replica can be created in proximity to the current position in order to offload an existing node, or it can be at a specific position in order to serve a specific set of users that experiences a bad QoE, like in the case of a failed migration target search. For either case, we need to find a suitable infrastructure unit and replicate our service instance to it. This is done by querying the infrastructure database of our orchestration system. Its entries are filtered according to a candidate's distance to the desired position and whether it is able to fulfill the computational requirements of the

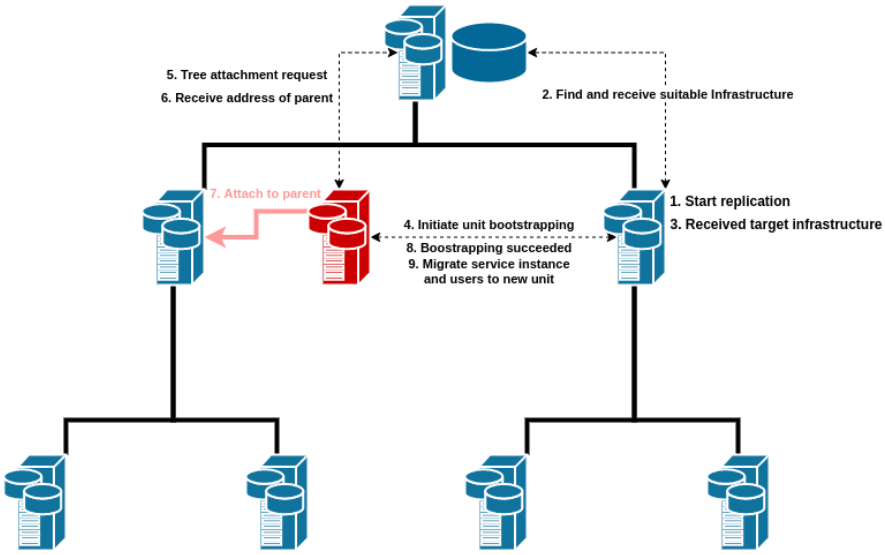

Figure 3.4: Network flow visualization of a replication procedure

respective service. Then, a set of possible nodes, sorted by suitability is returned. The replicating node is now trying to replicate its service instance to the best possible target node. In case the process fails, the next best target is tried until replication succeeded. In case a potential replication target successfully launches the respective deployment, it behaves like an administrator had issued a new deployment on it. Precisely, it means that it registers itself at the root of the service tree, from which it receives the address of its assigned parent node. The new replica then attaches to the parent. Once the replica is up and running, the users that were intended to be offloaded are redirected to it. Figure 3.4 visualizes the whole process of replication.

To terminate a service instance, the respective deployment is simply removed from the infrastructure unit and its monitoring hooks as well as its cooperative service tree functions are shut down. Furthermore, it signs off the service tree in order to prevent users from being assigned to a service instance that is not alive anymore. During regular operation of the system, neighbor nodes notice the absence of the terminated node and remove its ties to it. In case a child node becomes an orphan during the course of the termination of its parent, it notifies the root node, which assigns a new parent to the orphan.

Load Balancing

As already mentioned in Section 3.1.1, the root node of each service tree has additional responsibilities. One of these is offering a service discovery mechanism to users, which can query the address of a desired service for example via a DNS frontend of the root node. Once such a query arrives, the root node needs to find a suitable service instance which is able to provide a sufficient QoE to the user according to our defined goals. In the root node's index of the service tree, each node is stored alongside its geographic position. Using the index and a Geo-IP translation mechanism, the root node is then able to find the service tree instance which is physically closest to the user's position. By extension, it is likely that the returned closest node is also the one which can offer the best latency and therefore QoE to the user, according to our initial assumptions. If this is not the case, the user's QoE can be improved afterwards via migration. The root node keeps track of all user migrations and will return the service address of the saved migration target in favor of the address that would be inferred by considering the geographic position of the user.

Failures

Since this is a distributed application, failures are essential to consider. In our concept, we assume that omission failures, i.e. temporary or permanently failing network communication and nodes, and commission failures, i.e. off-protocol behaving nodes, can happen at any point in time. To address these potential issues, our concept needs measures that prevent inconsistencies in the service tree and help in recovering failure states. We must assume that the network as well as message processing of nodes is not reliable and messages in the overlay network can happen to get lost, to arrive more than once, or to arrive at an unexpected time. Therefore, when nodes send messages in the service tree, they assign these messages with an identifier. This identifier is used by recipients to drop duplicate messages. As a result, messages are idempotent. To compensate for lost messages, e.g. due to buffer overflow, networking or system issues, nodes are following a request-response communication pattern. When no response to a request was received after a certain amount of time, connections are reset in order to solve eventual connection state issues. Then, requests are resent. On top of handling erratic communication patterns, we also need to consider the case where nodes in a service tree crash due to runtime software errors, hardware failures or network errors that result in node partitioning. To detect incidents where a node becomes unavailable, direct vertical neighbor nodes—i.e. parent and children—send each other a health-check message back and forth in regular time intervals. In case a neighbor node does not respond in a reasonable time window, even after retries, the pinging node can assume

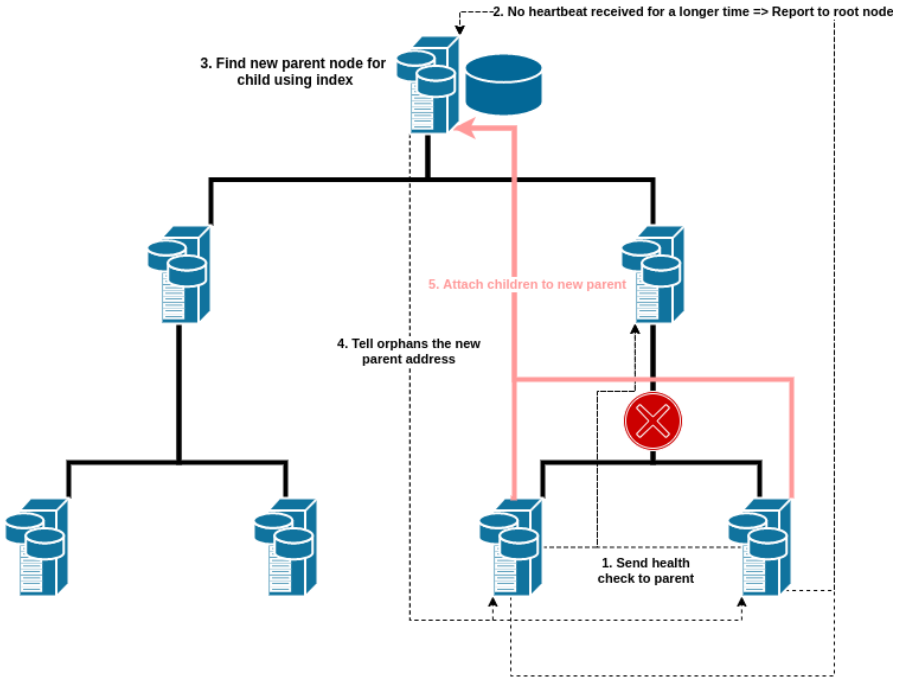

Figure 3.5: Network flow visualization of a failure handling procedure of orphaned nodes

that the neighbor is unavailable. In case the neighbor node was a direct child node, the parent abandons it by removing it from its neighborhood set, thereby stopping any successive communication with the unavailable child. In the opposite case where a child node detects that its parent is unavailable, the child needs additional recovery measures since its subtree is now effectively detached from the service tree. By reporting the unavailable parent node to the root node of the service tree, the root node can remove the failed node from the service tree topology and assign an alternative parent node to the orphaned subtree. The orphaned node then receives information about the new parent node and attaches to it. Now, the service tree has recovered from the node failure and normal operation can continue. In case a previously failed parent node becomes alive again, it recognizes the change in the topology configuration through the neighbor nodes which do not expect communication with the former neighbor node anymore. In order to operate normally again, the recovered node will contact the root node to reattach itself to the service tree again, now becoming a leaf node.

3.2 Design and Implementation of VineIO

With the described concept in mind, a Proof-of-Concept system was implemented in order to evaluate it. We called it VineIO: A container sidecar for realizing self-organizing platforms in Edge environments.

3.2.1 Implementation Goals

To fulfill our concept accurately, we needed to implement the system with several requirements in mind:

- *Portability:* Since we want to have an orchestration system that works in heterogeneous Edge environments, the implementation needs to work on most common hardware platforms. Precisely, the implementation needs to support multiple CPU ISAs, especially the popular x86 and ARM architectures. This requires the usage of a technology stack that runs on these architectures.
- *Efficiency:* VineIO needs to prevent unnecessary operational overheads in order to execute sufficiently well on smaller devices. Again, the chosen technology stack must not pose an obstacle to this requirement. Furthermore, a certain implementation efficiency needs to be achieved by good algorithmic design choices. Most notably, the implementation must not negate scalability benefits of the decentralized design.
- *Practicality:* Although VineIO is intended to be seen as Proof-of-Concept, the prototype must be able to fulfill all discussed mechanics of our concept. The prototype must be usable in practice to the full extent of our concept.
- *Modularity:* As explained in Chapter 2, the topic of service organization in Edge environments consists of several parts that have their own challenges. Therefore, the prototype must be constructed in a way that makes exchanging modules and components easy. For instance, different platform technologies and different metric assessment routines could be required for future research and therefore, the respective components should be exchangeable.

However, we also restricted the implementation scope in the following ways:

- *No feature-completeness*: As already described, the main purpose of VineIO is to evaluate the concept of self-organizing platforms in Edge environments, which is why we did not implement features that may be present in products from the industry, but do not influence the result of the evaluation. For instance, the prototype of VineIO supports only one platform backend, although it is designed in a way that makes adding more backends easy.

- *Simple scheduling routines*: The focus of this thesis lies more on the distributed nature and the Edge-awareness of the system than on optimizing the process of decision-making. Therefore, we use simple heuristics to gauge the state of the service cluster and have a reactive scheduler instead of more complex approaches which for instance might involve Machine Learning to enable proactive scheduling.

3.2.2 Technology Stack

In accordance to our implementational and conceptual requirements, the following technology stack was chosen to implement VineIO. We chose Python [72] as our programming language. It is one of the most popular languages [73] with a rich and versatile standard library, which fosters quick prototyping processes. Furthermore, Python code is highly portable. Its main interpreter, CPython [74], supports all relevant hardware platforms. Therefore, Python code is portable and suitable for heterogeneous Edge environments. However, Python has some notable properties that can pose an obstacle. Due to the Global Interpreter Lock (GIL) [75], multithreaded python applications tend to be inefficient. Also, as an interpreted language, the focus of the Python language is not the efficiency of its programs. However, we circumvented these issues by using Pythons multiprocessing capabilities in order to parallelize computation-bound tasks, while using its structural concurrency features for network-bound tasks.

For all communication tasks, we decided to use the message passing system ZeroMQ [76]. It is a mature technology that has well-supported libraries and is used by several big technology companies. Furthermore, it has several advantages for our use case. It is lightweight and suited to be used in embedded devices, which is beneficial in Edge environments. Moreover, it is advertised as "fast enough to be the fabric for clustered products" [76], which matches our use case perfectly. Furthermore, it can not only be used as a networking library, but also as a concurrency library, supporting several Interprocess Communication (IPC) modes. The set of offered features integrates well with VineIOs needs. There, it is used for IPC between subprocesses as well as TCP-based service tree communication. Since ZeroMQ does not offer means to serialize messages, we need to refer to dedicated serialization libraries for this purpose. We choose MsgPack [77], due to its efficiency and establishedness.

Containers are the de-facto standard cloud service deployment format as described in Chapter 2 and require no further introduction. Their ecosystem, ease of integration and light weight while enabling application portability benefits makes them an attractive choice for Edge services platforms. Therefore, we use OCI containers [23] in our platform layer for orchestrated services. Between the several alternative container runtimes, we choose the Docker Engine [78]. It supports multiple platforms, including

x86 and ARM [79]. Furthermore, its images can also support multiple ISAs, which means Docker can be used for heterogeneous environments [63]. Additionally, since it is an established technology that is commonly used during software development and operation, we expect it to be reliable enough for our purposes.

Due to the relevance of QoS-awareness in our concept, the chosen technology for monitoring user connections is significant for the operation of VineIO. To make the process of extracting latency information from connections efficient and non-intrusive while keeping it portable, we decided to use native features of the Linux kernel to fulfill this task. Precisely, we make use of eBPF [80], which is an extension of a kernel component called Berkeley Packet Filter. It enables users to inject JIT-compiled bytecode that was verified for crash-safeness onto a sandboxed platform that runs in the kernel space. From there, the code can access a wide range of internal kernel resources without significantly disturbing their operation. One of these resources is the networking stack of Linux. We leverage this technology by injecting code which accurately reports the latency of specific user connections to VineIO. To compile our eBPF code which is written in C, and inject and interact with it during runtime, we make use of the BPF Compiler Collection [81].

Besides these main pillars of VineIO, we utilize several other technologies for implementation. For instance, we use SQLite [82] as an in-memory database to store topology information in VineIO, together with its Spatialite extension [83] to support geographical queries. Also, NumPy [84] is an integral part of our system. We use it to calculate certain key values throughout the orchestrator, and it plays an important role in our capacity grid aggregation algorithm. Apart from the mentioned technologies, there are several others our system depends on. An exhaustive list can be found in the project's repository[2].

3.2.3 Architecture

Figure 3.6 visualizes the high-level system architecture of VineIO. In essence, there are three core functions that VineIO needs to implement: Monitoring, networking, and platform interactions. Therefore, we split our system accordingly, having three main modules called Monitoring, Action, and Overlay.

The Monitoring module runs a set of metrics collectors, assesses the gathered information and triggers orchestration actions at the Action module, if necessary. Furthermore, it provides an interface to share system state information to other modules.

The Action module acts as an abstraction of the platform layer, which consists of the Docker Engine, and encapsulates functions that execute and persist orchestration

[2] `https://gitlab.lrz.de/cm/2020-ralf-masterthesis/-/blob/master/Code/sop_orchestrator/pyproject.toml`

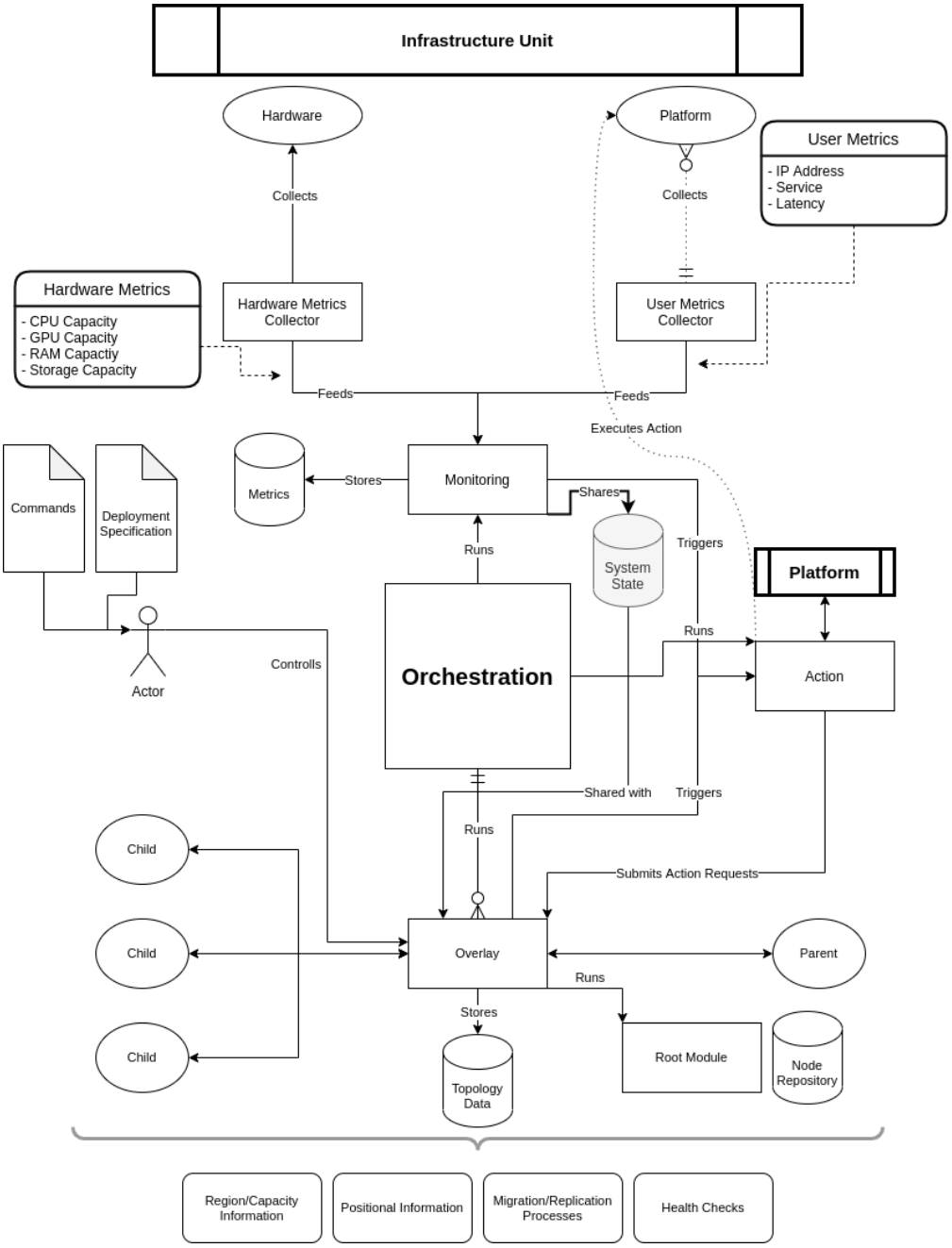

Figure 3.6: High-level Architecture of VineIO

actions. For instance, the logic of launching new deployments or the removal of existing ones is contained in this module. To fulfill its responsibilities, it communicates with the Monitoring module to create and remove monitoring hooks and with the Overlay module to propagate certain actions to other service tree nodes.

The Overlay module encompasses all networking functions and is responsible for maintaining the role of a node in the service tree. It receives several kinds of messages that contain data or commands and handles them accordingly. In some cases, it forwards commands to the Action module, e.g. in order to issue a new deployment. Also, in order to fulfill its data aggregation capabilities, it fetches hardware statistics from the Monitoring module.

Apart from these three main modules, there is also the Root module. It is only active in cases where service tree root nodes are maintained by the current platform. In that case, each service tree has its own independent instance of the Root module. The purpose of this module is to fulfill all root node functions that were described in Section 3.1. This means that the Root module assists a service tree by keeping book about all nodes in a service tree and providing structure-related management functions. It also represents an entry point to users, providing them with addresses to suitable service instances.

By applying the Ports & Adapters architectural pattern, also known as Hexagonal Architecture [85], we properly isolate these modules from each other and introduce well-defined interfaces in order to facilitate communication between them. This enables each module to run in its own subprocess, which can run in parallel without friction. While the three main processes manage a whole infrastructure unit and therefore maintain all services and their service tree functionality at once, the Root module is spawned per root node as explained.

There are additional minor but mentionable components that provides assistance functions to the rest of the program. For instance, VineIO also has access to an infrastructure database, which contains all VineIO platforms and is used during replication actions. Also, a Geo-IP-database is available which is able to translate IP addresses into geographic coordinates.

3.2.4 Implementation

The following section describes the individual components of the architecture in more detail, starting with the Monitoring module.

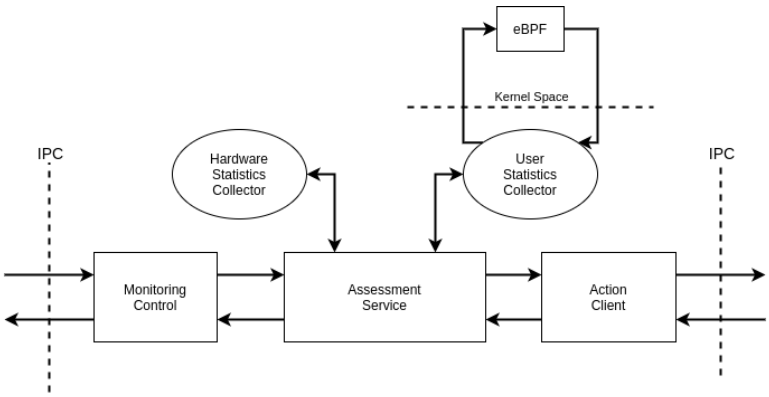

Figure 3.7: Architecture of the Monitoring module

Monitoring Module

The overall structure of the Monitoring module can be seen in Figure 3.7. The main input of the module comes from a set of so-called Collectors. These are the routines that fulfill the Monitoring phase of the MAPE control loop [62] of VineIO and are responsible for getting metrics from the infrastructure unit. There are two Collectors: One is responsible for hardware-related metrics, the other is responsible for user-related metrics. Hardware statistics are collected using standard operating system APIs and include the metrics CPU load, memory consumption, GPU load—if available—and storage occupation. To fetch these metrics, the Python libraries *psutil* [86] and *GPUtil* [87] are used. Notably, only Nvidia GPUs are recognized by the system. The Collector for user metrics consists of an eBPF backend code, which is injected into the TCP/IP stack of the Linux kernel and extracts the latency values of users which are specified by their IP address. The extracted values are stored in a ring buffer and can be fetched by the Python frontend of the Collector from kernel space into user space. Each Collector runs in its own co-routine and buffers its collected metrics, where they can be accessed by the Assessment Service. The Assessment Service implements the Analysis and Planning phase of VineIO's MAPE control loop [62] and checks in regular time intervals whether the system is in a normal state, as described in Section 3.1.2. In case an anomaly regarding the computational load or the user latencies was detected, it triggers and controls appropriate interventions. This is done via the Action Client, which acts as a Port [85] to the Action module and translates the commands into the respective messages to the Action process.

Apart from the data processing and interpretation tasks, the Monitoring module also provides a controlling interface to other components. For the Action module, this interface provides function to add or remove deployments from the monitoring list.

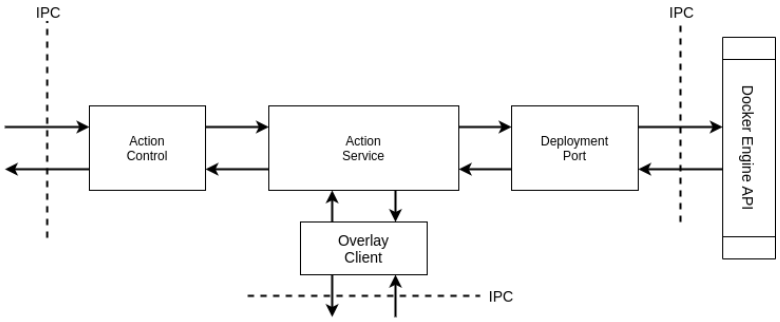

Figure 3.8: Architecture of the Action module

For the Overlay module, it provides access to the current hardware state.

Action Module

The Action module is able to receive action requests from other modules over its controller interface, which is located on the left in Figure 3.8. Such requests could be the deployment of a new service, with necessary modalities provided by a deployment specification. It could also be the termination and removal of a service from a given deployment name. In either case, the respective operations of accepted requests are forwarded to the Action Service, which takes care of executing them. For this, it uses the Deployment Port, which offers low-level platform layer operations to the Action module. In the current implementation of VineIO, the Python library *docker-py* [88] is used within the Deployment Port as a client for the Docker Engine API. It translates deployment-related actions into their respective container operations. For instance, when a new deployment is requested, the platform layer translates the request into a container creation procedure.

However, for action requests like user migrations or service replication, which both directly depend on the service tree for their correct execution, it uses its Overlay Client to hand over the distributed parts of an action to the Overlay process.

Overlay Module

The Overlay process encapsulates all service tree related functions, using a module structure which is visualized in Figure 3.9. Input in the form of service tree messages arrives at the Orchestration Controller, as well as migration and replication requests from the Action module. Also, administrational message from system operators are received by the Orchestration Controller. The controller is separated into two sub controllers: One for external TCP messages from the service tree and one for internal

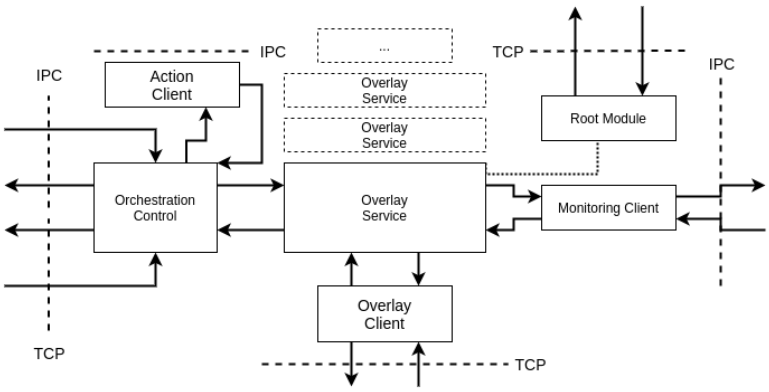

Figure 3.9: Architecture of the Overlay module

IPC messages. This design was implemented due to security reasons: There must be no way to use functions that were intended only for internal use from the outside of the system.

The Orchestration Controller manages a set of Overlay Services, which implement all tree related responsibilities of a service node. These are created by the controller when a new deployment or replica is created on the managed platform. Therefore, each deployment on the platform has its own Overlay Service, as it has its own service tree. The controller also shuts down Overlay Services in case of a service termination. Most service tree messages, like data exchange messages, that arrive at the controller are tied to a specific service tree. In many of these cases, the messages are simply forwarded to the corresponding Overlay Service, which handles these accordingly.

An Overlay Service has several responsibilities which are fulfilled in a concurrent manner. For one, it maintains all networking functions of a service tree node. To do this, it manages its view on its neighborhood with consists of children, parent, and sibling nodes. As discussed in our failure concept in Section 3.1.2, it performs maintenance tasks like health checks regularly. Also, it makes sure it is connected to the service tree properly and—if it is not—reattaches itself to the tree with the help of the root node. Besides maintenance, it handles all data exchanges messages that were received for the deployment it manages. Although it can receive and save data information at any time, it processes them not on demand, but in periodic time intervals. The respective loop then retrieves all gathered information and executes the data aggregation and propagation procedure which was described in detail in Section 3.1.2. Besides maintenance and data-related messages, there are also messages that are related to orchestration actions. Most notably, user migration messages need special handling. According to the information provided in the migration request and

the state and settings of the current platform, user migration requests are answered or forwarded as described in Section 3.1.2.

To communicate with its service tree neighborhood and the root node, each Overlay Service uses an own Overlay Client which manages respective sockets to children, parent, and root nodes. It is then used to e.g. propagate data, forward migration requests and communicate, send health checks and communicate with the root node.

Root Module

As already mentioned, the Root module runs as a subprocess of an Overlay Service in case the platform represents the root node of the respective service tree. However, it is not only accessible by this particular Overlay Service instance, but to the whole service tree. The Root Controller listens to the public network for any requests that are related to the service topology, or to user requests for an address of a service.

These are then issued on the Topology Repository of the Root module, which runs an SQLite in-memory database backend containing all service tree nodes. It used the Spatialite extension as described in Section 3.2.2 in order to efficiently serve queries that involve geospatial operations, like sorting service tree nodes according to their distance to a certain coordinate. Furthermore, successful migrations are stored in the database in order to keep track of which service is the current ideal one for each user. In general, the Topology Repository implements all functions that were discussed in Section 3.1.2. It can attach and remove nodes to and from the service tree, find the ideal parent for an orphaned or newly created service instance and, as mentioned, assign users to their ideal service instance. It also assists in replication actions by providing information about which infrastructure units are already in use by the service tree.

Other Components

Besides the mentioned modules, there are also some assistance utilities that are shared between the modules and used for certain tasks.

For instance, the infrastructure database adapter offers access to a database which contains all infrastructure units that can be potentially used as a platform in the current Edge environment. Originally, the infrastructure database was intended to be run externally. For the sake of prototyping however, we built an in-memory SQLite database which is extended by Spatialite behind the adapter. On system startup, it fills the database from the configuration file of VineIO, which contains a list of all infrastructure units and their properties. Consequently, each platform has its own copy of the infrastructure database locally available, which is for instance used during replication to filter and sort suitable targets according to hardware properties and

position. Since it is assumed to be static and is therefore only used in a read-only manner, there are no consistency issues between the platforms as long as they are configured properly.

Furthermore, VineIO has a read-only in-memory Geo-IP database adapter, which is used in several places to translate IP addresses into geographic coordinates. Behind the adapter lies the embedded GeoLite2 dataset from MaxMind which contains positional information for IP subsets. Users that have a bad QoE can be clustered according to their position by translating their IP addresses to positions, and users can receive a service instance that is in proximity to them by involving this database.

Due to the use of the Adapter pattern, each of these databases can be exchanged behind the adapter conveniently. This is useful to enable future use cases, but it also enhances the flexibility of the system greatly. For instance, the GeoLite2 dataset is based on the public Internet infrastructure. This is useful for real world usage, but can be a problem in setups that involve private networks. Therefore, VineIO provides an alternative implementation of the adapter, which can read a Geo-IP dataset from a CSV file and feed it into an in-memory, Spatialite-enhanced SQLite database.

3.2.5 Running and Interacting with the System

In order to use VineIO, we need to provide a configuration which specifies the behavior of the orchestrator, provides the orchestrator with essential data for operation, and helps the orchestrator to integrate with its infrastructure unit as well as Edge environment. This configuration can be passed in form of a YAML file [89]. VineIO needs to know several properties of the underlying hardware. Besides basic hardware capacity specification, like available RAM, it needs to know about the public address via which the orchestrator is available. The address can then be propagated accordingly to other nodes in the service tree as well as users. Furthermore, positional information is required so all geospatial operations of VineIO work correctly. On top of the own hardware information, it also needs a way to discover the infrastructure of the Edge environment. Currently, all available infrastructure units are listed in the configuration file, with all properties that are also given for the own infrastructure unit. With information about the underlying hardware available, the Monitoring part of the system needs to know which hardware load levels can be seen as critical for the service operation. Therefore, a section of the configuration file is dedicated towards operational policies. There, upper bounds for relative resource consumption are defined for all relevant components, like RAM and CPU. Besides these, there are options to configure a name for the platform and to specify the user connection tracking mode, which might need to be changed from the default depending on the configuration of the operating system.

When configured properly, VineIO is ready to be launched. We offer two ways of running the application. The first option is to run VineIO in a container. For this, we provide a Dockerfile which can be used to build an image that is suited for demo purposes. It contains all necessary dependencies and requires a few mounts in order to work. For instance, the configuration file that should be used by the container should be mounted. Furthermore, the Docker Engine socket of the host system needs to be offered to the VineIO container, so VineIO can deploy its orchestrated services on the host machine. Moreover, the container needs to be launched with the *–privileged* flag, since we need access to the kernel space in order to track user connections. The second option to run VineIO is to install it as a system package. This option requires that all system dependencies of VineIO, like SQLite and others listed in Section 3.2.2, need to be present on the operating system beforehand. If this is the case, VineIO can be installed into a virtual Python environment via its build and dependency management tool Poetry [90]. In the process of doing so, all Python libraries that we leverage will be installed. When this is accomplished, VineIO can be run within the virtual environment by launching its module, e.g. via *python3 -m sop_orchestrator*. Note that the procedure of installing VineIO can vary between different operating systems. The recommended way to try out VineIO is the container-based method.

A full reference of the available configuration options and their possible values, as well as instructions to install and run VineIO can be found in the repository of this thesis[3].

Administrators need a method to interact with a running VineIO system in order to issue a new deployment, remove a deployment, or view hardware statistics of a platform. VineIO offers a suitable API for these tasks which is implemented on top of the VineIO communication protocol. To be able to use this interface, we implemented a rudimentary control library which implements the protocol using ZeroMQ and MsgPack. On top of this library, we also implemented a command line interface (CLI). Using this CLI, administrators can fulfill all the tasks mentioned above. Additionally, the CLI is also able to communicate with the root node of a service tree and can be used to query service instances for a given user address. Due to the fact that user interface and communication logic is separated, it is easy to create other control applications on top of the library, which may e.g. offer a graphical user interface (GUI).

In the current state of the implementation, a smart user is needed to effectively make use of the system by fetching the ideal service instance from the root node. Ideally, this process would happen via DNS and would not require extra logic on the user other than the logic to consume the desired service. To still introduce this hidden user interaction with the root node during testing and evaluations, a simulator for a smart user was

[3] https://gitlab.lrz.de/cm/2020-ralf-masterthesis

created in the course of the thesis. It is written in Python and is based on top of the mentioned control library in order to fetch service instance addresses. Consequently, it is able to interact with orchestrated services that were correctly assigned by a root node. To simulate a realistic user, it regularly queries the intended service and accesses the received instance over HTTP in poisson-modelled time intervals [91]. It offers command line options to tweak the user behavior and specify a root node address, a spoofed user address and average waiting times. A Dockerfile was written to be able to spawn multiple container instances of the simulator, which makes testing the system easy in an automatic setting.

4 Evaluation

VineIO fulfills a complex set of requirements in order to be a usable prototype while representing our vision of self-organizing platforms. Therefore, we reflect on VineIO from different angles. First, we evaluate the implementation quality of VineIO. Then, we use VineIO to evaluate our concept of decentralized service orchestration. For this, we conduct a scalability test and showcase the use of VineIO in a distributed heterogeneous Edge environment.

4.1 Implementation Quality Analysis

During the implementation of VineIO, we focussed on multiple aspects. Besides accurately implementing our conceptual design, care has been taken to develop a system that is also usable in practice. Precisely, we made extensive efforts to ensure the correctness of VineIO. It should be usable as a viable software framework for future work. The fulfillment of these implementation goals is evaluated in the following.

4.1.1 Fulfillment of Concept

VineIO was developed in such a way that each instance of it can operate autonomously even in cases where connection failures occur. Real-time scheduling orchestration decisions happens solely based on locally gathered information from the Collectors of the Monitoring module. The execution of orchestration decisions might require other nodes in the service tree, but in case of temporary connection failure, the execution is simply aborted and retried after a pause. In these cases, platforms actively try to reconnect to the service tree until it succeeds. Consequently, VineIO can be characterized as resilient. Furthermore, it fulfills all QoS-related features that were discussed. Using eBPF, the latency of user connections is captured in an efficient and accurate way. This makes VineIO unique compared to the related works discussed in Section 2.3. It is also built in a way that makes it suitable to run on several hardware architectures. All leveraged software dependencies as well as the code of VineIO itself work on ISAs such as x86 and ARM. Considering all mentioned facts, our concept fulfills almost all implementation goals that were mentioned in Section 3.1. The only exception is the way user interact with VineIO. The current state of the implementation does not offer a

DNS frontend to users, as mentioned in Section 3.2.5. Instead, users must query their service instance over the VineIO communication protocol.

4.1.2 Testing

Ensuring that VineIO works as intended was a significant part in the development process. Typical helper tools like the Linter *flake8* [92], the type checker *mypy* [93] and the code formatter *black* [94] were all used to maintain code quality and catch common errors already during the process of programming. In addition to that, we used dedicated tests. To test the system effectively, two strategies were pursued: Unit tests and integration tests.

Unit Testing

Unit Testing was used to be able to quickly verify during the process of development that changes to the code do not affect the correctness of the system. Therefore, we wrote various test cases for all main modules of VineIO. We used *pytest* [95], together with several assistance plugins to simplify this process. Notably, the Ports & Adapters architecture made the process of testing modules in isolation easy. On module boundaries, which are represented by the Ports, we implemented mock facades that emulated expected or unexpected behavior of the other modules. This helped to catch many integration errors early.

In total, 44 different unit tests were created during the process of development. By parallelizing the execution of this amount of tests, we leveraged the *pytest-xdist* [96] add-on. Using this approach, running all tests was possible within a few seconds on the development machine.

Integration Testing

Although they run quickly, the disadvantage of unit tests is that they only regard components of a code base separately. Therefore, errors that happen during the interaction of several components are sometimes not caught by them. Since VineIO is a distributed application and is even separated into multiple isolated processes due to reasons stated in Section 3.2.3, we need additional measures to catch more potential errors. Ideally, we can verify the correct cooperation of the system's component by simulating real world usage to a certain extent, thereby testing the system from a blackbox perspective. For this reason, we implemented an extensive integration testing routine.

To fully test VineIO, it makes sense to have a small, multi-node virtualized network topology on the local development machine. The nodes in this topology represent

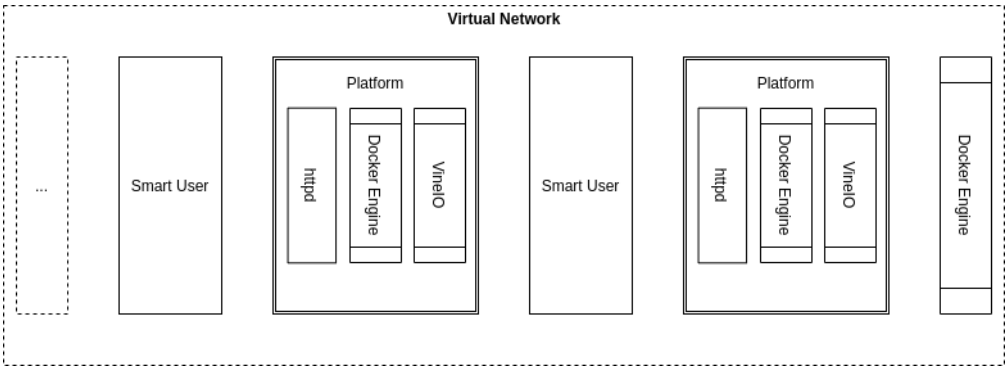

Figure 4.1: Containerized integration test network involving interconnected Docker-in-Docker platforms and smart users

different platforms and users. This enables us to test concrete use case scenarios. We implemented the setup for these scenarios using container technologies. In order to model infrastructure units accurately in our virtual network, we created an image of VineIO which is able to run Docker within a Docker container. This setup is visualized in Figure 4.1. As can be seen, the hierarchical Docker-in-Docker [97] setup enables the orchestrator to spawn service instances in sub-containers that are independent of the high-level virtual network, while still being accessible to the whole network. This approach has several advantages. For instance, all services instances will be reachable via the same address as the platform container. Furthermore, there is no global Docker Engine that needs to be shared between all platforms, which makes it possible to configure them like in production use cases. In general, the scenario of having multiple autonomous and independent platforms that communicate with each other over a common network topology is captured accurately by this structure, which makes our integration testing more realistic than alternative approaches that share a single Docker Engine. However, due to the nature of containers, the host kernel is still shared among all containers and sub-containers. In our use case, this does not cause any problems since conflicting accesses to the kernel do not occur. Mounting required system resources appropriately to the containers can therefore be done safely. Each container platform requires an own VineIO configuration as described in Section 3.2.5, including a suitable address, positional information and an infrastructure database that contains all platforms in the testing environment. Also, their Docker Engine needs to be configured so that no address conflicts occur between service instances. On top of that, a spoofed Geo-IP dataset in form of a CSV file needs to be fed into each platform in order to be able to test the geospatial features of the system. All respective configuration files are provided by the setup.

Alongside the Docker-in-Docker VineIO container platforms, a set of simulated test users which are based on the program described in Section 3.2.5 are part of the network. During integration testing, both a number of individual platforms and users are spawned and a test service tree is deployed. Then, users regularly access their assigned service instance while platforms execute their orchestration tasks as described. This normal state is used for a set of test scenarios that are triggered during the automated testing routine. A certain degree of control over the simulated network is necessary for this. For instance, we control the latency of connections using *tc* [98]. Regarding system load metrics we use a built-in feature of VineIO which enables a Collector to read system statistics from a JSON file instead of the appropriate operating system interfaces. To simulate a long idling time of a platform, we use *libfaketime* [99] in order to control the perceived date and time of VineIO.

For our integration test, we imagined the following scenario. We have 3 fog nodes that are located in Munich, Garching, and Mountain View, as well as 3 users that are also located in these cities. These users regularly access an HTTP service that is deployed on all fog nodes, which is simulated by a httpd service deployment in its default settings. These 3 service instances belong to the same service tree. In the regular state of this scenario, all users are matched to their respective fog node and have a sufficiently good QoE, while the fog nodes do not suffer from an overload. We are now interested in the reaction of the system in case of a bad QoE, a high system load, or an idling service instance. To cause a bad QoE for a randomly selected user, we utilize the command *tc* and add a constant delay overhead to its connection to its service instance. The now resulting network latency is above the delay threshold of the deployed service instance. In case the system reacts correctly, the user will be migrated to another service instance. This is reflected by the root node which eventually returns a different service instance address. After this experiment, the integration test routine removes a service instance from the tree and spoofs a high system load on another instance. Now, the system should react to this anomaly by replicating the service instance to the platform where a service instance was previously removed, since it is the only one available as a replication target. Finally, the test moves the time on all platforms forward. The platforms should now perceive a long idling time without any users for the orchestrated HTTP API, and terminate all service instances.

After playing all scenarios through, the logs of each orchestrator node and each user can be retrieved and analyzed in order to verify whether the control flows were according to the expectations.

Scope	Effective LoC
VineIO, excluding/including unit tests	6737/13474
VineIO client library and CLI	657
VineIO smart user	198

Table 4.1: Effective lines of source code measured with *cloc* [100]

4.1.3 Code Quality

To complement our testing efforts, the source code of VineIO was engineered in a solid manner. Features as well as assisting resources were implemented in 14329 effective lines of Python and C code as can be seen in Table 4.1. Tasks that are time-consuming and would tend to block other tasks of the system are offloaded to individual subprocesses, effectively circumventing Python's subpar multithreading performance. Due to this design choice, the CPU or network-bound—and therefore blocking—vertical slices *Monitoring*, *Actions* and *Overlay* do not hinder each other and can run effectively in parallel. Besides that, attention to best practices in Software Engineering has been spent during the development process. Components that do not belong to the same functional domain are loosely coupled. Moreover, components that are explicitly exchangeable are implemented against well-defined interfaces. This is an advantage that was inherited from the chosen Ports & Adapters architectural principle. For instance, the implementation facilitates the addition of a different platform layer, which could operate on Unikernels. Also, a different Assessment Service could be implemented to introduce a more sophisticated decision-making routine. These scheduling algorithms could also leverage metrics that are gathered by additionally introduced Collectors. In general, the system can be characterized as modular and can therefore be used as a framework for future work. Following modern Python idioms, the Python source code is almost fully annotated with types, which enhances the readability, reduces common errors that originate from dynamic typing and makes refactoring tasks easy. Also, algorithms were clearly separated in functions, methods, and classes. The use of the stylistic linter *flake8* and type checker *mypy* as well as pre-commit hooks fostered these properties during the process of development.

To conclude, the development of VineIO resulted in a well-written prototype that could be seen as not only an evaluation platform but a usable software product. The source code of all VineIO-related software products can be found in the project's repository[1].

[1] `https://gitlab.lrz.de/cm/2020-ralf-masterthesis`

4.2 Experimental Analysis

To validate our claims that we derived from the conceptual design and VineIO's grade of implementation, we introduce two different experiments. On the one hand, we measure the behavior of our decentralized service platform within different operational scales. To accomplish this, we emulate Edge environments of different sizes and measure the duration of all essential orchestration actions of VineIO. On the other hand, we find out whether the system can be used in practice using a federated, geographically distributed and heterogeneous hardware testbed which represents a realistic Edge environment. With the results of the experiments, we can then answer our initial research questions that we defined in Chapter 1.

4.2.1 Scalability and Reliability

As already concluded in various related works (cf. Section 2.3), a distributed control plane has the potential of solving bottlenecks of service platforms with a centralized control plane. Therefore, the self-organizing nature of our Edge service platforms can be seen as a key advantage over other systems.

The operational overhead of each service tree node does not depend directly on the size of the service tree, but rather on the cardinality of the node, i.e. the size of its neighborhood. However, the time-to-execute of the distributed orchestration actions *user migration* and *replication* depend to a certain degree on the size of the Edge environment. The duration of migrations depend on the latency between service tree nodes, since migration requests are routed during target search. This also applies to a certain extent to replications, since a replicating node needs to check whether a potential replication target is free to receive the service instance, which results in an additional network overhead. Also, suitable replication targets need to be sorted and filtered from the infrastructure database, which requires more computations the more entries the database contains. In cases where service tree nodes operate on a larger network topology, incurring larger network delays between tree hops, migrations and replications can take longer compared to network topologies that are smaller.

To quantify the mentioned expected overheads, we conduct experiments as described in the following. Due to the self-healing properties of VineIO and the autonomous operation capabilities of a platform, increasing service tree sizes should not have a negative effect on the reliability of individual service tree nodes. Also, we expect that increasing normalization times should be mainly a result of larger network topologies and not of larger service tree sizes, thereby not defeating the concept for larger-scale use cases.

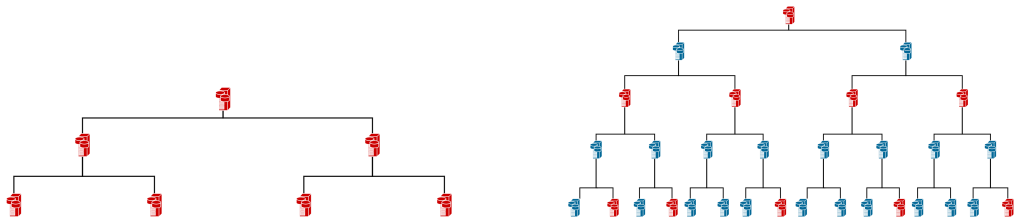

Figure 4.2: Topology A (left) and B (right), with active nodes marked in red

Methodology

It is a known problem in Edge computing research that evaluating Edge platforms in real environments is usually not an ideal method. Since these environments are often private, reproducibility of the evaluation is flawed significantly. Therefore, Edge computing systems are typically evaluated using either simulations or emulations [101].

Although simulators enable larger-scale measurements due to their simplification of the operation environment as well as the systems under test, we choose an emulation-based approach. We want to not only evaluate the concept of self-organizing platforms, but also evaluate whether our prototype VineIO can fulfill the goals of the concept. A simulation-based approach would not have been able to accomplish this to the same extent, since only a reduced version of VineIO would have been evaluated.

We pursue an emulation approach that is similar to our integration testing which we explained in Section 4.1.2. Our test region is Munich, over which we evenly distribute platforms that host a service, and users of this service. For a given network topology that contains the service tree as well as the users, we want to trigger all relevant orchestration decisions and observe how they are executed:

1. By causing a bad network latency, we want to evaluate how VineIO attempts to improve the QoE of a user. We expect that it migrates the user to another suitable service instance by leveraging the regional knowledge of the service tree.

2. By triggering a high system load on a service instance, we want to verify whether

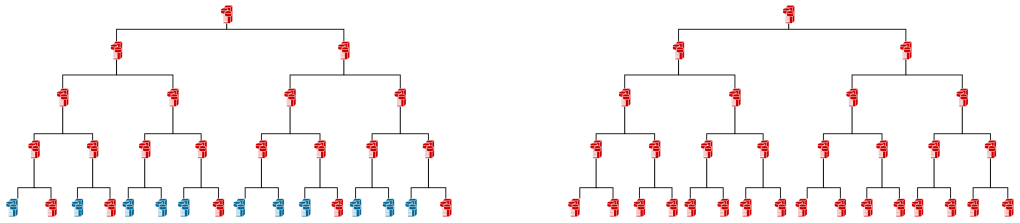

Figure 4.3: Topologies C (left) and D (right), with actives nodes marked in red

it correctly replicates itself to a suitable platform.

3. By simulating a long idling time, we want to check whether the system terminates service instances when they are not accessed anymore.

The general procedure of this scenario is already verified to work in our integration tests. In the course of this evaluation, we are interested in the influence of network topology and service tree sizes on the normalization times for the distributed orchestration actions. To measure it, we first introduce realistic network topologies.

The ideal way to model Internet topologies is to extract graphs from real Internet topology datasets like the CAIDA Macroscopic Internet Topology Data Kit (ITDK). As explained in Section 5.1.1, this turned out to be unreliable in our testing setup. Therefore, to roughly approximate Internet Edge topologies, we choose a tree structure to model our virtual test networks. In total, we introduce 4 varieties of network topologies that resemble such a structure. These are visualized in Figures 4.2 and 4.3. In all topologies, nodes are assumed to be homogeneous and have network connections as shown in the figures. Also, all connections between two nodes have the same general behavior, e.g. in terms of latency overhead. In topology A, we form a binary-tree network topology with 7 nodes. Each of these nodes hosts a VineIO platform as well as a user. Notably, they do not necessarily need to be geographically co-located. Instead, we distribute nodes and users randomly over the network topology. To measure the influence of network topology sizes on the operation of VineIO, we introduce another topology B, which resembles a bigger binary tree. However, the number of nodes and users stays roughly the same, 10 VineIO platforms and 10 users. Distances between nodes in a service tree are multiples of 2, which should increase the network latency between service tree nodes significantly. Topology C is based on the same network as topology B, but it involves 20 platforms and 20 users now, making the topology more densely occupied while doubling the service tree size. Topology D was constructed similarly—now inhabiting 30 platforms and users—and provides another reference point to analyze the effect of service tree sizes. We can run our experiment on these four topologies and compare the results to quantify the influence of service tree sizes, total user demand, and network topologies on VineIO. By comparing topology B, C, and D, we can analyze the effect of tree sizes on the normalization time of user migrations and service replication. Additionally, by comparing A with B, we can infer the additional overhead of longer network paths in an environment.

Setup

To conduct our experiment, we used 3 VMs which were provided by *Hasso Plattner Institut* (HPI). Each VM is based on an x86 ISA and has the same configuration of

computing resources, which includes a shared *Intel Core i7 9xx* CPU and 8 GB of dedicated RAM. Due to the larger amount of platforms and users that can be involved in larger topologies like C and D as well as the number of switches and links involved, running them on a single VM is infeasible. Therefore, we utilized MaxiNet [102] to run our experiment on a distributed set of devices. Since MaxiNet is based on *mininet* [103], we define our network topologies using means provided by *mininet*. For each of the 4 network topologies, we wrote a MaxiNet script that represents them. Since MaxiNet is based on *mininet*, these scripts resemble typical *mininet*-based experimental scripts to a large extent. Hosts that represent users and platforms are inserted into these topologies by connecting them to the appropriate switches. In each topology, a link has a configured network delay of 5 ms.

Similarly to our integration testing setup, users and platforms are run in containers. We leverage MaxiNets support for *containernet* [104] to accomplish this. VineIO platforms are based on the Docker-in-Docker image variant that we presented in Section 4.1.2. This way, each platform runs an own Docker Engine and platforms are modelled realistically. A VineIO configuration generator was written to provide each platform of the experiment with their own individual configuration. The generator distributes the platforms evenly over a given geographical area, which represents Munich in our case. Notably, the assignment of a node to a destined position happens randomly. Each platform then gets its own localized configuration, containing correct settings for the node itself as well as an infrastructure database that contains information about all generated nodes. The generator also distributes users over the same region and creates a Geo-IP database that contains IP address and assigned location of each user. On top of that, deployment specifications are created for each node. They differ depending on whether the root is the specified root node of the service tree. For our test service, we use the Docker image for *httpd* [105] [106] in its default configuration.

The script that automates our evaluation routine[2] works as follows: First, we generate configurations for all platform nodes, distributing them randomly but evenly over Munich. We then clean up leftover state and containers from eventual previous runs to prevent inconsistencies. Afterwards, the MaxiNet script which models the topology that is subject of the current run is uploaded to all VMs. This happens also to the respective node configurations and deployment specifications. The MaxiNet experiment is then executed. The modelled network topology is evenly distributed over all MaxiNet workers and the evaluation routine starts. The executed steps in this scenario mostly resemble the integration testing routine. After the experiment finishes, the automation script of our experiment gathers the log outputs of all participating hosts.

[2] `https://gitlab.lrz.de/cm/2020-ralf-masterthesis/-/blob/master/Code/evaluation/notebooks/10p5ms-tree/loop_experiment.sh`

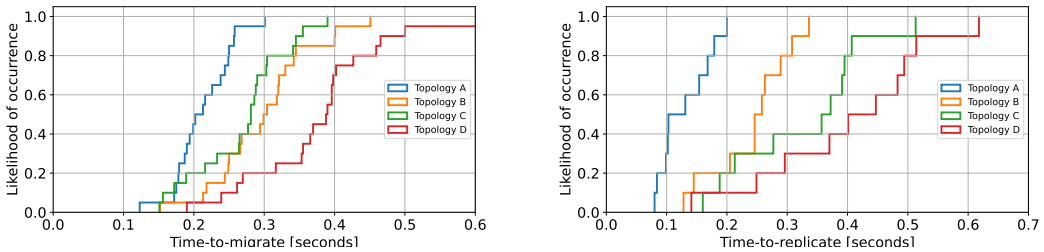

Figure 4.4: Empirical cumulative distribution functions on normalization times of migration (left) and replication (right) actions for all test topologies

Topology/p	1st	50th	75th	90th	99th
A	0.132	0.208	0.245	0.257	0.293
B	0.164	0.302	0.333	0.400	0.441
C	0.152	0.281	0.303	0.346	0.383
D	0.200	0.390	0.410	0.470	25.03

Topology/p	1st	50th	75th	90th	99th
A	0.080	0.117	0.165	0.181	0.198
B	0.130	0.252	0.283	0.311	0.333
C	0.163	0.365	0.394	0.418	0.503
D	0.150	0.424	0.491	0.524	0.608

Table 4.2: Percentiles of time-to-migrate (left) and time-to-replicate values (right) in seconds, measured in the experiment

This whole process is executed and repeated 20 times for each of the 4 network topologies. Due to the fact that each step of the experiment is automated, the setup provides consistency and repeatability of the results.

Results

Figure 4.4 visualizes empirical cumulative density functions (CDFs) for all 4 experiment categories, while Table 4.2 lists statistical metrics for them. The time-to-migrate is defined as the time distance between the event where a platform schedules a migration for a user cluster and the event where VineIO registers that the migration was completed successfully. We have evaluated 20 time-to-migrate samples per topology. The time-to-replicate is defined as the time distance between the event where a replication action is scheduled and the even where a replication was successfully found and verified. We do not include the last step of the actual service deployment on the new replica, since it obfuscates the results. The overhead of this step is however constant in any case and therefore not relevant for this evaluation. We have considered 10 time-to-replicate samples per topology in our evaluation.

By comparing the results of topologies A and B, we can see that migration and replication times are consistently larger in the experiment that involved the bigger

network topology. The median time-to-migrate of topology B is 94 ms larger than the median time-to-migrate of topology A. This is due to the fact that the bigger network topology results in larger network paths and therefore a larger network delay between the nodes in a service tree. We expected this result.

Comparing the results of topologies B and C, we can identify that although the service tree size doubled in C, the migration times did not change significantly in regular cases. The 50th percentiles of B and C differ only by 21 ms, which highlights the benefits from the regional target search procedure. Surprisingly, the time-to-migrate in topology C is consistently better by a small amount than topology B, although it is the bigger topology. We can explain this by the fact that topology C has a higher tree node density than topology B. As a result, network-node hop distances between service tree nodes must not be multiples of 2 anymore, but can be any number. This makes short 1-hop network paths possible between service tree neighbors which is not possible in topology B. Regarding the time-to-replicate, the results of topology C tend to be generally higher than the results of topology B. This is caused by a greater amount of potential replica targets that are checked, and the therefore greater potential network overhead.

One can see that time-to-migrate increased considerably in topology D, which contains 30 platforms and 30 users. The median case of D has a 109 ms higher duration than the median case of C. This effect can be attributed to the fact that in D, the leaf nodes of the tree topology are populated more densely than in C, where most nodes were not located in the 5th level of the topology. Therefore, the probability of having a larger network path between two service tree nodes increases in topology D, which affects the measured migration times. The time-to-replicate increased likewise, which is a result of the described relation to the infrastructure database size. Compared to the other topologies, the amount of potential replicas is the biggest in topology D.

Regarding termination actions, we could verify that terminations succeeded in all cases. Since a terminating node only signs off at the root node, the network overhead of a termination action does not depend on the service tree size. We therefore exclude respective measurements from our result presentation for brevity purposes.

We can conclude that service tree sizes do not have a direct effect on the migration of users. This is expected, since migrations are expected to happen regionally, referring only to more remote service tree nodes when possible. However, larger network paths increase the duration of migrations due to the incurred larger network overhead. Besides that, it can be inferred that the time-to-replicate strongly depends on the number of unavailable potential targets. In our experiment, only one possible replication target was available during the scenario. There is always only one idling platform while all other nodes already have the service deployed and are therefore unavailable for replication. This can result in prolonged target searches when the least optimal replica

candidate is the only one idling, resulting in corresponding network overheads.

4.2.2 Applicability in Heterogeneous Distributed Edge Infrastructures

To test whether VineIO can be used to organize heterogeneous and geographically distributed platform autonomously, we want to emulate a real-world use case. The focus of this evaluation lies on the infrastructure properties on which we run VineIO. In contrast to the scalability experiments, VineIO is not used in a containerized virtual network in the context of this experiment. Instead, it runs on distributed set of dedicated infrastructure units.

As explained in Section 4.1.1, VineIO fulfills all technology-related requirements to operate on heterogeneous hardware sets, since it is supported by various hardware platforms. Furthermore, we already uncovered in Section 4.2.1 that distributed hardware sets and the incurring network delays do not pose an obstacle to Vineio. However, it remains to be seen how VineIO respects the geographic and infrastructural differences during its orchestration routines.

We expect that geographically distributed as well as heterogeneous Edge environments do not hinder VineIO during its operation. Furthermore, due to our implementation logic, we expect that VineIO makes scaling decisions that respect geographic properties of all involved entities.

Methodology

In the scenario of this experiment, we have 4 dedicated VineIO platforms that belong to the same Edge environment and are located in two cities. Three platforms are based on small x86 devices and are located in Potsdam. Each of these runs an instance of the same HTTP API service, forming a service tree of size 3. One small ARM-based platform is located in Munich. It is idling and does not run any service, but is available to host one if scheduled correspondingly. Subject of this experiment is a user in Munich who accesses the mentioned HTTP API. Due to the distance to the other side of Germany, they experience a bad latency. The expected behavior of VineIO is to recognize this fact and try to migrate the user to one of the other 2 existing service instances in Potsdam. However, it should recognize that none of those are significantly closer to Munich.

To check whether the expected sequence of events occurs as described, we play this scenario through and observe the behavior of VineIO. Also, since an interesting metric is the duration of the whole replication operation, i.e. the time difference between the moment when the system recognizes the bad QoE and the time when the user was successfully migrated to the new replica in Munich, we capture it appropriately.

Table 4.3: Hardware setup of the Applicability Experiment

Hostname	Type	CPU	RAM	Architecture	Location
vm-20211019-005	VM	Intel Core i7 9xx	1 GB	x86	Potsdam
vm-20211019-006	VM	Intel Core i7 9xx	1 GB	x86	Potsdam
vm-20211019-007	VM	Intel Core i7 9xx	1 GB	x86	Potsdam
vinepi	RPi 3B+	Broadcom BCM2837	1 GB	ARM	Munich

Setup

Table 4.3 lists all involved hardware units of this experiment. The upper 3 VMs are part of the infrastructure that was provided by HPI, located in Potsdam. As can be seen from the hardware specification, all devices are configured equally and represent a resource-contrained x86 Edge device each. The 4th hardware unit of the experiment is a Raspberry Pi 3B+ [107] and is located in Munich. Like the HPI VMs, it represents a small Edge device, but instead of x86, it has an ARM ISA. Furthermore, it is connected to the Internet via WiFi instead of a more reliable Ethernet cable. Due to the smaller scale of the devices, their architectural and locational variations, this setup can be seen as a representation of a simple heterogeneous and federated Edge environment. We use a VPN to connect the two sites of this environment in order to enable communication between them.

All platforms run a containerized version of VineIO that shares a Docker Engine with its host system and therefore acts as a sidecar to the orchestrated service containers. The container of VineIO is correctly set up, having mounts for all host system resources required for tasks that require privileges, like user connection tracking. Each VineIO platform is configured individually, with respective values for its public address and geographical position. Furthermore, the provided infrastructure database is filled exclusively with the 4 platforms. Also, since the platforms do not operate in the public Internet but within a VPN, we spoof a Geo-IP-Database, providing appropriate geolocation data for all entities in this experiment.

To conduct our experiment, we wrote a script that automates all relevant aspects of it, making our experiment repeatable[3]. In the beginning, all containers and other state that may be left over from previous runs is cleaned up on each infrastructure unit. Afterwards, all 4 platforms are launched and on the 3 VMs in Potsdam, service

[3] `https://gitlab.lrz.de/cm/2020-ralf-masterthesis/-/blob/master/Code/evaluation/sop_demo/run_demo_evaluation.sh`

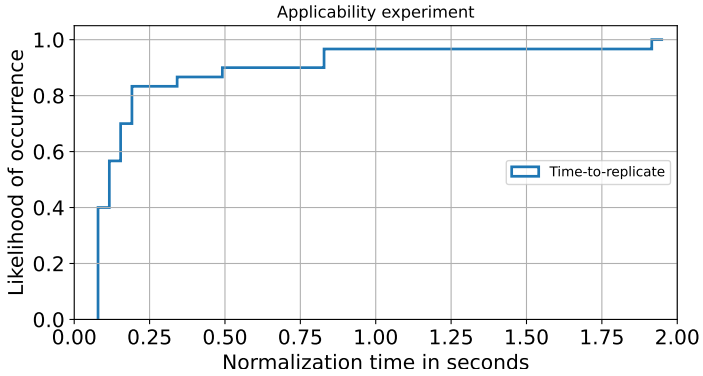

Figure 4.5: Empirical Cumulative Distribution Functions on normalization times of replication actions

instances of an HTTP mock API which is based on *httpd* [106] are deployed. The Munich-based machine which executes the experiment then accesses the HTTP API using *curl* [108] with an artificially added network delay that was introduced using *tc*. Notably, the delay only affects the infrastructure to be accessed in Potsdam. We also periodically check whether the expected replication action was already triggered on the respective unit in Potdam, by periodically querying the address on which we expect *vinepi* to host the HTTP API after replication. Once the request succeeds, the experiment run was successful and we shut down the infrastructure again, cleaning up any run-related state.

In total, we repeat this experiment 30 times. After each run, the captured logs of each platform is saved in order to be analyzed after all runs were completed.

Results

As expected, the HTTP API was replicated successfully from the simulated x86 Edge devices in Potsdam to the ARM Edge device in Munich in all cases. Architectural as well as geographical differences did not interfere with this procedure. Each time, the HTTP API worked equally well on all devices for the test user, except for the significantly better QoE on *vinepi*.

Figure 4.5 visualizes the measured time-to-replicate values via an empirical cumulative density function. Most durations are in the range between 100 and 200 ms, with a 75th percentile of 195 ms.

Notably, the time-to-replicate includes a previous migration attempt, which is not the case in the scalability experiment. VineIO first tries to optimize the QoE of the user by offloading them to a neighbor platform. However, it recognizes that none

of the service tree nodes are suitable judged by their geographical properties. This happens locally on the affected service platform without any network exchange. The normalization strategy then switches to replication, which is then executed accordingly. Therefore, although the normalization routine includes a migration process, it has the network overhead of a simple replication, which can also be seen in the measured values. These are especially low due to the fact that the topmost replica target, which is sorted according to distance, is *vinepi* in every case. Therefore, the only network overhead comes from the transmission of the replication request from Potsdam to Munich.

5 Conclusion

In this thesis, we designed a concept for self-organizing platforms in order to overcome challenges in orchestrating services at the Edge. We implemented this concept by creating VineIO, which represents both a usable system and a platform to evaluate the concept realistically. By monitoring the hardware of infrastructure units and taking it into account during orchestration decisions, VineIO can support heterogeneous Edge device landscapes (cf. RQ1). Since platforms can react in real-time to changes in hardware and user demand characteristics, dynamism in Edge environments does not pose an issue (cf. RQ2). VineIOs autonomous platforms form a tree structure which enables regional service orchestration in a bottom-up manner and circumvents the need of a centralized control plane. This is beneficial for managing large Edge service clusters without a need for manual intervention (cf. RQ3). Furthermore, by introducing eBPF-based user connection monitoring, we can distribute users over our service tree in a QoE-aware manner in real-time while being able to adapt service placements considering geographic properties of users and Edge devices (cf. RQ4).

To underline the benefits of VineIO, we evaluated it from different angles. Through extensive testing, we could verify that our system works correctly. By following software engineering best practices, we were able to create a solid, flexible and extensible framework for self-organizing platforms. We checked whether VineIO fulfills its claims in practice by first analyzing its behavior in different network scales, followed by its feasibility to use it in heterogeneous infrastructure environments.

Based on gathered metrics that we analyzed, we found out that regional operation of VineIO works during user migration actions regardless of service tree sizes. However, we also found out that replication actions have a network overhead that depends on the size of VineIO's infrastructure set. During tests in a heterogeneous and geographically distributed Edge environment, VineIO behaved as expected. Service replication across the federated infrastructure sets worked successfully in all experiments with a median duration of 132 ms. In every case, the replication action optimized the QoE of the user by replicating the desired service into their proximity.

To summarize, we were successfully able to answer our research questions by developing a concept, implementing its prototype and highlighting its characteristics experimentally.

5.1 Limitation and Future Work

Although the purpose of this thesis is fulfilled, there are limitations in our concept of self-organizing platforms as well as the implementation of VineIO. These are discussed in the following.

5.1.1 Scalability

As we could derive from the measured normalization times in Section 4.2.1, the advantages of platforms that manage themselves in a decentralized manner came through especially in migration actions. VineIO could fulfill these in the frame of our experiment. Regarding replications, a network overhead which depends on the number of unavailable replicas to check was discovered. This overhead can be mitigated by removing the need for this additional step. In a more sophisticated system, a dedicated and distributed database system that is writable would have accomplished this, while making the system also aware of appearing, moving, or disappearing devices on the Edge. State information that is relevant for the replication process, like the set of hosted deployments, could be considered without additional network overheads. In any case however, replication attempts can involve network overhead due to real-time failures, e.g. when a replication target does not have enough resources to host an additional service in the precise moment of replication. This potential overhead does not directly depend on the scale on which the system operates, i.e. the size of service trees or Edge environments.

To analyze the behavior of service trees more confidently, experiments on a larger scale would have been necessary. It would be interesting to verify whether the overhead of a single service tree node does not increase significantly even in large service trees. Also, measurements of time-to-migrate as well as time-to-replicate in these settings could have underlined our results.

During our scaling experiments, we used artificially created network topologies that form a tree structure. However, to increase the realisticness of the experiment, our test network could have been modelled according to real Internet topology traces. This was part of our initial testing strategy. We based our approach on EmuFog [48], which can generate MaxiNet scripts based on CAIDA ITDK datasets [109]. In order to be able to use EmuFog, we fixed various bugs. Precisely, we corrected the CAIDA topology parser so it correctly interprets column separators and city names that include space characters. Furthermore, we improved a flawed implementation logic in the topology link interpretation. Initially, EmuFog would only recognize a small subset of connections when a link involves more than two nodes, always using the first node as source and the remaining ones as destinations. This is not correct, since a link

connects all combinations of mentioned nodes. To give an example, in a link of 3 nodes the set of bidirectional edges between the nodes is not $\{(1,2),(1,3)\}$, but instead $\{(1,2),(1,3),(2,3)\}$. We corrected the implementation so that all possible connections are considered. However, the usage of MaxiNet on the correctly generated topologies turned out to be unreliable. Connections between hosts would unexplicably work in one direction, but not in the opposite direction, which does not correspond to the expected behavior of MaxiNet. Upon analyzing the code of MaxiNet, bugs were discovered. For instance, identities of imported classes are usually not equal on different worker nodes, which MaxiNet did not respect, leading to a flawed link implementation. The same issue arose when using traces from the Internet Topology Zoo [110] to model our test network topology. Unfortunately, we could not uncover the precise origin of MaxiNets behavior in the course of this thesis. To still be able to emulate Edge networks at a meaningful scale, we restrained the experiments to tree topologies that involve less complex network emulations. These proved to be reliable in the majority of runs, which made the compensation of failed runs feasible. All resources for the initial approaches are available in the repository of this thesis for reference[1,2]. The author will use the material to hand in pull requests to the affected upstream projects, providing the code of the bugs fixes that were created during this thesis.

5.1.2 Usability

As could be seen in the results of our applicability experiment in Section 4.2.2, VineIO proves to be portable as initially intended. In general, heterogeneous and geographically distributed Edge environments are supported. Service replications from an x86 device to an ARM device works correctly.

The usability of our prototype in real Edge environments still has certain limits that require a more extensive implementation. For instance, replication targets are successfully prioritized by distance, as was demonstrated. However, although the respective data is available, other factors like computational capabilities are currently not considered. This is an issue in cases where not all devices in an Edge environment are able to run all orchestrated services. For instance, some service only work on x86 ISAs, and some might only work efficiently when a dedicated GPU is available. Furthermore, some services require a certain degree of computing power in order to run without issues. Currently, these use cases are not supported by VineIO.

Furthermore, the current implementation of VineIO has portability limits. These do not come from inherent restrictions of the conceptual design, but from the fact

[1] `https://gitlab.lrz.de/cm/2020-ralf-masterthesis/-/tree/master/Code/evaluation/emufog`
[2] `https://gitlab.lrz.de/cm/2020-ralf-masterthesis/-/tree/master/Code/evaluation/topology_zoo`

that VineIO requires a C compiler suite on the devices it operates. This dependency comes from the eBPF parts of the platform, since these are JIT-compiled during runtime. Although the shipment of the respective dependencies with VineIO during application packaging is not a problem due to container technologies, the resulting image sizes tend to have a volume over 1 GB. This can be problematic for some resource-constrained Edge devices. It should be noted however that the compilation process itself does not pose a bottleneck, since the respective C source code spans only a few hundred lines of code, as described in Section 4.1. Also, it is possible to circumvent this issue by compiling the eBPF code ahead-of-time (AOT) to bytecode using *libbpf* [111]. This removes the need for a compiler suite during runtime and would therefore decrease the image size of VineIO significantly. Notably, libbpf is a new technology that is not well established yet, and it requires a reimplementation of our eBPF code [81]. Therefore, it was not adapted in the context of this thesis.

Another notable limitation of VineIO is that service images must be compatible with all hardware architectures that are available in the target Edge environment. Currently, there is no way to specify images that apply for selected hardware architectures only. One way to mitigate this issue is to pre-install the variants of an image accordingly on all infrastructure units, and leave out the image tag in the deployment specifications. Then, VineIO just uses the locally stored image, which is appropriate for the current platform.

5.1.3 Conceptual Design

Naturally, the concept of VineIO does not only have advantages. There are some shortcomings that are inherent to the idea of service trees.

For instance, root nodes fulfill responsibilities that are essential for the correct operation of the service tree. This centralization can make the root node a single point of failure of the whole tree, which is problematic. Ideally, all responsibilities of the service tree would be distributed equally among all tree nodes. However, especially its function as an entrypoint for users and new service tree nodes are hard to realize in a distributed manner. Solutions that involve a distributed sharded database could work in theory, but would have the disadvantage of longer operation times. For instance, finding the optimal service instance for a user might involve multiple network hops and their respective overhead when the service tree index is distributed among the nodes. This is problematic for users that want to connect to a service instance quickly. During the design of the system, we therefore prioritized the user experience and opted for the centralization of some features at the root node, being aware of the trade-offs. To mitigate the negative effects of this design decision, one could introduce replica backup instances for the root node that are synchronized with the main instance and

are enabled in case of errors or resource bottlenecks at the root node. Another solution would be to introduce a leader election mechanism (cf. [61]), where root nodes are elected and service trees restructured in case of errors.

Furthermore, our concept relies on the assumption that latency can be derived from geographical distance. As already proven in research [71], this is not the case in real Internet topologies. Potential solutions could involve datasets which quantify the relation to geographical distance and network latency based on the precise geographic region in which both ends of path are located. Such an approach has proven to be successful [71]. Also, utilizing network distance prediction approaches that focus on distance rankings [59] can be a viable solution.

5.1.4 Features

Although the purpose of this thesis is fulfilled, the resulting concept leaves room for optimizations. For instance, it would be more convenient for users to query service instance addresses over DNS. Such an approach could leverage systems like CoreDNS, which is extensible enough to be integrated with root modules of a service tree. Stateful service replication via CRIU [64] or Hcontainer [63] could make VineIO suitable for a wider range of use cases. Additional platform technologies like Unikernels could make VineIO more interesting for use cases where stronger service isolation guarantees are desired. More complex orchestration policies that realize a concept of operation costs, as described in [17], can make VineIO's behavior more efficient. Furthermore, more monitoring and scheduling capabilities that might consider additional network-level metrics and use more sophisticated decision-making algorithms can make VineIO even more effective. For instance, a scheduler that works proactively by using techniques like Machine Learning can yield better QoE results than our current reactive scheduling approach.

An extensive, categorized list of smaller and larger ideas to improvement the system can be found in the issue tracker[3] of VineIO. We consider these tasks out of scope of this thesis and leave them open for future work.

5.2 Outlook

Already within the scope of this thesis, we could underline conclusions of other related works which suggest that decentralized service platforms are useful in the context of Edge environments. Once Edge computing becomes more prevalent and the size and complexity of Edge service clusters increase, incentives for a system that meets

[3] https://gitlab.lrz.de/cm/2020-ralf-masterthesis/-/issues

the demands of Edge environments while being scalable will rise. A future where Edge components of smart cities are operated on self-organizing platforms could be possible. They could unify the operation of different innovative Edge services, for instance in the fields of autonomous vehicles and entertainment, and actively support user mobility. On a larger scale, these platforms could organize themselves even across cities, respecting inter-city-level metrics to operate smart cities more effectively.

List of Figures

List of Tables

Bibliography

[1] Eurostat. *Cloud computing - statistics on the use by enterprises*. [Online; accessed July 09, 2021]. URL: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises#Use_of_cloud_computing:_highlights.

[2] L. Corneo, M. Eder, N. Mohan, A. Zavodovski, S. Bayhan, W. Wong, P. Gunningberg, J. Kangasharju, and J. Ott. "Surrounded by the Clouds: A Comprehensive Cloud Reachability Study." In: Feb. 2021. DOI: 10.1145/3442381.3449854.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. "A scalable, commodity data center network architecture." In: *ACM SIGCOMM Computer Communication Review* 38.4 (Oct. 2008), pp. 63–74. DOI: 10.1145/1402946.1402967.

[4] S. Hoque, M. S. D. Brito, A. Willner, O. Keil, and T. Magedanz. *Towards Container Orchestration in Fog Computing Infrastructures*. July 2017. DOI: 10.1109/compsac.2017.248.

[5] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani. "Data Center Network Virtualization: A Survey." In: *IEEE Communications Surveys & Tutorials* 15.2 (2013), pp. 909–928. DOI: 10.1109/surv.2012.090512.00043.

[6] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu. *Emerging Trends, Techniques and Open Issues of Containerization: A Review*. 2019. DOI: 10.1109/access.2019.2945930.

[7] R. Debab and W. K. Hidouci. *Containers Runtimes War: A Comparative Study*. Nov. 2020. DOI: 10.1007/978-3-030-63089-8_9.

[8] Cloud Native Computing Foundation. *Kubernetes*. [Online; accessed June 23, 2021]. URL: https://kubernetes.io/.

[9] Cloudflare. *Glossary: Edge Server*. [Online; accessed June 21, 2021]. URL: https://www.cloudflare.com/learning/cdn/glossary/edge-server/.

[10] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue. "All one needs to know about fog computing and related edge computing paradigms: A complete survey." In: *Journal of Systems Architecture* 98 (Sept. 2019), pp. 289–330. DOI: 10.1016/j.sysarc.2019.02.009.

[11] N. Mohan. *Edge Computing Platforms and Protocols*. 2019.

[12] G. Ananthanarayanan, P. Bahl, P. Bodik, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha. "Real-Time Video Analytics: The Killer App for Edge Computing." In: *Computer* 50.10 (2017), pp. 58–67. DOI: 10.1109/mc.2017.3641638.

[13] V. Cardellini, F. L. Presti, M. Nardelli, and F. Rossi. *Self-adaptive Container Deployment in the Fog: A Survey*. 2020. DOI: 10.1007/978-3-030-58628-7_6.

[14] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli. "Geo-distributed efficient deployment of containers with Kubernetes." In: *Computer Communications* 159 (June 2020), pp. 161–174. DOI: 10.1016/j.comcom.2020.04.061.

[15] T. Goethals, F. D. Turck, and B. Volckaert. "Near real-time optimization of fog service placement for responsive edge computing." In: *Journal of Cloud Computing* 9.1 (June 2020). DOI: 10.1186/s13677-020-00180-z.

[16] Kubernetes. *Considerations for large clusters*. [Online; accessed July 09, 2021]. URL: https://kubernetes.io/docs/setup/best-practices/cluster-large/.

[17] A. Zavodovski, N. Mohan, S. Bayhan, W. Wong, and J. Kangasharju. "ICON." In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. ACM, Nov. 2018. DOI: 10.1145/3286062.3286065.

[18] J. Surbiryala and C. Rong. "Cloud Computing: History and Overview." In: *2019 IEEE Cloud Summit*. IEEE, Aug. 2019. DOI: 10.1109/cloudsummit47114.2019.00007.

[19] Jon Swartz. *How Amazon created AWS and changed technology forever*. [Online; accessed July 09, 2021]. URL: https://www.marketwatch.com/story/how-amazon-created-aws-and-changed-technology-forever-2019-12-03.

[20] P. Mell, T. Grance, et al. "The NIST definition of cloud computing." In: (2011).

[21] I. Mavridis and H. Karatza. "Lightweight Virtualization Approaches for Software-Defined Systems and Cloud Computing: An Evaluation of Unikernels and Containers." In: *2019 Sixth International Conference on Software Defined Systems (SDS)*. IEEE, June 2019. DOI: 10.1109/sds.2019.8768586.

[22] Docker Blog. *Docker Index Shows Continued Massive Developer Adoption and Activity to Build and Share Apps with Docker*. [Online; accessed June 21, 2021]. URL: https://www.docker.com/blog/docker-index-shows-continued-massive-developer-adoption-and-activity-to-build-and-share-apps-with-docker/.

[23] Open Container Initiative. *Open Container Initiative Website*. [Online; accessed June 21, 2021]. URL: https://opencontainers.org/.

[24] Open Container Initiative. *runc Github repository*. [Online; accessed June 21, 2021]. URL: https://github.com/opencontainers/runc.

[25] Open Container Initiative. *crun Github repository*. [Online; accessed June 21, 2021]. URL: https://github.com/containers/crun.

[26] Amazon Web Services. *Firecracker: Secure and fast microVMs for serverless computing*. [Online; accessed June 02, 2021]. URL: https://firecracker-microvm.github.io/.

[27] Google. *gVisor Github repository*. [Online; accessed June 02, 2021]. URL: https://github.com/google/gvisor.

[28] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. *Unikernels*. 2013. DOI: 10.1145/2451116.2451167.

[29] Alfred Bratterud. *IncludeOS - Run your application with zero overhead*. [Online; accessed June 21, 2021]. URL: https://www.includeos.org/.

[30] S. Lankes, J. Klimt, J. Breitbart, and S. Pickartz. "RustyHermit: A Scalable, Rust-Based Virtual Execution Environment." In: *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 331–342. DOI: 10.1007/978-3-030-59851-8_22.

[31] NanoVMs Inc. *NanoVMs website*. [Online; accessed June 02, 2021]. URL: https://nanovms.com/.

[32] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran. *A binary-compatible unikernel*. 2019. DOI: 10.1145/3313808.3313817.

[33] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. "Large-scale cluster management at Google with Borg." In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM, Apr. 2015. DOI: 10.1145/2741948.2741964.

[34] Apache Foundation. *Mesos*. [Online; accessed June 23, 2021]. URL: https://mesos.apache.org/.

[35] Docker. *Swarm mode overview*. [Online; accessed June 30, 2021]. URL: https://docs.docker.com/engine/swarm/.

[36] Canalys. *Global cloud services market reaches USD 42 billion in Q1 2021*. [Online; accessed June 02, 2021]. URL: https://canalys.com/newsroom/global-cloud-market-Q121.

[37] A. Khanna and S. Kaur. "Internet of Things (IoT), Applications and Challenges: A Comprehensive Review." In: *Wireless Personal Communications* 114.2 (May 2020), pp. 1687–1762. DOI: 10.1007/s11277-020-07446-4.

[38] D. N. Jha, K. Alwasel, A. Alshoshan, X. Huang, R. K. Naha, S. K. Battula, S. Garg, D. Puthal, P. James, A. Zomaya, S. Dustdar, and R. Ranjan. "IoTSim-Edge: A simulation framework for modeling the behavior of Internet of Things and edge computing environments." In: vol. 50. 6. Wiley, Jan. 2020, pp. 844–867. DOI: 10.1002/spe.2787.

[39] A. Corsaro and G. Baldoni. "fogØ5: Unifying the computing, networking and storage fabrics end-to-end." In: *2018 3rd Cloudification of the Internet of Things (CIoT)*. IEEE, July 2018. DOI: 10.1109/ciot.2018.8627124.

[40] L. Corneo, N. Mohan, A. Zavodovski, W. Wong, C. Rohner, P. Gunningberg, and J. Kangasharju. *(How Much) Can Edge Computing Change Network Latency*. Tech. rep. Germany. 9031. DOI: ng.

[41] ETSI. *Multi-access Edge Computing (MEC)*. [Online; accessed July 09, 2021]. URL: https://www.etsi.org/technologies/multi-access-edge-computing.

[42] M. Iorga, L. Feldman, R. Barton, M. Martin, N. Goren, and C. Mahmoudi. *Fog Computing Conceptual Model*. en. 2018-03-14 2018. DOI: https://doi.org/10.6028/NIST.SP.500-325.

[43] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. "The Case for VM-Based Cloudlets in Mobile Computing." In: *IEEE Pervasive Computing* 8.4 (Oct. 2009), pp. 14–23. DOI: 10.1109/mprv.2009.82.

[44] S. Corson and J. Macker. *RFC2501: Mobile ad hoc networking (MANET): Routing protocol performance issues and evaluation considerations*. 1999.

[45] J. S. Preden, K. Tammemae, A. Jantsch, M. Leier, A. Riid, and E. Calis. "The Benefits of Self-Awareness and Attention in Fog and Mist Computing." In: *Computer* 48.7 (July 2015), pp. 37–45. DOI: 10.1109/mc.2015.207.

[46] M. Ghobaei-Arani, A. Souri, and A. A. Rahmanian. "Resource Management Approaches in Fog Computing: a Comprehensive Review." In: *Journal of Grid Computing* 18.1 (Sept. 2019), pp. 1–42. DOI: 10.1007/s10723-019-09491-1.

[47] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee. "A Container-Based Edge Cloud PaaS Architecture Based on Raspberry Pi Clusters." In: *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*. IEEE, Aug. 2016. DOI: 10.1109/w-ficloud.2016.36.

[48] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran. "EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures." In: *2017 IEEE Fog World Congress (FWC)*. IEEE, Oct. 2017. DOI: 10.1109/fwc.2017.8368525.

[49] Y. Wang, J. Yang, X. Guo, and Z. Qu. "Satellite Edge Computing for the Internet of Things in Aerospace." In: *Sensors* 19.20 (Oct. 2019), p. 4375. DOI: 10.3390/s19204375.

[50] A. Lertsinsrubtavee, A. Ali, C. Molina-Jimenez, A. Sathiaseelan, and J. Crowcroft. "PiCasso: A lightweight edge computing platform." In: *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*. IEEE, Sept. 2017. DOI: 10.1109/cloudnet.2017.8071529.

[51] Kubernetes Multicluster Special Interest Group. *Kubernetes Cluster Federation*. [Online; accessed June 02, 2021]. URL: https://github.com/kubernetes-sigs/kubefed.

[52] Open Infrastructure Foundation. *Open Infrastructure Foundation*. [Online; accessed June 02, 2021]. URL: https://openinfra.dev/.

[53] Open Infrastructure Foundation. *StarlingX*. [Online; accessed June 02, 2021]. URL: https://www.starlingx.io/.

[54] OpenInfra Foundation. *OpenStack*. [Online; accessed May 20, 2021]. URL: https://www.openstack.org/.

[55] StarlingX. *Slides: StarlingX Project Overview*. [Online; accessed June 02, 2021]. URL: https://www.starlingx.io/collateral/StarlingX_Onboarding_Deck_for_Web.pdf.

[56] Eclipse Foundation. *Architecture: Eclipse fog05*. [Online; accessed May 20, 2021]. URL: https://fog05.io/docs/going-deeper/architecture/.

[57] Eclipse Foundation. *Zenoh*. [Online; accessed June 02, 2021]. URL: https://zenoh.io/.

[58] Y. Zhu and Y. Hu. "Efficient, proximity-aware load balancing for DHT-based P2P systems." In: *IEEE Transactions on Parallel and Distributed Systems* 16.4 (Apr. 2005), pp. 349–361. DOI: 10.1109/tpds.2005.46.

[59] H. Yin, X. Zhang, H. H. Liu, Y. Luo, C. Tian, S. Zhao, and F. Li. "Edge Provisioning with Flexible Server Placement." In: *IEEE Transactions on Parallel and Distributed Systems* 28.4 (Apr. 2017), pp. 1031–1045. DOI: 10.1109/tpds.2016.2604803.

[60] T. Ng and H. Zhang. "Predicting Internet network distance with coordinates-based approaches." In: *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE. DOI: 10.1109/infcom.2002.1019258.

[61] R. Casadei, D. Pianini, M. Viroli, and A. Natali. "Self-organising Coordination Regions: A Pattern for Edge Computing." In: *Lecture Notes in Computer Science*. Springer International Publishing, 2019, pp. 182–199. DOI: 10.1007/978-3-030-22397-7_11.

[62] IBM. "An architectural blueprint for autonomic computing." In: *IBM White Paper* 31.2006 (2006), pp. 1–6.

[63] A. Barbalace, M. L. Karaoui, W. Wang, T. Xing, P. Olivier, and B. Ravindran. *Edge computing: The case for heterogeneous-ISA container migration*. Mar. 2020. DOI: 10.1145/3381052.3381321.

[64] Virtuozzo. *CRIU*. [Online; accessed June 23, 2021]. URL: https://criu.org/Main_Page.

[65] system-nuts. *H-Container Github repository*. [Online; accessed June 23, 2021]. URL: https://github.com/systems-nuts/hcontainer-tutorial.

[66] Y. Liao, W. Du, P. Geurts, and G. Leduc. "DMFSGD: A Decentralized Matrix Factorization Algorithm for Network Distance Prediction." In: *IEEE/ACM Transactions on Networking* 21.5 (Oct. 2013), pp. 1511–1524. DOI: 10.1109/tnet.2012.2228881.

[67] T. E. Ng and H. Zhang. "A network positioning system for the internet." In: *USENIX Annual Technical Conference, General Track*. 2004, pp. 141–154.

[68] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. "iPlane: An information plane for distributed services." In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 367–380.

[69] P. Sharma, Z. Xu, S. Banerjee, and S.-J. Lee. "Estimating network proximity and latency." In: *ACM SIGCOMM Computer Communication Review* 36.3 (July 2006), pp. 39–50. DOI: 10.1145/1140086.1140092.

[70] H. Huang, H. Yin, G. Min, D. O. Wu, Y. Wu, T. Zuo, and K. Li. "Network distance prediction for enabling service-oriented applications over large-scale networks." In: *IEEE Communications Magazine* 53.8 (Aug. 2015), pp. 166–174. DOI: 10.1109/mcom.2015.7180524.

[71] Landa et al. "The Large-Scale Geography of Internet Round Trip Times." In.

[72] Python Software Foundation. *Python Website*. [Online; accessed June 02, 2021]. URL: https://www.python.org/.

[73] Stack Overflow. *Stack Overflow Developer Survey*. [Online; accessed June 02, 2021]. 2020. URL: https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-all-respondents.

[74] Python Software Foundation. *CPython Github repository*. [Online; accessed July 09, 2021]. URL: https://github.com/python/cpython.

[75] Python Wiki. *GlobalInterpreterLock*. [Online; accessed July 09, 2021]. URL: https://wiki.python.org/moin/GlobalInterpreterLock.

[76] The ZeroMQ Authors. *ZeroMQ: An open-source universal messaging library*. [Online; accessed July 02, 2021]. URL: https://zeromq.org/.

[77] Sadayuki Furuhashi. *MessagePack Website*. [Online; accessed June 23, 2021]. URL: https://msgpack.org/.

[78] Docker Inc. *Docker Website*. [Online; accessed July 02, 2021]. URL: https://www.docker.com/.

[79] Docker Inc. *Install Docker Engine*. [Online; accessed July 02, 2021]. URL: https://docs.docker.com/engine/install/.

[80] Cilium. *eBPF - Introduction, Tutorials & Community Resources*. [Online; accessed July 02, 2021]. URL: https://ebpf.io/.

[81] iovisor. *BPF Compiler Collection (BCC)*. [Online; accessed June 02, 2021]. URL: https://github.com/iovisor/bcc.

[82] SQLite Authors. *SQLite Home Page*. [Online; accessed June 02, 2021]. URL: https://www.sqlite.org/index.html.

[83] Alessandro Furieri. *SpatiaLite*. [Online; accessed June 02, 2021]. URL: https://www.gaia-gis.it/fossil/libspatialite/home.

[84] NumPy. *NumPy*. [Online; accessed June 23, 2021]. URL: https://numpy.org/.

[85] Alistair Cockburn. *Hexagonal architecture*. [Online; accessed July 09, 2021]. URL: https://alistair.cockburn.us/hexagonal-architecture/.

[86] PyPI. *psutil*. [Online; accessed July 09, 2021]. URL: https://pypi.org/project/psutil/.

[87] anderskm. *GPUtil Github repository*. [Online; accessed July 09, 2021]. URL: https://github.com/anderskm/gputil.

[88] docker. *docker-py Github repository*. [Online; accessed July 09, 2021]. URL: https://github.com/docker/docker-py.

[89] Oren Ben-Kiki, Clark Evans, Ingy döt Net. *YAML Ain't Markup Language (YAML™) Version 1.2*. [Online; accessed July 09, 2021]. URL: https://yaml.org/spec/1.2/spec.html.

[90] Sébastien Eustace. *Poetry - Python packaging and dependency management made easy*. [Online; accessed July 09, 2021]. URL: https://python-poetry.org/.

[91] H. Yu, D. Zheng, B. Y. Zhao, and W. Zheng. "Understanding user behavior in large-scale video-on-demand systems." In: *Proceedings of the 2006 EuroSys conference on - EuroSys '06*. ACM Press, 2006. DOI: 10.1145/1217935.1217968.

[92] pycqa. *flake8 GitLab repository*. [Online; accessed July 09, 2021]. URL: https://gitlab.com/pycqa/flake8.

[93] The MyPy Project. *Mypy - Optional Static Type Checking for Python*. [Online; accessed July 09, 2021]. URL: http://mypy-lang.org/.

[94] psf. *black Github repository*. [Online; accessed July 09, 2021]. URL: https://github.com/psf/black.

[95] Holger Krekel and pytest-dev Team. *pytest: Helps you write better programs*. [Online; accessed July 09, 2021]. URL: https://docs.pytest.org/en/6.2.x/.

[96] pytest-dev. *pytest-xdist Github repository*. [Online; accessed July 09, 2021]. URL: https://github.com/pytest-dev/pytest-xdist.

[97] docker. *Docker in Docker Image*. [Online; accessed July 09, 2021]. URL: https://hub.docker.com/_/docker.

[98] The Linux Foundation. *iproute2 Wiki page*. [Online; accessed July 09, 2021]. URL: https://wiki.linuxfoundation.org/networking/iproute2.

[99] wolfcw. *libfaketime Github repository*. [Online; accessed June 30, 2021]. URL: https://github.com/wolfcw/libfaketime.

[100] AlDanial. *cloc Github repository*. [Online; accessed July 09, 2021]. URL: https://github.com/AlDanial/cloc.

[101] A. Aral and V. Maio. "Simulators and Emulators for Edge Computing." In: Dec. 2020, pp. 291–311. ISBN: 9781785619403. DOI: 10.1049/PBPC033E_ch14.

[102] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl. "MaxiNet: Distributed emulation of software-defined networks." In: *2014 IFIP Networking Conference*. 2014, pp. 1–9. DOI: 10.1109/IFIPNetworking.2014.6857078.

[103] Mininet Project Contributors. *Mininet: An Instant Virtual Network on your Laptop (or other PC)*. [Online; accessed July 09, 2021]. URL: http://mininet.org/.

[104] containernet. *containernet Github repository*. [Online; accessed July 09, 2021]. URL: https://github.com/containernet/containernet.

[105] The Apache Software Foundation. *The Apache HTTP Server Project*. [Online; accessed July 09, 2021]. URL: https://httpd.apache.org/.

[106] The Apache Software Foundation. *httpd Image*. [Online; accessed July 09, 2021]. URL: https://hub.docker.com/_/httpd.

[107] Raspberry Pi Foundation. *Raspberry Pi 3 Model B+*. [Online; accessed July 09, 2021]. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/.

[108] Daniel Stenberg. *curl*. [Online; accessed July 09, 2021]. URL: https://curl.se/.

[109] CAIDA. *Macroscopic Internet Topology Data Kit (ITDK)*. [Online; accessed July 09, 2021]. URL: https://www.caida.org/catalog/datasets/internet-topology-data-kit/.

[110] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. "The Internet Topology Zoo." In: *Selected Areas in Communications, IEEE Journal on* 29.9 (Oct. 2011), pp. 1765–1775. ISSN: 0733-8716. DOI: 10.1109/JSAC.2011.111002.

[111] libbpf Contributors. *libbpf Github repository*. [Online; accessed June 21, 2021]. URL: https://github.com/libbpf/libbpf.