Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

MetaEdge – an Interoperable Framework for the Integration of Edge Computing Platforms

Nathanael Wenzel

Course of Study:	Informatik
Examiner:	Prof. Dr. Christian Becker
Supervisor:	Dr. rer. nat. Frank Dürr, Prof. Dr. Janick Edinger
Commenced:	April 26, 2023
Completed:	October 26, 2023

Abstract

In the current *computing continuum*, edge computing is an important field of research. Edge computing is a paradigm that revolutionizes traditional cloud-centric computing models by decentralizing data processing and analysis closer to the data source, often at or near the network's edge. This approach aims to alleviate the latency and bandwidth constraints associated with transmitting large volumes of data to distant cloud servers. By leveraging local computational resources, such as edge devices or servers, edge computing empowers real-time decision-making, enhances privacy, and enables applications in environments with limited or intermittent connectivity. Currently, there exist many different platforms for edge computing exist. However, these platforms often form a relatively encapsulated system, requiring a specific implementation or abstraction. Within this thesis, a proposal of an interoperable protocol for the integration of different edge platforms and devices is created. Different entities can use the protocol to provide or require computational power and resources, which allows efficient offloading in a heterogeneous environment. No homogenous platform is necessary, but heterogeneous devices can communicate via this protocol. Additionally, context is used to allow for specific optimization methods, e. g., to improve the connectivity or decrease the computation delay. Based on this protocol, a framework is implemented, called MetaEdge. This prototype is used to validate the suitability of the protocol and to show its effectiveness. The concept is proofed by creating multiple worker nodes which implement different edge platforms and runtimes. MetaEdge is able to orchestrate and coordinate these different tasks, and also use context of the worker and the network for different optimization strategies.

Kurzfassung

Im aktuellen *computing continuum* ist Edge Computing ein wichtiges Forschungsgebiet. Edge Computing ist ein Paradigma, das die traditionellen Cloud-zentrierten Computermodelle verändert, indem es die Datenverarbeitung und -analyse dezentralisiert. Diese Aufgaben werden näher an der Datenquelle, oft direkt am Netzwerkrand, ausgeführt. Dieser Ansatz zielt darauf ab, die mit der Übertragung großer Datenmengen an weit entfernte Cloud-Server verbundenen Latenzzeiten und Bandbreitenbeschränkungen zu verringern. Durch die Nutzung lokaler Rechenressourcen, z. B. Edge-Geräte oder -Server, ermöglicht Edge Computing Echtzeit-Berechnungen, verbessert den Datenschutz und ermöglicht es, Anwendungen in Umgebungen mit eingeschränkter oder unterbrochener Konnektivität auszuführen. Derzeit gibt es viele verschiedene Plattformen für Edge Computing. Diese Plattformen bilden jedoch häufig ein relativ gekapseltes System, das eine spezifische Implementierung oder Abstraktion erfordert. In dieser Arbeit wird ein Prototyp für ein interoperables Protokoll zur Integration verschiedener Edge-Plattformen und -Geräte erstellt. Verschiedene Einheiten können darüber Rechenleistung und Ressourcen bereitstellen oder nutzen, was ein effizientes *Offloading* in einer heterogenen Umgebung ermöglicht. Es ist keine homogene Plattform erforderlich, sondern heterogene Geräte können über dieses Protokoll kommunizieren. Darüber hinaus wird Kontext genutzt, um spezifische Optimierungsmethoden zu ermöglichen, z. B. um die Konnektivität zu verbessern oder die Rechenzeit zu verringern. Auf Grundlage dieses Protokolls wird ein Framework, MetaEdge, implementiert. Dieser Prototyp wird verwendet, um die Interaktion des Protokolls zu validieren und seine Wirksamkeit zu zeigen. Mehrere Edge-Plattformen und Laufzeiten werden in dem Konzept integriert und auf verschiedenen Instanzen gestartet. Dabei kann gezeigt werden, dass MetaEdge in der Lage ist, die verschiedenen Aufgaben zu orchestrieren und zu koordinieren sowie den Kontext der Teilnehmer und des Netzwerks für verschiedene Optimierungsstrategien zu nutzen.

Acknowledgments

This master thesis would not have been possible without the help of so many incredible people. I would like to thank all the people that helped and supported me during my studies, especially during the journey of my master thesis. It really means something to me and made this journey a lot more worth.

I am profoundly grateful to my esteemed supervisors, Professor Christian Becker, Professor Frank Dürr, and Professor Jannick Edinger, for their unwavering support, invaluable guidance, and exceptional expertise throughout the journey of crafting this master thesis.

Their open-mindedness and willingness to address my queries, as well as their invaluable contributions in refining the topic, have been pivotal in the development of this research. The communication in our meetings was always pleasant and productive, fostering an environment conducive to learning and growth.

I am particularly indebted to Professor Christian Becker for his instrumental role in identifying and shaping the direction of this thesis. Additionally, I wanted to express my gratefulness for the deep one-on-one sessions, they have been immensely beneficial. Moreover, you all have the ability to provide inspiration during moments of stagnation, which has been a beacon of light in some moments.

In times of respite, I found solace and rejuvenation in the embrace of my family. Their unwavering support and unconditional love provided the necessary sanctuary to recharge and refocus. I could always drop in on them and ask for advice.

I extend my heartfelt gratitude to my football team from early childhood. Their camaraderie and steadfast support, even when my participation was limited, have been a source of motivation and strength.

To all my friends in Stuttgart and the esteemed companions, I am thankful for your unwavering companionship and the countless moments of respite and laughter shared. Your presence has added immeasurable value to my life.

Lastly, I extend my appreciation to all those who provided resources, without which this work would not have been possible.

Thank you all for being the pillars of support in this endeavor.

Contents

1 Introduction **19**
- 1.1 Problem Statement . . . 19
- 1.2 Research Question . . . 20
- 1.3 Contribution . . . 20
- 1.4 Structure . . . 21

2 Background **23**
- 2.1 Distributed Computing . . . 23
- 2.2 Computation Offloading . . . 26
- 2.3 Protocols for Edge Computing . . . 27

3 Related Work **29**
- 3.1 Edge Computing Frameworks . . . 29
- 3.2 Discussion . . . 33

4 Requirements Analysis **35**
- 4.1 Problem Statement . . . 35
- 4.2 Usage scenario . . . 36
- 4.3 Functional Requirements . . . 36
- 4.4 Non-Functional Requirements . . . 38

5 MetaEdge **39**
- 5.1 System Model . . . 39
- 5.2 Protocols . . . 41
- 5.3 Architecture & Design . . . 56
- 5.4 Implementation . . . 60

6 Evaluation **67**
- 6.1 Experimental Setup . . . 67
- 6.2 Results . . . 69
- 6.3 Discussion . . . 80

7 Conclusion and Outlook **83**

Bibliography **85**

List of Figures

5.1 The MetaEdge System . . . 40
5.2 MetaEdge message header . . . 42
5.3 Message sequence for the worker registration . . . 43
5.4 Message sequence for the user registration . . . 45
5.5 Message sequence for the worker deregistration . . . 46
5.6 Message sequence to get a list of current workers . . . 47
5.7 Sequences for retrieving a state update of a worker . . . 48
5.8 Sequences for deploying a task . . . 49
5.9 Sequences for removing a task . . . 51
5.10 Sequences for executing a process . . . 53
5.11 Sequences for aborting a process . . . 55
5.12 Message sequence for the custom ping . . . 56
5.13 MetaEdge Broker Architecture . . . 57
5.14 MetaEdge User Architecture . . . 58
5.15 MetaEdge Worker Architecture . . . 59
5.16 The Tasklet system with consumers, producers, and brokers. . . . 66

6.1 Connectivity of the deployed devices in the virtualized setup . . . 69
6.2 Connectivity of the deployed devices in the Google Cloud setup . . . 69
6.3 Illustration of the centralized offloading capabilities . . . 71
6.4 Screenshots of the broker and affected worker during an exemplary offloading workflow . . . 71
6.5 Illustration of the decentralized offloading capabilities . . . 72
6.6 Screenshots of the broker and affected worker during an exemplary offloading workflow . . . 73
6.7 Illustration of the mixed offloading test case . . . 74
6.8 Screenshots of the broker and affected worker during an exemplary offloading workflow where the user sends a deployment request to the broker but does the execution with a direct connection to the worker . . . 75
6.9 Illustration of the setup when using the network context . . . 75
6.10 Screenshots of the logs of the broker and the local and remote worker . . . 76
6.11 Illustration of the setup when using the workload context . . . 76
6.12 Screenshots of the logs of the broker and the workers when using minimal Central Processing Unit (CPU) usage . . . 77
6.13 Duration of the local executions . . . 78
6.14 Computation delay when offloading the Python task . . . 78
6.15 Computation delay when offloading the OpenFaaS task . . . 80

List of Tables

6.1 Specifications of the host system used for virtual setup 67
6.2 Virtual test bed of integrated platforms and devices into MetaEdge 68
6.3 Specifications of the machines in the cloud setup 69
6.4 Task deployments used for evaluation 70

List of Listings

5.1 Excerpt of the task deployment definition on worker side 61
5.2 Custom data for a docker task handler 61
5.3 Interface for the Task Handler 64
5.4 Export a Docker image 65
5.5 Task description of an OpenFaaS function 65
6.1 Exemplary deployment description of the broker in the virtual setup 68
6.2 Exemplary task to compute Fibonacci numbers, implemented in Python 79

List of Algorithms

5.1 Basic scheduling algorithm 62
5.2 Minimal user authentication 64

Acronyms

AI Artificial Intelligence. 26

AIaaS AI-as-a-Service. 25

AMQP Advanced Message Queuing Protocol. 27

API Application Programming Interface. 33

AR Augmented Reality. 26

CNCF Cloud Native Computing Foundation. 31

CoAP Constrained Application Protocol. 28

CPU Central Processing Unit. 9

CRI-O Open Container Initiative-based implementation of Kubernetes Container Runtime Interface. 31

FaaS Function-as-a-Service. 25

GCP Google Cloud Platform. 68

Go Golang. 60

GPU Graphics Processing Unit. 32

gRPC gRPC Remote Procedure Calls. 60

HA High Availability. 31

HTTP Hypertext Transfer Protocol. 27

IaaS Infrastructure-as-a-Service. 25

IaC Infrastructure as code. 67

ICMP Internet Control Message Protocol. 44

ID Identifier. 42

IoT Internet of Things. 25

JVM Java Virtual Machine. 60

LwM2M Lightweight Machine to Machine. 28

MAPE Monitoring, Analysis, Planning, and Execution. 32

MCC Mobile Cloud Computing. 25

MEC Multi-Access Computing. 26

OCI Open Container Initiative. 31

OS Operating System. 19

PaaS Platform-as-a-Service. 25

REST Representational State Transfer. 27

RPC Remote Procedure Call. 27

RTT Round-Trip Time. 44

SaaS Software-as-a-Service. 25

TCP Transmission Control Protocol. 41

VM Virtual Machine. 27

VR Virtual Reality. 26

XaaS Everything-as-a-Service. 25

XMPP Extensible Messaging and Presence Protocol. 27

YAML YAML Ain't Markup Language. 65

1 Introduction

Mark Weiser's Vision of ubiquitous computing has become reality to a large extend [Wei91]. People interact with many different devices intuitively, which often are connected all together via the Internet. Today, this is also referred to the computing continuum, which spans from large datacenters in the cloud, to smaller devices at the fog and edge in the network, towards the end device of the user [RSS+20]. Additionally, the user has the more capable devices than ever before, for example in terms of storage or computing power, but also the used sensors (e. g., camera) are more sophisticated. However, at the same time, applications are becoming more complex and therefore require more resources, resulting in longer waiting times and higher power consumption of the local device. By using computation offloading, many of these issues seem to get fixed [KLLB13]. In this idea, so-called consumers offload some parts of their application to remote devices, so-called providers, which perform the computation and return the result back to the consumer. This leverages the combined power of different machines from which less capable devices can profit. In case the remote device has more computational power than the user, the execution of the application is faster. Additionally, the local device has the potential to consume less battery, since it does not have to perform complex computations. This is especially an advantage for mobile devices, which have limited battery capacity. However, the devices have to be connected to send data between them. For computation offloading, different models were developed. One of the first ideas was volunteer computing [KWA+01], where different computers provide their resources to form a virtual supercomputer. Later, cloud computing transformed the interaction model of businesses and private users [FZRL08]. However, such resources in the cloud are located relatively far away and therefore the latency is very high. To overcome the issue of latency, edge computing has emerged as a complement to cloud computing [GME+15]. Here, less powerful devices are positioned near the user and thus the latency is reduced.

1.1 Problem Statement

The devices in an edge computing environment are highly heterogeneous, because of the decentralization and different providers [HYW19; SEV+16]. They differ in various aspects, such as hardware, Operating System (OS), architecture, and more. Additionally, different architectural models and supporting system software were developed across different paradigms. Therefore, they do not share a common communication model, but each application model requires specific support. To increase the advantages of edge computing, context-aware algorithms are used [BSEB19]. Different platforms exist, which solve these problems. However, they often require custom implementations and are not interoperable with other systems.

1.2 Research Question

From the above description of the current problem, some research questions are derived to find a solution. The goal is to propose a coordination framework between edge devices, which offers high interoperability via a suitable protocol. It shall be possible for multiple various providers and platforms to interact with this framework in order to support the wide heterogeneity that such systems inherently exhibit. This leads to the first research question:

Research Question 1: *How to design an interoperable framework which coordinates between heterogeneous consumers and providers?*

The consumers in this system may have different use cases and preferences for their required task. They could select the target provider based on multiple different factors, such as computational power, but also the supported runtime or the latency to the provider can impact the decision. The protocol shall support to find a suitable edge node for the current task which the consumer can use. This is reflected in the following research question:

Research Question 2: *How can an interoperable protocol help edge clients to find a suitable edge node?*

As in previous points already mentioned, the devices are highly heterogeneous and may support different runtimes. As different runtimes are developed over time, they may not be known in advance when designing the system. Therefore, the protocol has to be extensible. The protocol should keep track of available runtimes and provide interoperability between the consumers and providers. This leads to the following research question:

Research Question 3: *How can different types of runtimes get integrated in Edge Computing to support heterogeneity and interoperability?*

1.3 Contribution

In this thesis, MetaEdge, a framework for the coordination and interoperability between entities in the edge is proposed. This includes design, implementation, and evaluation. MetaEdge provides a protocol specifically designed for interoperability and easy integration of different platforms and providers in the edge. It uses the properties and context of available entities for the coordination of different tasks.

A structured literature review for current edge environments is performed to find best practices for the system model. Additionally, one aspect is focused on interoperability between the devices at the edge.

During the development, support for different runtimes are integrated in the protocol. This allows devices to dynamically join the system with no pre-configuration of the whole system.

Finally, the system is tested in a heterogeneous edge computing environment, which is set up specifically for evaluation. It demonstrates the usefulness of the protocol and the framework in edge environments. By providing a protocol specifically designed for easy integration of different platforms into the edge could facilitate the development of new applications. They can use a common framework for the interoperability of consumers and providers.

1.4 Structure

This thesis is structured into seven chapters. Following this introduction, an overview and background information about distributed computing, computation offloading and interoperability is given in Chapter 2. The related work is presented in Chapter 3. It presents existing frameworks for orchestrating applications in the edge. In Chapter 4, a requirements analysis is performed to identify functional and non-functional requirements of the system. It also clearly defines the problem statement. The MetaEdge system is presented in Chapter 5, by firstly describing the system model and introducing the protocols. Then the architecture and the design is created and finally some insights in the implementation are highlighted. The evaluation of the system is elaborated in Chapter 6. It describes the experimental setup and presents the results, which are also discussed there. Finally, this thesis is concluded in Chapter 7 by summarizing the findings and pointing out potential future work.

2 Background

Within this thesis, a novel approach is provided which focuses on the interoperability for computation offloading in distributed systems, with a special focus on edge computing environments. This chapter presents some background information and provides insights into each topic, following the current literature. It summarizes different computing paradigms, according to their historical evolution. This starts at the presentation of cloud computing and moves on to the more current edge and fog computing.

Additionally, computation offloading is presented in more detail. It lists the need and advantage for offloading specific parts of the application and important mechanisms are presented. A good and comprehensive summary is can be found in Breitbach's dissertation [Bre22], which this section is based on.

Eventually, several popular protocols, which are also used in edge computing or similar environments, are introduced.

2.1 Distributed Computing

Distributed computing is a powerful paradigm which revolutionized the way of large-scale data processing and resource management. The underlying concept is relatively simple: A large task is broken down into independent computational tasks which are then distributed across multiple interconnected devices or nodes. This ensures an efficient and collaborative way of processing.

According to [vT16], a distributed system has some typical characteristics. First, it consists of a collection of computing elements, which are called nodes. The essence of a node is not further specified, meaning such a node can be a hardware device or a software process. They can also be high-performance mainframe computers or small devices in sensor networks. However, each node is able to behave independently of each other. Second, while there are multiple nodes in the system, from a user's perspective only the complete system as a whole is visible. In order to achieve this, the autonomous nodes need to collaborate. This means that in establishing this collaboration lies the heart of developing distributed systems.

In a distributed system, the participants can be consumers, producers, or both [BM02]. A consumer is defined as it runs a computationally intensive application and needs to harvest computing power. Therefore, it has to ship some of its computational tasks to other devices which support in computation. A provider can perform this computation on behalf of the consumer and returns the result. The consumer can profit of the shipped computation in several ways: It has not to do the computations itself, which can reduce the power consumption. Additionally, if the provider is more

powerful, this can speed up the computation as a whole. In some cases, a broker is necessary which acts as a central node with special responsibilities. It can focus on resource management and task placement which can help organize the consumers and producers.

Some of the first paradigms for distributed computing was *cluster computing* [BB99]. In cluster computing, rather homogenous nodes are used which are connected via a high-speed network. They are dedicated resources, which only purpose is to be a provider in a cluster. In contrast to grid or edge computing (see later), they are usually managed in a large data center. Depending on the capabilities and requirements, the cluster size can be adjusted. The main target of cluster computing is to reduce the response times and provide load balancing, while energy consumption is only a side factor [VLK+11]. A famous representative is the MapReduce algorithm [DG08]. Other examples of cluster schedulers are systems like Spark [ZCF+10], Borg [VPK+15], and Dryad [IBY+07].

In the 1990s, the term *grid computing* was invented to highlight the new features of this paradigm [FKT01]. It's an analogy to the electrical power grid, meaning it is always available when required and provides computing power. In grid computing, the devices are heterogeneous, loosely coupled computers. In contrast to cluster computing, they are not from the same organization, but from different organizations. When participating together in a grid, the devices form a virtual organization for resource sharing. Also, there is no clear distinction between consumers and providers, since the idea is that consumers will eventually also serve as provider. Therefore, they use other resources for some time but also provide their own resources for different participants. The providers expect to be consumers at some point in future and can therefore profit themselves, since the resources belong to the virtual organization.

Another form of distributed computing is *volunteer computing* [And04] which shares the same concept with grid computing: Providers across different organizations contributed to a powerful pool of resources. To the consumers, the providers appear as a transparent unit. In contrast to grid computing, the providers do not expect some kind of compensation of the consumers (e. g., by using their resources when needed). Also, the group of consumers is clearly defined from the start and the providers are often private end-users. A special focus in volunteer computing is placed on security, fault-tolerance, usability and incentive mechanisms.

2.1.1 Cloud Computing

Cloud computing is the dominating paradigm in current commercial products. A cloud consists of huge data centers distributed all around the globe which hosts many resources in a scalable manner. It provides transparent access to scalable computing power, thus allowing arbitrary use cases. One is not limited to specific applications, in contrast to grid computing, which was often bound to scientific use cases. This disruptive technology has got a huge push in 2006, when Amazon released their Elastic Compute Cloud. The key factors were the use of virtualization, which made it also possible for separating the resources for the users, and providing fast scaling. This allows to adjust the computing power based on the current required workload. Since then, it has experienced high market penetration, due to these factors.

The key aspects of cloud computing is to provide storage, processing power and services via virtualized resources [DLNW13]. It is physically distributed across many servers in different data centers. The data centers and servers are connected using distributed systems technologies. With the emerging of smartphones, which provide powerful portable resources, packed in a mobile

device, Mobile Cloud Computing (MCC) has been further developed. The mobile application is not fully computed on the mobile device, but also outside the smartphone. It is best suited for resource-restricted devices which can migrate part of the computation to available servers in the cloud. Since smartphones typically have a Wi-Fi connection, they are perfectly equipped for this use case.

However, the infrastructure itself is separated from the client and the distribution is done internally. This makes it still a relatively static approach of code offloading, where the architects have to define a static or dynamic system landscape. The closer this landscape is to the user, the more the performance is increased. However, due to the globally distributed resources, the latencies can often reach 100 ms [LYKZ10].

Differentiating the grid and the cloud, the cloud consists of scalable, almost unlimited resources on demand, where no prior commitment of the consumer is necessary [AFG+10]. This means, that no reservation of resources is required and the payment model for the services and resources is defined as a pay-as-you-go principle. For users, deploying in the cloud is easier than in grid computing [Gro09]. From an architectural point of view, cloud computing is a rather centralized distributed computing paradigm, where commercial data centers provide cloud resources [MOC+14]. Since the data centers are driven by companies, the resources are provided for profit, in contrast to grid where the providers also will be consumer at some point.

Different service models are developed in cloud computing, which consist of three main services [MG11]. With each service model, the underlying abstraction is increased and the user control is decreased. On the other side, the user profits from lower management effort, since he only can focus on his specific needs. Infrastructure-as-a-Service (IaaS) only provides the virtualized resources, with full control of the user. Platform-as-a-Service (PaaS) provides a managed platform, where the user can install and develop custom applications, but does not have to manage the underlying infrastructure. Software-as-a-Service (SaaS) abstracts the resources even more. Here, software is ready to use for the user, which typically scales automatically. The user does not have to worry about deployment, management, and other configuration, but also cannot change the underlying platform and infrastructure. Additionally, there consist other services, which can be seen as a specialization of one of the above services. For example Function-as-a-Service (FaaS), AI-as-a-Service (AIaaS), and Everything-as-a-Service (XaaS), to name a few [DFZ+15]. Common to all services is that the underlying hardware, and network are always fully transparent to the user.

2.1.2 Fog Computing

The *fog computing* paradigm has split from cloud computing and is closely related to edge computing. The main idea is to bring the resources closer to the end user and thus, reducing the latency and the overall network usage [BMZA12]. Many kinds of applications can profit from this strategy, especially real-time applications (e. g., real-time simulation) or Internet of Things (IoT) applications, where the sensor data can get aggregated earlier before sending it through the whole network. While the differentiation between the edge and fog is not clearly defined, it seems that fog computing has his focus on the infrastructure, while on edge computing the devices are in the focus. Initially, fog computing built a layer between the cloud and the edge of the network and was introduced by Cisco in 2012. The resources are less centrally clustered and distributed closer to the edge [JS16]. However, with this decentralization, different challenges did arise, like increased heterogeneity

because of the higher amount of decentralization and also the complexity did increase. For the offloading decision, the location context should be used to find a suitable fog node. Generally, fog computing can be seen as an extension to cloud computing, where computational resources are ubiquitous and distributed geographically at the edge by nature [HESB18].

2.1.3 Edge Computing

Finally, with the *edge computing* paradigm, one want to execute the computations at the edge of the network to keep the latency as low as possible and to keep the data close to the device [GME+15]. This can be achieved by integrating edge resources, the cloud, and devices which are topologically located between the edge and cloud. A general architecture can be described as follows: The edge devices produce the data at the edge. Fog nodes, which are between the edge and the cloud, offer computing power and act as a bridge to the cloud. In the cloud itself, some higher-level data analysis is performed on a global scale. Here are the most performant devices located. However, many data can already be processed without the need of a central entity [DB17].

As a nature of edge computing, the resources are distributed across several edge nodes. Through their decentralization and proximity to end-user devices, they provide ultra-low latency and a high bandwidth computing environment for connected devices. This is a key aspect for modern, latency-sensitive applications, e. g., real-time simulation, video processing, Augmented Reality (AR), or cloud gaming. Since the data is already processed at the edge, the data traffic to the core network is reduced.

Edge computing was further developed to Mobile Edge Computing, or Multi-Access Computing (MEC), nowadays. Here, dedicated servers are deployed at cellular base stations. The clients access the computational resources over the radio access network in a single-hop distance. Because the devices are inherently mobile (they are connected wirelessly and are often wearable) [CLWG15], special focus on mobility management has to be lied on and also disconnections must be handled.

2.2 Computation Offloading

The origin of computation offloading was introduced with *cyber foraging* [Sat01]. As seen in the previous paradigms, some resources located at different places can overtake the computation for a local device, which is generally described as computation offloading. While the mentioned paradigms do not limit to computation offloading, but can offer multiple services such as remote data storage, this thesis mainly focuses on computation offloading. The fundamental task is to transfer some kind of computation to a remote provider.

Computation offloading is prominent in resource-intensive domains, where relatively weak devices, like smartphones or wearables cannot handle the computations on their own, for example in Artificial Intelligence (AI), AR or Virtual Reality (VR), or computer vision [SBCD09]. The advantage when offloading some computation is reducing the execution delay and the response time, but also improving the energy consumption of the consumer, especially in edge computing systems. An important factor here is the context, since dependent on different attributes, offloading may be

beneficial or not. According to *Flores et al.* [FHT+15], the decision about offloading depends on different factors, which are categorizable into four different categories: what, when, where, and how.

What: This describes the partitioning or granularity of the offloaded application. Depending on the structure, either the whole application can get offloaded, only some parts or components, or even just single methods or threads [LLJL19]. It has to identify the tasks and split the application into offloadable and non-offloadable code. Different approaches exist, which do this manually [CBC+10] or automated [CIMN10].

When: The decision, when a task should be offloaded, is influenced by multiple factors. Generally, the time which is needed for offloading should be smaller that a local computation would take, to improve the performance [CIMN10]. However, other strategies exist, which can use other context to find a different optimization. For example, another goal is to minimize the energy consumption of the device [CBC+10]. This is influenced by the data size, which is shipped over the network, the bandwidth, and the amount of instructions, or task complexity. Additionally, the computational power of the remote device and the local device are important to consider. A dynamic decision does consider these factors, but also static approaches exist. A static approach does not use the context, but may be sufficient if a task is always computationally expensive and therefore, offloading would be most of the time beneficial.

Where: Another important aspect is to decide where the tasks should be placed. This can be done on a single or multiple providers and depends on the amount of parallelism and desired redundancy. If the application is offloaded to multiple different providers, the redundancy is increased and a failure of one provider can be corrected [OWZS13].

How: Additionally, different techniques for the offloading exist. Different systems use Remote Procedure Calls (RPCs), shipping Virtual Machines (VMs) or containers, or relying on a serverless architecture, which exploits stateless functions, following the FaaS pattern.

2.3 Protocols for Edge Computing

Several protocols in the edge computing environment exist. Some were developed with focus of edge computing, while others were adopted from existing platforms but suit well for edge computing. Most of the protocols have a focus on data transfer or reliable messaging between devices and applications.

One of the widest used protocol is Hypertext Transfer Protocol (HTTP), which was established for the Internet and is commonly used for cloud services. Architectural patterns and techniques, like Representational State Transfer (REST) and RPC, are often based on HTTP.

MQTT[1] was developed as a lightweight publish-subscribe messaging protocol for resource constrained devices in low-bandwidth networks. It has his main focus on IoT scenarios, where it provides efficient communication between edge devices and cloud services, and is used in edge environments [RND18]. Beside MQTT, other protocols like Advanced Message Queuing Protocol (AMQP) and Extensible Messaging and Presence Protocol (XMPP) also have their focus on messaging.

[1] https://mqtt.org

A specialized web transfer protocol is CoAP, which transmits the data in a binary format, but is able to translate to HTTP via a proxy [XJK22]. Its targets resource constrained devices in low-power networks and can be used for a binary REST representation with a small header.

Another protocol in the edge environment is Lightweight Machine to Machine (LwM2M) [PVM22]. It was originally built on top of Constrained Application Protocol (CoAP) and allows managing and communication with constrained devices in IoT and edge environments. It has built-in features for efficient device management capabilities, like firmware updates, configuration, and data reporting.

EdgeX foundry is a flexible and scalable edge platform with focus on interoperability between devices and applications in the IoT edge. It does not use custom protocols itself, but promotes the use of standardized protocols, like HTTP and MQTT.

Summing up, many protocols in the edge environment focus on data exchange with an already established architecture or structure of the available devices. No protocol directly enables or facilitates for building such structure dynamically.

3 Related Work

In this section, a structured literature review is performed to assess the related work in terms of computation offloading and edge computing frameworks. The current state of research is elaborated during the analysis, as well as research gaps are identified which are a basis for the requirements analysis. The current state is presented and an overview of current challenges and problems is given.

The focus of this research is on computing frameworks which allow offloading computation, especially at the edge. Systems which have their main focus on different problems, like application partitioning or are limited to a specific use case, are excluded.

One goal of this section is to analyze different frameworks, some coming from IoT while others have their focus on mobile computing, and try to find similarities.

In the following, an overview of different edge computing frameworks is given. The frameworks are evaluated for their fit of interoperability with other systems.

3.1 Edge Computing Frameworks

As already explained, edge computing itself, especially computation offloading, is not a new idea. There exists a vast amount of edge computing frameworks, each having a slightly different focus to improve the performance, ease of use, or provide another advantage. By spanning over the edge-cloud continuum, they can be divided into two categories. The first category contains the frameworks, which try to solve the problem “bottom-up”, with their general focus on IoT. Here, most often sensor data have to be pushed from small devices to the edge of the network where they can be further processed. The second category spans across the frameworks which try to bring cloud services closer to the end user by placing them in the edge. Instead of relying on a central cloud, they make use of edge nodes placed across the globe, making this a “top-down” approach.

3.1.1 Internet of Things

In this section, different frameworks which have their primary focus on IoT environments are explored and analyzed.

One prominent open source edge platform is EdgeX Foundry[1]. It describes itself as a middleware which connects things to the IT environment. The focus of EdgeX Foundry is on IoT, by connecting sensors and analyzing data. This is accomplished with a loosely coupled microservices architecture.

A blueprint of a data flow is the following: A sensor, attached to a so-called thing, collects data with a *Device Service*. The data is passed to the *Core Data* for local persistence. After, the data is passed to the *Application Services* where it gets transformed, formatted, or filtered. Lastly, *Edge Analytics* can trigger device actuation through a *Core Command Service*.

The key service within an EdgeX Foundry architecture are the *Device Services*, *Application Services*, *Core Metadata*, *Core Data*, and *Core Command*. The *Device Services* talk directly to the dedicated things via the used protocol of this thing. *Application Services* build a functions pipeline and ship the data to the enterprise applications, data lakes, cloud systems, and more. *Core Metadata* is a service which has the knowledge about all connected sensors and devices. It also has the information, which device manages which sensor and how one can communicate with the device. *Core Data* is responsible for persisting the sensor data at the edge, if it is desired. *Core Command* is a proxy service for sending an actuation request to a device or *Device Service*.

The architecture of EdgeX Foundry allows to build a tiered fog deployment. However, EdgeX Foundry promotes the usage of standardized protocols, and often the communication is done via a REST-API. Additionally, the devices and services have to be added at a central service before the deployment. It does not provide a possibility to self-register such a device.

In [LPP+22], the authors have pointed out that EdgeX Foundry lacks of container orchestration abilities. Thus, it is missing features such as dynamic deployment of microservices or dynamic resource management. This limits EdgeX Foundry fundamentally in terms of an edge computing platform, beside of its IoT gateway platform. In their work, they add Kubernetes support to EdgeX Foundry to improve the manageability, autoscaling capabilities and real-time monitoring.

In [SASK19, p. 115 ff.], the authors build an edge computing infrastructure for a face recognition application. They distribute the execution of the application stages among the available devices and gateways, in order to get the highest accuracy while keeping the device and gateways constraints. They construct a Multi-Choice Multi-Constraint Knapsack Problem to distribute the load based on the given constraints. This work is also based on [YTC17], where the authors proposed a framework which targets visual processing closer at the edge and utilizing heterogeneous types of devices. They also consider different hardware to potentially accelerate the computation by incorporating dynamic platform independence.

In [LCJZ18], the authors propose an edge computing framework to prove the correctness for cooperative processing. However, the system is very limited to their specific use case.

Another approach is done in [BS20], where the authors create a mobility-oriented retrieval protocol for computation offloading. They design a model for computation offloading in vehicular edge computing. With the proposed protocol, they can efficiently retrieve the output of processed data by using vehicles and road side units as communication nodes. The protocol itself is influenced by geographic routing and uses geolocation information of the network infrastructure and user.

[1] https://www.edgexfoundry.org

3.1.2 Mobile Computing

As in the previous section, different systems for edge computing were introduced, which had their focus coming from a IoT perspective. In the following, different frameworks are introduced that have their focus more on a mobile computing perspective, trying to integrate patterns from cloud computing at the edge.

One of the main dominant systems used for orchestrating cloud services is Kubernetes[2]. Kubernetes provides a rich ecosystem for deployment of orchestrating across different devices. It is based on containers, or pods, which get deployed on nodes belonging to the same cluster. If one node dies, the application can automatically get brought up on other nodes. Each service is duplicated across a specified amount of devices which allows load balancing between the nodes. If the load increases, an autoscaler can automatically add necessary resources and increase the amount of deployed pods. Additionally, through its virtualized solution, it allows isolation and rolling deployments. This containerized virtualization via the pods makes it possible to not only run a specific application on the nodes, but each node is able to run everything. Generally, it provides High Availability (HA) within multiple clusters.

Some of the commercially used systems which are based on Kubernetes are Nokia Edge Network Controller[3] and OpenShift[4]. OpenShift, for example, adds additional features for usability and security to the system and does not use Docker[5], but the Open Container Initiative (OCI) Image format and Open Container Initiative-based implementation of Kubernetes Container Runtime Interface (CRI-O) as runtime, which are both open source projects and specifications. Incoming traffic is handled similar to Kubernetes, by distributing it across the routes within the cluster. It combines the flexibility of cloud-services, virtualization, microservices and containerization with the speed and efficiency of edge computing to increase the functionality, reduce the latency and improve the bandwidth.

One academic project on top of Kubernetes is KubeEdge[6], which connects and coordinates between the edge and cloud environment[XSXH18]. It has reached quite some population and is an incubating project of the Cloud Native Computing Foundation (CNCF). KubeEdge uses the same network protocol infrastructure and same runtime environment on the edge and in the cloud, to enable a seamless communication between edge nodes and cloud servers. The architecture consists of four main components, namely the *KubeBus*, *EdgeCore*, *MetadataSyncService*, and the *EdgeController*. The KubeBus contains the network protocol stack, EdgeCore is a lightweight edge agent and the EdgeController is a controller plugin for Kubernetes. Additionally, it has a distributed metadata store and synchronization service. When deploying an KubeEdge system, the *CloudCore* component runs in the cloud and manages the edge devices. CloudCore consists of the *EdgeController* and *DeviceController*. Edge devices register themselves and join a cluster. The application and services will then be distributed from CloudCore to the edge devices. This is achieved with the help of YAML files which contain the description and details of the service. Edge devices receive and serve the application, which is accessible via HTTP and MQTT [KVM20].

[2] https://kubernetes.io
[3] https://www.nokia.com/networks/products/edge-network-controller/
[4] https://www.redhat.com/en/technologies/cloud-computing/openshift
[5] https://www.docker.com/
[6] https://kubeedge.io

One additional project, which sits on top of KubeEdge and other edge platforms, is FabEdge[7]. In [YNL+22], the authors introduce a KubeEdge Wireless platform for building an Edge-Mesh architecture. Their focus is on resource allocation and service scheduling problems, and they want to enable an easy migration of services within the network. They extend each node with monitoring capabilities of their node status and generate a computing and communication topology according to the regional network characteristics. In the end, they optimize the network strategy using AI. In [KK23], the authors enhance the scheduling approach in KubeEdge. They argue that the initial scheduling approach, using EdgeMesh allows service discovery and load-balancing by distributed user traffic to each pod in the cluster. However, in an edge environment, the load-balancing function has some drawbacks, because the pods are distributed over the edge nodes and latency is introduced when a request is forwarded between these nodes. Their proposed solution is a local scheduling scheme, which processes the traffic at the local edge node without forwarding it to a remote node. This way, they want to provide low latency, which improves the throughput of the cluster. Other projects building on top of Kubernetes are K3s[8] and K0s[9]. Both aim to reduce the footprint and simplify the configuration of Kubernetes, but still provide a CNCF certified Kubernetes distribution. Similar to KubeEdge, K3s has reach the status of a sandbox project at CNCF.

In [BYB+23], the authors argue that existing solutions based on Kubernetes perform poorly at the edge, because Kubernetes is designed for reliable, low latency, high bandwidth cloud environments. They propose Oakestra to overcome those limitations. Their solutions is a hierarchical, lighweight, flexible and scalable orchestration framework. Different edge infrastructure providers can integrate and constraints are used to describe the capabilities of worker nodes.

In [BHQT22], the authors built a serverless-based framework for managing complex MEC solutions, NEPTUNE. They want to place the functions on the edge nodes, according to the user locations and avoid the saturation of single nodes. Additionally, they exploit Graphics Processing Unit (GPU) resources, when available. The resources are allocated dynamically to meet foreseen execution times. Their solution uses K3s as the underlying system. To allocate the functions on the nodes, they create a mixed integer programming problem which minimizes the network delays. The user can set response time requirements which are taken into account during the optimization. This work extend [BMQ19], where the authors designed a decentralized self-management and serverless computing platform: PAPS. Here, a large-scale edge topology is partitioned into delay-aware communities, each having an own leader which provides a reference allocation of resources and places containers inside the community. Additionally, the system contains a supervisor which has a global view of the topology, and is using a Monitoring, Analysis, Planning, and Execution (MAPE)-loop to continuously update the state. This work itself is based on [BMG+19], in which the authors exploit a FaaS model to the edge to bring computation to the continuum in the form of microservices. They consider specific context and user requirements to select the computation location. Their proposed system is A3-E, which allows stateless and lightweight functions to be autonomously fetched, deployed, and exposed by heterogeneous providers. They can reduce the latency by 90 % using the edge instead of cloud and decrease the battery consumption by 74 % when the application is offloaded. Additionally, the deployment time is reduced by 70 %. However, their solution is based

[7] https://github.com/FabEdge/fabedge
[8] https://k3s.io
[9] https://k0sproject.io

on OpenWhisk[10], which uses virtualization to provide containerized functions. For communication, they use a HTTP RESTful Application Programming Interface (API). Other popular platforms to provide FaaS models at the edge, beside OpenWhisk, are OpenFaaS[11] and Knative[12].

In [GFD22], the authors complain that a big problem of traditional FaaS models is the so-called cold-start of the functions. This describes the necessary time, when a function is initially deployed and loaded into memory. This happens, because of the serverless nature: If a function is not used for some time, it is shut down to save resources. The solution of the authors is to replace the containers with a more lightweight WebAssembly runtime, to overcome the issues of a cold-start. Another solution is presented in [PB20], which argues that existing FaaS solutions are not specifically designed for the edge but generally built for powerful hardware. The authors design a novel FaaS platform, specifically designed for the edge, to get the advantages of fewer data transfer to the cloud and reduced latency. They face the challenges of constrained and dynamically shared resources. According to their perspective, FaaS applications in the edge will not span across multiple locations, since this would increase the latency, and the applications are generally monolithic, which simplifies node management and removed the need for a load balancer. In their design, they use HTTP for the communication protocol to deploy the functions and use a reverse proxy to map the requests to the functions.

A similar approach as [PB20] is done by the authors in [SP18]. They also see the issues of FaaS platforms in the edge and implement a more lightweight solution, Faasm [SP20]. According to their results, virtualization provides several advantages. This has multiple stages: First, “pure” virtualization enables sharing a physical machine, then containers enable sharing an operating system. Third, serverless aims to share a runtime. The problem of containers however is still undesirable overhead and an obstruction of sharing features. As a solution, they also propose using WebAssembly to isolate different tasks and to reduce the footprint and cold-start initialization time. The same idea is followed in [HR19] and [HRS+17].

Another solution is proposed with ENORM [WVMN20]. This is a fog computing framework which also uses containerized virtualization of applications. Additionally, a priority is given to the applications to prioritize or delay the execution of the applications. Instead of the other FaaS solutions, this is a full framework which provides auto-scaling mechanisms, too. Additionally, it provides a custom protocol for the handshake, deployment, and termination of edge nodes. However, the control and management is centralized in the cloud server, which starts and terminates the edge nodes. They are not able to join the system themselves. Additionally, this only takes the CPU resources into account and does neglect other resources, like GPU.

3.2 Discussion

During the analyzation of the presented solutions, it gets clear that Kubernetes is one of the dominant systems. While this allows to have standardized platforms, the interoperability is not fully given, since they do not provide a protocol to communicate with different systems. Some exceptions exist,

[10] https://openwhisk.apache.org
[11] https://www.openfaas.com
[12] https://knative.dev/

for example [WVMN20]. However, they often lack of other features, e. g., only a centralized node acts as communication server or that a cluster has to be defined on startup which decreases the dynamism of the platform. The interoperability between different platforms seems to be not focused by any system.

4 Requirements Analysis

In this section, a problem statement is defined which the proposed system aims to solve. The goal of this thesis is to design a framework for the integration of different edge computing platforms. A special focus is given the interoperability protocol between the entities in the system. Then, a usage scenario is created to better visualize what the problems of current systems are, helping to understand what can be expected of the new framework. After, a requirements analysis is performed, split into functional and non-functional requirements. The functional requirements define, which major functions the system will have, while the non-functional requirements describe specific quality goals of the system.

First, the problem statement is defined in Section 4.1. In Section 4.2, a usage scenario is proposed which describes in detail how the interaction of the entities with MetaEdge will optimally look like. Then, the functional requirements are broken down from the problem statement in Section 4.3. These are used to develop the core components of the system and are driven by the research question from Section 1.2. In Section 4.4, the non-functional requirements are defined which describe what constraints and goals the system must meet.

4.1 Problem Statement

As presented in Section 3.1, the current edge computing landscape is very heterogeneous and different solutions exist. One dominant factor is the use of virtualized resources and containers, however also some other ideas emerge, which try to overcome some the issues by using more lightweight solutions. Additionally, many solutions leverage the FaaS paradigm to provide specific functions via the network. However, these do not share a common model and the implementation is limited to specific abstractions. Therefore, the problem is stated as follows: The current edge computing landscape is heterogeneous, and the approaches differ in abstractions and implementations, meaning they support different programming languages or require specific application models. Additionally, the computing nodes can consist of many different devices which differ in their hardware, OS or their location in the network. In this thesis, a framework is proposed which allows heterogeneous hardware to register as edge nodes. Additionally, it does not require the services to be written in one specific programming language, but the framework supports heterogeneous platforms. These nodes provide specific services to be directly computed at the edge of the network. As they are based on different hardware, they require different amounts of time to compute the tasks. Some may even provide hardware acceleration. Beside the latency in the network, these are interesting information for a client to make a decision where a task should be computed. Therefore, the framework shall list the services, which are available at the nodes, in addition to their metadata like CPU power or hardware acceleration. Some nodes may already run a service while others first need to deploy and start the service. The client shall be able to select a specific node based on his needs and use this service to offload the computation.

4.2 Usage scenario

This section presents a fictional usage scenario which helps to understand why an interoperable framework between different edge platforms is necessary. Max in an enthusiastic cyclist who often travels long distances in training. Therefore, he is very mobile and often switches his current network. However, he has some advanced training simulation and analysis devices which he uses to optimize his training effects. This covers the navigation, analysis of his vitality functions (such as heart beat or muscle regeneration), and others. Because this is a very sophisticated system, he has not the ability to do all the computations on a local board computer. Additionally, the computations should be very fast, to provide fast feedback. Since the latency to some cloud resources is too high, an edge computing approach is necessary. However, this cannot be a static provider, since he covers different areas during his training, as already mentioned. As a solution, the system uses MetaEdge, to be able to interoperate with different providers.

When he starts his training, the application connects with the MetaEdge system and registers as user. Additionally, other clients and providers join the network which want to offload other tasks or provide their computational power. The system has different tasks, which need to be offloaded, and analyzes the available providers. Since the tasks have different requirements, it selects a suitable provider for each task. This may be the fastest provider for navigation, but a specific provider which has some hardware acceleration for the vitality functions. As his training progresses, he switches into other networks, resulting in different latencies and available providers. The system detects these changes and because it is interoperable, it can offload the tasks new providers, which support the platform for his specific tasks.

4.3 Functional Requirements

For the MetaEdge system, following functional requirements are formulated. These requirements are directly derived from the problem statement and usage scenario, as well as from the research questions.

R_F1 – Computational Offloading: The main function of MetaEdge is to offload arbitrary tasks to other edge computing platforms. This includes the fundamental tasks of deploying and executing code on another machine, called worker, which join the MetaEdge system. In general, MetaEdge should provide the protocol and infrastructure to send the messages needed for deploying and executing. Developing applications for this system should be simple and require little custom programming.

It shall be possible to offload arbitrary applications, therefore it is necessary to define a suitable protocol and messages which allow the description of such tasks. Additionally, some sort of abstraction shall be included to divide the protocol and the actual client implementation. Furthermore, the system shall be modular to allow extensibility and adding new functions.

R_F2 – Platform independence: As already seen, different edge computing systems have emerged during the time. To allow the integration and interoperability of different systems, MetaEdge should be platform independent and runtime agnostic. It should not restrict the applications

to specific platforms, but allow a wide range, usable via a common protocol. In fact, it should not make any assumptions about the underlying platform. The details need to be handled by the underlying system itself, while the protocol only enables the interoperability.

The shipped tasks can be very different by nature, e. g., already compiled executables, packed in containers, or depending on another runtime, like Python or WebAssembly. All of these options should be covered in the MetaEdge system.

The negotiation about the actual used task shall be done via the protocol. To allow adaptability, the network infrastructure and communication should provide the features necessary to exchange the necessary data.

R_F3 – Heterogeneity support: As in the nature of edge computing, the devices are potentially very different. It ranges from laptops, desktop PCs, smartphones to edge servers or even cloud servers which could be users or workers. Therefore, the offloading system shall be executable on all these devices. Specifically, this means that the framework is independent of hardware, OS, supported runtimes and platforms, or network connection. Additionally, the offloaded applications are heterogeneous theirself and are written in different languages, could be packed in a container or are directly compiled into an executable. The offloading system shall therefore provide a uniform abstraction of these tasks. Even if the tasks and participants are heterogeneous, it shall mediate the data exchange and enable compatible participants to work together.

R_F4 – On-demand computation: The MetaEdge system shall not only enable computation offloading, but also provide a means to execute a specific task on demand at a specific worker node. To decrease the execution delay, a task shall be able to get deployed at suitable worker nodes. From these nodes, a user can select the worker matching his requirements the best, and request the execution at this node. This enables the system to proactively deploy and cache applications, and execute them when needed.

R_F5 – Dynamic registration of users and workers: In order to keep the system versatile, the users and workers should be able to dynamically join and leave the system. This represents very well an edge environment, where the devices are very fluctuant. Since the structure is not known beforehand, no fixed setup can be chosen. Additionally, the participants can change in a dynamic network.

R_F6 – Context-awareness: The system shall use different context dimensions to select the best worker node matching the task description. This means that context about the device, the task, and the network has to be measured. The protocol shall reflect the context via suitable messages, to provide the knowledge also to the clients.

R_F7 – Service discovery: Due to the dynamic nature of the edge environment, the tasks are not deployed to fixed locations. Therefore, users have to find suitable workers for their offloading tasks. To provide an effective offloading mechanism, a service discovery is necessary to inform the users, where different applications are deployed. The protocol shall directly integrate such feature to not have the need for an additional protocol.

4.4 Non-Functional Requirements

Complementary to the functional requirements, the following non-functional requirements are identified.

$R_{NF}1$ – Usability: The MetaEdge system should be easily usable, to motivate other developers to use the system. For this, the framework should provide access to an API which can be used by the stakeholders. Three different stakeholders are connected with the system: Application users, workers, and application programmers. For each stakeholder, the system shall be conveniently usable. For users, MetaEdge shall be seamlessly integrated into the application. Application programmers shall be able to use the API of the framework to integrate MetaEdge into their system. The interfaces have to be clearly defined to reduce potential ambiguities of the system. On the other side, providers or the worker nodes shall be able to easily integrate MetaEdge into their system, too. The interoperability between the participants shall then be coordinated via the underlying protocol.

$R_{NF}2$ – Performance: As the need for edge computing environments is often a reduced latency and increased bandwidth, this cannot be neglected in MetaEdge. The main advantage of computation offloading is, in this case, reduced execution times. In order to keep this as low as possible, the system shall be able to quickly identify possible worker nodes for the given task and ship the task to the identified node. If this process takes too long, the benefits of edge computing are lost.

The proposed protocol for MetaEdge shall only have low overhead, to reduce the requirements for the protocol and support other requirements, such as performance. If the protocol is heavy-weight, this could again introduce bigger latencies, because more data has to be shipped. Therefore, the protocol shall only send as much data as necessary.

$R_{NF}3$ – Scalability: Scalability is the ability to handle increased system load. Since edge computing is very dynamic, MetaEdge also expects to handle increased load. It does not only have to handle one user and one worker, but their may be several hundreds or thousands of user, workers, and tasks, which have to be scheduled, shipped, and executed. All of these factors influence the load of the system. Therefore, it shall be able to handle high load of the system and still satisfy the necessary performance.

$R_{NF}4$ – Robustness: As nodes in the edge environment often have an increased fluctuation, increased number of communication link failures, or other malicious behavior, the system shall be able to cope with such errors and provide a suitable solution to continue the normal workflow. For the user, the errors shall be handled transparently and reduce the noticeable interruptions.

$R_{NF}5$ – Extensibility: Furthermore, extensibility is a very important aspect in the area of edge computing. It constantly evolves and introduces new technologies or architectures. For example, other edge systems may integrate new devices, support different OS, or use different runtimes. To easily add such features to the MetaEdge system, its design shall be modular, which improves the integration of such new features. Especially, the protocol shall be able to work with other platforms and runtimes, not described so far, or allow an easy change in the scheduling technique.

5 MetaEdge

The main artifact of this thesis is MetaEdge. Its design answers research question 1. Additionally, the proposed protocol for MetaEdge will help to answer research questions 2 and 3. The protocol acts as a basic assurance that the systems, which implement this protocol support the required messages and functions. Overall, MetaEdge enables different kind of entities to interoperate in an edge environment, by providing computational resources and offloading tasks. The heterogeneity is resolved via the protocol, making them interoperable.

In this chapter, the system model is introduced first in Section 5.1. This provides an overview and insights into the components and the structure of the system. Then, in Section 5.2, the used protocols are presented. Third, the architecture and design of the software components is explained in Section 5.3. The different kind of interactions are described and explained, here. A special focus is given to allow interoperability. Finally, the implementation of the prototype is introduced in Section 5.4.

5.1 System Model

The vision of MetaEdge is to provide a framework, which can integrate different edge computing platforms. Therefore, it uses a custom protocol for distributed computation offloading at edge environments. It integrates different kind of participants, or entities, namely users, workers, and a broker. Users want to offload computation, while workers can provide their computing capabilities and execute tasks on behalf of the user, described in $\mathbf{R_F1}$. The system is illustrated in Figure 5.1. It visualizes two users and two workers which are registered at the broker. One user deployed a task at a worker and both users use this worker to offload a process.

The protocol is intended for general use in an edge computing environment to offload computation. The primary goal is to enable interoperability between different devices, regardless of the actual hardware or platforms of the participants, according to $\mathbf{R_F2}$. The interaction between the devices is based on the protocol, which defines the possible actions and operates as a standard. By implementing this protocol, the devices can be sure that the specific actions are supported.

To orchestrate the devices, a central broker is part of the protocol, too. This broker mediates between the users and workers. It is a central connection point, used for service and device discovery. In general, his job is to orchestrate the devices, who want to register, deploy tasks or execute processes. The broker will help to especially fulfill the requirements $\mathbf{R_F5}$ and $\mathbf{R_F7}$.

The framework provides a custom protocol, which each participant has to implement. This protocol enables to register at the broker, create a peer-to-peer connection link to the target machine, and to deploy different tasks and execute processes. The task can be any arbitrary kind of executable. The protocol does not make any further assumptions, to support as many platforms as possible

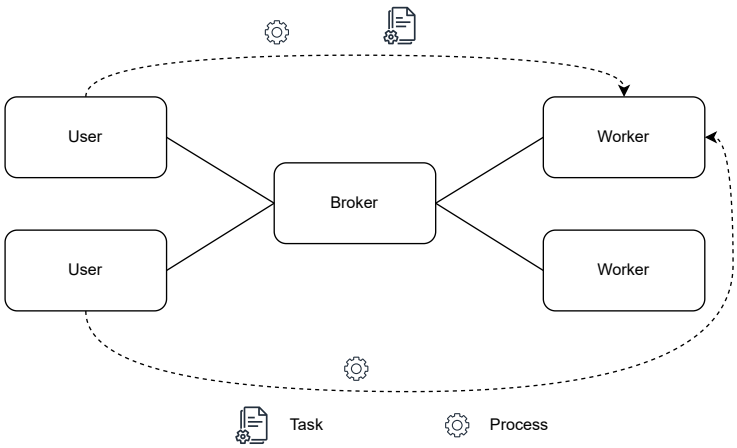


Figure 5.1: The MetaEdge System

for execution. The compatibility between the user and worker node is negotiated via the protocol and integrates the possibility for different context to fulfill requirement $\mathbf{R_F6}$. If the worker node accepts the user node's task, he can deploy and handle it. The actual implementation, how the task gets deployed and executed, is outside the protocol. The worker node can use an abstraction for deployment, e. g., use a VM or a containerized solution, or directly run the task on bare metal. An execution is triggered as a process from a task. When a worker finishes the process, he returns the result via the protocol. If he encounters an error, he sends and error message back to the user to inform the user that something went wrong at this worker node, so the user can reschedule the offloaded computation. Each worker node informs the broker about his system information, currently deployed tasks, and current workload.

Each participant (beside the broker), can either be user or worker. Both types have to register at the broker, but the workers additionally have to provide their system configuration. This includes their available hardware, supported runtimes, or other factors working as constraints of the computation. When the user wants to offload a task, he sends a request to the broker to ask for available workers. This request has to contain different constraints, which are necessary to execute the task. For example, this could contain the underlying hardware architecture, a necessary runtime, or other installed libraries which are necessary during execution. The broker will provide a list of potential workers, from which the user can select the one which suits the best. Alternatively, the broker can directly deploy or execute the task at a suitable worker, in the style of a proxy for the user. When the broker returns the list of available workers, the user can schedule the deployment and execution itself. This selection can be based on different properties of the connection, e. g., hop distance or latency, to the actual worker. The broker does not have this knowledge, since the connection can be different between the devices. Therefore, the user is able to find a specific optimization which is not possible at the broker side. After establishing a peer-to-peer connection between the user and worker, the task can get deployed. For each new task deployed, the worker sends an update to the broker to inform about the additional task. This information is helpful for future users, since they do not have to deploy the task again, but can use the worker who already has deployed the task. To prevent overload of a worker, a monitoring approach is added to the protocol. The workers continuously send updates of their workload to the broker, in order to avoid overload. When the worker detects an error during execution or gets an execution request with a task he has not deployed, he sends an error message, informing about the problem.

All-in-all, the users and workers are not bound to a specific runtime or hardware. Via the protocol, the supported platforms are listed, to negotiate between the users and workers. To show the suitability of this framework, a one-broker-approach is used during implementation. This could be extended to multiple brokers across different regions for large-scale deployments. However, this requires interaction between the different brokers, extending the custom protocol, which is not the focus of this thesis.

5.2 Protocols

As described in Section 5.1, MetaEdge consists of three different components, implemented in software. First, the broker which acts as a central registration service and manages all the connections and provides potential workers to the users. Second, the user which want to offload some computation. For this, he has to register, request a list of potential workers, and finally connect to a worker to deploy and execute his task. Third, a worker which receives the computation requests. In order to make these devices interoperable, a suitable protocol is defined in this section. The protocol must ensure that the data sent between devices is interoperable and that proper communication occurs between the devices. For a computation request of a user, suitable workers shall be selected.

To summarize, the protocol has to enable the following actions:

Registration Users and workers register at the broker to join the system.

Task deployment Users who want to offload computation first have to deploy the specific task at a suitable worker. To get the target task on the worker, the protocol provides a defined workflow.

Process execution After a task is deployed at a worker, an execution of a process has to be triggered to actually run the task. This enables to run a task multiple times with different input, as well as different users can use the task.

Context updates In order to find an optimal scheduling, the workers have to send updates of their workload. If they have to do too much work, the execution could be delayed.

Service and Device discovery In order to find a suitable worker for a user, the list of available workers has to be transmitted to the users, so they can find suitable workers themselves.

The protocol is using Transmission Control Protocol (TCP), to provide reliable connection between two entities. The following sections propose a general message layout, as well as present the different sequences of the protocol workflow. Each sequence describes a custom *command* in the MetaEdge protocol.

5.2.1 General Message Layout

An important aspect of the protocol is to identify corresponding messages. For this, the general message layout of the protocol is designed as followed. It consists of a fixed size header and a variable length of data, or payload. The header contains four different entries, namely an identifier, the version of the protocol, the command, and the length of the payload. This layout has several

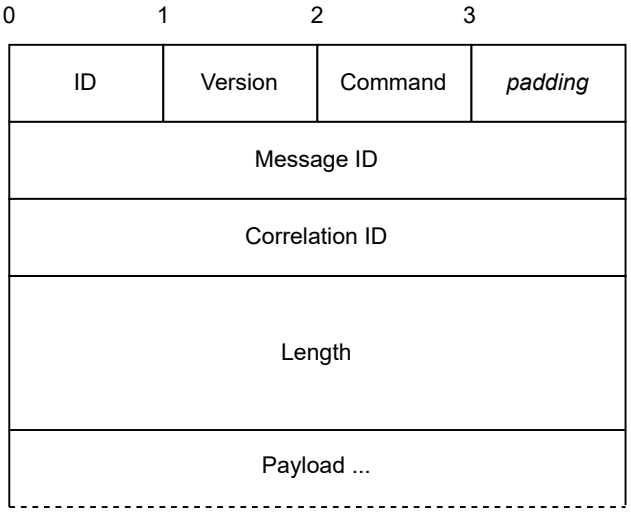


Figure 5.2: MetaEdge message header

features: The identifier marks this message as a MetaEdge message type. If a message does not contain this identifier, the client knows that it does not communicate with another participant in MetaEdge, and he can drop the connection. The version field specifies the actual version of the MetaEdge protocol. This enables to add additional features, which can be added in later versions. Furthermore, it tells the peer, which version the sender supports and understands, thus this is useful for negotiating the correct protocol version. The third entry, command, specifies the message type request. By having this type in the header, the different command options can easily get handled. The payload itself is dependent of this type. Therefore, no general layout can be specified for the payload. However, for each message, the payload has to be defined. This imposes no problem, since the length is known in advance (due to the length field in the header), and the message can get parsed. The header also contains a message Identifier (ID) and a correlation ID. The message ID is a unique identifier of the current message, while the correlation ID is used if the message contains a reply to a previous message, and contains the message ID of the initial request. The last entry, length, is used to separate different messages, e. g., when two consecutive messages are sent. The receiver only has to read the message until the specified length, knowing that it has the complete content of the message, including its payload.

Figure 5.2 visualizes the header. The identifier, version, and command are each one byte in size. Since the payload could get very long if big tasks are sent, this has to handle large numbers. Therefore, the field describing the length of the message has eight bytes in size to describe 64-bit integers, using big-endian.

5.2.2 Error Messages

Since several errors can occur during the communication, a dedicated error message is used to reply to failed requests and indicate the reason, why it failed. This message contains the error code and a correlation ID for the initial request. Following error codes are defined:

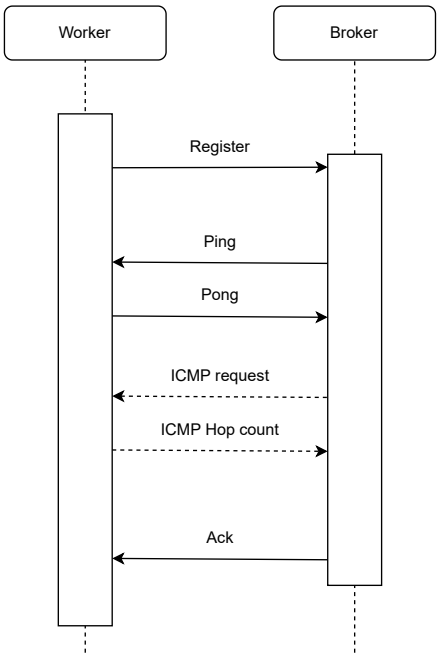


Figure 5.3: Message sequence for the worker registration

Version not supported This code indicates that the used version is not supported by the other participant.

Unknown command This code indicates that the command or message type is not known by the receiver.

Failed to parse message This code indicates that the receiver had an error parsing the message.

Registration error This code indicates that the registration was not successful, for example when already too many participants are using the system.

Deployment error This code indicates that a worker could not deploy a task, for example because it does not support the runtime or has not enough disk space left.

Task not deployed This code is used to indicate that the requested task is not deployed, e. g., a worker cannot execute a process, because it has not deployed the required task.

Execution error This code indicates that the execution failed, for example when the parameters were malformed or another error occurred during the execution.

Unknown process This code indicates that the requested process does not exist on the worker side. For example, the user wants to terminate a specific process, but the worker has not started this process.

5.2.3 Worker Registration

The first important step for computation offloading in MetaEdge is to register the worker. The registration is necessary, in order to enter the network, as well as enabling a peer-to-peer connection from the users to the worker, later. The protocol for registration is defined in the following way: First, the worker sends a registration message to the broker. The broker parses the message to get the static properties of the worker and creates a unique ID for the worker. Additionally, he sends a *Ping* request to the worker, which this one has to immediately answer with a *Pong* reply. Both of these messages do not contain further data, but are only used as markers. When the broker then receives the *Pong* message, he knows the Round-Trip Time (RTT) to this worker. Additionally, the broker uses Internet Control Message Protocol (ICMP) messages to get the number of hops to this worker. Both information, latency and hop count, are used as dynamic properties of this worker. Dynamic properties complement the static properties. Both information are used as worker context. When these requests are finished, the broker sends an acknowledgment to the worker. The acknowledgment contains the assigned worker ID and a list of user tokens. These tokens identify registered users which can communicate with the worker, using this token. Each user token is unique for each user-worker-pair. If the broker encounters an error, he sends an error message to the worker instead of an acknowledgment. An error can occur if the broker does not understand the request, e. g., it does not support the used version, or if some data is missing. When the broker receives the acknowledgment, he knows the registration was successful. The message flow for this registration process is illustrated in Figure 5.3

In the following, the content of the messages are elaborated.

Ping Contains no further data as payload

Pong Contains no further data as payload

Register Contains the information of the worker. This includes the following data:

Connection How the user can connect to the worker. Especially which port the worker listens to

Static Properties Contains the static properties which can be used for context when scheduling the tasks. To provide some minimal information about the capabilities of the worker, the following data is contained in this section:

- CPU
- GPU
- OS
- architecture
- supported platforms
- available libraries

Ack Describes a general acknowledgment of the received message. Contains no further data

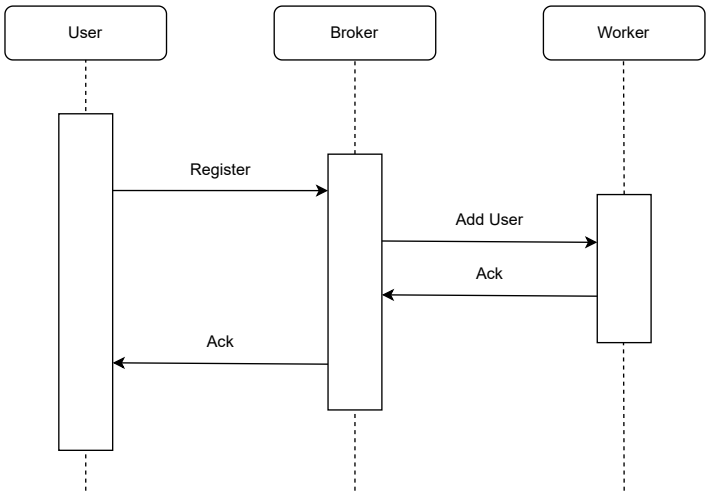


Figure 5.4: Message sequence for the user registration

Generally, the registration message contains the port on which the worker listens to new connections. Additionally, it contains the properties and constraints of the worker, namely the operating system (OS), the architecture, information about the CPU and GPU, supported platforms and runtimes, as well as installed libraries, which can be used when offloading a task.

After registration, the worker has to send state updates in a certain interval to the broker. These state updates accomplish several things. First, the broker still knows that the worker is running, as the status updates work as heartbeat messages. Therefore, no dedicated heartbeat messages are necessary, which reduce the overall load and network usage of the system. Second, the current workload of the worker is contained in these messages. By sending these values, the broker does know how much the worker is used and if he has free resources.

Additionally, the broker starts to send ping messages to the worker. With these ping messages, the latency to the worker is monitored, and the broker can select the workers based on their round trip time (RTT).

5.2.4 User Registration

Similar to the worker, the user has to register, too, to participate in the system. However, the user registration is simpler than the worker registration, because the broker does not have to check the network connectivity to the user. Here, the user has to send a register message, but no more data is necessary. The broker does not have to check the network connectivity to the user, but has to inform each registered worker about this new user. This achieves that only registered users can use the system. Additionally, it prevents misuse, as each request can get tracked back to the user. When the broker receives the *Registration* message from the user, he creates a unique ID for this user. Then, he sends a token to each worker, identifying this unique user-worker-pair. If a worker registers after a user, the broker can use this user ID to create the token. After each worker is informed, the broker sends an acknowledgment to the user, containing the created user ID. Similar to the worker registration, the broker can reply with an error message if he does not understand the request. The process is illustrated in Figure 5.4.

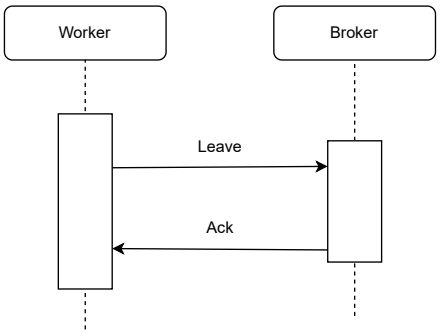


Figure 5.5: Message sequence for the worker deregistration

The message types defined in the process are summarized here:

Register User Informs the broker that the user wants to register. Has no additional payload.

Add User Informs the worker about a new user in the system. Contains the user token as payload.

After a successful registration, the user start to send ping messages to the broker to monitor the latency. No dedicated leave message is necessary for the user. He just stops participating in the system. As he only uses other resources, but does not provide own resources, the other participants are not affected.

5.2.5 Worker Leave

When a worker wants to leave the system, he has to send a *Leave* message, indicating that he will become unavailable. This ensures, that the broker can unregister the worker and that no further tasks are deployed, and no processes executed on this worker. In Figure 5.5, the message sequences are presented.

The worker sends the leave request to the broker. This request has to contain the ID of the worker, in order to properly identify the worker and unregister the correct one. When everything was successful, the broker replies with an *Ack* message. When the worker receives this message, he knows that he was successfully unregistered. Otherwise, he could retry the deregistration. No new error messages are defined in this process, beside the already known ones.

The message types are summarized here:

Leave The leave request of the worker. Contains the worker ID as payload.

5.2.6 User Leave

Since the user does not provide any services but only uses them, there is no dedicated deregistration protocol defined. He can just disconnect and later reconnect. If he reuses his assigned ID, he even can skip the second registration process.

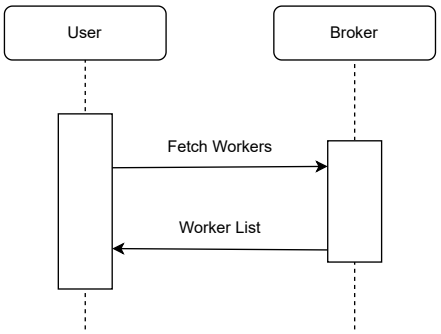


Figure 5.6: Message sequence to get a list of current workers

5.2.7 Fetch Workers

Before the user can directly connect with a worker, he needs a list of the available workers which are registered at the broker. He can request this list with a *Fetch Workers* message, which is sent to the broker. This message contains constraints, which describe specific requirements which the workers have to fulfill. The broker responds with a *Worker List* message, containing all matching workers. Figure 5.6 illustrates the sequence.

The message types contain the following details:

Fetch Workers This message is sent to the broker. It contains the following data:

User ID Used to identify the user

Constraints List of constraints for the workers. This can contain any static (Section 5.2.3) or dynamic properties (Section 5.2.8) of the worker.

Worker List This contains a list of all matching worker to the request. This message contains the following information for each worker:

Connection The connection information to this worker

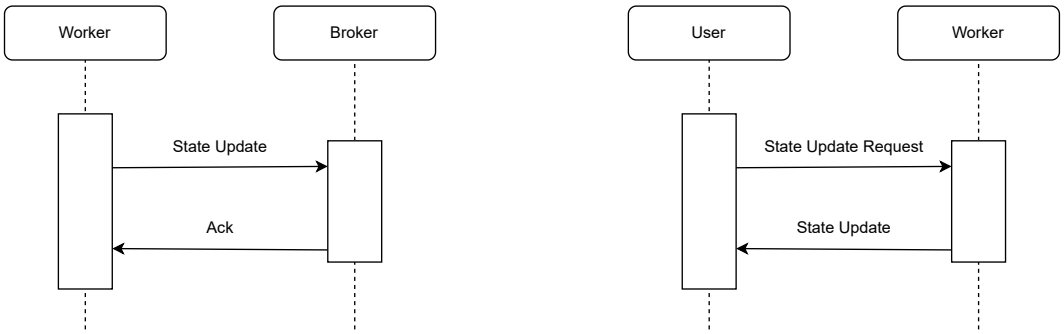
Static Properties Static properties of the worker, as defined in Section 5.2.3

Dynamic Properties Contains the workload and other dynamic properties of the worker, for example:

- CPU usage
- GPU usage
- free memory
- free storage

Deployed Tasks This list contains the tasks which the worker currently has deployed and read to run

Worker ID Describes the worker ID used for identification



(a) Message sequence for a self-initiated state update of the worker

(b) Message sequence for a triggered state update of the worker

Figure 5.7: Sequences for retrieving a state update of a worker

5.2.8 State Update

To integrate a monitoring functionality into the protocol, it defines a *State Update* sequence. This sequence is defined in two ways, illustrated in Figure 5.7a and Figure 5.7b. With this protocol, the current state information of a worker can be obtained.

First, the worker can send self-initiated state updates, in this case to the broker. When the broker receives this message, he can update the worker state in his local memory. To acknowledge the message, he sends an *Ack*. Second, a client, in this case a user, can request the current state of the worker. Here, the worker responds to this request with his *State Update*.

The content of the defined messages types uses following information:

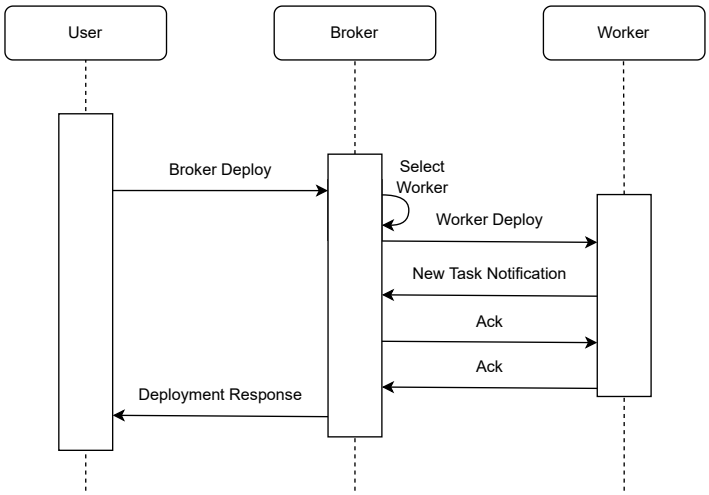
State Update This contains the information of the *internal dynamic state* of the worker. For example, this contains the current workload, free memory, and free storage.

State Update Request This message contains no further data in his payload.

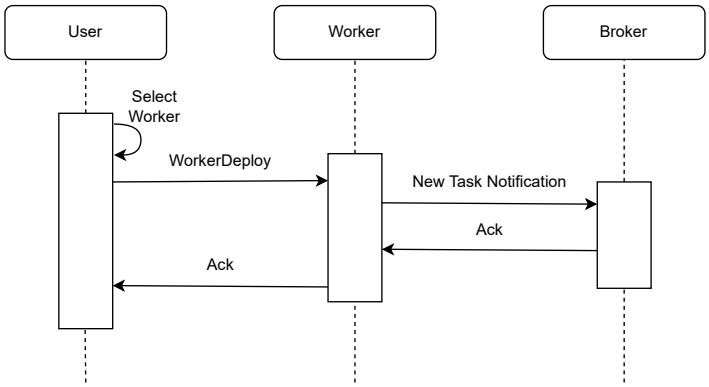
5.2.9 Task deployment

To provide an offloading functionality, the protocol has to support a way of deploying the tasks at the worker. To consider different needs, this will be possible in two different ways, presented in Figure 5.8. Either the user instructs the broker to do the deployment as a proxy of the user, or the user deploys the task directly at the worker himself.

Both workflows will be described below. The first option for the user is to deploy a task via the broker (see Figure 5.8a). Here, he has to send a deployment message to the broker which contains the task and the task description. Additionally, the message contains constraints which the workers have to meet in order to select a suitable worker. These constraints can be hardware constraints, like the architecture of the worker, software constrains, like supported runtimes, or dynamic properties, for example the current workload of the worker. The broker receives the message and has to filter the registered workers to find workers which meet the requirements. From this filtered list, he selects one of the worker to which the task will be deployed. The broker then forwards the deployment request to this broker. When the worker receives the deployment request, he sends a notification to



(a) Message sequence to deploy a task via the broker



(b) Message sequence to deploy a task directly from the user to the worker

Figure 5.8: Sequences for deploying a task

the broker to inform him that he will deploy this task. Based on this notification, the broker can then finally decide if this deployment is valid. This notification would not be absolutely necessary here, since the broker already knows that the worker should deploy this task. However, this message is included to keep the process at the worker the same, when the user directly deploys a task. After the broker has acknowledged the notification, the worker can eventually perform the necessary steps to deploy the task locally. When the deployment is finished, he sends an *Ack* message back to the broker. This indicates a successful deployment and the broker can inform the user which worker has deployed the task. With this information, the user could establish a peer-to-peer connection with the worker directly when he wants to execute a process based on this task. If the task is already deployed at the selected worker, the broker directly returns an acknowledgment, containing the worker with the deployed task.

The second option for deployment is that the user decides itself where it would like to deploy the task, presented in Figure 5.8b. This involves slightly more work on user side, however the user has more control about the actual task placement and can find an optimization for his needs. First, the user has to have a list of registered workers. He can get this list via the protocol from Section 5.2.7. The user can then use the list to find the best suitable worker itself. For example, he

can ping the workers to get the latency and select the worker with the lowest one. Since the network connection between the different participants may be different, the latency between the broker and the worker and the latency between the user and the worker may also be different. When the user has identified the best worker for his use case, he can send a deployment message directly to the worker. Additionally, he can select multiple workers to distribute the task on multiple workers. When the worker receives the deployment request, he sends a notification to the broker to inform him, that this worker wants to deploy the given task. This notification contains an identifier of the task, the worker, and the user. The broker is then able to decide if the task should be deployed at this worker. For example, if the user has already deployed too many tasks or if the task is deployed at multiple different workers already, the broker can intervene and reject the deployment. When the broker accepts the deployment, he replies with an *Ack* message. Additionally, the broker knows that the worker has now deployed this specific task. After the acknowledgment of the broker, the worker finishes the local deployment and sends an acknowledgment to the user. This approach has several advantages: The user can deploy the task on the worker with the lowest latency (or other properties) to the user. Additionally, the broker still has the option to cancel the deployment, for example if the user has not enough permissions, or if he has already used his contingent for task deployments. Besides that, the task only needs to be sent in one message, whereas in the approach where the broker acts as a proxy, it has to be sent first to the broker and then to the worker. This saves some network usage and makes the protocol more efficient.

The following message types and payload are defined in this sequence:

Broker Deploy This message is sent from the user to the broker to initiate a task deployment. It contains the following data:

User ID Identifies the user which sent the request.

Task ID specifies the ID of the task

Platform Describes which platform the task is using and needs to run.

Constraints Specifies constraints to select a suitable worker

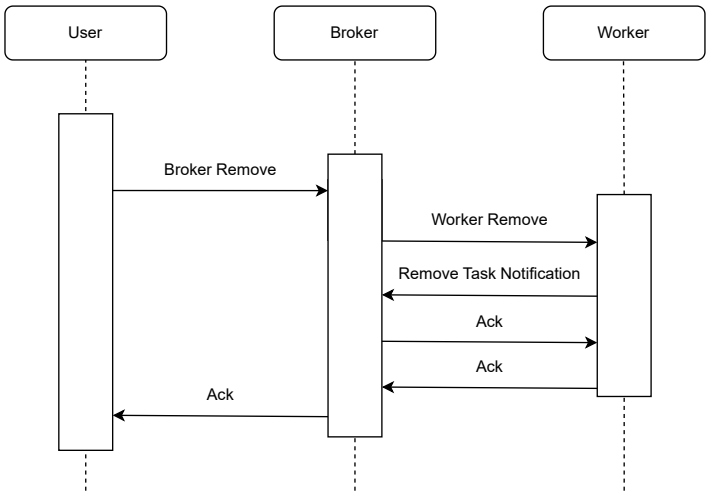
Data Contains the data of the task, which is needed for deployment. This is just an array of bytes, since arbitrary tasks can be sent with this protocol. To reduce the transmitted data, the task is generally transmitted as a compressed archive. For example, this can be an executable. The protocol does not make any assumptions about the task layout. Both, the user and worker, must be able to convert the task for deployment.

Custom data This describes custom data, which is necessary to successfully deploy the task on the target platform. This data is not generalizable for all tasks, but may necessary for specific tasks and platforms. For easy extensibility, the YAML format is used, here, which allows the tasks and clients to define their required data.

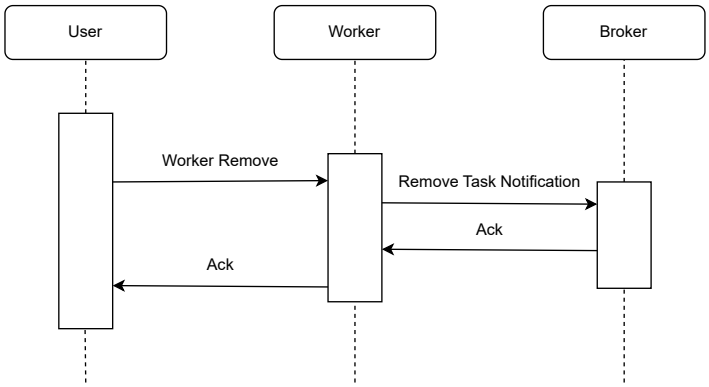
Worker Deploy This message is directly sent to the worker. It contains the same data as the message sent to the broker, but the constraints are removed since the worker is already selected. Additionally, the user ID is switched with an identifier of the user-worker pair.

Add Task Notification This message is sent from the worker to the broker. It contains the following data:

User Token the user-worker pair for the request.



(a) Message sequence to remove a task via the broker



(b) Message sequence to remove a task directly from the worker

Figure 5.9: Sequences for removing a task

Task ID The task ID which will be deployed.

Worker ID The worker ID which wants to deploy the task.

Deployment Response This message is specifically sent from the broker to the user to inform that the task has been deployed. As payload, it contains the data of the worker who has deployed the task. This includes static properties, dynamic properties and connection information. With the connection information, the user is able to connect directly with the worker by using the given IP address and port.

5.2.10 Remove Task

Beside deployment, there should also exist the possibility to remove old tasks, for example when the user does not need it anymore. Removing the task will free some resources, such as disk usage. Similar to the deployment, the protocol provides two different sequences, one which uses the broker as a proxy and one which uses a direct connection between the user and the worker (Figure 5.9).

In Figure 5.9a, the removal process which uses the broker as a proxy is illustrated. The workflow is similar to the task deployment sequence. The user sends a message to the broker to specify which task should be removed from which worker. However, this message already contains the ID of target worker, therefore the broker does not have to identify the worker first. The broker forwards this message to the actual worker. Before the worker removes the task locally, he sends a notification to the broker, which again indicates that the worker will remove this task. After the broker accepts this notification, the worker can remove the task and sends a confirmation to the broker. The request could fail if the worker detects that other users are currently using this task, and therefore he cannot remove this one. The broker in turn sends the confirmation, in form of an *Ack* message, to the user.

Additionally, the user can send the removal request directly to the worker (Figure 5.9b). This request has to contain the task ID, as well as the identifier of the user. The worker then verifies if the task is not used by other users and sends a message to the broker, indicating that the task will be removed. This way, the broker knows that the worker has the task not deployed, anymore and can update his worker cache. He sends an acknowledgment to the worker, to notify that he can now safely remove the task. Eventually, the worker sends an acknowledgment to the user, indicating that the task is now removed.

While the initial trigger in the previous case is necessary, when a user explicitly wants to remove a task, this is not necessary in all cases. If the worker detects, that a task is not used for long time, he can decide to initiate the removal process on its own. In this case, he sends the removal request directly to the broker, which does the same acknowledgment as above, and removes the task. Only the interaction between the worker and user does not happen.

To summarize, the following messages types are defined, here:

Broker Remove This message is sent from the user to the broker. The payload has to contain following information:

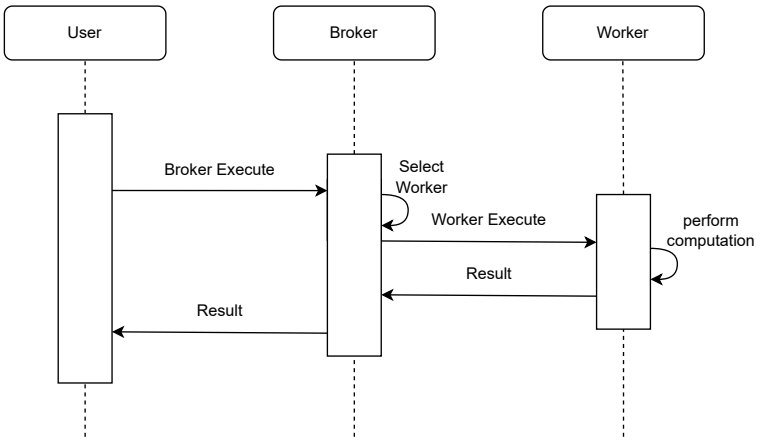
- **User ID** Specifies, which user has sent the request
- **Worker ID** Specifies the target worker, where the task should get removed
- **Task ID** Specifies the task, which should get removed

Worker Remove This message is sent to the worker, either from the broker or user. It has to contain the following information:

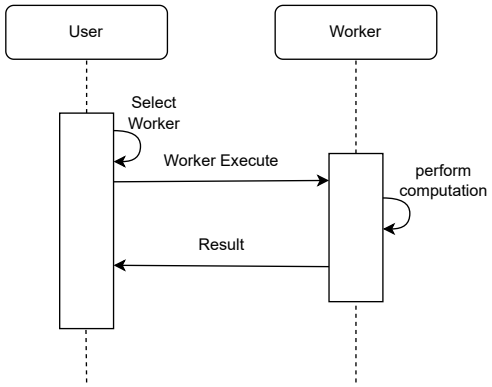
- **Task ID** Specifies the task, which should get removed
- **User Token** The user-worker pair

Remove Task Notification This message is sent from the worker to the broker. It contains the following data:

- **Worker ID** The worker who sends the notification
- **Task ID** The task which the worker will remove locally



(a) Message sequence send an execution request via the broker



(b) Message sequence to execute a process directly between the user and the worker

Figure 5.10: Sequences for executing a process

5.2.11 Execute Process

Finally, the users have to be able to execute the deployed tasks. Similar to the deployment process, two different possibilities exist here, too (see Figure 5.10). The first one is the execution via the broker (Figure 5.10a). Here, the user sends an execution request to the broker, containing the target task ID, parameters and constraints. Additionally, the request contains the user ID to identify the user and a process ID. This process ID can then be used to identify the process. By directly including this ID in this request, there is no second message necessary which informs about the created process. This reduces the number of messages and therefore simplifies the protocol and increases the speed. The broker then selects an appropriate worker, based on the task and the given constraints, and forwards the execution request to this worker. The worker receives the request, performs the computation and returns the result, if successful. If he cannot execute the task, he returns an error message. The broker then forwards either the result or error message to the user.

The other option corresponds to the direct deployment process and is illustrated in Figure 5.10b. After the user has a list of the available workers, as explained in Section 5.2.7, he can decide on his own which worker should perform the process. After he has selected a suitable worker, he sends the message, which contains the task for this process, the parameters, the identifier for the user as well

as the process ID. As before, the worker executes the task and performs the computation. After execution, he returns the result or, in case something went wrong, sends an error message. When the user receives the result or error message, he knows that the task has completed.

The following messages are used in this sequence:

Broker Execute This message is sent from the user to the broker. It contains the following information:

Task ID describes the ID of the task

Process ID specifies the target ID of the process, under which the user want to start the task

User ID the ID of the user

Parameters Parameters which are necessary to start the task. Sine the type depends on the task, arbitrary data has to be able to be sent. Therefore, this contains just an array of bytes and the applications are responsible to decode the data

Constraints Specifies the constraints to find a suitable worker.

Worker Execute This message is sent to the worker to initiate a process execution. The content is similar to the message, sent to the broker, but does not contain any constraints. Additionally, the user ID is switched with a token, identifying the user-worker-pair.

Result This message transports the result of an execution. Since the results are dependent of the tasks, this also contains arbitrary data. Hence, the data is encoded as an array of bytes and the receiver's application is responsible to decode the data.

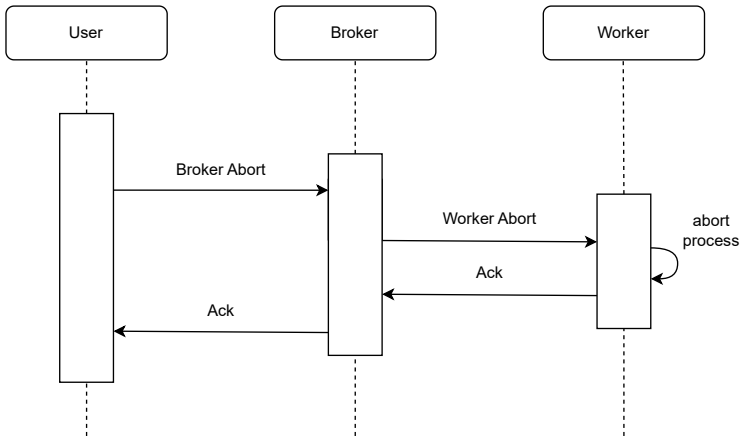
5.2.12 Abort Process

Eventually, the protocol provides a means to abort a running process. This can be useful, if the user does not need the execution result anymore, for example because another worker which started the same process has already finished the computation. The general sequences can be seen in Figure 5.11. Figure 5.11a describes the abortion of a process via the broker. Here, the user sends a *Abort Process* message to the broker which contains the process ID. Now it gets clear why this ID was created for the process beforehand. The broker forwards this request to the worker which executes the given process. The worker will then stop the process and return an *Ack* Message to the broker. Again, the broker will just forward this message to the user. The sequence for a direct abortion of a process, between the user and a worker, is even simpler (Figure 5.11b). Instead of sending the message to the broker, the abortion message is sent to the worker. The worker stops the process and informs the user. In both cases, the user knows that the process is no longer executed. However, this interferes with the process execution protocol. Therefore, instead of sending the result of the execution, the worker has to send a message which indicates that the process was aborted. The sequence of the execution protocol does not change in this case, but another reply is received.

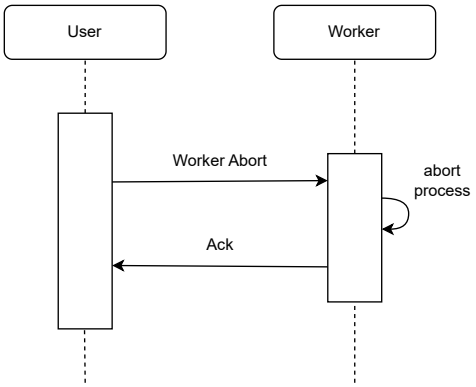
If the execution cannot be stopped, e. g., because the user has not started it before, the worker returns an error message.

The following messages are defined in this sequence:

Broker Abort This message is sent from the user to the broker. It contains the following data:



(a) Message sequence send a process abort request via the broker



(b) Message sequence to abort a process directly between the user and the worker

Figure 5.11: Sequences for aborting a process

User ID Identifies the user of this request

Process ID Specifies the process which should get stopped. Only the user who started the process knows this ID.

Worker Abort This message is sent to the worker, either from the user or the broker. It contains the following data:

User Token The identifier of this user-worker pair. When the broker acts for the user as a proxy, the broker also uses the user ID.

Process ID The process which should get stopped.

5.2.13 Ping

Additionally, the protocol provides a message sequence to check the latency with a worker. This is necessary, to measure the connection quality of a worker. If the latency is too high, a user may want to select a different worker for offloading. The ICMP already defines a message which achieves

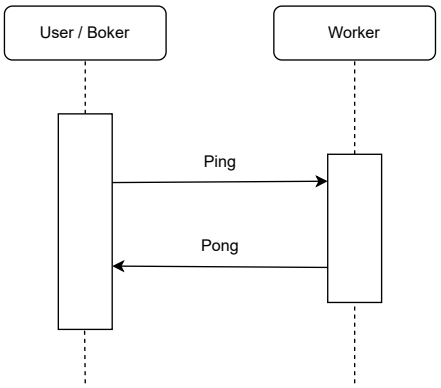


Figure 5.12: Message sequence for the custom ping

this behavior, however these packets may get less prioritized from routers. By providing a custom message, the actual latency inside the protocol can be measured. However, it's not possible to get the hop-count with this custom message, too. In this case, ICMP must be used as a fallback.

The message flow is visualized in Figure 5.12. Both, the user and broker, can send a ping message to the worker. The worker immediately responds with an *Ack* message. After the sender has received the *Ack*, he can easily compute the RTT and define the latency to this worker.

In this sequence, the following messages are used:

Ping This message indicates a Ping request. It contains no further data.

Pong This message confirms the request.

5.3 Architecture & Design

This section provides a deeper insight into the architecture of the individual components. It reveals the responsibilities of the components to enable what is needed in detail to achieve their goals. Additionally, to provide an easy-to-use framework, a public API has to be usable by other components.

As previously described, the main software artifacts are the broker, the user, and the worker. The broker is a central connection point for the users and workers and keeps a cache of the workers and their deployed tasks. The user registers at the broker to deploy and execute tasks at the worker, which can be done with peer-to-peer connections or using the broker as proxy. Both, the user and the broker, have a task scheduler to select the best worker for the current task. Finally, the worker registers at the broker to provide its computing resources. It receives the tasks and has to deploy and execute them. The actual implementation of the task handler is up to the worker, therefore the framework provides an API to communicate via standardized API calls. The task handler can deploy and execute the tasks, depending on the workers capabilities. The software components of the broker are described in Section 5.3.1, the user is described in Section 5.3.2, and the components for the worker are described in Section 5.3.3.

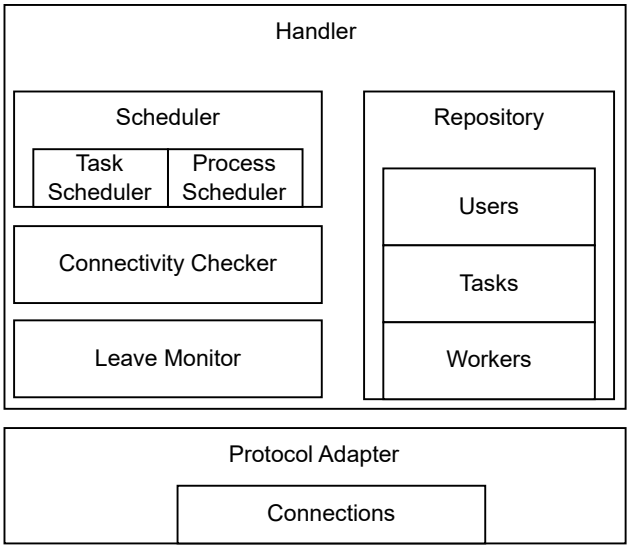


Figure 5.13: MetaEdge Broker Architecture

5.3.1 Broker

The architecture of the broker is presented in Figure 5.13. It uses a layered architecture, to abstract the different components. The lower layer is an adapter for the protocol. It handles all incoming and outgoing connections and parses the messages of the clients (users and workers). It's responsible to communicate with the other participants, using the MetaEdge protocol. On top of this layer is the *Handler*, which receives the parsed messages from the connection layer and can itself pass messages to the underlying layer which are then sent to other participants. This Handler does the actual implementation of the protocol and has to forward the messages or provide the correct answers. To keep track of all registered user, workers, and deployed tasks, the Handler manages multiple repositories which contain this information. Additionally, the Handler contains a scheduler, to be able to select suitable workers, based on the request. It also contains a connectivity checker, which is able to validate the connection to the workers. Eventually, a *Leave Monitor* is also part of the Handler, which helps to fulfill $\mathbf{R_{NF}4}$. This monitor verifies if the worker is still active. If no update is received from a worker for a specific amount of time, the worker is considered dead and removed from the system.

5.3.2 User

The proposed framework introduces a layered architecture for the user, similar to the broker. It is presented in Figure 5.14. The different layers are used to abstract and hide different details of the system. The lowest layer is a protocol adapter, responsible for the actual communication and transporting the messages. It sends the messages to the other entities in the system and parses the received messages for the upper layers of the user. On top of the adapter is the *Handler* component, which contains the system logic. This component implements the workflow of the protocol on the user side. It provides an API to send (and receive) different messages and acts as a bridge between the client application on top and the lower layer which actually sends the messages. Therefore, it

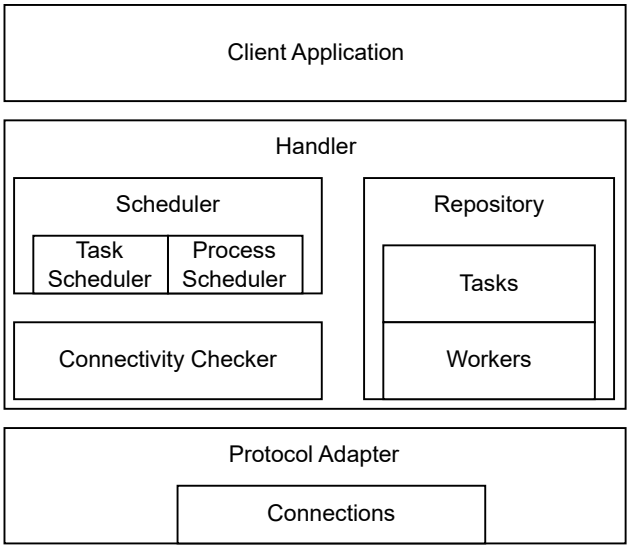


Figure 5.14: MetaEdge User Architecture

adds some necessary data to the different commands before forwarding the request to the adapter. The client application can use the handler via an API. The application can then implement their use case and use the MetaEdge system to do the offloading.

Similar to the broker, it contains a scheduler to select a worker for offloading. The information of available workers and tasks is stored in dedicated repositories. The handler additionally contains a *Worker Monitor*, to monitor the given workers. The worker monitor is responsible to update any changes of the worker, for example if the connectivity changes (e. g., increased hop count), or if the workload increases. With this monitor, the effort on the broker is reduced, because the users can update suitable workers and remove a worker from the list if the workload exceeds a specific threshold, in a self-organized way.

5.3.3 Worker

The third software artifact is the worker. The architecture of the worker component is illustrated in Figure 5.15. As the previous components, this is organized in a layered architecture, too, to abstract the different responsibilities of the layers. Again, the lowest layer is an adapter layer for the protocol messages. It transforms the outgoing messages for the underlying communication protocol and parses the incoming messages for the upper layers. Additionally, this layer connects to the broker and accepts connections from the users.

On top of this adapter is the layer for the *Worker Handler*. This component uses the interface to the adapter to send the messages and provides and interface for the actual *Task Handler*. It manages the deployed tasks and processes of the user. Therefore, it has to use different repositories, containing the deployed tasks and current processes of the users at the current worker, which also help to fulfill $\mathbf{R_F4}$. These caches are used to maintain the privileges for the specific task or process. A user which deploys a task at a worker becomes the owner of this task. Only the owner is allowed to remove the task from the worker. Other users can only use the task, but do not have the privileges to remove this

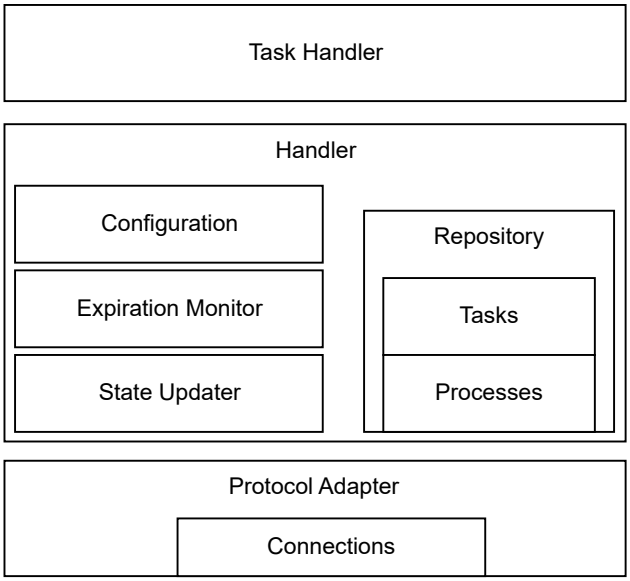


Figure 5.15: MetaEdge Worker Architecture

task. A similar approach is used for the processes. Since the actual execution time of an offloaded process is not beforehand for sure, a user may start the same process at different workers. However, he only needs the result of the fastest worker. Therefore, he cancels all further executions. To avoid that users cancel a process of another user, only the user who created the process is allowed to abort the execution.

One exception for the removal of a task is given to the worker. If a task is not used for a long time, he can remove the specific task by himself. Therefore, the Worker Handler contains an *Expiration Monitor* component, which monitors the tasks and their processes. Additionally, the Worker Handler component contains a *State Updater*, which monitors the current state of the worker and sends updates to the broker. These updates contain information about the workload, like used CPU, but can also contain some task-specific information.

When a worker registers at a broker, he has to provide a list, defined by the protocol, which contains system information of this worker, like supported runtimes, platforms, architecture, and other data. This information is then used by the broker and users to find suitable workers for a specific task.

The third layer is the *Task Handler*, which contains the actual implementation of the specific platform to handle the tasks. This component is only responsible to actually deploy the task and execute the processes. Since the deployment varies across different platforms, this component contains platform-specific code. When a new runtime or platform is added to the system, this is the only component which has to be modified for implementation. During the registration process, the information which platform the Task Handler implements has to get published. To keep this component simple, the interface only contains the functions to deploy and remove a task, as well as start and abort a process. Since this interface is very slim, it's easier to integrate different platforms and increases the usability, which contributes to the requirements $\mathbf{R_{NF}1}$ and $\mathbf{R_{NF}4}$.

5.4 Implementation

This section describes the implementation of the protocol and the core components of the system, according to their architecture. In Section 5.4.1, the details for the implementation of the protocol are presented. The implementation of the scheduler and connectivity checker are presented in Section 5.4.2 and 5.4.3, respectively. The framework for custom task handlers is explained in Section 5.4.5 in more details.

According to requirements $\mathbf{R_F2}$ and $\mathbf{R_F3}$, MetaEdge should be platform independent and be able to run on any device. Additionally, it has to provide fast performance according to $\mathbf{R_{NF}2}$ and be easily usable ($\mathbf{R_{NF}1}$). Furthermore, the overhead of the system and protocol should be low, in order to not restrict low-power devices. Many programming languages provide solutions to the mentioned requirements. For example, C and C++ are popular representatives for compiled languages. Compilers exist for various amount of platforms. Since they run on bare metal, no additional runtime is required on top. On the other side, programming languages like Java provide platform independent solutions, but require additional runtimes, for example the Java Virtual Machine (JVM). The JVM interprets the platform independent byte code for the specific machine. This still provides fast performance, but requires more resources. The execution performance is often only reduced slightly, but advantages such as faster prototyping or increased security often predominate. A relatively new language is Golang (Go), which is a compiled language and therefore provides the advantages of C/C++. However, it also promises faster prototyping and is a type safe language for increased security. In the environment of network and cloud services, Go is very well established. For example, Docker[1] and containerd[2], the predominant container virtualization solutions, are written in Golang. To profit of these advantages, Go was chosen as programming language. The platform independence is still given, since the code can get compiled for the different platforms.

5.4.1 Protocol

The protocol is implemented using gRPC Remote Procedure Calls (gRPC). gRPC is a framework for RPC which can run in any environment. This simplifies the prototype a lot, since the parsing of the messages does not have to be implemented. Additionally, it is based on HTTP/2 and protocol buffers. This provides an easy request-response communication and automatic serialization of messages. Therefore, a dedicated message ID to correlate messages is not necessary, because this is handled by the underlying architecture. The serialization provides an efficient binary data format, which improves the performance in comparison to a text-based format, for example XML.

In the prototype, gRPC is used as the adapter interface. As seen in Section 5.3, this is the lowest layer for the different components. This adapter could easily get exchanged to use another data format or implement a custom protocol definition.

For both, the worker and broker, an own service is defined which describes the incoming interaction possibilities with this entity. The user does not have an own service, since this entity initiates the interactions and only creates requests, but does not have to answer to incoming requests. Aside

[1] https://www.docker.com/
[2] https://containerd.io/

Listing 5.1 Excerpt of the task deployment definition on worker side

```
message DeployRequest {
    string userToken = 1;
    string taskId = 2;
    string platform = 3;
    bytes chunk = 4;
    string custom = 5;
}

message DeployResponse {
    string custom = 1;
}

service Worker {
    rpc Deploy(stream DeployRequest) returns (DeployResponse);
}
```

Listing 5.2 Custom data for a docker task handler

```
platform: docker
id: docker-image-id
custom:
    remote: docker-image-id
```

from that, the gRPC framework directly allows sending errors, without the need to define custom error messages. In Go, these errors can get handled after a function call instead of processing the result. For each other described message in Section 5.2, the according protobuf[3] messages get defined, which is the data format used in the framework.

For example, the deployment of a task to the worker is implemented as shown in Listing 5.1. Since the data of a task can have a huge size, a stream is used to keep the single messages small. The task itself is divided into multiple chunks, and sent as an array of bytes inside the *chunk* field. The message definition includes all necessary information defined by the protocol. Via the platform attribute, the workers for a task are matched. The additional task and platform specific information are provided via a custom field, which contains the data as a YAML encoded string, and can contain arbitrary data. As already mentioned, this message allows a user to deploy a task which he has locally on his side. However, this is not the only solution, where a task could be stored. Depending on the provider or task type, a task can be stored on a third party registry, e. g., the public Docker registry. In this case, not the actual task is sent with the message, but the information, how the worker can deploy the task and where he gets the task. For example, a task handler for the Docker platform has to check if the custom data contains a *remote* field (Listing 5.2). If this field is present, he has to pull the given image from the docker registry. This way, various amount of possible deployment techniques are allowed to be used with the protocol.

[3] https://protobuf.dev/

Algorithm 5.1 Basic scheduling algorithm

```
procedure SCHEDULE(W,C)
    suitableWorkers ← {}
    for all w ∈ W do
        suitable ← VerifyConstraints(w.C_w, C)
        if suitable then
            suitableWorkers ← suitableWorkers ∨ w
        end if
    end for
    target ← SelectOne(suitableWorkers)
    return target
end procedure
```

5.4.2 Scheduler

Both, the user and the broker have to implement a scheduler to select matching workers for the specific tasks. Basically, any kind of scheduler can be used in this case, since the scheduler itself is not tightly coupled to the protocol. In fact, since they are only responsible for selecting a suitable worker, a user application can use a custom scheduler, which is optimized for his exact use case. For example, the scientific community has already developed multiple different schedulers for edge computing, focusing on different aspects. In the dissertation of Breitbach [Bre22], different scheduling approaches are developed to enhance decentralized scheduling, do a proactive task placement, or focusing on energy consumption. While the user can highly benefit of the internal knowledge and optimization of the application, and therefore use a very specifically tailored scheduling approach, the broker does not have this specific knowledge and also can be used by several different users as proxy. Therefore, he has to select a suitable worker based on the given constraints in the *Deployment* or *Execution* message. As already presented, the user provides information about a suitable worker within these constraints.

A basic scheduling algorithm is developed in this case, outlined in Algorithm 5.1. The idea is to find a suitable worker, without any more knowledge of the system. With this scheduler, the broker is then able to ship the deployment or execution to a dedicated worker. In case of deployment, the broker returns the information about the worker to the user, so that the user could be able to establish a connection with the worker directly. In case of an execution request, the broker selects a matching worker, then ships the execution to the worker and waits for completion. After completion, he sends the result back to the user. The algorithm is explained below:

Let W be the set of registered workers at the broker, and C_w the constraints for a specific worker. Let C be the set of given constraints.

Inside the *SelectOne* function, again an arbitrary method could be used. In a simple case, this just uses the first available worker, a more sophisticated approach could select the fastest worker or another worker which matches with some previous context.

5.4.3 Connectivity Checker

The connectivity checker has to rely partly on the MetaEdge protocol, but also has to use a different technique to get the desired data. The connectivity checker has to monitor two aspects:

- latency to other participants
- hop count to other participants

Both could be realized using the ICMP protocol and sending ping messages to the target. But ICMP messages have no special priority and if the router is under high load, he could delay the processing of ICMP messages (or even drop them), resulting in a wrong RTT. However, it's not possible from a higher (layered) protocol to get the hop count to another participant. As a solution, the ICMP messages are used to determine the hop count by limiting the time-to-live and verifying if the message still receives the target. To get the RTT to the specific worker, a custom Ping message is sent via the MetaEdge protocol. The worker implementation immediately has to answer the message. If the worker is under high load and cannot immediately answer the ping request, but has to delay the response, the client will recognize this with a higher RTT than in reality. Fortunately, this does not impose significant bad effects, since a scheduler who optimizes the workload will select this worker less often in that case, therefore potentially reducing its load and normalizing his ping responses.

5.4.4 User authentication

Within this protocol, users and workers take distinct roles. However, this enables various attack points. In the following, one possible attack point is targeted and presented, how a minimal line of defense is implemented.

To request offloading, the user has to provide his ID, which can then get checked from the broker. A worker does not own a user ID, however when he receives a request, this user ID is contained in the message. At this point, a malicious worker can pretend that he is the same user and start offloading other processes with the user's identity. To mitigate this problem, the user ID will not get sent to the worker, but unique user-worker tokens are generated. These tokens are only valid for one user-worker connection. When both participants register, the broker creates an ID for them. The user knows his ID, but can fetch a list of the workers, which contains the IDs of available workers. However, the user will not know other user IDs. At the same time, the worker gets sent a hashed token, whenever a user registers. This way, the worker never knows other user IDs directly. In this scenario, the broker is used as a central trust instance, because he knows all user IDs and worker IDs. Therefore, this does not protect against a malicious broker. The pseudocode is shown in Algorithm 5.2.

5.4.5 Task Handler

Each worker, who joins the system, has to provide a task handler for his supported platform. The worker communicates via an interface with the task handler, which is shown in Listing 5.3. The interface itself is very small. It only provides methods for the deployment and execution of a task

Algorithm 5.2 Minimal user authentication

```
procedure WorkerRegistration
    id ← createId()
    tokens ← ∅
    for all u ∈ users do
        tokens ← hash(id, u.id)
    end for
    return tokens
end procedure
procedure UserRegistration
    id ← createId()
    for all w ∈ workers do
        token ← hash(w.id, id)
        w.SendUserToken(token)
    end for
    return id
end procedure
procedure Deployment(worker, userId)
    token ← hash(w.Id, userId)
    worker.Offload(token)
end procedure
```

Listing 5.3 Interface for the Task Handler

```
type TaskHandler interface {
    Deploy(Task) error
    Remove(taskId) error

    Execute(Process) error
    Abort(processId) error
}
```

and process. The details about the implementation depend on the underlying platform of the task. Various task handler were implemented in this thesis to prove the functionality. In the following sections, some of them are presented.

Python

This task handler is able to execute simple python scripts. Here, the task data is an archive with the python file. Additionally, it needs an entry point into the task, which is given via the custom data of the protocol. When the task handler deploys the task, it unwraps the archive and stores the content on disk. To execute a process, it loads the file given as entry point. Additionally, each request contains an array of strings as parameters. These parameters are used to start the underlying process. The given task itself is responsible to correctly parse the parameters, as the other components do not have these details.

Listing 5.4 Export a Docker image

```
docker save image:latest | gzip > image_latest.tar.gz
```

Listing 5.5 Task description of an OpenFaaS function

```
version: 1.0
provider:
    name: openfaas
    gateway: http://127.0.0.1:8080
functions:
    hello-world:
        lang: golang-middleware
        handler: ./hello-world
        image: hello-world:latest
```

Docker

The implementation of this task handler is able to run simple docker containers. It uses the Docker Engine, which has a Go SDK available, to deploy and execute the containers. When a user wants to deploy a Docker image, the image has to be sent as the task data via the protocol. Therefore, the image first has to be exported into an archive, which can be done with a command as shown in Listing 5.4. The Dockerfile of the image already specifies the entry point, hence no custom entry point has to be sent with the protocol. However, such an image can get very big, which delays the deployment because many data has to be sent, especially if the deployment is done via the broker. Alternatively, a Docker image can get loaded via a central registry, e. g., the Docker Hub[4]. In this case, the image is not transferred via the protocol, but the task contains a custom field, which specifies which image should get pulled. The task handler has to parse the custom field to get this information. When a process should get executed, the task handler creates a container of the image. Again, the parameters are added, but the actual implementation of the task is responsible to correctly parse the parameters, since they are only given as strings.

OpenFaaS

Another platform which was integrated in the scope of this thesis was OpenFaaS. As mentioned in Section 3.1.2, OpenFaaS is a popular platform for serverless functions. In this case, it is implemented using faasd[5], which enables to run OpenFaaS functions without Kubernetes. To describe functions, OpenFaaS uses YAML Ain't Markup Language (YAML). Functions are deployed with a YAML file, which contains the details of the function. Therefore, this file is used as the task data and sent via the protocol. The task handler reads the file and uses faasd to deploy the function. An exemplary description is show in Listing 5.5 In order to be able to deploy this function with faasd, the user or

[4] https://hub.docker.com/
[5] https://github.com/openfaas/faasd

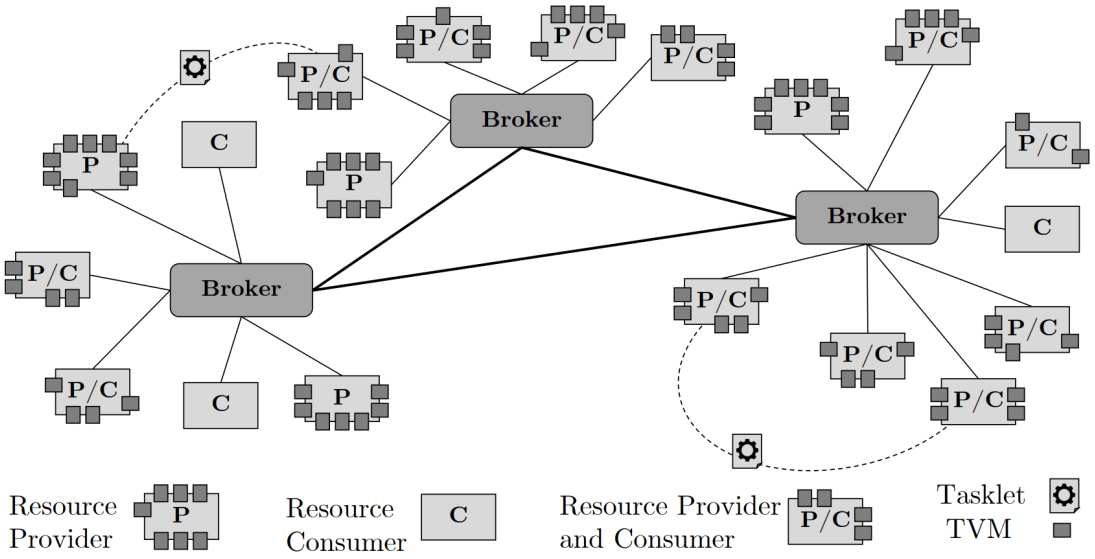


Figure 5.16: The Tasklet system with consumers, producers, and brokers.

another third party first has to publish the function, which uploads the resulting image to the docker registry.

Tasklet

The Tasklet system is another computation offloading system, introduced in [SEP+16]. It defines producers and consumers, which connect to a broker, illustrated in Figure 5.16. Consumers want to offload code and producers perform the computation. However, in this system, consumers can also be producers. The orchestration is done via the broker, which tells a consumer which producers have available resources. In order to integrate the Tasklet system into MetaEdge, the architecture had to get adapted. The end user cannot connect directly with a broker from the Tasklet system, but has to connect with the MetaEdge broker. On the other hand, the task handler of the worker has to integrate a complete Tasklet system and connect to the middleware. When a user deploys a Tasklet, the task handler receives the data of the Tasklet. But the execution of a Tasklet is only started, when an execution request is received. Then the task handler forwards the Tasklet to a producer and receives the result. The result is just a byte array, which is then returned to the end user. The end user's application is responsible for parsing the result, since the MetaEdge protocol has no knowledge about the Tasklet implementation itself.

6 Evaluation

In this chapter, the proposed MetaEdge framework and protocol will be evaluated, using the implemented prototype. During the evaluation, the goal is to achieve the required interoperability from requirement $\mathbf{R_F2}$ and show the interoperability between different users and workers. Additionally, the overhead of the protocol is measured to evalute the goal of requirement $\mathbf{R_{NF}2}$. For the evaluation, two test beds were set up that are described in Section 6.1. Multiple workers supporting different platforms and implementations were set up and joined the network, creating a heterogeneous environment. This environment was used to simulate the different kind of providers in an edge computing scenario. In Section 6.2, the results of the evaluation are shown. Finally, the results from the evaluation are discussed in Section 6.3.

6.1 Experimental Setup

This evaluation is done with a virtual test bed. Within this test bed, the goal is to create a heterogeneous edge environment with several different devices. The devices support different runtimes and platforms used in edge computing for academic research as well as in industry. The virtual setup is done with Vagrant[1], a ruby-based application for the creation and configuration of lightweight, reproducible, and portable development environments. This allows for a controlled environment with no side effects. The reproducibility is given via the Vagrantfile, as it follows the Infrastructure as code (IaC) paradigm. In Listing 6.1 is shown, how an exemplary device is created. It gets 2 GB of memory and 1 CPU core to use. Additionally, a static IP address is created to communicate in the private network, as well as port forwarding is enabled to also forward requests from the host system to the guest. The end-device, a powerful Debian server, creates several VMs based on the given configuration. While the setup is reproducible in this way and different devices can get simulated, the network itself is homogenous since all virtualized resources are located on the same physical server. Each device is reachable within one hop, shown in Figure 6.1, and the latency is below 0.5 ms, which does not represent a real-world scenario. However, for the first evaluation of the protocol and framework, this is acceptable. In Table 6.1, the system specification of the host system, the Debian server, are listed.

Processor	12 core 3.60GHz Intel Xeon CPU E5-1650 v4
RAM	32 GB
Storage	450 GB

Table 6.1: Specifications of the host system used for virtual setup

[1] https://www.vagrantup.com/

Listing 6.1 Exemplary deployment description of the broker in the virtual setup

```
Vagrant.configure("2") do |config|

config.vm.define "broker" do |broker|
    broker.vm.box = "debian/buster64"
    broker.vm.hostname = "broker"
    broker.vm.network "private_network", ip: "192.168.33.10"
    broker.vm.network "forwarded_port", guest: "9000", host: "9000"
    broker.vm.provider :libvirt do |libvirt|
        libvirt.memory = 2048
        libvirt.cpus = 1
    end

    broker.vm.provision :shell, path: "./hack/vagrant/provision_broker.sh", env: {"PORT" => "
9000"}
end
```

Name	Short Description	Platform
WP1	Worker which implements the python runtime	Python
WP2	Worker which implements the python runtime	
WD	Worker which has Docker installed	Docker
WF	Worker which uses faasd to deploy OpenFaaS functions	OpenFaaS
WT	Worker which can run the Tasklet system	Tasklet
U1	User of the MetaEdge system	
U2	User of the MetaEdge system	
B	Broker of the MetaEdge system	

Table 6.2: Virtual test bed of integrated platforms and devices into MetaEdge

For evaluation, several different edge environments are investigated, listed in Table 6.2. For each edge environment, an own worker is created which runs in its own VM. Each VM can use one CPU core, has 20 GB storage and 2 GB RAM. Each worker uses one task handler, implemented in Section 5.4.5, simulating an edge platform.

To counteract the issues of the virtual test bed, a second setup is created, which specifically adds some heterogeneity to the network. In this setup, only two different workers which support the Python runtime and OpenFaaS are added. The deployment is done on three instances in the Google Cloud Platform (GCP), see Table 6.3. On one instance, a broker, user, and worker are hosted, which have low latency. This instance is called “machine-A” is placed in the region “us-west”. On two distant instances, called “machine-B” and “machine-C” only a worker is hosted which connects to the system. This instance is placed in the region “us-central”. The latency between these instances is 40 ms, which may not perfectly represent an edge infrastructure but provides suitable insights to a scenario, where a distance server connects to the system.

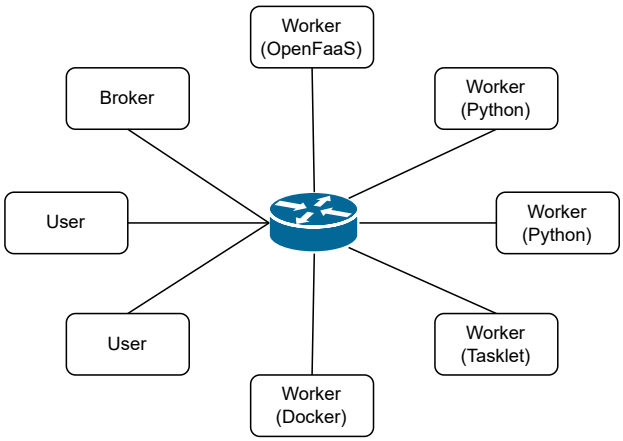


Figure 6.1: Connectivity of the deployed devices in the virtualized setup

Device name	machine-A	machine-B	machine-C
Processor	2 vCPU	2 vCPU	2 vCPU
RAM	1 GB	1 GB	1 GB
Storage	10 GB	10 GB	10 GB
Location	us-central	us-west	us-west
Type	Broker, User, Worker	Worker	Worker

Table 6.3: Specifications of the machines in the cloud setup

6.2 Results

In this section, the results of the different experiments are presented. First, the framework is used to ship the complete offloading process to the broker, which is then responsible to distribute and manage the different offloading requests (Section 6.2.1). Then, in Section 6.2.2, the users use their own scheduler to distribute their tasks to specific workers. Both experiments show the feasibility of the proposed protocol while the second process highlights the self-organizing features of the protocol. To also highlight that the system is able to handle centralized and decentralized offloading concurrently, both methods get combined in Section 6.2.3. In Section 6.2.4 and Section 6.2.5, the decision process is extended to multiple constraints. This does not only consider the supported

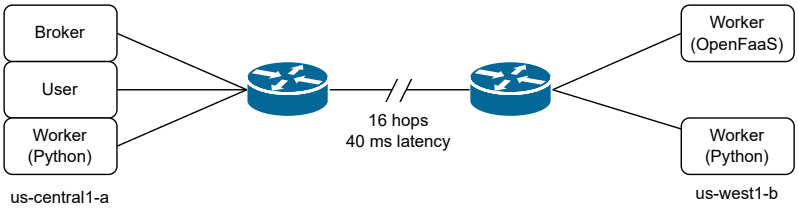


Figure 6.2: Connectivity of the deployed devices in the Google Cloud setup

Task No.	Platform	Deployed by
1	Python3	user 1
2	Docker	user 2
3	OpenFaaS	user 2
4	Tasklet	user 2

Table 6.4: Task deployments used for evaluation

runtime, but many different features. Here, dynamic properties such as available CPU resources or the hop distance are considered. To allow for network heterogeneity, this experiment is not conducted in a virtualized environment, but on the setup made with GCP.

6.2.1 Centralized Offloading

The first experiment validates the broker of the protocol, in particular the suitability of a centralized deployment and execution process via the broker. The basic setup is used as described above. One broker is started, together with different workers, which all provide a different configuration to execute offloaded processes. Additionally, two users are started, which will deploy and execute tasks. The users are instructed to only communicate with the broker, but not with the workers directly. After each participant has registered at the broker, the first user deploys a task via the broker. The different tasks are listed in Table 6.4.

After the deployment, both users start offloading processes and sending requests to the broker. Even though the broker does not know the details of the task itself, he is able to forward the requests to the respective workers, since the required platform for the task is provided via the protocol. With his built-in scheduler, he selects a worker with a suitable runtime to deploy the task and then selects the worker with the deployed task to execute the processes.

Figure 6.3 visualizes the offloading path for both users with an exemplary python task. Only the first user deploys the task, while both users are able to offload a process. The broker selects the worker and orchestrates the distribution of tasks and processes.

In Figure 6.4, a screenshot of the logs of the broker and the affected worker, which offloads the python task, is shown. The logs list that multiple workers register at the broker. The broker (Figure 6.4a) receives a request for a task deployment and selects a worker. After, he receives two requests to execute processes for the given task, which he further forwards to the selected worker. The logs of the worker (Figure 6.4b) record that a task was received and deployed, and then two processes were started.

6.2.2 Decentralized Offloading

To further emphasize the self-organizing capabilities of the protocol, the users do now schedule the offloading and processing themselves, without the broker, presented in Figure 6.5. The users and workers both register at the broker, but then start to mainly communicate directly to each other. This reduces the amount of messages and increases the performance, since the messages are directly sent

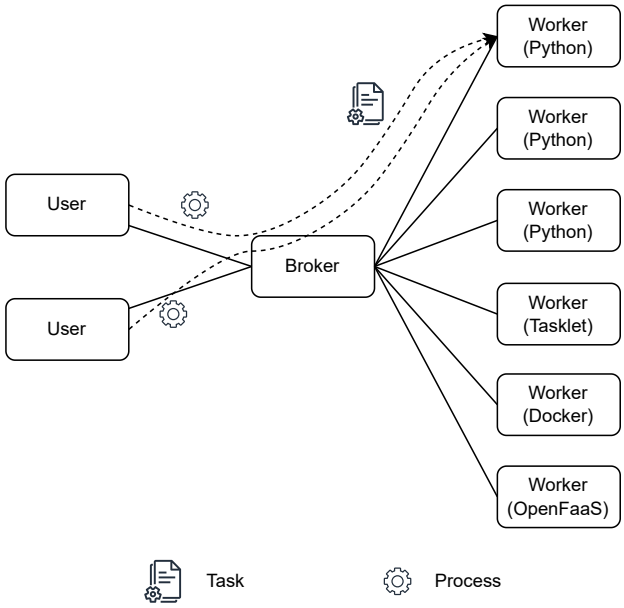


Figure 6.3: Illustration of the centralized offloading capabilities

```
Start broker ...
Broker listening address={"IP":"::","Port":9000,"Zone":""}
Registered worker id=06b27ab3-412c-4957-8bc7-18f9a73607e1
Registered worker id=4d24fe83-3f66-447d-b300-c63484b8b72c
Registered worker id=f7b38ecc-a5bc-4445-9769-278b9a361bb3
Registered worker id=18f0f119-20e8-4bf5-a085-79517f312c4b
Registered worker id=1160dd07-0332-485c-98fd-3613af107319
Registered worker id=b5b797aa-93bd-4d0e-bae6-4c642055894f
Registered user id=a0309e8b-0fca-421c-9e9d-bfa71d428213
Registered user id=4ba61802-cb95-4323-9bcf-7f07def18b23
Forward task deployment task id=python-fibonacci user id=a0309e8b-0fca-421c-9e9d-bfa71d428213
Selected worker Address=192.168.33.102:9502 ID=4d24fe83-3f66-447d-b300-c63484b8b72c
worker informs about task deployment task id=python-fibonacci worker id=4d24fe83-3f66-447d-b300-c63484b8b72c
Forward process execution process id=d7a5fedf-1edb-4581-ae33-53c608ea5e97 task id=python-fibonacci user id=a0309e8b-0fca-421c-9e9d-bfa71d428213
Selected worker Address=192.168.33.102:9502 ID=4d24fe83-3f66-447d-b300-c63484b8b72c
Forward process execution process id=7cb126e7-4752-40da-89c8-fced9b6540b1 task id=python-fibonacci user id=4ba61802-cb95-4323-9bcf-7f07def18b23
Selected worker Address=192.168.33.102:9502 ID=4d24fe83-3f66-447d-b300-c63484b8b72c
```

(a) Screenshot of the logs from the broker orchestrating the offloading

```
Start worker ...
Create Python Task Handler
Worker service listening address={"IP":"::","Port":9502,"Zone":""}
Registered id=4d24fe83-3f66-447d-b300-c63484b8b72c
Deploy Task id=python-fibonacci platform=python3
Execute process id=d7a5fedf-1edb-4581-ae33-53c608ea5e97 task=python-fibonacci
Execute process id=7cb126e7-4752-40da-89c8-fced9b6540b1 task=python-fibonacci
```

(b) Screenshot of the logs from the affected worker during the offloading

Figure 6.4: Screenshots of the broker and affected worker during an exemplary offloading workflow

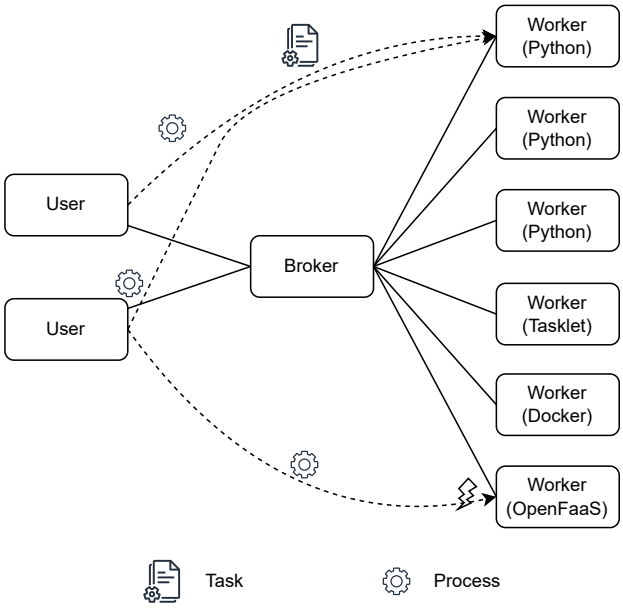


Figure 6.5: Illustration of the decentralized offloading capabilities

to the ultimate receiver. Additionally, the users have more control about which worker performs the computation. To further evaluate the robustness of the protocol, one user is instructed to select an unsuitable worker for offloading. Here, the worker should respond with an error message, indicating that he cannot offload the task because of missing runtimes.

A screenshot of the logs is again shown in Figure 6.6. Figure 6.6a presents that the broker just registers that the worker wants to deploy a task and can update his internal cache. The logs of the affected worker are shown in Figure 6.6b. As before, this worker deploys the task and performs the computations. An example of a failed deployment is shown in Figure 6.6c, where an unsuitable worker was selected. However, this did not crash the system, but the worker was able to respond with the corresponding error message.

6.2.3 Mixed Offloading

In this test, the user instructs the broker to offload a task, but then uses the direct communication link to the respective worker to improve the performance by eliminating the additional hop via the broker when executing a task. Again, the python task is selected as test case in this illustration (Figure 6.7a). The message flow is shown in Figure 6.7b.

A screenshot of the logs is shown if Figure 6.8. As can be seen, the broker forwards an execution request and selects a suitable worker (Figure 6.8a). The execution request is then sent directly to the worker, therefore the broker has no log entry to forward the execution request. The worker receives the deployment and execution request (Figure 6.8b).

```
Start broker ...
Broker listening address={"IP":"::","Port":9000,"Zone":""}
Registered worker id=06b27ab3-412c-4957-8bc7-18f9a73607e1
Registered worker id=4d24fe83-3f66-447d-b300-c63484b8b72c
Registered worker id=f7b38ecc-a5bc-4445-9769-278b9a361bb3
Registered worker id=18f0f119-20e8-4bf5-a085-79517f312c4b
Registered worker id=1160dd07-0332-485c-98fd-3613af107319
Registered worker id=b5b797aa-93bd-4d0e-bae6-4c642055894f
Registered user id=a0309e8b-0fca-421c-9e9d-bfa71d428213
Registered user id=4ba61802-cb95-4323-9bcf-7f07def18b23
worker informs about task deployment task id=python-fibonacci worker id=f7b38ecc-a5bc-4445-9769-278b9a361bb3
```

(a) Screenshot of the logs from the broker during orchestration

```
Start worker ...
Create Python Task Handler
Worker service listening address={"IP":"::","Port":9501,"Zone":""}
Registered id=f7b38ecc-a5bc-4445-9769-278b9a361bb3
Deploy Task id=python-fibonacci platform=python3
Execute process id=10447de3-95bf-4390-bd47-d73671371ea0 task=python-fibonacci
Execute process id=bd2b61af-5a09-4272-94b0-9cf12f2b916f task=python-fibonacci
```

(b) Screenshot of the logs from the affected worker during the offloading

```
Start worker ...
Create Faasd Task Handler
Worker service listening address={"IP":"::","Port":9505,"Zone":""}
Registered id=b5b797aa-93bd-4d0e-bae6-4c642055894f
Deploy Task id=python-fibonacci platform=python3
Failed deployment
```

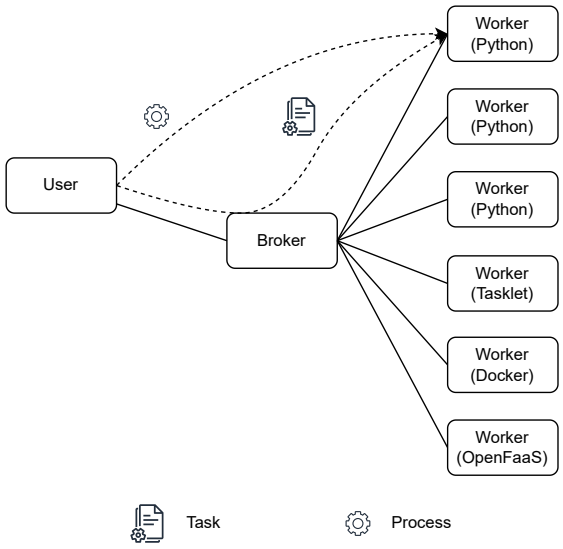
(c) Screenshot of the logs from the worker of a failed deployment

Figure 6.6: Screenshots of the broker and affected worker during an exemplary offloading workflow

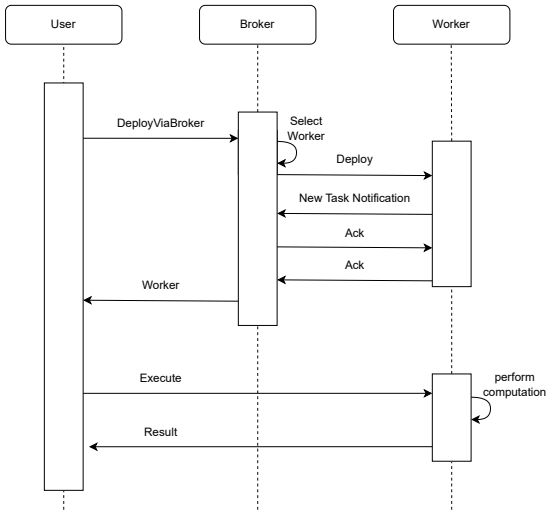
6.2.4 Network Context

This test verifies the ability to select different workers based on the network context, meaning hop distance or latency. To achieve a better real-world scenario with different latencies, the test bed from Figure 6.2 is used. In order to select workers based on the network context, two specific schedulers are implemented: The first scheduler selects the worker with minimal hop distance while the second scheduler selects the worker with lower latency. In a first run, only the broker and user are started on “machine-A”, but not the local worker. This ensures that the distant worker on “machine-B” is at least selected once. Then, the worker on “machine-A” joins the system and a second run is started. Since the network connectivity should be checked during registration, the broker should know that the new worker is closer in terms of latency and hop distance and hence select this worker. In Figure 6.9, the setup is shown. The offloading to the distant worker is illustrated in (1). Here, the local server has not joined the system, yet. In (2), the local worker participates in the system. Therefore, this worker can be used for offloading.

The logs are presented in Figure 6.10. The broker forwards the previously used python-task for offloading. Since no other worker is available, in both cases the schedulers select the remote worker. Then, the worker on “machine-A” joins the system. Since the network connectivity is evaluated during the registration proces, the broker now knows that this worker is very close and has little



(a) Illustration of the mixed offloading scenario



(b) Illustration of the mixed offloading message flow

Figure 6.7: Illustration of the mixed offloading test case

latency. The second deployment and offloading process is started now and with this information, the schedulers select in both cases the local worker, proving that the network information are considered.

6.2.5 Workload context

Eventually, the following test uses the workload context of the workers to select the worker with the lowest CPU usage. This setup is again based on the virtual test bed, presented in Figure 6.1 to exclude other heterogeneities in the system, however only two identical workers join the system. Both workers implement a python runtime, since a python task will be offloaded again. The first worker, called “worker-A”, has no additional configuration. The second worker, “worker-B”, will

```
Start broker ...
Broker listening address={"IP":"::","Port":9000,"Zone":""}
Registered worker id=87c1be0f-af20-443a-8e52-20b14cb97a4c
Registered worker id=a8ca4bd9-b140-41d2-ac36-b69470b82048
Registered worker id=cff971be-3d5d-4926-b92d-6319335d46c3
Registered worker id=f004ca9e-69d3-4d0e-bf7a-70dc50b0bc7b
Registered worker id=0c9e3707-2428-4923-be05-3b48bbf41f6d
Registered worker id=f4c1b6be-6ddf-42d6-9511-6d8a0270ea83
Registered user id=3078110c-5c18-4c7a-83f1-4e748c94657a
Forward task deployment task id=python-fibonacci user id=3078110c-5c18-4c7a-83f1-4e748c94657a
Selected worker Address=192.168.33.101:9501 ID=87c1be0f-af20-443a-8e52-20b14cb97a4c
worker informs about task deployment task id=python-fibonacci worker id=87c1be0f-af20-443a-8e52-20b14cb97a4c
```

(a) Screenshot of the logs from the broker during orchestration

```
Start worker ...
Create Python Task Handler
Worker service listening address={"IP":"::","Port":9501,"Zone":""}
Registered id=87c1be0f-af20-443a-8e52-20b14cb97a4c
Deploy Task id=python-fibonacci platform=python3
Execute process id=cef8ea46-f905-4474-b10b-f1bf933b61f8 task=python-fibonacci
```

(b) Screenshot of the logs from the affected worker during the offloading

Figure 6.8: Screenshots of the broker and affected worker during an exemplary offloading workflow where the user sends a deployment request to the broker but does the execution with a direct connection to the worker

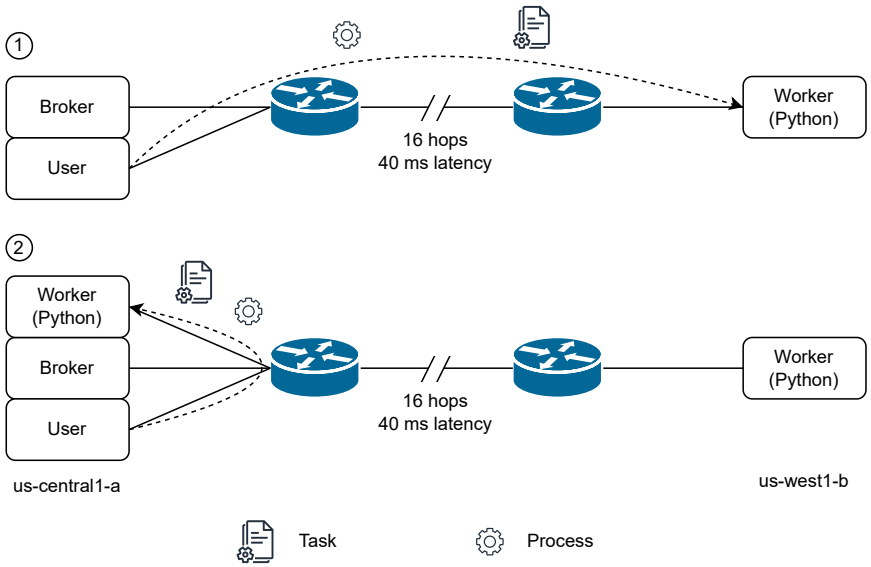


Figure 6.9: Illustration of the setup when using the network context

```
Start broker ...
Broker listening address={"IP":"::","Port":9000,"Zone":""}
Registered worker id=d8364353-0fa3-47f3-ae5c-b7095e57b66b
Registered user id=c0005c3d-e306-4320-a7af-6e6c0517972d
Forward task deployment task id=python-fibonacci user id=c0005c3d-e306-4320-a7af-6e6c0517972d
Selected worker Address=34.83.28.111:9500 ID=d8364353-0fa3-47f3-ae5c-b7095e57b66b
worker informs about task deployment task id=python-fibonacci worker id=d8364353-0fa3-47f3-ae5c-b7095e57b66b
Forward process execution process id=54f3c069-c889-45a0-b40a-5e78ecc31996 task id=python-fibonacci user id=c0005c3d-e306-4320-a7af-6e6c0517972d
Selected worker Address=34.83.28.111:9500 ID=d8364353-0fa3-47f3-ae5c-b7095e57b66b
Registered worker id=9fb96430-30c4-45d6-bd37-adbc838120d9
Forward task deployment task id=python-fibonacci user id=c0005c3d-e306-4320-a7af-6e6c0517972d
Selected worker Address=127.0.0.1:9500 ID=9fb96430-30c4-45d6-bd37-adbc838120d9
worker informs about task deployment task id=python-fibonacci worker id=9fb96430-30c4-45d6-bd37-adbc838120d9
Forward process execution process id=84b0c1bc-088a-4c42-8b11-23d007bf2347 task id=python-fibonacci user id=c0005c3d-e306-4320-a7af-6e6c0517972d
Selected worker Address=127.0.0.1:9500 ID=9fb96430-30c4-45d6-bd37-adbc838120d9
```

(a) Screenshot of the logs of the broker who schedules the offloading to the workers

```
Start worker ...
Create Python Task Handler
Worker service listening address={"IP":"::","Port":9500,"Zone":""}
Registered id=d8364353-0fa3-47f3-ae5c-b7095e57b66b
Deploy Task id=python-fibonacci platform=python3
Execute process id=54f3c069-c889-45a0-b40a-5e78ecc31996 task=python-fibonacci
```

(b) Screenshot of the logs of the remote worker

```
Start worker ...
Create Python Task Handler
Worker service listening address={"IP":"::","Port":9500,"Zone":""}
Registered id=9fb96430-30c4-45d6-bd37-adbc838120d9
Deploy Task id=python-fibonacci platform=python3
Execute process id=84b0c1bc-088a-4c42-8b11-23d007bf2347 task=python-fibonacci
```

(c) Screenshot of the logs of the local worker

Figure 6.10: Screenshots of the logs of the broker and the local and remote worker

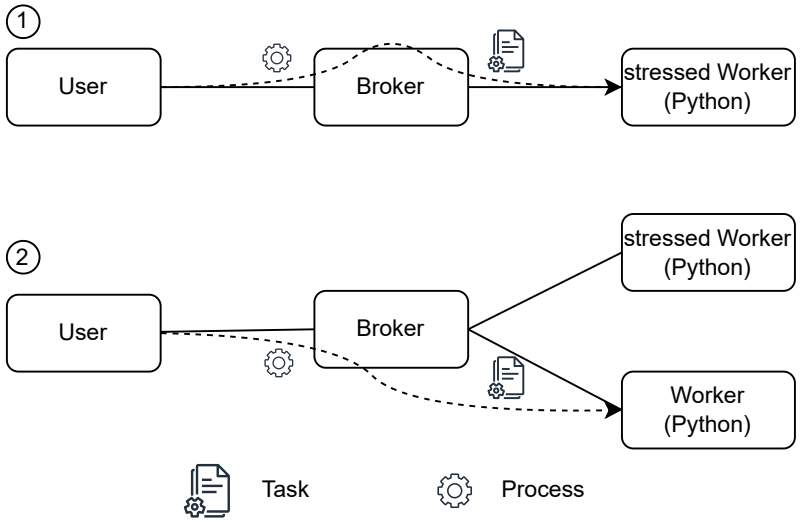


Figure 6.11: Illustration of the setup when using the workload context

be set under high stress to utilize roughly 50 % of his CPU resources. Similar to Section 6.2.4, two runs are started. In the first run, only “worker-B” join the system to be used as offloading instance. In the second run, “worker-A” participates, too. The worker is selected at the broker, whose scheduler uses the worker with the lowest CPU usage. Since “worker-A” does not process any other tasks, his workload should be lower and hence get selected as offloading instance. The setup is also shown in Figure 6.11.

```
Start broker ...
Broker listening address={"IP":"::","Port":9000,"Zone":""}
Registered worker id=cf07e6ee-dee4-48c1-8539-5290dcab58a7
Registered user id=32566c83-c463-48fc-9f07-e8849edf0dc1
Forward task deployment task id=python-fibonacci user id=32566c83-c463-48fc-9f07-e8849edf0dc1
Selected worker Address=192.168.33.103:9503 ID=cf07e6ee-dee4-48c1-8539-5290dcab58a7
worker informs about task deployment task id=python-fibonacci worker id=cf07e6ee-dee4-48c1-8539-5290dcab58a7
Forward process execution process id=e43599e2-aadd-4355-a06c-983aa7433161 task id=python-fibonacci user id=32566c83-c463-48fc-9f07-e8849edf0dc1
Selected worker Address=192.168.33.103:9503 ID=cf07e6ee-dee4-48c1-8539-5290dcab58a7
Registered worker id=4da779d3-747e-48b0-9742-7c7fe1e879a3
Forward task deployment task id=python-fibonacci user id=32566c83-c463-48fc-9f07-e8849edf0dc1
Selected worker Address=192.168.33.101:9501 ID=4da779d3-747e-48b0-9742-7c7fe1e879a3
worker informs about task deployment task id=python-fibonacci worker id=4da779d3-747e-48b0-9742-7c7fe1e879a3
Forward process execution process id=805de462-eb11-495a-9a45-ea4366a1a26b task id=python-fibonacci user id=32566c83-c463-48fc-9f07-e8849edf0dc1
Selected worker Address=192.168.33.101:9501 ID=4da779d3-747e-48b0-9742-7c7fe1e879a3
```

(a) Screenshot of the logs of the broker who schedules the offloading to the workers

```
Start worker ...
Create Python Task Handler
Worker service listening address={"IP":"::","Port":9503,"Zone":""}
Registered id=cf07e6ee-dee4-48c1-8539-5290dcab58a7
Deploy Task id=python-fibonacci platform=python3
Execute process id=e43599e2-aadd-4355-a06c-983aa7433161 task=python-fibonacci
```

(b) Screenshot of the logs of the worker under stress

```
Start worker ...
Create Python Task Handler
Worker service listening address={"IP":"::","Port":9501,"Zone":""}
Registered id=4da779d3-747e-48b0-9742-7c7fe1e879a3
Deploy Task id=python-fibonacci platform=python3
Execute process id=805de462-eb11-495a-9a45-ea4366a1a26b task=python-fibonacci
```

(c) Screenshot of the logs of the worker with no further stress

Figure 6.12: Screenshots of the logs of the broker and the workers when using minimal CPU usage

As can be seen in Figure 6.12, the broker selects the stressed worker, when this is the only one available (Figure 6.12a and Figure 6.12b). As soon as the other worker joins, the new one is selected since his workload is lower (Figure 6.12c).

6.2.6 Offloading Performance

Finally, the duration time for offloading processes is investigated. In this test, the delay at the user between sending the request and receiving the result is measured, as well as the time the process actually takes at the worker. To also show the influence of network latencies, the cloud setup with GCP (Figure 6.2) is used, and the processes are offloaded to the distant worker. To not show only the performance of one specific task type, two different platforms are presented, namely the integration with Python and OpenFaaS. Both tasks implement a simple algorithm to compute Fibonacci numbers. An example is shown in Listing 6.2, which contains the essentials for the task, implemented in Python. The execution request is sent with the parameter “1”, to immediately have the result available. First, a baseline of the execution durations is shown in Figure 6.13. On average, 11.19 ms are needed to compute the result with the plain Python task and 58.18 ms are necessary to get the result with the OpenFaaS solution.

In Figure 6.14, the different delays are shown when the python task is offloaded, as well as the latency between the devices. The performance is evaluated with a direct connection link between the user and the worker (Figure 6.14a, (1)), as well as using the broker as proxy (Figure 6.14b, (2)). In (1), the worker needs on average 12.17 ms to compute the result. The user receives the

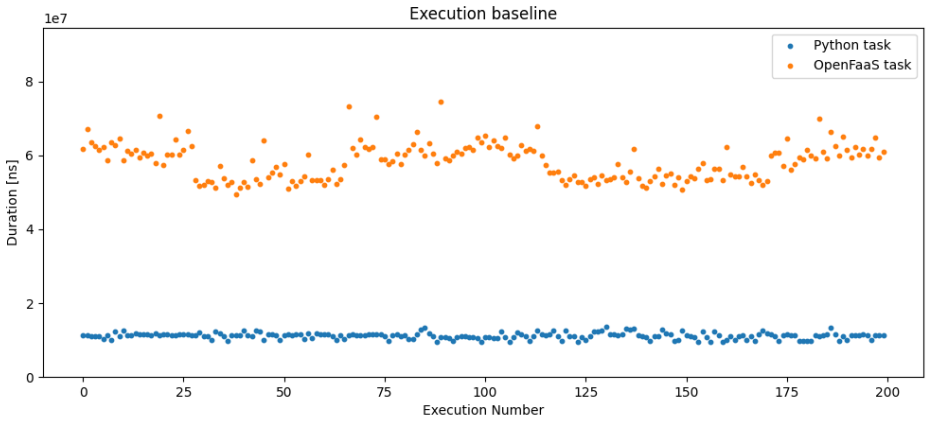
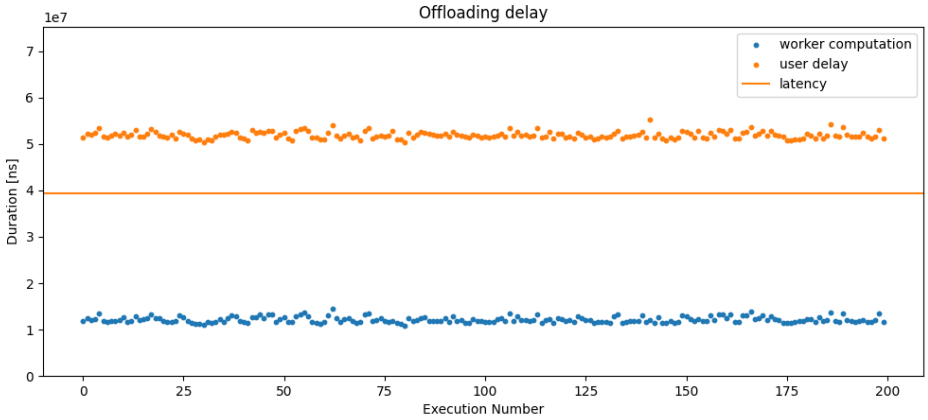
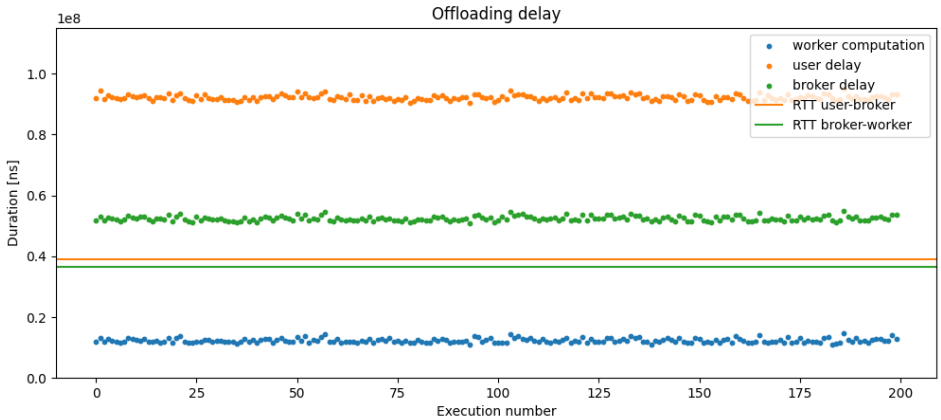


Figure 6.13: Duration of the local executions



(a) Computation delay with a direct connection



(b) Computation delay with broker

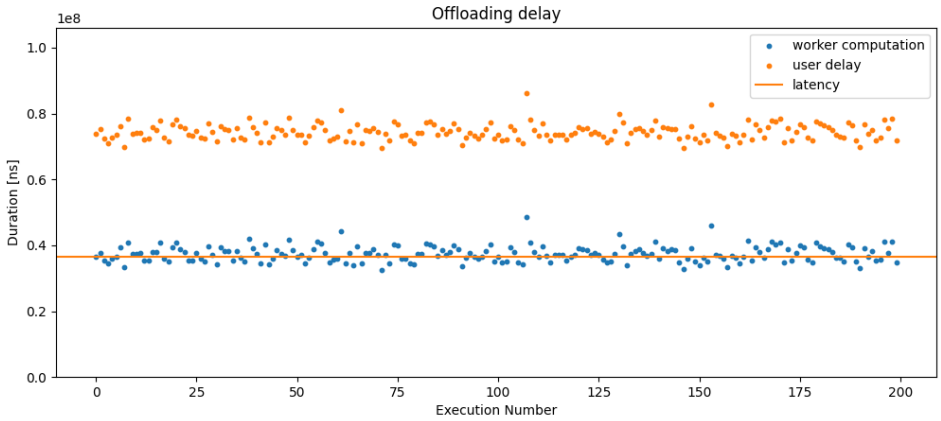
Figure 6.14: Computation delay when offloading the Python task

Listing 6.2 Exemplary task to compute Fibonacci numbers, implemented in Python

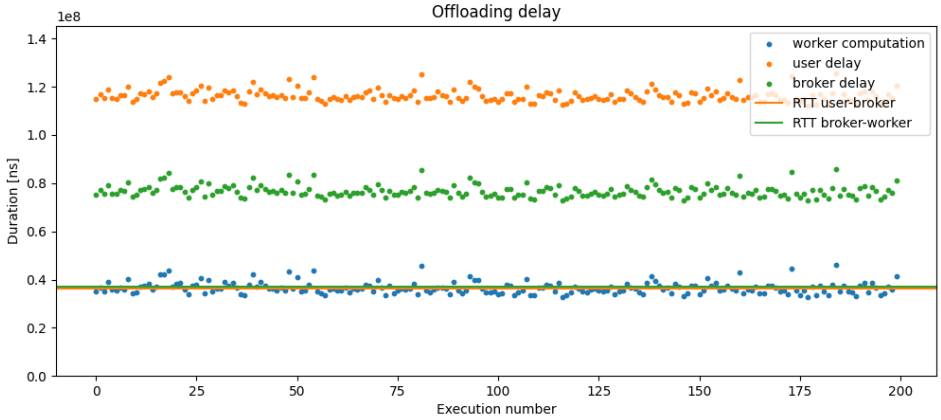
```
import sys
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
if __name__ == "__main__":
    n = int(sys.argv[1])
    result = fibonacci(n)
    print(f"{result}")
```

computed result after 51.92 ms. Most of the delay happens due to the latency, which is 39.38 ms on average. After excluding the RTT, the difference between the duration of the worker's computation and receiving the result is 0.36 ms. With the given latency, the overhead of the MetaEdge protocol and framework is only 0.92 %, compared to the execution time on worker side. Assuming a delay of 10 ms, the overhead is still only 3.62 %. In (2), the worker computes the result on average on 12.31 ms. However, the result is received by the user only after 92.22 ms. This big latency is introduced, because the broker has to forward the result, which basically doubles the latency. The RTT between the broker and the worker is 36.37 ms and the RTT from the user to the broker is 39.16 ms. Excluding the latency from the delays, the broker adds a delay of 3.73 ms and the delay at the user is in total 4.37 ms. The overhead in this case is 10.26 % at the broker and 11.16 % at the user. This high number probably results from the scheduling algorithm at the broker, because for each request, the broker has to find a suitable worker. If the RTT is set to 10 ms, the overhead at the broker is 37.31 % and at the user 43.72 % in total.

In Figure 6.15, the different delays for the OpenFaaS task are shown, together with the latencies between the participants. A direct connection link between the user and the worker is used in Figure 6.15a ((3)), and the broker proxies the offloading requests in Figure 6.15b ((4)). In (3), the worker computes the result on average in 37.40 ms, which is received from the user after a total delay of 74.36 ms. The RTT of the messages is 36.45 ms. Again, after excluding the latency, the delay introduced due to MetaEdge is 0.51 ms, which is an overhead of 1.40 %. Reducing the latency to 10 ms, the overhead increases to 5.10 %. In the test case (4), the worker computes the result in 36.60 ms, but the result is only received after 116.36 ms. As above, the latency is doubled because the broker proxies the request to another destination. The RTT from the broker to the worker is 36.93 ms, and the latency from the user to the broker is 36.12 ms. Hence, the broker adds a delay of 3.17 ms and the total delay due to MetaEdge on user side is 6.70 ms. Here, the overhead because of the broker is 8.58 % and the total overhead after the user receives the result is 18.56 %. These results match well with the previous test, which suggest that using the broker as proxy harm the performance.



(a) Computation delay with a direct connection



(b) Computation delay with broker

Figure 6.15: Computation delay when offloading the OpenFaaS task

6.3 Discussion

As shown in the evaluation, MetaEdge allows the integration of different platforms and runtimes into an edge environment and to offload a various amount of different tasks. The protocol is able to provide the worker capabilities and consider specific needs of the users, in order to choose a valid offloading target. Since the protocol allows for custom task data, it is easily extendable and additional providers can get integrated. Additionally, this framework allows very slim devices to participate in an edge environment, since it is not backed on a whole system, but basically relies on the protocol, which can get integrated by different endpoints.

By using different schedulers, it is possible to optimize for specific needs. Since the users are free to use their own scheduler, they have more control where their data actually gets shipped.

The best performance is achieved when a user directly connects to a worker, to avoid any additional delays introduced by the broker. This also allows the user to find the best worker for his optimization approach. Using the broker as a proxy simplifies the implementation of the user, however this introduces large latencies, which may be undesirable in an edge scenario.

7 Conclusion and Outlook

The edge computing environment is developing in a very fast way. As shown in this thesis, current frameworks often rely on virtualized environments, backed by Kubernetes for orchestrating and exploiting the FaaS pattern. In this work, the focus was targeted towards an interoperability protocol, which can be used by different systems and does not need virtualized environments. Each participant in this system communicates his features or needs. This enables also very resource-limited devices to participate, even without virtualization.

With the proposed framework, MetaEdge, it was shown that different platforms can be integrated, allowing for heterogeneous implementations. Users in the system can detect suitable workers with the help of a central broker. Additionally, arbitrary tasks can get offloaded via the protocol, where users can use independent optimizations strategies, based on the context. The execution of a task is decoupled from the deployment, to increase the performance of an offloaded process. This is achieved in two ways: First, the user can directly establish a connection to the targeted worker. Second, to offload a process, only one message has to get sent from the user to the worker, to keep the delay as small as possible. The protocol is not limited for a specific use case, but targeted for different applications in edge computing environments.

The framework was evaluated in different test beds, using virtualized and real-world environments. The evaluation verified the functionality of the protocol, as well as indicated that the protocol is very lightweight and does not introduce a big overhead.

Future Work

While this work introduced a novel protocol for edge computing, there exist multiple opportunities for future improvements. First, the design was implemented using a single broker. To increase the scalability, a multi-broker approach could get implemented, allowing more participants to join. This requires a synchronization mechanism between the brokers. Additionally, this can increase the availability of the system, since the failure of a broker can get covered by another broker. This is already one popular feature of current edge computing platforms.

Second, more different workers could be integrated in the system, increasing the heterogeneity even more. They can also provide some hardware features, which could increase the execution of specific tasks. Custom schedulers will be necessary in this case to find an optimization.

Additionally, the schedulers can get equipped with predictive algorithms, for example to decrease the energy consumption or estimated execution time. Therefore, the energy consumption of the users when they try to offload their tasks should get measured, too. Furthermore, MetaEdge focuses

only on offloading, but this depends on local and remote resources as well as connectivity between the devices. Using this information can help to decide when offloading is useful and when a local computation is preferable.

Another important aspect is the mobility of the devices, which may be increased in edge computing scenarios. To support such mobile devices, specific handover algorithms should be integrated. This can affect the selection of other workers or switching to another broker, if available.

Bibliography

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia. "A View of Cloud Computing". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: https://doi.org/10.1145/1721654.1721672 (cit. on p. 25).

[And04] D. Anderson. "BOINC: a system for public-resource computing and storage". In: *Fifth IEEE/ACM International Workshop on Grid Computing*. 2004, pp. 4–10. DOI: 10.1109/GRID.2004.14 (cit. on p. 24).

[BB99] M. Baker, R. Buyya. "Cluster Computing: The Commodity Supercomputer". In: *Softw. Pract. Exper.* 29.6 (May 1999), pp. 551–576. ISSN: 0038-0644 (cit. on p. 24).

[BHQT22] L. Baresi, D. Y. X. Hu, G. Quattrocchi, L. Terracciano. *NEPTUNE: Network- and GPU-aware Management of Serverless Functions at the Edge*. 2022. arXiv: 2205.04320 [cs.SE] (cit. on p. 32).

[BM02] R. Buyya, M. Maheswaran. "A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing". In: *Software: Practice and Experience* 32 (Feb. 2002). DOI: 10.1002/spe.432 (cit. on p. 23).

[BMG+19] L. Baresi, D. F. Mendonça, M. Garriga, S. Guinea, G. Quattrocchi. "A Unified Model for the Mobile-Edge-Cloud Continuum". In: *ACM Trans. Internet Technol.* 19.2 (Apr. 2019). ISSN: 1533-5399. DOI: 10.1145/3226644. URL: https://doi.org/10.1145/3226644 (cit. on p. 32).

[BMQ19] L. Baresi, D. F. Mendonça, G. Quattrocchi. "PAPS: A Framework for Decentralized Self-management at the Edge". In: *Service-Oriented Computing*. Ed. by S. Yangui, I. Bouassida Rodriguez, K. Drira, Z. Tari. Cham: Springer International Publishing, 2019, pp. 508–522. ISBN: 978-3-030-33702-5 (cit. on p. 32).

[BMZA12] F. Bonomi, R. Milito, J. Zhu, S. Addepalli. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 13–16. ISBN: 9781450315197. DOI: 10.1145/2342509.2342513. URL: https://doi.org/10.1145/2342509.2342513 (cit. on p. 25).

[Bre22] M. Breitbach. "Computation Offloading for Fast and Energy-Efficient Edge Computing". Dissertation. University of Mannheim, 2022 (cit. on pp. 23, 62).

[BS20] A. Boukerche, V. Soto. "An Efficient Mobility-Oriented Retrieval Protocol for Computation Offloading in Vehicular Edge Multi-Access Network". In: *IEEE Transactions on Intelligent Transportation Systems* 21.6 (2020), pp. 2675–2688. DOI: 10.1109/TITS.2020.2991376 (cit. on p. 30).

[BSEB19] M. Breitbach, D. Schäfer, J. Edinger, C. Becker. "Context-Aware Data and Task Placement in Edge Computing Environments". In: *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom*. 2019, pp. 1–10. DOI: 10.1109/PERCOM.2019.8767386 (cit. on p. 19).

[BYB+23] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, J. Ott. "Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing". In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, July 2023, pp. 215–231. ISBN: 978-1-939133-35-9. URL: https://www.usenix.org/conference/atc23/presentation/bartolomeo (cit. on p. 32).

[CBC+10] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl. "MAUI: Making smartphones last longer with code offload". In: vol. 2010. Oct. 2010, pp. 49–62. DOI: 10.1145/1814433.1814441 (cit. on p. 27).

[CIMN10] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik. *CloneCloud: Boosting Mobile Device Applications Through Cloud Clone Execution*. 2010. arXiv: 1009.3088 [cs.DC] (cit. on p. 27).

[CLWG15] Z. Cheng, P. Li, J. Wang, S. Guo. "Just-in-Time Code Offloading for Wearable Computing". In: *IEEE Transactions on Emerging Topics in Computing* 3.1 (2015), pp. 74–83. DOI: 10.1109/TETC.2014.2387688 (cit. on p. 26).

[DB17] S. K. Datta, C. Bonnet. "An edge computing architecture integrating virtual IoT devices". In: *2017 IEEE 6th Global Conference on Consumer Electronics (GCCE)*. 2017, pp. 1–3. DOI: 10.1109/GCCE.2017.8229253 (cit. on p. 26).

[DFZ+15] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, B. Hu. "Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends". In: *2015 IEEE 8th International Conference on Cloud Computing*. 2015, pp. 621–628. DOI: 10.1109/CLOUD.2015.88 (cit. on p. 25).

[DG08] J. Dean, S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492 (cit. on p. 24).

[DLNW13] H. Dinh Thai, C. Lee, D. Niyato, P. Wang. "A survey of mobile cloud computing: Architecture, applications, and approaches". In: *Wireless Communications and Mobile Computing* 13 (Dec. 2013). DOI: 10.1002/wcm.1203 (cit. on p. 24).

[FHT+15] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, R. Buyya. "Mobile code offloading: from concept to practice and beyond". In: *IEEE Communications Magazine* 53.3 (2015), pp. 80–88. DOI: 10.1109/MCOM.2015.7060486 (cit. on p. 27).

[FKT01] I. Foster, C. Kesselman, S. Tuecke. *The Anatomy of the Grid - Enabling Scalable Virtual Organizations*. 2001. arXiv: cs/0103025 [cs.AR] (cit. on p. 24).

[FZRL08] I. Foster, Y. Zhao, I. Raicu, S. Lu. "Cloud Computing and Grid Computing 360-Degree Compared". In: *2008 Grid Computing Environments Workshop*. 2008, pp. 1–10. DOI: 10.1109/GCE.2008.4738445 (cit. on p. 19).

[GFD22] P. Gackstatter, P. A. Frangoudis, S. Dustdar. "Pushing Serverless to the Edge with WebAssembly Runtimes". In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2022, pp. 140–149. DOI: 10.1109/CCGrid54584.2022.00023 (cit. on p. 33).

[GME+15] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, E. Riviere. "Edge-Centric Computing: Vision and Challenges". In: *SIGCOMM Comput. Commun. Rev.* 45.5 (Sept. 2015), pp. 37–42. ISSN: 0146-4833. DOI: 10.1145/2831347.2831354. URL: https://doi.org/10.1145/2831347.2831354 (cit. on pp. 19, 26).

[Gro09] R. L. Grossman. "The Case for Cloud Computing". In: *IT Professional* 11.2 (2009), pp. 23–27. DOI: 10.1109/MITP.2009.40 (cit. on p. 25).

[HESB18] M. Heck, J. Edinger, D. Schaefer, C. Becker. "IoT Applications in Fog and Edge Computing: Where Are We and Where Are We Going?" In: *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. 2018, pp. 1–6. DOI: 10.1109/ICCCN.2018.8487455 (cit. on p. 26).

[HR19] A. Hall, U. Ramachandran. "An Execution Model for Serverless Functions at the Edge". In: *Proceedings of the International Conference on Internet of Things Design and Implementation*. IoTDI '19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 225–236. ISBN: 9781450362832. DOI: 10.1145/3302505.3310084. URL: https://doi.org/10.1145/3302505.3310084 (cit. on p. 33).

[HRS+17] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, J. Bastien. "Bringing the Web up to Speed with WebAssembly". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 185–200. ISSN: 0362-1340. DOI: 10.1145/3140587.3062363. URL: https://doi.org/10.1145/3140587.3062363 (cit. on p. 33).

[HYW19] N. Hassan, K.-L. A. Yau, C. Wu. "Edge Computing in 5G: A Review". In: *IEEE Access* 7 (2019), pp. 127276–127289. DOI: 10.1109/ACCESS.2019.2938534 (cit. on p. 19).

[IBY+07] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: Association for Computing Machinery, 2007, pp. 59–72. ISBN: 9781595936363. DOI: 10.1145/1272996.1273005. URL: https://doi.org/10.1145/1272996.1273005 (cit. on p. 24).

[JS16] A. Jain, P. Singhal. "Fog computing: Driving force behind the emergence of edge computing". In: *2016 International Conference System Modeling & Advancement in Research Trends (SMART)*. 2016, pp. 294–297. DOI: 10.1109/SYSMART.2016.7894538 (cit. on p. 25).

[KK23] S.-H. Kim, T. Kim. "Local Scheduling in KubeEdge-Based Edge Computing Environment". In: *Sensors* 23.3 (2023). ISSN: 1424-8220. DOI: 10.3390/s23031522. URL: https://www.mdpi.com/1424-8220/23/3/1522 (cit. on p. 32).

[KLLB13] K. Kumar, J. Liu, Y.-H. Lu, B. Bhargava. "A Survey of Computation Offloading for Mobile Systems". In: *Mobile Networks and Applications* 18 (Feb. 2013). DOI: 10.1007/s11036-012-0368-0 (cit. on p. 19).

[KVM20] E. Krishnasamy, S. Varrette, M. Mucciardi. *Edge Computing: An Overview of Framework and Applications*. English. Tech. rep. https://prace-ri.eu/wp-content/uploads/Edge-Computing-An-Overview-of-Framework-and-Applications.pdf. December 2020 (cit. on p. 31).

[KWA+01] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Leboisky. "SETI@home-massively distributed computing for SETI". In: *Computing in Science Engineering* 3.1 (2001), pp. 78–83. DOI: 10.1109/5992.895191 (cit. on p. 19).

[LCJZ18] C. Long, Y. Cao, T. Jiang, Q. Zhang. "Edge Computing Framework for Cooperative Video Processing in Multimedia IoT Systems". In: *IEEE Transactions on Multimedia* 20.5 (2018), pp. 1126–1139. DOI: 10.1109/TMM.2017.2764330 (cit. on p. 30).

[LLJL19] L. Lin, X. Liao, H. Jin, P. Li. "Computation Offloading Toward Edge Computing". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1584–1607. DOI: 10.1109/JPROC.2019.2922285 (cit. on p. 27).

[LPP+22] S. Lee, L.-A. Phan, D.-H. Park, S. Kim, T. Kim. "EdgeX over Kubernetes: Enabling Container Orchestration in EdgeX". In: *Applied Sciences* 12.1 (2022). ISSN: 2076-3417. DOI: 10.3390/app12010140. URL: https://www.mdpi.com/2076-3417/12/1/140 (cit. on p. 30).

[LYKZ10] A. Li, X. Yang, S. Kandula, M. Zhang. "CloudCmp: Comparing Public Cloud Providers". In: Nov. 2010, pp. 1–14. DOI: 10.1145/1879141.1879143 (cit. on p. 25).

[MG11] P. M. Mell, T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, USA, 2011 (cit. on p. 25).

[MOC+14] T. Mastelic, A. Oleksiak, H. Claussen, I. Brandic, J.-M. Pierson, A. V. Vasilakos. "Cloud Computing: Survey on Energy Efficiency". In: *ACM Comput. Surv.* 47.2 (Dec. 2014). ISSN: 0360-0300. DOI: 10.1145/2656204. URL: https://doi.org/10.1145/2656204 (cit. on p. 25).

[OWZS13] K. Ousterhout, P. Wendell, M. Zaharia, I. Stoica. "Sparrow: Distributed, Low Latency Scheduling". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 69–84. ISBN: 9781450323888. DOI: 10.1145/2517349.2522716. URL: https://doi.org/10.1145/2517349.2522716 (cit. on p. 27).

[PB20] T. Pfandzelter, D. Bermbach. "tinyFaaS: A Lightweight FaaS Platform for Edge Environments". In: *2020 IEEE International Conference on Fog Computing (ICFC)*. 2020, pp. 17–24. DOI: 10.1109/ICFC49376.2020.00011 (cit. on p. 33).

[PVM22] M. Pappalardo, A. Virdis, E. Mingozzi. "An Edge-Based LWM2M Proxy for Device Management to Efficiently Support QoS-Aware IoT Services". In: *IoT* 3.1 (2022), pp. 169–190. ISSN: 2624-831X. DOI: 10.3390/iot3010011. URL: https://www.mdpi.com/2624-831X/3/1/11 (cit. on p. 28).

[RND18] T. Rausch, S. Nastic, S. Dustdar. "EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications". In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 2018, pp. 191–197. DOI: 10.1109/IC2E.2018.00043 (cit. on p. 27).

[RSS+20] D. Rosendo, P. Silva, M. Simonin, A. Costan, G. Antoniu. “E2Clab: Exploring the Computing Continuum through Repeatable, Replicable and Reproducible Edge-to-Cloud Experiments”. In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 2020, pp. 176–186. DOI: 10.1109/CLUSTER49012.2020.00028 (cit. on p. 19).

[SASK19] K. Siozios, D. Anagnostos, D. Soudris, E. Kosmatopoulos. “IoT for smart grids”. In: *Cham, Switzerland: Springer* (2019) (cit. on p. 30).

[Sat01] M. Satyanarayanan. “Pervasive computing: vision and challenges”. In: *IEEE Personal Communications* 8.4 (2001), pp. 10–17. DOI: 10.1109/98.943998 (cit. on p. 26).

[SBCD09] M. Satyanarayanan, P. Bahl, R. Caceres, N. Davies. “The Case for VM-Based Cloudlets in Mobile Computing”. In: *IEEE Pervasive Computing* 8.4 (2009), pp. 14–23. DOI: 10.1109/MPRV.2009.82 (cit. on p. 26).

[SEP+16] D. Schafer, J. Edinger, J. M. Paluska, S. VanSyckel, C. Becker. “Tasklets: "Better than Best-Effort"Computing”. In: *2016 25th International Conference on Computer Communication and Networks (ICCCN)*. 2016, pp. 1–11. DOI: 10.1109/ICCCN.2016.7568580 (cit. on p. 66).

[SEV+16] D. Schäfer, J. Edinger, S. VanSyckel, J. M. Paluska, C. Becker. “Tasklets: Overcoming Heterogeneity in Distributed Computing Systems”. In: *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 2016, pp. 156–161. DOI: 10.1109/ICDCSW.2016.22 (cit. on p. 19).

[SP18] S. Shillaker, P. R. Pietzuch. “A Provider-Friendly Serverless Framework for Latency-Critical Applications”. In: 2018. URL: https://api.semanticscholar.org/CorpusID:51997872 (cit. on p. 33).

[SP20] S. Shillaker, P. Pietzuch. “Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 419–433. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shillaker (cit. on p. 33).

[VLK+11] G. Valentini, W. Lassonde, S. Khan, N. Min Allah, S. Madani, J. Li, L. Zhang, L. Wang, N. Ghani, J. Kołodziej, H. Li, A. Zomaya, C.-Z. Xu, P. Balaji, A. Vishnu, F. Pinel, J. Pecero, D. Kliazovich, P. Bouvry. “An Overview of Energy Efficiency Techniques in Cluster Computing Systems”. In: *Cluster Computing* 16 (Mar. 2011), pp. 1–13. DOI: 10.1007/s10586-011-0171-x (cit. on p. 24).

[VPK+15] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes. “Large-Scale Cluster Management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741964. URL: https://doi.org/10.1145/2741948.2741964 (cit. on p. 24).

[vT16] M. van Steen, A. Tanenbaum. “A brief introduction to distributed systems”. English. In: *Computing* 98.10 (2016), pp. 967–1009. ISSN: 0010-485X. DOI: 10.1007/s00607-016-0508-7 (cit. on p. 23).

[Wei91] M. Weiser. “The Computer for the 21 st Century”. In: *Scientific American* 265.3 (1991), pp. 94–105. ISSN: 00368733, 19467087. URL: http://www.jstor.org/stable/24938718 (visited on 10/13/2023) (cit. on p. 19).

[WVMN20] N. Wang, B. Varghese, M. Matthaiou, D. S. Nikolopoulos. “ENORM: A Framework For Edge NOde Resource Management”. In: *IEEE Transactions on Services Computing* 13.6 (2020), pp. 1086–1099. DOI: 10.1109/TSC.2017.2753775 (cit. on pp. 33, 34).

[XJK22] R. Xu, W. Jin, D. Kim. “Knowledge-based edge computing framework based on CoAP and HTTP for enabling heterogeneous connectivity”. In: *Personal and Ubiquitous Computing* 26 (Apr. 2022), pp. 1–16. DOI: 10.1007/s00779-020-01466-4 (cit. on p. 28).

[XSXH18] Y. Xiong, Y. Sun, L. Xing, Y. Huang. “Extend Cloud to Edge with KubeEdge”. In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 2018, pp. 373–377. DOI: 10.1109/SEC.2018.00048 (cit. on p. 31).

[YNL+22] T. Yang, J. Ning, D. Lan, J. Zhang, Y. Yang, X. Wang, A. Taherkordi. “Kubeedge Wireless for Integrated Communication and Computing Services Everywhere”. In: *IEEE Wireless Communications* 29.2 (2022), pp. 140–145. DOI: 10.1109/MWC.004.2100038 (cit. on p. 32).

[YTC17] S.-W. Yang, O. Tickoo, Y.-K. Chen. “A framework for visual fog computing”. In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2017, pp. 1–4. DOI: 10.1109/ISCAS.2017.8050297 (cit. on p. 30).

[ZCF+10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. Boston, MA: USENIX Association, 2010, p. 10 (cit. on p. 24).

All links were last followed on October 23, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature