Self-Organizing Fog Computing Systems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

by

Vasileios Karagiannis, MSc
Registration Number 11743105

to the Faculty of Informatics
at the TU Wien
Advisor: Prof. Dr. Stefan Schulte

The dissertation has been reviewed by:

____________________  ______________________
Paul Pop                  Boris Koldehofe

Vienna, 1st February, 2022

____________________
Vasileios Karagiannis
Self-Organizing Fog Computing Systems

DISSERTATION
zur Erlangung des akademischen Grades
Doktor der Technischen Wissenschaften
eingereicht von
Vasileios Karagiannis, MSc
Matrikelnummer 11743105

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Prof. Dr. Stefan Schulte

Diese Dissertation haben begutachtet:

________________________________________
 Paul Pop

________________________________________
 Boris Koldehofe

Wien, 1. Februar 2022

________________________________________
Vasileios Karagiannis
Abstract

Fog computing is a novel computing paradigm which enables the execution of applications on compute nodes which reside both in the cloud and at the edge of the network. Various performance benefits, such as low communication latency and high network bandwidth, have turned this paradigm into a well-accepted extension of cloud computing. So far, many fog computing systems have been proposed, consisting of distributed compute nodes which are often organized hierarchically in layers. Such systems commonly rely on the assumption that the nodes of adjacent layers reside close to each other, thereby achieving low latency computations. However, this assumption may not hold in fog computing systems that span over large geographical areas, due to the wide distribution of the nodes. In addition, most proposed fog computing systems route the data on a path which starts at the data source, and goes through various edge and cloud nodes. Each node on this path may accept the data if there are available resources to process this data locally. Otherwise, the data is forwarded to the next node on path. Notably, when the data is forwarded (rather than accepted), the communication latency increases by the delay to reach the next node.

This thesis aims at tackling these problems by proposing distributed algorithms whereby the compute nodes measure the network proximity to each other, and self-organize accordingly. These algorithms are implemented on geographically distributed compute nodes, considering image processing and smart city use cases, and are thoroughly evaluated showing significant latency- and bandwidth-related performance benefits. Furthermore, we analyze the communication latency of sending data to distributed edge and cloud compute nodes, and we propose two novel routing approaches: i) A context-aware routing mechanism which maintains a history of previous transmissions, and uses this history to find nearby nodes with available resources. ii) edgeRouting, which leverages the high bandwidth between nodes of cloud providers in order to select network paths with low communication latency. Both of these mechanisms are evaluated under real-world settings, and are shown to be able to lower the communication latency of fog computing systems significantly, compared to alternative methods.
Kurzfassung


I would like to note my gratitude for being able to conduct the research of this thesis with funding from the European Commission’s Horizon 2020 research and innovation programme under grant agreements: No. 764785 Fog Computing for Robotics and Industrial Automation (FORA), and No. 732505 Lightweight Computations for Networks at the Edge (Lightkone). I am also very grateful for my stay at the Distributed Systems Group of TU Wien which provided a pleasant working environment and brilliant colleagues. Last but not least, I would like to thank my family for being supportive of my aspirations.
Erklärung zur Verfassung der Arbeit

Vasileios Karagiannis, MSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Februar 2022

Vasileios Karagiannis
Contents

Abstract v

Contents xiii

Publications xv

1 Introduction 1
   1.1 Fog Computing System Overview 1
   1.2 Problem Statement 3
   1.3 Contributions 5
   1.4 Thesis Outline 7

2 Background 9
   2.1 Internet of Things 9
   2.2 Cloud Computing 11
   2.3 Edge Computing 13
   2.4 Fog Computing 13

3 Self-Organization 17
   3.1 Overview 17
   3.2 Related Work 19
   3.3 Algorithms for Self-Organization 25
   3.4 Evaluation 39
   3.5 Summary 54

4 Context-Aware Routing 57
   4.1 Overview 57
   4.2 Related Work 58
   4.3 Context-Aware Routing Mechanism 60
   4.4 Evaluation 74
   4.5 Summary 87

5 edgeRouting 89
   5.1 Overview 89
Publications

This thesis is based on the following articles which have been published in the proceedings of scientific conferences, workshops, and journals. Parts of these articles are included in verbatim throughout the thesis without being explicitly referenced.


Additional scientific articles which have been published during the course of this thesis but are not discussed hereinafter are listed below:


• Thomas Hiessl, Vasileios Karagiannis, Christoph Hochreiner, Stefan Schulte, and Matteo Nardelli. Optimal placement of stream processing operators in the fog. In *International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10. IEEE, 2019

In the first chapter of this thesis, we present an introduction which focuses on the addressed problems, along with the specific ways these problems are tackled. Specifically, we first describe the overall concept of a fog computing system in Section 1.1 and then we discuss the addressed problems in Section 1.2. After that, we mention the relevant contributions in Section 1.3 and we provide an overview of the structure that is used for the remainder of this thesis in Section 1.4.

1.1 Fog Computing System Overview

In this section, we introduce the basic concepts of fog computing, and fog computing systems. While a more elaborate view on background information is provided in Chapter 2, here we present some fundamental information which is necessary for the understanding of the addressed problems and respective contributions of this thesis.

Computing at the edge of the network is motivated mostly by the high round-trip times (i.e., high communication latency) and the bandwidth bottlenecks that can be observed in the end-to-end communication between devices at the edge and in the cloud [190]. Fog computing emerged to cope with such issues by exploiting the computational resources of the various compute nodes that span from the cloud to the edge of the network [201]. These nodes may belong to cloud operators (e.g., Amazon and Microsoft data centers) [173], network operators (e.g., networking devices with spare or added computational capabilities such as base stations and access points) [54] [219], or privately-owned dedicated compute nodes at the edge of the network (e.g., cloudlets and fog nodes) [195]. By exploiting such nodes, fog computing aims at processing data, e.g., from Internet of Things (IoT) devices, close to the data sources [36]. This relieves the Internet backbone of network traffic and hinders the formation of network bottlenecks [134]. Moreover, since the data is processed close to the data sources, fog computing is able to provide low latency and high bandwidth [188] [229].
Since fog computing can incorporate various compute nodes from the cloud to the edge of the network, fog computing systems may need to scale to a large degree [15]. When the number of participating compute nodes becomes too large to be centrally coordinated, distributed algorithms can be employed to enable the nodes to self-organize in a decentralized manner [46]. This usually entails the functionality to form a structure (e.g., a hierarchical structure of compute nodes, as shown in Fig. 1.1) such that each node has a partial view of the system by communicating only with a limited number of neighbor compute nodes. A set of neighbor compute nodes can also be referred to as a group (also shown in Fig. 1.1) [165, 185, 206].

To facilitate applications on top of a system structure of distributed compute nodes, each participant may need to process information and share it with the rest of the nodes in the system. Typically, an application can consist of multiple computations which are executed on various compute nodes [87]. Each node is able to meet different application requirements. For example, a node at the edge of the network may be able to process data faster than a remote cloud node [195]. Nevertheless, interactions with a cloud node are sometimes essential due to the increased processing and storing capacity [187]. While exchanging information between neighbor nodes might be a straightforward transmission, propagating data towards other nodes of the system can become complex, especially when the number of participating compute nodes grows [95]. To cope with this issue, more advanced routing mechanisms can be implemented to provide messaging over the system structure [95].
In this thesis, we focus on two fog computing aspects: i) distributed algorithms for self-organization, and ii) routing mechanisms. Both of these aspects can be considered essential for the operation of a fog computing system. Distributed algorithms aim at creating a suitable system structure of compute nodes that can communicate with each other with low latency and high bandwidth. Routing mechanisms, on the other hand, focus on propagating information over the system structure.

1.2 Problem Statement

Notable advances in creating fog computing systems propose a hierarchical structure with the cloud at the top, and various compute nodes organized in layers below, i.e., between the cloud and the IoT devices at the edge of the network [133, 161, 186]. In such structures, reduced communication latency is achieved by processing the data on the nodes of the lower layers. This latency reduction depends on the network proximity among the nodes of the adjacent layers, and does not come automatically due to the hierarchical organization of the nodes [161]. Thus, in distributed systems with nodes that may span over large geographical areas (such as in fog computing), measuring the network proximity and organizing the nodes accordingly is necessary to ensure the efficiency of the hierarchical structure. However, most current approaches for creating fog computing systems rely on hierarchical structures but do not take actual proximity measurements, and do not discuss how the nodes of such a structure are discovered and organized in a way that fulfills latency- and bandwidth-related goals (e.g., [13, 77, 118]). This might not pose a concern for statically configured small-scale systems, but may become a problem when the task of creating the system structure needs to be automated, e.g., for large-scale systems. A further repercussion of not considering how an efficient hierarchical structure can be created, is that the overhead associated with organizing the nodes into a hierarchy, i.e., the control messages required for creating appropriate logical links among the nodes of a fog computing system, is not analyzed. This repercussion may lead to scalability issues in case the overhead grows significantly (and causes network bottlenecks) when many nodes join the system. Therefore, a Research Question (RQ) arises:

Research Question I:
How to create a scalable fog computing system consisting of distributed compute nodes which are organized considering network proximity?

While scalability and the overhead of organizing the nodes of a fog computing system are usually not discussed in state-of-the-art research, there are approaches that try to circumvent the assumption that the nodes of adjacent layers reside close to each other. Prominent approaches employ location coordinates and communication latency to estimate the network proximity between compute nodes [169, 207, 209]. By using the communication latency as an indicator of network proximity (e.g., using round-trip times), the nodes measure the values of network latency to reach each other, and form
1. Introduction

logical links with the nodes of the lowest latency values [169]. However, the value of latency measurements may not be a reliable indicator of network proximity since it can fluctuate, e.g., due to network bottlenecks [79]. When using location coordinates as an indicator of network proximity, the compute nodes are assumed to be preconfigured with coordinates that can be used for estimating the proximity between nodes, e.g., according to Euclidean distance [81, 207]. Nevertheless, there is no guarantee that the distance from location coordinates corresponds to network proximity [113]. In addition, preconfiguring each node with coordinates that actually correspond to network proximity may become impractical in large-scale systems since the number of nodes might be too large to allow individual configuration for each node.

Another potential problem in fog computing systems may arise from the way the data is routed. Many fog computing systems route the data on a path which starts at the data source, and goes through various edge and cloud compute nodes until the data is accepted for processing [23, 160, 217]. The compute nodes on this path are usually assumed to be ordered based on their proximity to the data source. Thus, the first compute nodes are able to process the IoT data with low communication latency, but this latency increases when nodes farther along the path are utilized for the processing [120]. This way, the compute node closest to the data source with adequate available computational resources, is utilized for processing the IoT data with low latency. Typically, the compute nodes at the edge of the network provide limited computational resources [130, 230]. As a result, only a fraction of the IoT data is accepted by nearby compute nodes (i.e., the first nodes on path), while the rest of the data is more likely to be processed in a remote cloud node [212].

In such systems, every time a compute node forwards the data, the communication latency is increased by both the time required by a compute node to decide whether to accept the data or not, and the time needed to reach the next node. Therefore, this latency accumulates while the IoT data is forwarded by nodes on path. Furthermore, routing the data through intermediate nodes to the compute node that accepts it, may increase the number of network hops traveled, and potentially the utilization of network bandwidth [112]. Hence, the second RQ arises:

**Research Question II:**

*How to enable the distributed compute nodes of a fog computing system to route data towards other nodes in an efficient manner?*

Most current approaches that aim at avoiding the forwarding by nodes on path, use direct transmissions to a destination node that is usually selected by a centralized component that makes decisions based on a placement algorithm. However, the overhead of the placement algorithm may increase significantly in large-scale systems [108]. Additionally, such systems typically consider end-to-end metrics, and do not take into account alternative network paths with detours through edge nodes, which may have higher network bandwidth, and lower communication latency.
1.3. Contributions

Therefore, this thesis addresses critical problems which occur during the creation and operation of a fog computing system. On the one hand, RQ I aims at tackling the problem of building a system consisting of distributed compute nodes which are organized based on proximity in a scalable manner. On the other hand, assuming that the essential algorithms to build such a system are available, the problem of enabling efficient routing among the nodes may arise. The reason that this problem is likely to occur is that the nodes of fog computing systems may need to communicate with very low latency [40]. To achieve this, it is imperative that there are mechanisms in place, which aim at routing data through the network paths with the lowest communication latency. Thus, in a fog computing system that is already organized based on network proximity, RQ II focuses on the efficient routing of data, which targets low latency.

1.3 Contributions

Based on the RQs discussed in Section 1.2, this thesis makes the following contributions to the state of the art of fog computing systems:

**Contribution I: Algorithms for Self-Organization**

To aid in answering RQ I, we propose self-organizing fog computing systems. A self-organizing fog computing system can be created using a distributed algorithm that runs on all the compute nodes of the system in order to execute the required control operations for achieving the desired system structure [112].

This contribution can be considered relevant for the general field of distributed algorithms that aim at organizing a number of compute nodes that may be too large to be centrally organized. This field is very important nowadays because more and more computations are executed on a set of nodes which rely on self-organization to form and maintain an appropriate system structure [25]. Algorithms developed in this context often find use in pervasive computing, fog and edge computing, and IoT scenarios [46]. In contrast to existing approaches, we design algorithms for the self-organization of fog computing systems which span over large geographical areas, and we also compare them regarding communication latency, network bandwidth, and scalability. Specifically, this contribution advances fog computing systems in the following ways: i) The process of adding new nodes is defined (by the algorithm) which makes it possible to model the resulting overhead in order to examine scalability aspects. ii) Integrating network proximity measurements in the self-organization enables the nodes to consider the proximity to each other, and repeat the self-organization with each new node that joins the system. This ensures that even when the compute nodes are geographically distributed, the network proximity to each other is taken into account while creating the system structure.

The results of the implementation of our distributed algorithms in a real-world setting show that the proposed algorithms reduce the communication latency of latency-sensitive processes by 27–43%, and increase the available network bandwidth by 36–86%, compared to alternative methods. For latency-tolerant processes, the reduction in the communication latency is 15–32%, and the increase in the available network bandwidth is 14–49%.
Furthermore, we analyze the scalability of our algorithms, and we show that a flat structure (i.e., without layers) scales better than the commonly used layered hierarchy due to generating less overhead when the size of the system grows. Contribution I has been originally presented in [112, 113, 115], and is discussed extensively in Chapter 3 of this thesis.

**Contribution II: Context-Aware Routing**

In response to RQ II, we propose a mechanism for context-aware routing in fog computing systems. Instead of routing the IoT data along the path of compute nodes until a node with adequate resources is found, our mechanism aims at sending the data directly to that node. This eliminates the overhead of sending the data through intermediate nodes, and consequently lowers the communication latency of performing IoT computations. Since the target node is not known a priori, our mechanism relies on the context of the data, and on information from previous transmissions. Based on these, the proposed mechanism first examines which compute node has accepted data of the same context in the past. Then, the IoT data is routed on a path that starts from that node, thereby bypassing other compute nodes that typically forward such data due to not having enough available computational resources to process it.

This contribution includes a system model for executing IoT applications in fog computing systems, and a mechanism for realizing the proposed context-aware routing. Even though alternative approaches from the literature that rely on the context of the data to improve the performance of IoT applications exist, they do not typically consider leveraging the context to improve routing aspects. By doing that, we manage to design a context-aware routing mechanism which achieves low communication latency and bandwidth utilization. Based on this mechanism, we also implement a fog computing system which is able to employ predictive methods such as Reinforcement Learning (RL) to further improve the routing of data. Interestingly, the context-aware routing of this contribution is decoupled from the way the participating compute nodes are discovered and organized. Consequently, this routing approach can be applied to any fog computing system, including a system of compute nodes created using the algorithms of Contribution I.

To show the benefits of context-aware routing, we build a fog computing system using real-world geographically distributed compute nodes, and we perform an extensive prototype- and simulation-based evaluation considering a smart energy application. According to our results, the proposed routing approach lowers the communication latency of sending IoT data by up to 23%, and reduces the hop count by up to 73%, compared to an alternative method. Contribution II has been originally presented in [108], and is discussed extensively in Chapter 4 of this thesis.

**Contribution III: edgeRouting**

To further pursue an answer for RQ II, we analyze the different delays that contribute to the latency of sending data from IoT devices to edge and cloud compute nodes offered by cloud providers. Based on this analysis, we propose edgeRouting which is a routing approach for reducing the communication latency. According to edgeRouting, when sending data from an IoT device to a suitable compute node of a cloud provider, this
data is routed through the closest available edge compute node. This way, the data is sent to the edge node using the connection of an Internet provider (i.e., with potentially limited bandwidth), and from there, the data is routed again to the suitable compute node with higher bandwidth offered by a cloud provider. Thus, this approach manages to leverage both the low propagation delay of edge compute nodes, and the high bandwidth among edge and cloud compute nodes of cloud providers, in order to achieve low latency.

This contribution includes the analysis of the latency of sending IoT data to distributed compute nodes at the edge and the cloud, and the design of a routing approach which sends the data in a manner that reduces this latency. Specifically, we analyze existing alternatives from the literature, and we derive shortcomings which affect the communication latency in a negative manner, e.g., most current approaches consider end-to-end communication while disregarding network paths with detours that may provide higher bandwidth. By taking into account such shortcomings, we manage to devise the proposed edgeRouting that considers detours through edge compute nodes. Similar to Contribution II, Contribution III is also a routing approach which can be applied to any fog computing system, including a system built using the distributed algorithms from Contribution I.

To evaluate the proposed edgeRouting, we build a fog computing system of distributed compute nodes, and we implement edgeRouting along with alternative routing approaches from the literature. Then, we perform experiments considering IoT use cases, and we produce results which show that the proposed approach reduces the communication latency by up to 55% compared to the alternatives. Contribution III has been originally presented in [114], and is discussed extensively in Chapter 5 of this thesis.

1.4 Thesis Outline

The rest of this thesis abides by the following structure:

- Chapter 2 provides background information for this thesis, and introduces the concepts of IoT, cloud computing, edge computing, and fog computing. To this end, this chapter also discusses comprehensive examples for each introduced concept.

- Chapter 3 presents the design and functionality of two distributed algorithms that enable the self-organization of the compute nodes of a fog computing system. In addition, in this chapter we deploy an implementation of these algorithms on geographically distributed compute nodes, and we discuss the differences between our approach and alternative methods.

- In Chapter 4 we propose a mechanism for context-aware routing in fog computing systems. This mechanism maintains a history of previous transmissions, which can be leveraged based on various strategies in order to route the data in an efficient manner. We also discuss such strategies which can be based on predictive methods.
1. Introduction

such as RL. Finally, we present a prototype- and simulation-based evaluation of the proposed mechanism, which compares our approach to an alternative method.

- In Chapter 5, we analyze the latency of sending IoT data to distributed compute nodes of a fog computing system, and we propose an approach using detours for reducing this latency. Moreover, we perform experiments using a setup of actual distributed compute nodes in order to compare the proposed approach to alternative methods, and we present our findings.

- Finally, Chapter 6 concludes this thesis by summarizing the presented contributions along with their corresponding assumptions and requirements, and by revisiting the target RQs. In addition, in this chapter we provide an overview of our ongoing work, as well as our future plans for further research on fog computing systems.
Background

This chapter introduces well-established concepts which are necessary for understanding the contributions of this thesis. Specifically, in the following sections we discuss background information on the IoT which aims at utilizing sensors and actuators to sense and affect the surrounding physical environment, the integration of the IoT with cloud computing resources, as well as the newer concepts of edge and fog computing. For the better understanding of these concepts, we also provide relevant figures which demonstrate visually comprehensible examples.

2.1 Internet of Things

The IoT envisions a world in which everyday objects (such as wearables, house appliances, and phones) connect to the Internet. Such objects may use this connectivity to exchange, store, and process information in order to sense and/or affect the surrounding environment [144]. Thus, the IoT is a paradigm that enables everyday objects with sensors and actuators to communicate with each other, and collaborate in order to reach common goals [18, 24]. For example, smart traffic lights that arrange their schedule based on the actual traffic perceived through cameras deployed all over a city [145], empty parking spots that notify nearby drivers, etc. This paradigm has a profound potential to improve the quality of life in modern societies due to the wide range of applicable use cases such as: assisted living, transportation, healthcare, industrial automation, etc. [82, 103, 151].

The main enabler of this concept is the presence of a variety of things around us, which integrate a communication module and can be uniquely or collectively addressed [82].

From the perspective of users, the impact of the IoT can be noticeable in both domestic and working environments through the utilization of applications (e.g., Web or smartphone apps) that provide an interface to monitor, control, and analyze values of the surrounding things. From the perspective of companies, the IoT provides business opportunities to
2. Background

A typical IoT setup is depicted in Fig. 2.1. This setup includes various small devices with sensors and/or actuators, also referred to as IoT devices, which communicate with each other via wired or wireless means. Wireless low power communication however, e.g., over Bluetooth or Zigbee, is more common due to providing a longer lifespan for devices that operate on batteries [80]. These devices are usually scattered in order to be able to monitor and/or affect a larger area. While being scattered, they use wireless transmissions to send sensor measurements or to receive actuator commands. In such setups, there is usually a central coordination component that acts as a gateway to the Internet, and exposes access to all the sensors and actuators through an Application Programming Interface (API) [245].

Since the gateway implements an API, the device that hosts the gateway is usually more computationally powerful than the IoT devices. Thus, other functionalities can also be featured in this device, such as the creation of automatic events to turn on/off actuators based on the sensor measurements [51]. Users can add/remove/configure events through the API. The API is usually based on a communication protocol such as Hypertext Transfer Protocol (HTTP) which allows access through the Internet or the Local Area Network (LAN).

When the goal is to realize large-scale applications, the gateway can also be used for sending all the information from the IoT devices to the cloud [121]. This can have two prime benefits: First, having the same cloud node collecting information from many gateways means that events can be created on the cloud node, which include various devices that may be deployed in different areas. Consequently, the user applications that communicate with the cloud can retrieve more information, and have control over an
increased number of devices. Second, the cloud can provide more computational resources which can be used for serving more users, and for implementing additional algorithms that mine information in order to provide the users with various features and interesting analytics [122].

Due to such advantages, cloud computing resources are used widely for IoT applications. For this reason, in the following section we discuss more concretely the deployment of IoT applications in the cloud, and in general, the cloud computing paradigm.

2.2 Cloud Computing

Cloud computing refers to the paradigm of enabling on-demand network access to computational resources such as networks, storage, servers, and services, that can be provisioned and released automatically with minimal interactions with the provider [61]. This means that based on this paradigm, a user can utilize a client application (e.g., using smartphones, or laptops) to acquire immediate access to computational resources which are delivered over a network. These resources are usually located in the provider’s data center. In order to serve a larger number of users, the computational resources of one or more data centers of a cloud provider are combined into a shared pool of resources. From this pool, a set of resources can be virtualized as a single compute node. A user gains access to the resources of this node through an API. This way, resources can be assigned and released dynamically to compute nodes, based on the needs of the user [98].

There are three main service models whereby users can acquire access to cloud computing resources [157]: i) Software as a Service (SaaS) is the model which allows users to execute applications that run in the cloud rather than on the local resources of the user devices. An example of SaaS is the use of online text editors such as Google Docs, or online storage such as Dropbox. ii) Platform as a Service (PaaS) is a model aimed for application developers. When using this model, a user gains access to programming languages, libraries, and development tools supported by the provider, which can be used for designing applications that eventually run in the cloud. Thus, in this model the developer designs an application, while the cloud provider handles deployment and execution aspects. An example of PaaS is the Google App Engine. iii) Finally, Infrastructure as a Service (IaaS) is a service model for users who want to be able to provision computational resources arbitrarily, and have control over the underlying Operating System (OS) as well. An example of IaaS is the Google Compute Engine, or the Amazon Elastic Compute Cloud.

The use of these service models provides storing, processing, and management of data in a pool of resources that can be virtually unlimited. This makes the cloud an excellent platform for the deployment of IoT applications [220]. In fact, since the amount of data produced by IoT devices can be massive (while the computational capabilities of these devices may be very poor), the cloud is imperative for the support of IoT applications, especially when these applications handle information from millions of devices [88].
2. Background

An example of an IoT setup with cloud computing resources is depicted in Fig. 2.2. The IoT devices connect to a gateway which implements an API, similar to Fig. 2.1 which shows an IoT setup without cloud computing resources. However, Fig. 2.2 shows that many gateways can connect to the cloud which is able to serve a larger number of users, while also providing access to more IoT devices.

When combining the IoT with the cloud, data from different locations is collected, processed, and analyzed through various applications in order to create interesting and meaningful information for the users. The resulting IoT-cloud paradigm connects the things to the Internet, enabling a variety of large-scale applications, such as energy management, vehicular networking, and smart cities [28].

The wide range of use cases for combining the IoT with cloud computing can significantly increase the number of Internet-connected devices, and consequently the amount of data that travels through the Internet [21,234]. This huge amount of data may overwhelm the network causing bottlenecks which in turn can result in increased latency and low bandwidth [63]. To cope with such issues, the utilization of edge computing resources has been proposed. Edge computing suggests the use of compute nodes which reside in proximity of the IoT devices. This hinders the accumulation of data in one central location thereby avoiding the formation of bottlenecks. Furthermore, since the data can now be processed close to the data sources, the communication latency is reduced compared to utilizing a remote cloud node [20].

Due to the benefits of edge computing, significant efforts have been made in the distributed systems community to realize the processing of data at the network edge. Such efforts, along with the edge computing paradigm in general, are discussed in the next section.
2.3 Edge Computing

Due to the advent of the IoT, a huge amount of data generated by IoT devices is sent to the cloud for processing [177, 213]. Since remote cloud compute nodes alone may not be able to process this data in a timely manner [41], edge computing has been proposed for reducing the communication latency [183]. To achieve this, edge computing relies on the utilization of distributed compute nodes which aim at processing the IoT data in the proximity of the IoT devices [113, 143]. Driven by the potential benefits of edge computing, cloud providers extend their network of data centers with additional sites which provide computational resources at the network edge. For instance, Google utilizes Edge Points of Presence in various regions around the world, and Microsoft uses Edge Zones in population centers which are far away from cloud infrastructure [2, 6].

Therefore, edge computing refers to the paradigm of utilizing the technologies developed in the context of cloud computing, but closer to the edge of the network where the IoT devices reside [48]. Presumably, this preconditions that there are available computational resources at the edge of the network, which are capable of implementing the cloud computing service models. This way, the individual computational resources at the edge can be considered as a shared pool of resources, which is available to nearby users.

The edge computing paradigm can be especially useful for IoT applications which require access to computational resources with very low communication latency [27, 242]. Such applications can be related to, e.g., industrial settings and smart grids [30, 59, 102, 180]. Meeting the latency requirements of such applications may not be possible when using only traditional cloud computing resources due to the potentially massive load of incoming traffic, and the long physical distance between the cloud and the IoT devices [242].

A typical edge computing setup may include IoT devices which generate data and consume commands, gateways, nearby compute nodes, and users. An example of such a setup is depicted in Fig. 2.3. The IoT devices communicate with a gateway that sends/receives information to a nearby compute node. The users also connect to a nearby compute node, thereby avoiding high-latency communication with a remote cloud.

Since the emergence of edge computing, other similar paradigms for executing computations at the edge of the network have also been proposed [140, 226]. A notable paradigm focusing on processing IoT data is the fog computing paradigm which is presented in the next section.

2.4 Fog Computing

The fog computing paradigm is an extension of cloud computing that includes both cloud resources and resources at the edge of the network [69]. This paradigm is motivated by the need to meet requirements of IoT applications such as mobility support, location awareness, and very low latency [40, 176]. According to the fog computing paradigm, this can be achieved when using an interplay of cloud and edge resources, which can be particularly useful for data management and analytics [40].
2. Background

There is a number of characteristics which make fog computing an important extension to the cloud [40]: i) Fog computing includes edge compute nodes with location awareness, i.e., the nodes are aware of the proximity to other nearby nodes. This can facilitate applications with very low latency requirements such as gaming, augmented reality, and video streaming. ii) In contrast to the cloud computing paradigm, fog computing envisions a wide geographical distribution of resources and applications. For example, the use of access points on highways, which host applications for users in moving vehicles. iii) As a consequence of the wide geographical distribution, a very large number of participating nodes is expected, which span large areas. iv) Large-scale deployment of sensor devices which monitor the environment. v) Heterogeneity of compute nodes which may be utilizing the resources of the edge network (e.g., access points), the core network (e.g., routers and switches), or the cloud (e.g., data centers).

Due to such characteristics, fog computing specializes in applications which require both nearby compute nodes with location awareness, and remote cloud nodes which can provide global coordination [33]. Such applications are expected to be critical in the advancement of various modern technologies such as connected vehicles and big data analytics [38, 152].

A typical fog computing setup is depicted in Fig. 2.4. As shown in this figure, the IoT devices connect to a gateway which communicates with a nearby compute node. In addition, the nearby compute nodes can also communicate with the cloud. As a result, applications that require low latency can be deployed on nearby compute nodes, while...
Figure 2.4: Example of a fog computing setup with IoT devices, gateways, compute nodes, and users. The light blue nodes represent nearby compute nodes, whereas the dark blue node represents the cloud.

applications that do not have strict latency requirements can also be deployed in the cloud.

A distinguishing characteristic to separate fog from edge computing is that the fog envisions a hierarchy of compute resources which span from the cloud to the edge of the network [40]. Edge computing, on the other hand, aims at pushing the computations towards the edge of the network wherever there are available computational resources without explicitly including interactions with the cloud [103, 195].

At this point, it should be noted that the current literature of edge and fog computing can have inconsistencies regarding the utilized nomenclature [40, 222, 239]. This may be the outcome of having many research groups working on the general topic of computing at the edge of the network at the same time. As a result, each group may be using slightly different terminology. The terminology and the modeling decisions that we make in this thesis rely on related works from the literature [11, 181]. Nevertheless, additional definitions and assumptions are introduced when needed. Notably, this introduced information applies within the context of this thesis, but may not be generally applicable, especially when considering systems which do not align with the theoretical principles of fog computing.
CHAPTER

Self-Organization

In this chapter, we present the first contribution of this thesis, which is based on [112, 113, 115]. Contribution I is motivated by RQ1 and the very large number of participating compute nodes which may be present in fog computing systems (as discussed in Section 1.2). To manage such a large number of nodes, we present distributed algorithms which enable all the participating compute nodes of a fog computing system to self-organize without manual user configuration. To present these algorithms intelligibly, first we give an overview of the proposed approach in Section 3.1. Then, we discuss related approaches from the literature in Section 3.2. Afterwards in Section 3.3 we discuss in detail the proposed distributed algorithms, and in Section 3.4 we present an evaluation of these algorithms based on a prototype implementation using a real-world setup of nearby and remote compute nodes. Finally, we summarize the findings of this contribution in Section 3.5.

3.1 Overview

For this contribution, we design algorithms so that each time a new compute node joins a fog computing system, this new node and the existing nodes take network proximity measurements to each other. Then, based on these measurements, the nodes self-organize into a predefined structure (either hierarchical or flat) in a plug-and-play manner. The proposed algorithms are designed to be agnostic of the utilized network proximity measure. We do this to make our algorithms more flexible since in some fog computing systems, one proximity measure may be more appropriate (or available) than others. Nevertheless, we make sure that our algorithms can theoretically work with latency measurements and number of hops, although practically (i.e., in the evaluation in Section 3.4) we use only the number of hops as an indicator of network proximity. The reason we focus on these two measures is that they are typically associated with network proximity in end-to-end communication [196].
Figure 3.1: Example of a system structure with low and high proximity logical links.

We position this work in the area of distributed algorithms for the coordination of compute nodes [46]. Such algorithms aim at organizing a number of nodes that may become too large to be centrally coordinated, which can find use in various IoT scenarios [46]. This is an important aspect in today’s networks because more and more computations are performed on a potentially large set of compute nodes which require self-organization to achieve the desired system structure [25]. Potential use cases include, e.g., multimedia applications. For example, an image processing application may require computations for user-generated data with low communication latency, while utilizing a network of (non-mobile) compute nodes [16]. In case such applications are deployed on large-scale networks (such as the Internet), the compute nodes may be agnostic of the underlying physical network topology [92, 236]. When two compute nodes communicate with each other, all the physical links on the path between these two nodes serve as a single logical link [124]. This logical link however, hides the information of network proximity. Consequently, a compute node may not know which other compute nodes reside in close network proximity, i.e., which other compute nodes can be reached with low communication latency.

Therefore, in a network of compute nodes as shown in Fig. 3.1, assuming that the spatial distance between nodes in the figure corresponds to network proximity, the compute nodes can form various logical links to each other. Through these links, the nodes exchange data (initially acquired by a data source, e.g., images from a smartphone as shown in the figure) in order to perform the required computations in a distributed manner. Some of these links may connect nodes in low network proximity (straight lines in Fig. 3.1), and bear lower communication latency than others that connect nodes in high network proximity (dashed lines in Fig. 3.1). Even though the network proximity between the compute nodes is abstracted by the logical links, it is still possible for the nodes to take measurements, e.g., using round-trip times, or hop count. This way, each node can acquire information about the network proximity, and consequently, the communication latency to reach other nodes. Thus, while a simple approach may organize the participating compute nodes disregarding the network proximity between the nodes, taking measurements and organizing the compute nodes accordingly, can reduce the
communication latency significantly (this is also shown in the evaluation in Section 3.4).

For this reason, our goal is to allow the compute nodes of fog computing systems to self-organize based on network proximity. The proposed algorithms enable the compute nodes to take network proximity measurements, and to select logical links between nodes in close network proximity automatically. As a result, we manage to reduce the communication latency, to improve the Quality of Service (QoS) of the applications, and to allow the system to scale to a large degree while incorporating various geographically distributed compute nodes. This can be particularly useful for systems that span over large geographical areas [56, 125, 167, 225].

Our approach can be used either as the basis for a novel fog computing system, or as a supplement to already proposed systems which rely on a hierarchical or flat system structure, but do not describe how to create it. This contribution includes the design and implementation of two distributed algorithms that enable the self-organization of the nodes of a fog computing system based on network proximity. In addition, we implement the proposed algorithms, and we experiment with image processing and smart city use cases. Our results show that in fog computing systems with geographically distributed compute nodes, self-organization reduces the communication latency of latency-sensitive processes in both hierarchical and flat structures by 27–43%, and increases the available network bandwidth by 36–86% (compared to alternative methods). When executing latency-tolerant processes, self-organization reduces the latency by 15–32%, and increases the available network bandwidth by 14–49%. Moreover, based on empirical results and by using predictive methods, we show that a flat structure is able to scale better than the commonly used layered hierarchy due to generating less overhead when the size of the system grows.

### 3.2 Related Work

This chapter investigates self-organizing fog computing systems that consist of geographically distributed compute nodes. Accordingly, related work is primarily from the field of fog computing system organization and establishment. Most related approaches in this field utilize either a hierarchical or a flat structure for organizing the various compute nodes of a fog computing system [104, 118], as shown in Fig. 3.2. Notably, both Fig. 3.2a and Fig. 3.2b depict the exact same network of geographically distributed compute nodes, which is why the disposition of the nodes is identical. However, each approach may change the logical layout of the system (i.e., hierarchical or flat), and consequently the incurred communication latency when executing applications (as shown in Section 3.4).

To present existing approaches in a comprehensive manner, we divide them into these two categories based on the utilized system structure. Specifically, fog computing systems that form hierarchical structures are presented in Section 3.2.1 and fog computing systems that form flat structures are discussed in Section 3.2.2. Afterwards, in Section 3.2.3, we
discuss work from the field of self-organization and self-organizing clustering, which is also important related work for our approach.

3.2.1 Hierarchical Structures

Most current state-of-the-art research approaches for creating fog computing systems utilize a hierarchical structure. For instance, Kiani et al. propose a hierarchical structure with geographically distributed compute nodes, and investigate related performance benefits based on simulations that consider either negligible or significant delay between compute nodes. Gao et al. present a hierarchical fog computing system with dynamic resource allocation based on traffic prediction, and show latency reductions using extensive simulations. Adhikari et al. propose a hierarchical fog computing system for executing applications based on priorities. In this system, it is assumed that there are gateways which receive tasks from IoT devices, and offload these tasks to the compute nodes of the hierarchy, while no algorithm to achieve this hierarchy is presented.

Even though such approaches can be used for creating hierarchical fog computing systems, they do not provide algorithms that take into account actual network proximity measurements. This may compromise the efficiency of the system, as discussed in Section 1.2. Moreover, none of these works provides results from real-world implementations with distributed compute nodes. In this chapter, we design algorithms for creating hierarchical fog computing systems based on network proximity, and we also implement them considering actual use cases. Furthermore, we use logic similar to the aforementioned approaches as baselines (in Section 3.4.2), in order to show the differences of considering network proximity measurements in a real-world setting.

Regarding further related work, Skarlat et al. propose the FogFrame framework for building a hierarchical structure with the cloud at the top, and various compute nodes below, which are organized in layers. To create this structure, FogFrame relies on a fog controller which decides where to place each new node that joins the system, based on
the Euclidean distance from preconfigured location coordinates. Thus, this work relies on two assumptions: i) Each node is preconfigured with location coordinates. ii) The distance from these coordinates actually corresponds to network proximity. Instead of relying on preconfigured information, in our approach the nodes self-organize based on the proposed distributed algorithms. Moreover, the self-organization takes into account actual network proximity measurements (i.e., network hops, or latency measurements) in order to organize the geographically distributed compute nodes more efficiently.

Saurez et al. [199] present a fog computing system which consists of geographically distributed fog and cloud compute nodes. To create this system, each new compute node becomes a child (in the hierarchy) of a preexisting parent node. This way, while new compute nodes join, the system grows in a hierarchical manner. Even though this work targets geographically distributed compute nodes, the proposed algorithms do not consider an increasing number of nodes, which may create scalability concerns. In this chapter, we propose algorithms for organizing the compute nodes, and we examine the behavior of these algorithms when the system size grows.

Mortazavi et al. [160] present CloudPath which operates on a hierarchical tree-like overlay due to assuming that network devices (e.g., routers or switches) between the edge and the cloud act as compute nodes. Thus, CloudPath does not actually create the hierarchical structure but nevertheless, the data is routed through a hierarchy with the edge nodes as leaves, the cloud as root, and various compute nodes in the middle. This approach does not measure the network proximity between nodes, but since compute nodes exist on the network path of the data, network proximity can be considered evident. Even though a similar assumption has also been proposed by Ascigil et al. [23], it should be noted that such approaches rely heavily on infrastructure support. For example, the network operator is required to route all the CloudPath traffic to specific nodes. In our approach, the nodes use distributed algorithms in order to self-organize into a (hierarchical or flat) structure without assuming cooperation from the network operator.

Nguyen et al. [168] present ICN-Fog which operates on a hierarchical (or flat) structure whereby each node has a logical link to the cloud, and also to compute nodes in proximity. The specific manner that this structure is created is not discussed, and the proximity between nearby nodes is assumed without any actual measurements. However, the authors mention that the communication of the nodes can be based on a wireless protocol, e.g., 6LoWPAN. Notably, assuming wireless communication simplifies the problem since the nodes can be organized based on discovery, e.g., each node forms logical links to the nodes in range. However, this solution does not apply to wide-area distributed systems with nodes that are not within wireless range. Our approach on the other hand, aims at organizing the nodes of fog computing systems that may span over large geographical areas.
3. Self-Organization

3.2.2 Flat Structures

Even though most state-of-the-art systems in fog computing form hierarchical structures (as discussed in Section 3.2.1), there are also approaches which utilize flat structures to organize the compute nodes of fog computing systems [118]. For instance, Casadei and Viroli [47] focus on the problem of coordinating distributed compute nodes at the edge of the network, and design an approach for dynamically partitioning the system into areas each one governed by an elected manager-node. Rabay’a et al. [184] present a fog computing system whereby the compute nodes are organized into a peer-to-peer overlay which provides performance benefits due to minimizing the interactions with the cloud. Yu et al. [241] present a fog computing system consisting of geographically distributed fog compute nodes which are organized into a flat-structured virtualization plane, and a hierarchical control plane, that aim at facilitating large-scale IoT.

Such approaches can be used for creating flat fog computing systems. However, they do not take into account the network proximity of the compute nodes. Moreover, none of the aforementioned works shows results from real-world implementations with geographically distributed compute nodes. In this chapter, we design algorithms for creating flat fog computing systems based on network proximity, and we also implement them in a real-world setting. Furthermore, we use logic similar to the aforementioned works as a baseline (in Section 3.4.2), in order to show the differences to our approach in real-world scenarios.

Regarding further related work, Song et al. [210] present Peer Data Sharing (PDS) which is a mechanism for organizing compute nodes at the edge of the network in order to exchange data. The organization of the nodes in PDS occurs opportunistically based on wireless discovery. Thus, this approach relies on wireless communication to enable the communication between nodes in proximity. Furthermore, PDS relies on the assumption that the target geographical area, and the number of participating compute nodes remain limited. Therefore, the authors do not address the problem of organizing the nodes of large-scale systems, or systems with nodes that span over large geographical areas. On the contrary, for our approach we design algorithms aimed for geographically distributed compute nodes. Moreover, we consider large-scale systems by examining and evaluating scalability aspects.

Tato et al. [216] present Koala which is an overlay network for organizing geographically distributed compute nodes. Each node in Koala maintains a routing table with the addresses of other nodes of the system. To create this table, every new node requests information from existing nodes, and measures proximity using latency and hop count. This work is presented only through experimental simulations. Hence, empirical results and technical information regarding measuring the network proximity between geographically distributed nodes are not provided. In this chapter, we address these matters, and we present the insights of our efforts. Moreover, we measure the overhead of organizing the nodes in order to analyze the system’s scalability. This is not discussed in the context of Koala.

Jiang and Tsang [90] propose organizing the compute nodes of a fog computing system into
an overlay network. This overlay is able to facilitate offloading of computational tasks to compute nodes, considering delay-related requirements in order to provide fog computing services. To account for the proximity between nodes, the proposed algorithms consider a function which outputs the disutility of the communication delay. In order to produce results for the evaluation of this work, the authors model the disutility function with a range of preconfigured values, and perform extensive experimental simulations. Therefore, this approach assumes that the communication delay between nodes is known a priori. In our approach, we design algorithms that integrate network proximity measurements in order to create fog computing systems without any prior knowledge. Moreover, we deploy an implementation of our algorithms on geographically distributed compute nodes, and we show results from real-world fog computing systems.

To sum up the discussion, Table 3.1 provides an overview of the related work. As shown in this table, there are various approaches in the literature that can be used for organizing the compute nodes of a fog computing system into a hierarchical or flat structure. However, most of these works do not consider actual network proximity measurements, and do not provide a scalability analysis. Furthermore, to the best of our knowledge, none of them provides a comparison of the two structures considering real-world scenarios with geographically distributed compute nodes. In this chapter, we design algorithms for hierarchical and flat self-organization that target fog computing systems which span over large geographical areas, and we also compare them regarding communication latency, network bandwidth, and scalability.

### Table 3.1: Overview of related work.

<table>
<thead>
<tr>
<th>Authors</th>
<th>Proximity measurements based on</th>
<th>Scalability analysis</th>
<th>Presented system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adhikari et al.</td>
<td>✔</td>
<td></td>
<td>Simulated</td>
</tr>
<tr>
<td>Gao et al.</td>
<td>✔</td>
<td></td>
<td>Simulated</td>
</tr>
<tr>
<td>Kiani et al.</td>
<td>✔</td>
<td></td>
<td>Simulated</td>
</tr>
<tr>
<td>Skarlat et al.</td>
<td>✔</td>
<td>Coordinates</td>
<td>Implemented</td>
</tr>
<tr>
<td>Saurez et al.</td>
<td>✔</td>
<td>Coordinates</td>
<td>Implemented</td>
</tr>
<tr>
<td>Mortazavi et al.</td>
<td>✔</td>
<td></td>
<td>Emulated</td>
</tr>
<tr>
<td>Nguyen et al.</td>
<td>✔</td>
<td></td>
<td>Not defined</td>
</tr>
<tr>
<td>Casadei and Viroli</td>
<td>✔</td>
<td>✔</td>
<td>Simulated</td>
</tr>
<tr>
<td>Rabay’a et al.</td>
<td>✔</td>
<td>✔</td>
<td>Simulated</td>
</tr>
<tr>
<td>Yu et al.</td>
<td>✔</td>
<td>✔</td>
<td>Not defined</td>
</tr>
<tr>
<td>Song et al.</td>
<td>✔</td>
<td>✔</td>
<td>Implemented</td>
</tr>
<tr>
<td>Tato et al.</td>
<td>✔</td>
<td>Hops, latency</td>
<td>Simulated</td>
</tr>
<tr>
<td>Jiang and Tsang</td>
<td>✔</td>
<td></td>
<td>Implemented</td>
</tr>
<tr>
<td><strong>Our Approach</strong></td>
<td>✔</td>
<td>✔</td>
<td>Implemented</td>
</tr>
</tbody>
</table>
3. Self-Organization

3.2.3 Self-organization

There is also work in the literature that does not target fog computing systems explicitly, but provides self-organizing concepts for the organization of distributed nodes, and is therefore of relevance to this chapter. For example, Casadei et al. [46] propose creating a system structure for edge computing, using a self-organizing approach that divides the nodes into regions, and each region is coordinated by a leader node. However, such edge computing approaches focus on nodes that reside close to each other, and do not consider interactions with nodes in distant geographical areas, e.g., with remote cloud nodes. Leitão et al. [129] propose a distributed algorithm for peer-to-peer communication that allows multiple nodes to self-organize into groups which are referred to as super-peers. However, such peer-to-peer approaches aim at facilitating information exchange among distributed nodes, and do not focus on leveraging the network proximity between the nodes in order to lower the latency, which is a prime goal of fog computing. Dabek et al. [57] propose a coordinate system based on a two-dimensional Euclidean model that can be used to estimate latency between nodes with a potential percentage of error. This approach targets file-sharing applications that do not typically have stringent latency requirements. Therefore, some potential error can be tolerated. However, such an error may be unacceptable for fog computing applications with very low latency requirements [10]. Audrito et al. [25] propose a gradient algorithm for finding paths between nodes in order to create efficient self-organizing systems. In this approach, each node interacts with the other nodes through broadcasts. Thus, it is assumed that each node communicates with neighbor nodes without knowing their network addresses, since a broadcast reaches all the other nodes that are listening [68]. However, such approaches are not applicable to geographically distributed compute nodes that communicate over the Internet, which is our target environment. The reason is that Internet protocols such as the Transmission Control Protocol (TCP) or HTTP provide end-to-end communication, and do not support broadcast operations (i.e., sending messages without knowing the destination Internet Protocol (IP) address) [179, 200].

Additionally, there is related work from the field of self-organizing clustering in ad hoc networks. Ad hoc refers to self-organizing networks with a high degree of mobility which causes the network’s topology to change rapidly and unpredictably [97, 156]. For example, Liao et al. [135] propose a clustering algorithm to enable hierarchical communication among nodes in wireless sensor networks. Johnen et al. [97] present a clustering algorithm that focuses on stability by ensuring that the network converges towards the desired system structure. Mitton et al. [156] propose a stabilization algorithm for clustering in ad hoc networks, which considers scalability aspects.

The discussed approaches rely on the ability of the nodes to discover all the other nodes within wireless range. This does not apply to geographically distributed compute nodes that communicate with each other over the Internet (i.e., using also wired means), which is the scope of our work.
3.3 Algorithms for Self-Organization

In this section, we present the design and functionality of two algorithms for organizing the geographically distributed compute nodes of a fog computing system. To this end, we provide the application model in Section 3.3.1, the system model in Section 3.3.2, and a description of the proposed system functionality in Section 3.3.3. Afterwards, we present algorithms for organizing geographically distributed compute nodes into a hierarchical and a flat structure in Sections 3.3.4 and 3.3.5, respectively.

3.3.1 Application Model

Our application model is based on a widely-used fog computing model from the literature [87]. In order to leverage the multiple compute nodes of a fog computing system, every application is executed in a distributed manner by many nodes [187]. For this reason, each application is assumed to consist of multiple processes. These processes are independently deployable, but may interact with each other through dependencies (i.e., the output of one process may become the input for another) in order to execute the functionality of the application as a process composition [87]. Thus, while running, each process performs application-related tasks such as: analytics, machine learning, aggregating, storage, etc. The input data of these processes is acquired from the data sources which can be either devices with integrated sensors (e.g., from the IoT) [101], or other processes. The final output of the processes can be a command for storage, notification, or actuation of a device that integrates an actuator.

Since fog computing aims at facilitating applications with processes that may require low communication latency, we distinguish the processes based on latency requirements [87]. Therefore, a process can be either latency-sensitive or latency-tolerant, e.g., for actuation commands or for sensor monitoring, respectively [74]. This distinction enables the handling of different processes in different ways. In general, latency-sensitive processes should be deployed on compute nodes close to the data source where the communication latency is low. Latency-tolerant processes on the other hand, can also be deployed on nodes outside the proximity of the data source, e.g., in a remote cloud node. For example, Fig. 3.3 shows the application model of an image processing application. Specifically, Fig. 3.3a shows the processes of the application, and Fig. 3.3b shows the placement of these processes on distributed compute nodes that span from the cloud to the edge of the network.

A more concrete example of such an image processing application (as shown in Fig. 3.3), is a smart doorbell. Such an application can operate as follows: First, the doorbell camera (as shown at the bottom of Fig. 3.3b) captures the image feed periodically, and sends pictures to a nearby compute node that executes the motion detection process. When this process detects the motion of a person, the image with the person is sent to the face recognition process. Then, the face recognition process retrieves the faces of the house residents from the cloud storage, and compares these faces with the image from the motion detection process. If there is a match, an actuation command is sent back in order
3. Self-Organization

(a) The processes of an image processing application.

<table>
<thead>
<tr>
<th>Latency-sensitive processes</th>
<th>Latency-tolerant processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motion detection</td>
<td>Analytics</td>
</tr>
<tr>
<td>Face recognition</td>
<td>Storage</td>
</tr>
</tbody>
</table>

(b) The placement of the processes on distributed compute nodes.

Figure 3.3: Example of the application model based on an image processing application.

to open the door automatically. Furthermore, the face recognition process sends all the comparisons to the cloud for storage and analytics purposes, e.g., to maintain a record of visitors. Notably, a smart doorbell may not be an application with stringent low latency requirements. Nevertheless, executing some of the involved processes with low latency (e.g., in order to open the door quickly in the face of residents), may increase the QoS significantly. Other processes on the other hand, might not be affected by latency (e.g., performing monthly analytics regarding visitors). Thus, as latency-sensitive we consider the processes for which lower latency may result in higher QoS and as latency-tolerant we consider the processes for which lower latency does not affect the QoS significantly. Hence, using this system model, and distinguishing between latency-sensitive and latency-tolerant processes, may benefit various applications.

There are various approaches for the placement of the processes on the available compute nodes, most of which are based on optimization methods [19, 38, 110, 224]. Such approaches can be considered complementary to our approach, since the system structure and the latency between nodes are usually required as input of the placement algorithms [26, 109, 111].

3.3.2 System Model

Our system model is based on well-known system models from the literature [11, 173, 201]. We define a fog computing system as an environment that includes multiple geographically distributed compute nodes that span from the cloud to the edge of the network, along...
with IoT devices which integrate sensors and actuators [101, 221].

The amount of computational resources that each compute node provides can vary, commonly but not exclusively, based on the location of the node. This means that the compute nodes closer to the edge tend to integrate less resources than those closer to (or in) the cloud [187]. Typically, when a node integrates a large amount of computational resources, access to these resources is granted through dynamically created Virtual Machines (VMs) which retain only a fraction of the initial computing power. In this thesis, we consider a compute node as a set of computational resources either physical or virtual, which are provided through an API that can be accessed using the node’s IP address (and port number).

Moreover, we make the following assumptions: Since we target geographically distributed compute nodes, we assume that these nodes communicate with each other through the Internet. We also assume that the first compute node of the system, which is also the root in hierarchical structures (e.g., the top node in Fig. 3.2a) is always a cloud compute node, because the cloud can provide a global contact point with a static IP address that can be used by other nodes that want to join the system. Thus, all nodes are assumed to know this address, and to use it to bootstrap the communication with the fog computing system.

Additionally, we assume that all the compute nodes are non-mobile (i.e., the addresses of the nodes do not change during the runtime of the system), while the data sources might be mobile [46]. For the case that nodes join and leave (i.e., fail or disconnect) the system concurrently, we precondition a mechanism that maintains the system structure, as discussed in previous work [115]. By doing this, we decouple the problem of handling unexpected joining and leaving of compute nodes that are highly dynamic, and we focus on the problem of creating efficient system structures. We therefore assume that multiple compute nodes join the system sequentially in a transactional manner.

Interestingly, there are other approaches in the literature which provide concepts and mechanisms for the aforementioned assumptions. For example, cases with nodes that may become unavailable at any moment can be addressed using a reactive or proactive fault-tolerance mechanism [137]. The former aims at reducing the effect of failures, while the latter tries to avoid failures altogether [125]. There are also approaches which focus on migrating applications between nodes in order to perform load balancing tasks [117, 136]. Additionally, some approaches from the literature cope with node mobility. Even though the nodes are assumed non-mobile in our work, other approaches deal with mobility using, e.g., opportunistic routing [231], or context of users [158].

In the following, we provide the basic notation of the system model, which is also summarized in Table 3.2. The compute nodes that intend to join the system are denoted as \( n_1, n_2, \ldots, n_N \). The first compute node \( n_1 \) is used as the initial contact point for other nodes to join. Each new node that requests to join the system, is denoted as \( n_{\text{new}} \). The existing node that acts as the contact point is noted as \( n_{\text{cp}} \).

In order to communicate with the rest of the compute nodes in the system, each node needs to select neighbors. Neighbors are defined as two nodes that are connected by
3. Self-Organization

Table 3.2: The basic notation of the system model.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n_1, n_2, \ldots, n_N)</td>
<td>Set of all the nodes of the system.</td>
</tr>
<tr>
<td>(n_{new})</td>
<td>New node to be added to the system.</td>
</tr>
<tr>
<td>(n_{cp})</td>
<td>Existing node to be used as contact point.</td>
</tr>
<tr>
<td>(n_{pro})</td>
<td>Existing node closest to (n_{new}).</td>
</tr>
<tr>
<td>(n_{far})</td>
<td>Existing node farthest away from its neighbors.</td>
</tr>
<tr>
<td>(S)</td>
<td>Maximum group size.</td>
</tr>
<tr>
<td>(G_{new})</td>
<td>New group for adding (n_{new}) and (n_{cp}).</td>
</tr>
<tr>
<td>(G_{cp})</td>
<td>The group of the contact node (n_{cp}).</td>
</tr>
<tr>
<td>(G_{upd})</td>
<td>Updated group that includes (n_{new}).</td>
</tr>
<tr>
<td>(P(n_i, n_j))</td>
<td>Network proximity between (n_i) and (n_j).</td>
</tr>
</tbody>
</table>

a logical link, i.e., they know each other’s IP addresses. Nodes may have more than one neighbor: When many nodes know each other’s IP addresses, they are considered a group. A group is defined as a complete graph (or clique) of nodes, i.e., every pair of nodes in a group is connected by a logical link. For instance, the nodes \(n_1, n_2, n_3,\) and \(n_6\) in Fig. 3.4a (and also the nodes \(n_1, n_3, n_7,\) and \(n_9\) in Fig. 3.4b) belong to the same group, and consider each other neighbors. Groups are assumed to have a fixed maximum size \(S\) which can be used for controlling the layout of the structure. In Fig. 3.4a for example, \(S = 4\) ensures a hierarchical tree-like structure with groups that can have up to four nodes. Furthermore, maintaining a fixed \(S\) for both hierarchical and flat structures enables creating comparable fog computing systems which differ only in the utilized system structure. This aids in drawing valuable conclusions regarding the benefits of each structure, as discussed in Section 3.4.

When a new group of nodes is created, e.g., due to new nodes joining the system, it is referred to as \(G_{new}\). In order to communicate with other compute nodes, each node joins (at least) one group (which includes neighbors), and all the groups together constitute the fog computing structure, as shown in Fig. 3.4. Thus, a structure of nodes is defined as the high-level architecture of the system, which shows the manner whereby the compute nodes are interconnected and communicate with each other (as shown in Fig. 3.4) \[112\]. A hierarchical structure has a root (e.g., \(n_1\) in Fig. 3.4a), and the nodes are organized in layers, whereas a flat structure does not have a root or layers \[112\]. Further information on hierarchical and flat structures is provided in Sections 3.3.4 and 3.3.5 respectively, including similarities and differences. For instance, an example of a difference is that in a hierarchical structure the nodes cannot belong to more than 2 groups, as shown in Fig. 3.4a whereas in flat structures this is not a constraint, e.g., \(n_3\) in Fig. 3.4b belongs to 3 groups. The goal when creating either structure is to select neighbors in close network proximity for each node, so that the communication through the structure occurs with low latency and high bandwidth.

For organizing the compute nodes based on network proximity, which may be essential to
3.3. Algorithms for Self-Organization

Figure 3.4: Examples of fog computing structures according to the proposed system model (when the compute nodes join sequentially, i.e., \( n_1 \) is the first, \( n_2 \) is the second, \( n_3 \) is the third, etc.).

Fog computing systems (as discussed in Section 1.2), the following additional notation is used. The network proximity between two nodes \( n_i \) and \( n_j \) is denoted as \( P(n_i, n_j) \), and represents the network distance, e.g., based on network latency (round-trip or one-way), or the number of hops, between \( n_i \) and \( n_j \). The selection and implementation of specific network proximity indicators (i.e., number of hops, or network latency), are discussed further in our evaluation in Section 3.4.1. Thus, when we mention the proximity between compute nodes hereinafter, we refer to network proximity.

According to the proposed algorithms (which are presented in Sections 3.3.4 and 3.3.5), when joining, \( n_{\text{new}} \) selects the group that contains the node of the closest proximity, which is referred to as \( n_{\text{pro}} \). When a group exceeds the capacity \( S \), the node that resides the farthest away from the others (within the group), noted as \( n_{\text{far}} \), leaves the group. For a group of size \( S \), when the nodes of the group are \( n_1, n_2, \ldots, n_S \), for \( n_{\text{far}} \) applies:

\[
\sum_{i=1}^{S} P(n_{\text{far}}, n_i) = \max \left\{ \sum_{i=1}^{S} P(n_1, n_i), \ldots, \sum_{i=1}^{S} P(n_S, n_i) \right\}
\]

This means that \( n_{\text{far}} \) is the node for which the sum of the proximity values towards the rest of the nodes in the group, is the maximum among all the nodes of the group. Since all the neighbors have the exact same view of the group they belong to, even if this metric points to more than one node, it is possible to have a common strategy for selecting a unique \( n_{\text{far}} \), e.g., using the first occurrence.
3. Self-Organization

In Fig. 3.5 we show the high-level architecture of a compute node, along with the interactions with its neighbors, and with IoT devices. The IoT devices and the compute nodes of the system can be geographically distributed in various locations, but are expected to communicate with each other through the Internet. The goal is to process the generated IoT data by nodes in proximity thereby with low latency and high bandwidth. To execute an application, the IoT devices send the IoT data to nearby compute nodes which use this data as input for the hosted processes. In order to reach other nodes which reside farther away, the executed processes send data to the neighbor nodes, which is how data travels through the system.

To achieve this functionality, each compute node has four internal components, as shown in Fig. 3.5. The Communication component is used for the communication with other nodes and with the IoT devices. The Distributed Algorithm component runs the proposed algorithms in order to handle the self-organization of the compute nodes, and to select neighbors. The neighbors are stored in the Groups along with the proximity measurements. The Processes component runs the processes hosted by the node, and uses the Groups in order to send data to the neighbors (through the Communication component). Notably,
the granularity of the components aims at facilitating the compatibility of alternative fog computing systems (e.g., [96] or [207]) with our algorithms. To achieve this, the Groups are also accessible through the Communication component. This way, an external component that handles the execution of the processes can also leverage the groups in order to discover compute nodes in proximity, and to abide by the system structure of the utilized algorithm (implemented in the Distributed Algorithm component). In the following, we design two distributed algorithms (one that results in a hierarchical structure, and one that results in a flat structure) which can be used as the logic of the Distributed Algorithm component.

### 3.3.4 Hierarchical Structures

As discussed in Section 3.2, various approaches have been proposed for organizing the compute nodes of a fog computing system in a hierarchical manner. In this section, we further advance the typical hierarchical structure by designing a distributed algorithm for the self-organization of geographically distributed compute nodes based on proximity. This can be essential for fog computing systems that span over large geographical areas since in such cases, close proximity cannot be assumed, as discussed in Section 1.2.

In the hierarchical structure, the compute nodes form a layered hierarchy, and communicate with the neighbors of adjacent layers, as shown in Fig. 3.4a. Thus, by sending data to neighbors, the data travels up/down the hierarchy. Notably, Fig. 3.4a shows nine compute nodes that have joined the system sequentially (i.e., \( n_1 \) is the first, \( n_2 \) is the second, \( n_3 \) is the third, etc.). However, the arrangement of these nodes seems abstract. This can be the effect of organizing the compute nodes based on proximity. Every time a new node joins, this new node along with existing nodes self-organize, so that the new compute node has neighbors in close proximity. In the following, we define some additional notation which is exclusive to the hierarchical structure, and then we show how such a structure can be created.

In addition to the already mentioned definitions (presented in Section 3.3.2), for the hierarchical structure we use the terms parent, sibling (i.e., same parent), and child for neighbors that reside above, next to, or below a node, respectively. Notably, in the hierarchical structure each node (e.g., \( n_6 \) in Fig. 3.4a) belongs to (at most) two groups: one that includes a parent and siblings (e.g., \( n_1, n_2, n_3, n_6 \)), and one that includes children (e.g., \( n_6, n_5, n_4, n_8 \)). Exceptions are the root of the hierarchy, which belongs to one group due to not having a parent or siblings (e.g., \( n_1 \)), and the leaves of the hierarchy, which belong to one group due to not having children (e.g., \( n_7, n_3, n_5, n_9, n_8 \)).

To organize the compute nodes of a fog computing system into a hierarchical structure based on proximity, we design the Hierarchical Proximity-Integrated (HPI) algorithm which is described in Algorithm 3.1 from the perspective of a new compute node which requests to join the system. HPI operates as follows: Initially, there is only one node which acts as the contact point \( n_{\text{cp}} \) for each new node \( n_{\text{new}} \) (Line 1). Upon request (Line 2), \( n_{\text{cp}} \) examines the group of its children. If this group is empty, then \( n_{\text{cp}} \) sends
3. **Self-Organization**

**Algorithm 3.1: HPI (Hierarchical Proximity-Integrated)**

<table>
<thead>
<tr>
<th>Procedure: joinSystemHPI(Address (n_{cp}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (\text{joinResponse} = \text{joinRequest}(n_{cp}))</td>
</tr>
<tr>
<td>2. if (\text{joinResponse.isEmpty()}) then</td>
</tr>
<tr>
<td>3. (G_{\text{new}} = \text{new Group}(n_{\text{new}}, n_{cp}))</td>
</tr>
<tr>
<td>4. (\text{notifyNeighbors}(G_{\text{new}}))</td>
</tr>
<tr>
<td>5. else (G_{cp} = \text{joinResponse.getGroup()})</td>
</tr>
<tr>
<td>6. (\text{addNeighbors}(G_{cp}))</td>
</tr>
<tr>
<td>7. (\text{notifyNeighbors}(G_{cp}))</td>
</tr>
<tr>
<td>8. if (G_{cp}.size &gt; S) then</td>
</tr>
<tr>
<td>9. (n_{far} = \text{findNFar}(G_{cp}))</td>
</tr>
<tr>
<td>10. if (n_{far} == n_{new}) then</td>
</tr>
<tr>
<td>11. (n_{pro} = \text{findNPro}(G_{cp}))</td>
</tr>
<tr>
<td>12. (\text{joinSystemHPI}(n_{pro}))</td>
</tr>
<tr>
<td>13. else</td>
</tr>
<tr>
<td>14. (G_{cp}.\text{remove}(n_{far}))</td>
</tr>
<tr>
<td>15. end</td>
</tr>
<tr>
<td>16. end</td>
</tr>
<tr>
<td>17. end</td>
</tr>
<tr>
<td>18. end</td>
</tr>
</tbody>
</table>

back an empty response (Line 3) which triggers \(n_{\text{new}}\) to create a new group with \(n_{cp}\) and \(n_{\text{new}}\) (Line 4), and then to notify \(n_{cp}\) (Line 5). This means that \(n_{\text{new}}\) is added as the first child of \(n_{cp}\).

If the group of children of \(n_{cp}\) is not empty, \(n_{cp}\) responds with the group of its children (Line 6). Then, \(n_{\text{new}}\) joins this group as a child of \(n_{cp}\) (Line 7), and notifies the neighbors about the arrival of a new node (Line 8), i.e., the neighbors also add \(n_{\text{new}}\). The arrival of a new node triggers all the siblings of the group to count the neighbors in the group that \(n_{\text{new}}\) joined. If the number of nodes in this group is smaller or equal to \(S\), then \(n_{\text{new}}\) stays at its current position as a child of \(n_{cp}\) (Line 17). However, if the number of nodes is greater than \(S\), i.e., the group exceeds its capacity (Line 9), then all the nodes of this group calculate the sibling that resides the farthest away from the others, i.e., \(n_{far}\) (Line 10), and remove \(n_{far}\) from the group (Line 15). Notably, the same functionality can be achieved if only one sibling calculates \(n_{far}\), and shares this information with the rest of the nodes in the group. The difference is that the latter case utilizes less computational resources (because only one node calculates \(n_{far}\)), but more network resources (because the IP address of \(n_{far}\) needs to be sent to all the nodes in the group). Thus, since in such systems it is more common that the network becomes a bottleneck, and not the lack of computational resources \([210]\), we propose that all nodes calculate \(n_{far}\) so that no additional network overhead is induced.

At this point, there are two cases for \(n_{far}\). If \(n_{far}\) is the same as \(n_{\text{new}}\) (Line 11), then \(n_{far}\) finds the sibling with the closest proximity \(n_{pro}\) (Line 12), departs from its current
Algorithm 3.2: Perspective of existing nodes

1  Procedure: uponJoin(Group $G_{upd}$, Address $n_{cp}$) 
2     if $G_{upd}$.size $>$ $S$ then 
3         $n_{far}$ = findNFar($G_{upd}$) 
4         $G_{upd}$.remove($n_{far}$) 
5     if $n_{far}$ == this then 
6         $G_{upd}$.remove() 
7         joinSystemHPI(Address $n_{cp}$) 
8  end 
9 end

Algorithm 3.2 shows what happens according to HPI from the perspective of the existing nodes. Upon the arrival of a new node, all the neighbors receive the updated group (Line 1) which is sent by the new node. Then, each node counts the neighbors in order to check if the group exceeds the capacity $S$ (Line 2). If the capacity of the group is exceeded, then all nodes find $n_{far}$ (Line 3), and remove it from the group (Line 4). Subsequently, the node that is $n_{far}$ (Line 5) deletes the group (Line 6), and requests to join the system again using the initial contact point (Line 7), i.e., the root of the system.

To make the process of adding new compute nodes more comprehensible, Fig. 3.6 shows a system with five compute nodes, i.e., $n_1, n_2, \ldots, n_5$, and how a new compute node $n_6$ joins according to Algorithm 3.1. Initially, as shown in Fig. 3.6a, $n_6$ requests to join from the root of the system. Then, $n_1$ responds with the IP addresses of its children (i.e., $n_2, n_3, n_4$), $n_6$ joins this group, and the neighbors (i.e., $n_1, n_2, n_3, n_4$) also add $n_6$. The addition of a new node triggers all the neighbors to count the size of the group. In this case, the size is five (i.e., larger than the maximum group size $S$ that equals four), which means that all nodes calculate $n_{far}$ which is $n_4$. Thus, the new compute node $n_6$ stays in this group, and $n_4$ leaves and requests to join again through the root $n_1$, as shown in Fig. 3.6b. Since $n_4 == n_{far}$ in the group of $n_1$, $n_4$ requests to join through the closest sibling which is $n_6$. Then, $n_4$ joins the group of $n_6$ as a child, as shown in Fig. 3.6c. The size of this group is three, i.e., smaller than $S$, which means that no further actions are required and the algorithm terminates.

This algorithm may raise the question of why $n_{far}$ is selected among the siblings, while the parent of the group is not considered. This is done in order to avoid that the following scenario occurs: $n_{new}$ joins a group of the structure. Then, the parent $n_{par}$ of that group, being the farthest away from its neighbors (i.e., $n_{par}$ is calculated as $n_{far}$), leaves the group, and requests to join the system again using $n_{pro}$ as contact point (Line 13). This way, $n_{new}$ travels downwards the hierarchical structure, in order to find a group with nodes in close proximity. The second case is that $n_{far}$ is one of the other siblings (i.e., not $n_{new}$). In this case, $n_{far}$ departs from its current group, and joins the system again using the root as contact point. This way, the most distant node of each group departs and joins again from the root of the hierarchy in order to find siblings in close proximity.
3. Self-Organization

Figure 3.6: Example of the process of HPI step-by-step when using a maximum group size that equals four.

A system that implements HPI self-stabilizes when all the operations that are triggered by a new compute node finish. The number of these operations depends on the system size. As described by Algorithm 3.1, new compute nodes start from the root, and travel downwards the hierarchy until a group with nodes in proximity is found, or a new group is created. Thus, the larger the number of layers in the hierarchy, the larger the number of groups that may need to be examined before the algorithm finishes. This means that a hierarchical structure that is long in depth may require more operations. Notably, what determines the number of layers in the hierarchy is the proximity of the nodes. If many nodes from the same location join, it is likely that every new node will follow the path of the previous ones (while traveling downwards the hierarchy), and force the hierarchical structure to grow in depth. However, assuming that the nodes are geographically scattered, the hierarchy will grow first in breadth and then in depth because the new nodes will follow different paths. This limits the operations required for the algorithm to finish, and the system stabilizes. Further evidence regarding the stabilization of a system that follows the approach of HPI is provided in Section 3.4.3.

Another aspect of HPI that should be discussed is the distribution of the computational
3.3. Algorithms for Self-Organization

resources within groups. This is an important aspect because even though the data can travel up/down the hierarchical structure (as previously discussed), it is the compute nodes within the same group that provide additional computational resources with the lowest communication latency. Therefore, if a group has a high amount of computational resources (considering the sum of the resources of all the compute nodes within the group), it might be the case that this amount is enough to meet the requirements of the applications. Consequently, the data does not need to travel to other parts of the hierarchy, thereby reducing the communication latency by sending data only to nearby nodes. When organizing the compute nodes according to HPI, the groups tend to have nodes that reside close to each other. Thus, in systems that include multiple nodes in close proximity (e.g., at the edge of the network) with rather limited computational resources, HPI may lead to having some groups that have lower computational resources than others (i.e., groups which include only edge nodes). On the contrary, organizing the nodes in groups, e.g., randomly, would actually be more likely to distribute the computational resources evenly among the groups. Nevertheless, in fog computing systems the communication latency is likely to contribute more to increased overall application execution delay than the lack of computational resources (as shown in Section 3.4.3). Thus, even though the approach of HPI might not distribute the computational resources evenly, it can reduce the overall application execution delay significantly more than a random approach, by reducing the communication latency within the groups. This is also shown in the evaluation in Section 3.4.3.

3.3.5 Flat Structures

While hierarchical structures represent the common practice, flat structures have also been proposed for fog computing (as discussed in Section 3.2). In this section, we design a distributed algorithm that enables the nodes of a fog computing system to self-organize into a flat structure based on proximity. This is especially useful for fog computing systems that span over large geographical areas since in such cases, the assumption that the nodes reside close to each other may not hold (as discussed in Section 1.2). As shown in Fig. 3.4b, a flat structure is different from a hierarchical structure in that: i) There is no root. Thereby, the structure can expand towards any direction, contrarily to the hierarchical structure which expands downwards. ii) In contrast to hierarchical structures whereby nodes cannot belong to more than two groups (as discussed in Section 3.3.4), in flat structures the number of groups that each node can belong to, may not be limited [115]. As a result of these differences, the data does not travel exclusively towards the cloud. Instead, the data can travel through the structure in any direction. By doing this though, a cloud node can also be found. For this reason, the flat structure may be more appropriate when there are multiple geographically distributed cloud compute nodes in a fog computing system, which are preferred to be arranged among the other nodes, rather than on top of them (i.e., in a hierarchy).

Notably, Fig. 3.4b shows nine compute nodes that have joined the system sequentially. The arrangement of these nodes though seems to be abstract again (similar to Fig. 3.4a)
3. Self-Organization

Algorithm 3.3: FPI (Flat Proximity-Integrated)

1 Procedure: joinSystemFPI(Address \( n_{cp} \))
2 GroupList = joinRequest(\( n_{cp} \))
3 for \( i = 1 \) to GroupList.size() do
4 \( n_{pro} = \) findNPro(GroupList)
5 \( G_{n_{cp}} = \) findGroup(GroupList, \( n_{pro} \))
6 addNeighbors(\( G_{n_{cp}} \))
7 notifyNeighbors(\( G_{n_{cp}} \))
8 if \( G_{n_{cp}}.size > S \) then
9 \( n_{far} = \) findNFar(\( G_{n_{cp}} \))
10 if \( n_{far} == n_{new} \) then
11 GroupList.remove(\( G_{n_{cp}} \))
12 else
13 \( G_{n_{cp}}.remove(n_{far}) \)
14 joinFlag = true
15 break
16 end
17 else
18 joinFlag = true
19 break
20 end
21 if joinFlag != true then
22 \( G_{new} = \) new Group(\( n_{new}, n_{cp} \))
23 notifyNeighbors(\( G_{new} \))
24 end

because when a new compute node joins, some of the nodes self-organize based on proximity, and the arrangement of the structure changes. Since the flat structure has no root, a new compute node can request to join from any existing node of the system as long as its IP address is known. Furthermore, since in flat structures the number of groups that each node belongs to is not limited, this number is decided dynamically according to the utilized algorithm.

To organize the compute nodes of a fog computing system into a flat structure, we design the Flat Proximity-Integrated (FPI) algorithm which is described in Algorithm 3.3 from the perspective of a new compute node which requests to join the system. In this algorithm, each new node \( n_{new} \) joins through \( n_{cp} \) (Line 1) which can be any existing cloud node of the system (as discussed in Section 3.3.2).

Upon request, \( n_{cp} \) responds with a list of the groups that it belongs to (Line 2), which includes all the neighbors of \( n_{cp} \). Then, \( n_{new} \) measures the proximity to each one of these neighbors in order to find the node with the closest proximity \( n_{pro} \) (Line 4). Afterwards,
3.3. Algorithms for Self-Organization

$n_{new}$ finds the group that includes $n_{pro}$ (Line 5), joins this group (Line 6), and notifies the neighbors about the arrival of a new compute node (Line 7), i.e., the neighbors also add $n_{new}$. This triggers all the nodes of this group to count their neighbors. If the number of neighbors is smaller or equal to $S$ (Line 17), $n_{new}$ stays in this group and no further actions are required (i.e., a flag is raised in Line 18, and the process stops in Line 19).

If the number of neighbors is greater than $S$ (Line 8), then the nodes calculate $n_{far}$ (Line 9), and remove it from the group. Similar to Algorithm 3.1, $n_{cp}$ is excluded from the calculation of $n_{far}$. If $n_{far}$ equals $n_{new}$ (Line 10), $n_{new}$ removes this group from the list acquired by $n_{cp}$ (Line 11), and examines the rest of the groups in the list by following the same process again (Line 3). If $n_{new}$ equals $n_{far}$ in all the groups of $n_{cp}$, $n_{new}$ creates a new group that includes $n_{new}$ and $n_{cp}$ (Line 22). However, if $n_{new}$ is not the most distant node in one of the groups of $n_{cp}$ (Line 12), then $n_{new}$ remains in that group and removes $n_{far}$ (Line 13) since the group is at capacity. Then, $n_{far}$ requests to join again by following the same process (Line 1), and using $n_{cp}$ as contact.

From the perspective of the existing nodes, the process is the same as with HPI, so Algorithm 3.2 applies to FPI as well (after replacing HPI with FPI in Line 7). However, the contact node used by $n_{far}$ in FPI is different than in HPI. Specifically, when a new node joins a group, all the nodes of this group receive the update (Line 1). Then, each node examines if the size $S$ is exceeded (Line 2). If it is, all nodes find $n_{far}$ (Line 3), and remove it from the group (Line 4). The node that is $n_{far}$ (Line 5) deletes the group (Line 6), and joins the system again through the contact that was used by the new node which triggered $n_{far}$ to leave (Line 7).

Fig. 3.7 shows the process of adding new compute nodes according to FPI graphically. In a system of five compute nodes, i.e., $n_1, n_2, \ldots, n_5$, a new compute node $n_6$ requests to join using $n_5$ as contact, as shown in Fig. 3.7a. Then, $n_6$ joins the group of $n_5$ that contains the node in closest proximity $n_{pro}$, and the neighbors in this group (i.e., $n_1, n_2, n_3, n_5$) add $n_6$ as well. The addition of $n_6$ triggers all the neighbors to count the group size. In this case, the size is five (i.e., larger than the maximum group size $S$ that equals four). This means that all nodes calculate $n_{far}$ which is $n_1$. Therefore, the new compute node $n_6$ stays in this group, while $n_1$ leaves and requests to join again using $n_5$ as contact, as shown in Fig. 3.7b. $n_1 == n_{far}$ in the group of $n_5$ that contains $n_6$, which means that $n_1$ will join the other group of $n_5$, i.e., the group that contains $n_5$, and $n_4$, as shown in Fig. 3.7c. Since this group is not at capacity, i.e., the group size equals three, no further actions are required and the FPI algorithm finishes.

This algorithm raises the question of why $n_{far}$ joins through $n_{cp}$ specifically, when rejoining the system. In general, the reason for having $n_{far}$ replaced by a node in closer proximity (i.e., $n_{new}$), is based on the goal to select neighbors for each node so that the communication among them takes place with low latency and high bandwidth. For the same reason, we examine the option of having $n_{far}$ (after leaving) join through its closest neighbor ($n_{pro}$). This seems reasonable because this way, both $n_{new}$ and $n_{far}$ are placed in groups with nodes in proximity. We also examine the case that $n_{pro}$ is used as contact.
3. Self-Organization

for $n_{\text{new}}$ upon the join request, and until $n_{\text{cp}}$ equals $n_{\text{pro}}$, in order to place each new node in the group of the closest neighbor.

However, after experimenting with large-scale systems (i.e., 1,000–10,000 nodes), we notice that both of these options can trigger changes that spread throughout the system. This happens because when $n_{\text{far}}$ leaves and joins another group at capacity, one of the preexisting nodes of that group leaves and joins again. This behavior continues until $n_{\text{far}}$ joins a group which is not at capacity. However, in large-scale systems, the number of groups to be examined can be prohibitively large to allow this process to finish, resulting in additional overhead that may compromise the scalability of the system by creating bottlenecks. Therefore, to limit this overhead, $n_{\text{far}}$ requests to join through $n_{\text{cp}}$ which means that the resulting changes of a new node joining affect $n_{\text{cp}}$ and its neighbors, while the rest of the system remains stable.

A system that follows FPI self-stabilizes when the new node has joined a group of the contact node (either preexisting or new), and no more operations are pending. Since in FPI only the groups of the contact node are examined by a new node, the number of operations required for stabilization depends on the number of groups of the contact. This means that if the contact node belongs to many groups, many operations need to be completed before the system stabilizes, while a small number of groups means fewer operations. However, when the contact nodes are selected randomly, i.e., there is no global point of entry for new nodes (which is the case in FPI), the number of groups that each node belongs to, remains limited [115]. This means that the operations required for the algorithm to finish remain limited as well, and the system stabilizes. Additional evidence about the stabilization of a system that implements FPI is provided in Section 3.4.3.

Regarding the distribution of the computational resources within groups, FPI behaves similarly to HPI. This means that when organizing the compute nodes according to FPI, the groups tend to have nodes that reside close to each other. Therefore, in systems with multiple compute nodes in close proximity (e.g., at the edge of the network) that have rather limited computational resources, FPI may result in having some groups that have

---

Figure 3.7: Example of the process of FPI step-by-step when using a maximum group size that equals four.
lower computational resources than others. Nevertheless, the communication latency (and consequently the overall application execution delay, as shown in Section 3.4.3) is likely to be significantly reduced with FPI, due to organizing the compute nodes based on network proximity. This is also shown in the evaluation in Section 3.4.3.

3.4 Evaluation

To evaluate our algorithms, in this section we present an implementation of the proposed system, as presented in Section 3.3.3. The evaluation environment, including technical details of the implementation, and decisions regarding system parameters, is discussed in Section 3.4.1. Afterwards in Section 3.4.2, we describe the alternative methods which we use as baselines. Finally, we perform various experiments considering an image processing and a smart city use case, and we present our findings (which focus on communication latency, network bandwidth, utilization of computational resources, scalability, and stabilization) in Section 3.4.3.

3.4.1 Evaluation Environment

For this evaluation, we create fog computing systems using multiple geographically distributed compute nodes which are either virtual, i.e., using cloud services, or physical, i.e., using Raspberry Pi 4 single-board computers. Since our approach targets geographically distributed systems, we use 18 compute nodes in three regions around the world, namely eastern America, central Europe, and eastern Asia, as shown in Fig. 3.8. In each region, we assume that there are: one cloud compute node (i.e., nodes: 1, 7, and 13 in Fig. 3.8), and five compute nodes at the edge of the network (i.e., the rest of the nodes in Fig. 3.8). Hereinafter, we refer to the cloud compute nodes as high-end nodes, and to the rest of the compute nodes as low-end nodes. We use this terminology because the computational resources at the edge of the network are expected to be limited compared to the cloud. Notably, in our experiments some low-end nodes are provided by cloud services (as shown in Fig. 3.8) and have similar resources to the high-end nodes. This does not affect our experiments significantly because our results focus on network-related metrics. Nevertheless, considering the location and the resource diversity of the compute nodes of the system aims at creating realistic scenarios for the experiments. The goal of this setup is to enable the nodes to self-organize into a hierarchical or a flat structure, such that each compute node has neighbors in close proximity.

With this goal in mind, we develop a prototype that implements the proposed algorithms (presented in Section 3.3), and enables all the compute nodes of the system (shown in Fig. 3.8) to self-organize into a hierarchical or a flat structure in order to execute applications. Our prototype is developed using the Java Spring framework which produces an executable Java Archive (JAR) file that can be executed on any node with a Java Virtual Machine (JVM). We do this for portability reasons, so that any compute node (virtual or physical) can run the prototype independently of the underlying OS. Nevertheless, all the nodes in our system use the Debian 10 OS (the Raspberry Pi computers use the
3. Self-Organization

Debian 10-based Raspbian). For the communication between the compute nodes, we use Representational State Transfer (REST)-based APIs due to the extensive adoption of REST which provides interoperability with Internet services over [HTTP](106).

Even though the proposed algorithms are agnostic of the utilized proximity measure, and can theoretically work with either network latency or number of hops, as indicators of network proximity, in this evaluation we use the latter. The reason is that the number of hops between distributed compute nodes can be considered static, which makes it a steady indicator of network proximity [155, 182]. On the contrary, proximity measures that rely on latency may prove unreliable due to fluctuations that can follow random patterns [228]. In fact, it has been observed that in environments with multiple geographically distributed nodes, different latency measurements indicate different nodes as the closest [235]. It should be noted that hops might also cause irregularities, i.e., it is possible that nodes that reside nearby in terms of network hops, have high latency due to bandwidth bottlenecks. However, while nodes that reside far from each other (in terms of network proximity) are likely to have to deal with bandwidth bottlenecks [190], the communication between nodes that reside nearby is not as prone to bandwidth bottlenecks. Thus, connecting nodes according to network proximity can actually increase the available bandwidth [240]. This means that even though bandwidth bottlenecks between nodes in close network proximity might occur, they are rather unlikely. Therefore, we assume that the communication latency can be reduced by decreasing the number of hops between nodes, and increasing the network bandwidth. This has also been confirmed in the literature [116]. Further evidence to support this decision (i.e., using hops as an indicator of network proximity when creating fog computing systems), is provided by our results (in Section 3.4.3) which

![Figure 3.8: Location of the compute nodes used in the evaluation (the cloud icon next to a location signifies a Google cloud node, and the Raspberry Pi icon signifies a Raspberry Pi node).](image-url)
show reduced communication latency and increased network bandwidth when measuring the network proximity between compute nodes using network hops.

Thus, to measure the proximity between two compute nodes, we count the number of hops in the network path that connects these two nodes. In order to find this number, the prototype makes system-calls to Traceroute [10] (a command-line tool for network diagnostics integrated into all major OSs such as Windows, Linux, and MacOS). If a node belongs to an organization which uses a firewall that blocks the traffic from Traceroute, we consider the number of hops until this firewall as a representative indicator of network proximity for this node. In our experiments, this happens with the nodes that are provided by the Raspberry Pi boards, and results in a hop count that ignores the last hop (i.e., the hop count is shortened by one). Thus, our prototype is able to count the network hops between any two nodes. To find the number of hops, Traceroute sends Internet Control Message Protocol (ICMP) messages that are processed by all the routers on path [75]. Notably, only routers (or devices that operate on the network layer of Open Systems Interconnection (OSI)) are counted. This means that switches, hubs, and other network devices (that operate on the data link layer of OSI) do not affect the number of hops. This is very positive in our case, because even though the communication to the first router (within the LAN) may be very fast, it might involve a large number of network devices (e.g., wireless/wired bridges and hubs) that could rapidly increase the hop count without increasing the network latency. This is avoided by not considering such devices when measuring the number of hops.

Since in Section 3.3 we introduce the proposed structures with a group size that equals four (as shown in Fig. 3.4), initially, we retain the same group size for this evaluation in order to make the organization of the system more comprehensible. However, when scrutinizing the experiments in the next sections, we also discuss how the group size affects the results, and in general, how it affects the behavior of the system. The initial contact point of the system is always a cloud compute node (as discussed in Section 3.3.2), and is selected randomly (using the uniform distribution) among the cloud nodes. In the hierarchical structure, the same compute node acts as contact for all the other nodes to join, as discussed in Section 3.3.4. In the flat structure, the contact node may change, but is again selected randomly using the uniform distribution among the cloud nodes that have already joined, as discussed in Section 3.3.5.

### 3.4.2 Evaluation Baselines

As discussed in Section 3.2, most approaches for building fog computing systems rely on the proximity of neighbor nodes without taking actual network proximity measurements. To represent such hierarchical fog computing structures (such as [13, 77, 118] which are discussed in Section 3.2.1), we implement an algorithm whereby the nodes self-organize into a hierarchical structure without integrated proximity, which we refer to as Hierarchical Proximity-Agnostic (HPA). In HPA, each new node becomes a child in an existing group which is not at capacity. If all the groups are at capacity, a leaf node becomes a parent for the new node. This results in a hierarchical structure which
3. **Self-Organization**

grows first in breadth and then in depth. The arrangement of the compute nodes in this structure depends on the order that the nodes join.

To represent flat fog computing structures that do not take network proximity measurements (such as [47, 184, 241] which are discussed in Section 3.2.2), we implement an algorithm which we refer to as Flat Proximity-Agnostic (FPA). In FPA, the nodes self-organize into a flat structure without integrated proximity. Similar to HPA, the arrangement of the nodes in FPA depends on the order that the nodes join, but also on the nodes which are used as contacts (which are selected randomly using the uniform distribution). The goal of both of these algorithms (i.e., HPA and FPA) is to serve as baselines in order to examine the potential benefits of applying self-organization with integrated proximity to organize geographically distributed compute nodes in fog computing.

Furthermore, we implement an algorithm which adds all the nodes to the same group. This means that every node has all the other nodes as neighbors. In contrast to the other approaches whereby each compute node can communicate directly only with a limited number of neighbors, in this algorithm each compute node can communicate directly with all the other nodes of the system. Thus, since in this approach all the nodes are directly connected to each other, this algorithm is expected to have the lowest communication latency. For this reason, we refer to this algorithm as the **Optimal latency algorithm (OPT)**. Similar to HPI and FPI, OPT is also proximity-integrated. Notably, the approach of OPT may not be realistic in fog computing because it assumes that every node maintains a global view of the system, which may require a large amount of computational resources (especially in large-scale systems). However, this might not be possible since the compute nodes at the edge of the network have limited computational and storage resources [187, 218]. Nevertheless, we use OPT to represent the lowest possible communication latency.

### 3.4.3 Evaluation Results

In this section, we perform various experiments which aim at showing how each algorithm performs regarding communication latency, network bandwidth, utilization of computational resources, execution delay, scalability, and stabilization. Since the order that each node joins the system may affect the arrangement of the structure, for each algorithm we perform 50 experiments with nodes that join randomly using the uniform distribution. This means that for each of the examined algorithms (i.e., HPA, HPI, FPA, FPI, and OPT), the 18 compute nodes (shown in Fig. 3.8) join the system sequentially (in a random order) 50 times, and form fog computing structures according to the corresponding algorithm. For each experiment, we take various measurements which are presented below. Even though we have experimented extensively with this setup, we present the results of 50 experiments (for each algorithm) because we found this number to be large enough to represent the general case.
3.4. Evaluation

Communication Latency

To evaluate the communication latency among the compute nodes for each algorithm, we assume an image processing use case. Such use cases are common for the evaluation of fog computing systems \[53, 67\]. Specifically, in our experiments we consider a user device at the edge of the network, e.g., a smart doorbell, that takes a picture using an application (a similar use case has been presented in Section 3.3.1). This application includes various latency-sensitive and latency-tolerant processes which can be related to pattern recognition, analytics, storage, etc. As discussed in Section 3.3.1, the processes for which low latency may increase the QoS for the users, are considered latency-sensitive (e.g., pattern recognition). On the other hand, the processes for which low latency may not affect the QoS significantly (e.g., analytics and storage), are considered latency-tolerant.

Instead of executing the application on the user device (which may have limited computational resources or power supply), the application’s processes are distributed on the compute nodes of the fog computing system (i.e., the computations of the application are offloaded to the compute nodes). The latency-sensitive processes are hosted on the compute nodes in proximity, while the latency-tolerant processes are hosted on compute nodes which reside farther away.

Each low-end node receives an image file (from a user device) that needs to be processed by processes that are hosted on the compute nodes. Thus, this file is sent through the structure to all the other compute nodes of the system. Presumably, the nodes that receive the file first are the neighbors which are provided by our algorithms. To take latency measurements, we measure the delay from the time that the initial compute node sends the file until the time each of the other nodes receives the file. This delay includes communication latency and parsing of the HTTP headers but excludes further processing from the processes.

To examine the incurred delay for latency-sensitive processes, for each low-end node we measure the time period until the image reaches the nodes in proximity. As nodes in proximity, we consider the three nodes which receive the file first. The reason that we consider three nodes specifically, is that the group size equals four (as discussed in Section 3.4.1), i.e., one sender and three receivers. Notably, the group size does not apply to OPT because it creates only one group that includes all the compute nodes. Nevertheless, we still consider as compute nodes in proximity the three closest nodes in order to make the results comparable among all the algorithms.

The latency measurements of the latency-sensitive processes which are hosted on compute nodes in proximity are shown in Fig. 3.9, for each one of the algorithms. Each box plot considers all 50 values (from the 50 experiments). However, the outliers are omitted because due to their distance from the interquartile range, they change the scale of the figure, and impair the visual representation of the results. Notably, the outliers have higher values than the shown maximum, but do not actually change the interpretation of the shown results. The values of Fig. 3.9 are the outcome of using an image file of 4.8 Megabytes (MB) which is a reasonable size of an image taken for pattern recognition.
3. Self-Organization

As shown in Fig. 3.9, HPA has an average delay of 1,435 milliseconds (ms). By applying self-organization with integrated proximity in the hierarchical structure, as implemented in HPI, the average delay drops to 811 ms which indicates a reduction of about 43%. By using a flat structure instead of the hierarchy, FPA has an average delay of 1,193 ms which means about 17% less than HPA. When applying self-organization with integrating proximity in the flat structure, FPI has an average delay of 876 ms. This means about 39% less than HPA, about 27% less than FPA, and about 7% more than HPI. Furthermore, by examining the average values for different file sizes (which are shown in Table 3.3), we note that HPI and FPI provide similar latency benefits independently of the size of the file.

Finally, OPT has an average delay of 507 ms which means about 65% less than HPA, about 37% less than HPI, about 58% less than FPA, and about 42% less than FPI. Notably, in Fig. 3.9 we notice that the average delay of OPT is above the upper quartile. This happens because there are many outliers which are numerically distant from the

Table 3.3: Communication latency (in ms) to nodes in proximity for different file sizes.

<table>
<thead>
<tr>
<th></th>
<th>HPA</th>
<th>HPI</th>
<th>FPA</th>
<th>FPI</th>
<th>OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2 MB</td>
<td>Average 893</td>
<td>479</td>
<td>795</td>
<td>533</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>St. dev. 764</td>
<td>592</td>
<td>683</td>
<td>608</td>
<td>121</td>
</tr>
<tr>
<td>2.4 MB</td>
<td>Average 1,065</td>
<td>581</td>
<td>947</td>
<td>627</td>
<td>201</td>
</tr>
<tr>
<td></td>
<td>St. dev. 904</td>
<td>715</td>
<td>824</td>
<td>700</td>
<td>228</td>
</tr>
<tr>
<td>4.8 MB</td>
<td>Average 1,435</td>
<td>811</td>
<td>1,193</td>
<td>876</td>
<td>507</td>
</tr>
<tr>
<td></td>
<td>St. dev. 1,190</td>
<td>843</td>
<td>950</td>
<td>786</td>
<td>690</td>
</tr>
</tbody>
</table>
3.4. Evaluation

Figure 3.10: Communication latency to the rest of the nodes in the system.

Table 3.4: Communication latency (in ms) to the rest of the nodes in the system for different file sizes.

<table>
<thead>
<tr>
<th>File Size</th>
<th>HPA</th>
<th>HPI</th>
<th>FPA</th>
<th>FPI</th>
<th>OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2 MB</td>
<td>3,250</td>
<td>2,236</td>
<td>2,665</td>
<td>2,329</td>
<td>1,359</td>
</tr>
<tr>
<td>St. dev.</td>
<td>1,679</td>
<td>1,213</td>
<td>1,362</td>
<td>1,200</td>
<td>783</td>
</tr>
<tr>
<td>2.4 MB</td>
<td>4,021</td>
<td>2,694</td>
<td>3,348</td>
<td>2,821</td>
<td>1,839</td>
</tr>
<tr>
<td>St. dev.</td>
<td>2,299</td>
<td>1,548</td>
<td>1,896</td>
<td>1,549</td>
<td>1,254</td>
</tr>
<tr>
<td>4.8 MB</td>
<td>5,220</td>
<td>3,564</td>
<td>4,061</td>
<td>3,456</td>
<td>2,673</td>
</tr>
<tr>
<td>St. dev.</td>
<td>2,804</td>
<td>2,649</td>
<td>2,134</td>
<td>1,731</td>
<td>2,067</td>
</tr>
</tbody>
</table>

interquartile range. The reason that |OPT| exhibits this behavior is that initiating transmissions to many nodes at the same time may cause an (upload) bandwidth bottleneck which can increase the communication latency of the transmissions invoked by latency-sensitive processes. This shows that in fog computing, trying to improve performance by connecting many compute nodes to each other, may cause the opposite effect. Further experimentation shows that when increasing the size of the system, this behavior becomes increasingly noticeable. For this reason, the notion of a group (i.e., nodes that maintain a limited number of neighbors), and algorithms to create the groups—such as the algorithms proposed in this chapter—become crucial.

We also measure the delay for reaching the rest of the nodes in the system, which host latency-tolerant processes. Even though the communication latency may not affect latency-tolerant processes, low latency still improves the QoS of the users. Fig. 3.10 shows the delay to reach all the nodes of the system apart from the nodes in proximity (which are shown in Fig. 3.9). The average values and the standard deviations along with additional file sizes are shown in Table 3.4.

As shown in Fig. 3.10, |HPA| has an average delay of 5,220 ms. By applying self-organization
with integrated proximity, HPI lowers the delay to 3,564 ms which indicates a reduction of about 32%. With a flat structure, FPA has an average delay of 4,061 ms which means about 22% less delay than HPA. FPI which applies self-organization with integrated proximity in the flat structure, has an average delay of 3,456 ms which is about 34% less than HPA, about 3% less than HPI, and about 15% less than FPA. Finally, when all the nodes are connected to each other, OPT has an average delay of 2,673 ms which is about 49% less than HPA, about 25% less than HPI, about 34% less than FPA, and about 23% less than FPI.

Therefore, we note that applying self-organization with integrated proximity reduces the communication latency of latency-sensitive processes for both hierarchical and flat structures by 27–43%. Furthermore, for latency-tolerant processes, self-organization with integrated proximity reduces the communication latency by 15–32%. In addition, with this experiment we note that even though hierarchical structures represent the common practice in fog computing, FPA outperforms HPA for both latency-sensitive (about 17% reduction) and latency-tolerant processes (about 22% reduction), while FPI performs slightly better than HPI for latency-tolerant processes (about 3% reduction), but mildly worse for latency-sensitive processes (about 7% reduction). The reason that flat structures perform better is that in a flat structure, each node can have more neighbors than in the hierarchical (even with the same group size) due to the unconstrained number of groups (as discussed in Section 3.3.5). Thus, the chances of finding neighbors in close proximity increase. In HPI, every new node travels through the hierarchy in search of nodes in proximity, which is why more groups are examined than in FPI. This means that even though each node in HPI may have fewer neighbors than in FPI, the nodes that do become neighbors are more carefully selected, thereby achieving the lowest communication latency for latency-sensitive processes.

When experimenting with various group sizes and different numbers of nodes in the system, we only observe slight changes in the communication latency. Nevertheless, a pattern in these changes can be noticed. If the number of nodes in the system is smaller or equal to the group size, there are no differences among the examined algorithms since there is only one group of nodes. However, when the number of nodes increases, and the number of groups in the system becomes equal to the group size, then the proposed algorithms start showing significant differences. This happens because when the system reaches this size, enough nodes have selected neighbors according to the utilized algorithm for the system to form the target structure. For instance, by using a group size that equals four, the results we present in this section apply when the system grows to more than 16 nodes. After this, the percentages of latency reduction remain similar with slight increases when the system size grows.

Network Bandwidth

Estimating the bandwidth capacity between two nodes in fog computing scenarios can be challenging because the various devices and applications which utilize the same network affect the available network bandwidth [94]. This happens because the bandwidth capacity
Table 3.5: Network bandwidth (in MB/sec) to nodes in proximity for different file sizes.

<table>
<thead>
<tr>
<th>File Size</th>
<th>HPA</th>
<th>HPI</th>
<th>FPA</th>
<th>FPI</th>
<th>OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2 MB</td>
<td>1.344</td>
<td>2.505</td>
<td>1.509</td>
<td>2.251</td>
<td>9.524</td>
</tr>
<tr>
<td>2.4 MB</td>
<td>2.254</td>
<td>4.131</td>
<td>2.534</td>
<td>3.828</td>
<td>11.94</td>
</tr>
<tr>
<td>4.8 MB</td>
<td>3.345</td>
<td>5.919</td>
<td>4.023</td>
<td>5.479</td>
<td>9.467</td>
</tr>
</tbody>
</table>

of the links that connect the participating compute nodes may be shared among all the nodes in the network [55]. Thus, in our fog computing environment which includes geographically distributed compute nodes that communicate with each other through the Internet, measuring the bandwidth between nodes may produce unreliable results since other nodes of the Internet may be using the same network links. For this reason, instead of measuring the bandwidth capacity between compute nodes for each algorithm (i.e., HPA, HPI, FPA, FPI, and OPT), we examine the changes in the available network bandwidth based on the required time to transfer files through the structure.

To examine the changes in the available network bandwidth for latency-sensitive processes, we analyze the amount of time required to transfer files of different sizes to the compute nodes in proximity. In Table 3.3, we notice that, e.g., HPA requires an average of 893 ms to send a 1.2 MB file to the nodes in proximity. This indicates that the involved network links are able to transfer 1.2 MB in 893 ms, or 1.34 MB per 1 second. Similarly, we can estimate the size of the data per second (MB/sec) that can be transferred to compute nodes in proximity for all the algorithms. Table 3.5 shows these measurements.

By examining Table 3.5, we note that HPI which applies self-organization with integrated proximity, increases the available network bandwidth compared to the simple hierarchical approach (i.e., HPA), by about 76-86% for all evaluated file sizes. Similarly, FPI increases the available network bandwidth compared to FPA, by about 36-51% for all evaluated file sizes. OPT has the highest bandwidth due to utilizing only direct communication links between nodes.

The reason that HPI and FPI increase the available network bandwidth of latency-sensitive processes (by about 36-86%) is because the bandwidth capacity of the links that connect nodes in close proximity tends to be higher than when the nodes reside farther apart. This is a crucial observation of fog computing that current research often claims without actual empirical results, and without quantifying the extent of the potential bandwidth gains [197, 204].

Similar to Table 3.5 which is based on the average values of Table 3.3, we also create Table 3.6 which is based on Table 3.4, and shows bandwidth measurements for latency-tolerant processes. According to Table 3.6, HPI increases the available network bandwidth by about 46-49% compared to HPA, while FPI increases the available network bandwidth by about 14-19% compared to FPA. Again, OPT which uses only direct communication links between nodes, has the highest network bandwidth.
3. Self-Organization

Table 3.6: Network bandwidth (in MB/sec) to the rest of the nodes in the system for different file sizes.

<table>
<thead>
<tr>
<th>File Size</th>
<th>HPA</th>
<th>HPI</th>
<th>FPA</th>
<th>FPI</th>
<th>OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2 MB</td>
<td>0.369</td>
<td>0.537</td>
<td>0.45</td>
<td>0.515</td>
<td>0.883</td>
</tr>
<tr>
<td>2.4 MB</td>
<td>0.597</td>
<td>0.891</td>
<td>0.717</td>
<td>0.851</td>
<td>1.305</td>
</tr>
<tr>
<td>4.8 MB</td>
<td>0.92</td>
<td>1.347</td>
<td>1.182</td>
<td>1.389</td>
<td>1.796</td>
</tr>
</tbody>
</table>

Therefore, by further analyzing the time required to send files of various sizes to the compute nodes of the system, we show that the proposed algorithms enable the compute nodes to communicate with higher network bandwidth. This means that using self-organization with integrated proximity (i.e., HPI and FPI) increases the available network bandwidth for both hierarchical and flat structures. Specifically, we note that self-organization increases the available network bandwidth of latency-sensitive processes by about 36-86%, and for latency-tolerant processes by about 14-49%.

Utilization of Computational Resources

To ensure that the proposed algorithms do not interfere with the execution of the applications, we examine the utilization of Central Processing Unit (CPU) and Random Access Memory (RAM). We do this on the nodes which run the proposed algorithms on single-board computers (Raspberry Pis), because these are the nodes with the least amount of computational resources.

Using Top [9] (a command-line tool for monitoring the utilized computational resources of the running processes), we monitor the resource utilization of the processes that handle the self-organization according to each of the proposed algorithms. CPU utilization remains at 1% with occasional spikes which do not exceed 10%, and RAM utilization remains steady at around 9%. Since the proposed algorithms are executed in a distributed manner, no node has to perform all the organizational operations, and resource utilization remains low. Hence, these results support our decision to use low-end nodes for creating fog computing systems as presented in Section 3.3.3.

Execution Delay

Since the benefits of the proposed algorithms stem from reducing the communication latency, in this section we examine the execution delay of the processes in order to put the communication latency benefits into perspective. Notably, if the execution delay is very high compared to the communication latency, it means that reducing the communication latency results in a small reduction of the overall latency of executing processes in the system (since the overall latency includes both communication latency and execution delay). However, if the execution delay is small compared to the communication latency, it means that the overall latency depends more on the communication latency. Hence,
3.4. Evaluation

Table 3.7: Execution delay (in ms) of image processing applications for different file sizes.

<table>
<thead>
<tr>
<th></th>
<th>Face detection</th>
<th>Body detection</th>
<th>Smile detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2 MB</td>
<td>Average 296</td>
<td>231</td>
<td>315</td>
</tr>
<tr>
<td></td>
<td>St. dev. 27</td>
<td>23</td>
<td>21</td>
</tr>
<tr>
<td>2.4 MB</td>
<td>Average 518</td>
<td>423</td>
<td>622</td>
</tr>
<tr>
<td></td>
<td>St. dev. 13</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>4.8 MB</td>
<td>Average 960</td>
<td>793</td>
<td>1,275</td>
</tr>
<tr>
<td></td>
<td>St. dev. 26</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

a significant reduction in the communication latency (such as 27–43% as presented in Section 3.4.3) can cause a noticeable reduction in the overall latency.

The execution delay represents the time required by a compute node to execute a process. Thus, this delay depends on the resource capacities (e.g., CPU and RAM) of the utilized compute node, and on the computations that are performed by the process. The processes that we execute for this experiment are related to image processing which is the examined use case of this evaluation. Specifically, we execute a face detection process (e.g., for face tracking), a body detection process (e.g., for counting people), and a smile detection process (e.g., for emotion recognition). All of these processes are implemented to use machine learning-based Haar cascade classifiers which are provided pre-trained by the OpenCV library [227].

Since the execution delay depends on the computational resources of the utilized nodes, we measure this delay again on the compute nodes with the least amount of computational resources (i.e., the Raspberry Pis). This way, the measurements represent the maximum execution delay which can be reduced if the processes are executed on nodes with more computational resources, e.g., in the cloud. To acquire results which are representative of the general case, we execute each process 50 times. The results include the average values and standard deviations of executing each process for different file sizes, and are shown in Table 3.7.

Notably, each process has a different execution delay because a different cascade classifier is used, i.e., the computations are different for each process. Specifically, we note that for an image file of 1.2 MB the face detection process requires, on average, 296 ms the body detection process needs, on average, 231 ms and the smile detection process takes, on average, 315 ms. We also note that the execution delay grows when the file size increases. This happens because larger files require more time to be processed, which increases the execution delay. Nevertheless, the previous observation that the execution delay of detecting bodies is, on average, less than detecting faces which is less than detecting smiles, always applies.

By comparing the execution delay of the processes (shown in Table 3.7) with the
communication latency required to reach compute nodes in proximity (shown in Table 3.3), and remote compute nodes (shown in Table 3.4), excluding the approach of OPT (which may not produce realistic results, as discussed in Section 3.4.2), we note the following:

The communication latency of remote compute nodes is always much higher than the execution delay. For an image file of 1.2 MB, for example, the communication latency of HPI (2,236 ms) is about 6.5 times larger than the execution delay of the face detection process (296 ms). The communication delay of compute nodes in proximity, on the other hand, is not as high. Nevertheless, for an image file of 1.2 MB the execution delay of all the processes is still lower than the communication latency of any approach (i.e., HPA, HPI, FPA, FPI). For an image file of 2.4 MB the execution delay is also lower, with the exception of the smile detection process which has an execution delay (622 ms) that is about 7% larger than the communication delay of HPI (581 ms). Finally, for an image file of 4.8 MB the execution delay is lower or comparable to the communication latency, with the exception of the smile detection process which has an execution delay (1,275 ms) that is about 57% higher than the communication latency of HPI (811 ms), 6% higher than FPA (1,193 ms), and 45% higher than FPI (876 ms).

Therefore, by considering these results, and by taking into account that these execution delays represent maximum values, we note that in a system with geographically distributed compute nodes, the communication latency is much higher than the execution delay. However, if the computations of the executed processes are very intensive, or if the input data is very large, then the execution delay may become comparable to the communication latency. Notably, in many IoT scenarios the input data is rather small, e.g., sensor values [60]. This can result in even lower execution delay. Hence, we conclude that the communication latency plays a critical role in the overall latency of executing applications in the system. This further advocates that algorithms which enable the compute nodes to self-organize and communicate with low communication latency, are crucial for achieving low latency in fog computing systems.

**Scalability**

While performing the experiments regarding communication latency (in Section 3.4.3) and network bandwidth (in Section 3.4.3), we noticed that the incurred network traffic of the nodes that use OPT is slightly higher than with the other algorithms. However, despite the extensive experimentation, with a system of 18 nodes we were not able to produce results which show a pattern that explains this observation. Even though slight changes in the network traffic seem trivial in small-scale systems, in larger systems, such changes may scale and eventually lead to scalability concerns due to network bottlenecks. For this reason, in this section we examine the scalability of the proposed algorithms considering the incurred network overhead.

To evaluate scalability, we examine the network overhead in a system with growing size, i.e., a system in which new nodes are added consecutively, and follow the proposed algorithms in order to self-organize into the desired system structure. Systems with a growing size may find real-world use within the context of smart cities. Smart cities may
3.4. Evaluation

Integrate an increasing number of nodes due to new nodes being added for facilitating new applications, but also, due to the city itself expanding geographically [189]. Furthermore, we consider this a good example of a growing system because smart cities have been proposed as a target use case for fog computing [161].

In order to take scalability measurements that can be extrapolated to large-scale systems, first we use software to extend the fog computing structures of the proposed algorithms with a large number of compute nodes, and we measure the organizational network overhead. Then, we use regression (which is a predictive method) in order to model this overhead, and to create a function which outputs the overhead of our algorithms based on the system size. By examining the slope of this function, we are able to draw conclusions regarding the sensitivity of the network overhead when the system size changes. Therefore, we still use the same code of the prototype, but each node assumes that it is part of a larger system. To achieve this, we benchmark the code of the prototype into a simulation which creates fog computing systems similar to the ones of the previous experiments. However, unlike the previous experiments, we are now able to examine the behavior of our algorithms in a setting with a configurable number of participating compute nodes. The simulator we use for this purpose is based on a simulator for hierarchical and flat fog computing systems from the literature [112].

For this experiment, we create fog computing systems with 1,000 nodes which self-organize according to the proposed algorithms. We consider 1,000 nodes to be a satisfactory system size for the regression analysis due to the high coefficients of determination in the regression results. Specifically, with the overhead data from 1,000 nodes, both HPI and OPT have coefficients of determination with values higher than 0.9. The coefficient of determination is a widely-used statistical measure, also known as $R^2$, which shows how close the regression line is to the data. This statistical measure has a value between 0 and 1, and shows how well the model fits the data, i.e., values close to 1 indicate that the model explains the variability of the data. Usually, a coefficient of determination with a value higher than 0.6 is considered adequate [85, 154].

To gather the overhead data, the system starts with one compute node, and new nodes join dynamically according to each one of the examined algorithms, until the system reaches 1,000 nodes. For each new node that joins the system, we count the number of control messages which need to be exchanged in order to find neighbors. Thus, this experiment allows us to measure the overhead of the required control operations without considering the application traffic. When counting control messages, we consider requests to join the system, to notify neighbors, to measure proximity, etc. (as described by Algorithms 3.1 and 3.3 in Section 3.3). Since Traceroute sends messages incrementally to count the hops between two nodes, the number of required messages to emulate Traceroute equals to the number of hops between these nodes.

For the proximity-agnostic algorithms (i.e., HPA and FPA), the number of control messages remains negligible even with 1,000 nodes. This happens because these algorithms are designed to serve as baselines, and only minimal effort is put into the self-organization
of the nodes, which results in low overhead. However, in [OPT, HPI, and FPI] this overhead increases due to the multiple control messages which are required for the self-organization. Fig. 3.11 shows the number of required control messages for each new node that joins the system according to [OPT, HPI, and FPI] Since these results exhibit a similar pattern which resembles a straight line, we perform a linear regression analysis (using the least squares method) with two goals in mind. First, we want to examine if there is a relationship between the number of nodes (as the independent variable) and the overhead from control messages (as the dependent variable). If such a relationship exists, which seems to be the case for [OPT], there might be scalability concerns since the system may need to scale to a point that the overhead creates network bottlenecks, and interferes with the application traffic. Second, regression can be used as a prediction method for estimating future values, i.e., with even more than 1,000 nodes, in order to examine how steeply the overhead changes when the system scales.

In Table 3.8, we show the regression analysis results which are based on the overhead from [OPT, HPI, and FPI] in Fig. 3.11. [OPT] and [HPI] have high coefficients of determination meaning that the regression line explains the data well. However, [FPI] has a coefficient of determination which is close to zero. This is a surprising result because by visualizing the data (in Fig. 3.11), it is evident that all the points of [FPI] follow the pattern of a straight line. To interpret the coefficient of determination of [FPI], we examine the definition: \( R^2 = 1 - \frac{SS_{res}}{SS_{tot}} \) with \( SS_{res} \) being the residual sum of squares (i.e., difference between actual data and predicted data), and \( SS_{tot} \) being the total sum of squares of the differences between the dependent variable and its mean. In linear models, \( SS_{tot} \) can also equal the residual sum of squares plus the explained sum of squares. The explained sum of squares measures the variation of the data.

If \( SS_{res} \) equals zero, it means that the data falls perfectly on the regression line (i.e., predicted values), and \( R^2 = 1 \) (similar to [OPT] and [HPI]). When \( SS_{res} \) grows, it means that the data starts deviating from the regression line, and \( R^2 \) decreases. However, when the data has very low variance, the explained sum of squares decreases. This means that...
3.4. Evaluation

Table 3.8: Regression analysis of the data in Fig. 3.11

<table>
<thead>
<tr>
<th></th>
<th>OPT</th>
<th>HPI</th>
<th>FPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R^2$</td>
<td>0.956</td>
<td>0.931</td>
<td>0.029</td>
</tr>
<tr>
<td>Standard Error</td>
<td>440.526</td>
<td>66.682</td>
<td>5.046</td>
</tr>
<tr>
<td>Variance</td>
<td>4,473,771</td>
<td>64,484</td>
<td>26</td>
</tr>
<tr>
<td>Intercept</td>
<td>-26.861</td>
<td>263.127</td>
<td>45.687</td>
</tr>
<tr>
<td>Slope</td>
<td>7.166</td>
<td>0.848</td>
<td>0.003</td>
</tr>
</tbody>
</table>

$SS_{tot}$ becomes similar to $SS_{res}$ and $R^2$ approaches zero. This is why it is possible to have a prediction model with high accuracy even with low $R^2$ as long as the variance of the data is very low. To verify that this is the case for FPI, we examine the variance of the data, but also the standard error which shows how far is the data from the regression line. In Table 3.8 we note that FPI has very low variance and a low standard error, which indicate the accuracy of the regression line for estimating the overhead of the current and future values. OPT and HPI have higher standard errors due to the variance of the data.

Thus, the combination of two statistical measures ($R^2$ and standard error) indicates that the overhead of each algorithm can be predicted according to the regression lines. Each regression line can be created based on the values of Table 3.8 and considering that for each new node that joins the system (independent variable $X$), the resulting overhead (dependent variable $Y$) will be $Y = \text{Slope} \cdot X + \text{Intercept}$. The slope, which shows the rate of increase, indicates that in OPT with a slope that equals 7.1, the overhead grows radically which may lead to scalability concerns very fast. In HPI, the slope equals 0.8 which is not as high as OPT, but still affects the overhead noticeably. Finally in FPI, the slope is 0.003 meaning that the overhead remains almost stable while the system size grows. This advocates scalability.

We also perform a regression analysis for different group sizes. By doing this, we note that the statistical measures remain very similar independently of the group size. Specifically, OPT and HPI maintain high $R^2$, variances, and standard errors while FPI maintains low $R^2$, variance, and standard error. This means that the regression lines can be used to explain both actual and predicted data. The slopes of the regression lines are also similar, with OPT and HPI having growing overhead, while in FPI the overhead remains almost stable. However, we also note that the intercept increases along with the group size. This happens because in larger group sizes, the proximity to more nodes needs to be measured, and more nodes need to be notified when a new node joins. The messages required for these operations increase the overhead. Therefore, when the group size grows, the overhead grows, and the intercept grows as well. Essentially, this means that for larger group sizes, the values of the y-axis in Fig. 3.11 increase, but the lines of the figure remain very similar.

Hence, in all the algorithms (i.e., OPT, HPI, and FPI) a larger group size increases the
resulting overhead. In addition, in Opt and HPI, there is a dependency between the number of nodes and the overhead, which indicates that while the system size grows, the overhead grows as well. However, this dependency is exceedingly weak in FPI. This means that while the system size grows, the overhead remains almost stable.

The reason that FPI exhibits this behavior is that there is no central entity in the system, and when new nodes join, only the neighbors are affected while the rest of the system remains stable independently of the number of nodes. On the contrary, when new nodes join in Opt and HPI, this might trigger changes that spread all over the system, which is why the overhead grows. This is a significant finding because it shows that when a system uses a centralized approach, even in a hierarchical manner which is the common case in state-of-the-art approaches for organizing fog computing systems, it is possible that the overhead accumulates. This might lead to scalability concerns. As suggested by our results, the increase in the overhead can be reduced significantly by using a flat structure.

Stabilization

As discussed in Section 3.4.3, for Fig. 3.11 we add compute nodes sequentially, and for each one, we count the control messages that need to be exchanged in the system until the new node has joined, and no further operations are pending. This means that Fig. 3.11 also shows the number of messages that need to be exchanged until the system stabilizes and a new compute node can be added.

When using Opt, we note that while the system size grows, the system requires an increasing number of messages (and thereby operations) to stabilize. This means that a large number of nodes may interfere with the normal operation of the algorithm, if a new node is added before the system has stabilized. When using HPI, the rate of increasing operations drops significantly, which means that the system can scale to a large degree before stabilization becomes a concern. Finally, when using FPI, the number of required operations is the lowest, and the rate of increase is close to zero (as discussed in Section 3.4.3). This means that FPI stabilizes before the other approaches, and that the stabilization is not affected by the system size.

Notably, these results align with the discussions of stabilization in Sections 3.3.4 and 3.3.5. In HPI, the system stabilization is affected by the number of layers in the hierarchy, which is why when the hierarchical structure grows, the algorithm requires more operations for the system to stabilize. In FPI, on the other hand, the stabilization does not depend on the system size, which is why the number of required operations for the system to stabilize does not increase when the system grows.

3.5 Summary

In this chapter, we design and implement distributed algorithms for self-organizing fog computing systems that span over large geographical areas. By applying the proposed
algorithms with integrated proximity, we show that fog computing systems can execute applications with lower communication latency and higher network bandwidth than when using alternative methods. Specifically, according to the experiments we perform considering actual use cases, we show that the proposed algorithms reduce the communication latency of latency-sensitive processes by 27–43%, and increase the available network bandwidth by 36–86%. For latency-tolerant processes we note a reduction of 15–32% in the communication latency, and an increase of 14–49% in the available network bandwidth. We also examine the scalability of our algorithms based on empirical results and using predictive methods, and we show that flat structures can scale better than the commonly used hierarchical structures, due to producing less overhead when the system size grows. However, hierarchical structures are shown to be particularly effective at reducing the communication latency for latency-sensitive processes. Therefore, we reach the conclusion that flat structures with integrated proximity provide solid benefits regarding communication latency, bandwidth, and scalability, while hierarchical structures provide similar or better latency and bandwidth benefits, but exhibit a lower degree of scalability. Considering all the provided results, we claim that self-organization has the potential to further advance the efficiency of fog computing systems despite the utilized system structure.
CHAPTER 4

Context-Aware Routing

In this chapter, we present the second contribution of this thesis, which is based on [108]. Contribution II is motivated by RQ II, and addresses the problem of routing IoT data to the participating compute nodes of a fog computing system. The routing approach we present in this chapter is decoupled from the way the participating compute nodes are discovered and organized. As a result, our routing approach can be applied to any set of distributed compute nodes that communicate with each other to enable the execution of applications on nearby and remote nodes, including but not exclusively, a group of nodes that has been created automatically using the algorithms of Chapter 3.

To describe the contribution of this chapter in a comprehensive manner, first we provide an overview of the proposed approach in Section 4.1. Afterwards, we present related works from the literature in Section 4.2, and a detailed description of the proposed routing approach in Section 4.3. Finally, in Section 4.4 we discuss results from the evaluation of our approach, which includes experiments conducted over a real-world setup of distributed compute nodes, and in Section 4.5 we summarize our findings.

4.1 Overview

In this section, we propose a mechanism for sending information from IoT devices to nearby and remote compute nodes based on the context of the routed data. To achieve this, our mechanism maintains a history of all nodes that have accepted data of a particular context in the past. By processing this history and using predictive methods, the proposed context-aware routing mechanism sends the data directly to the closest node that tends to accept data of the same context.

This approach can be particularly useful for fog computing systems that forward IoT data among compute nodes, until a node with sufficient computational resources accepts the IoT data for processing. For such systems, the proposed mechanism avoids the
forwarding by nodes on path (which can be problematic as discussed in Section 1.2), and reduces the communication latency of routing data over a fog computing system. We evaluate this approach using both prototype- and simulation-based experiments which show reduced communication latency by up to 23%, and lower number of hops traveled by up to 73% (compared to a baseline method).

4.2 Related Work

There are various approaches in the literature that propose mechanisms for enabling compute nodes of fog computing systems to either accept IoT data and process it, or to forward the data to other compute nodes. For instance, Tong et al. [217] propose a mechanism for forwarding peak workloads from mobile devices to compute nodes towards the cloud in order to increase the computation capacity of a system which includes various edge and cloud nodes. Chekired et al. [50] present a mechanism for processing industrial IoT data based on priorities such that high priority data is processed on compute nodes close to the edge, and low priority data is forwarded towards the cloud. Ascigil et al. [23] propose a system with various compute nodes on the path from the edge to the cloud, and design a mechanism based on deadlines, to either accept or forward a workload. Mortazavi et al. [160] design a platform (also presented in the related work of Chapter 3) which assumes that network devices between the edge and the cloud act as compute nodes, and are able to process the data on a path towards the cloud.

Notably, the aforementioned approaches assume that the data is routed on a path from the edge towards the cloud, and propose mechanisms to enable the compute nodes on path to either accept or forward this data. However, none of these approaches considers bypassing compute nodes on path to avoid the extra overhead. For this reason, this chapter presents a mechanism that takes into account the context of the IoT data for avoiding busy compute nodes on path in order to process the data with reduced communication latency. Other approaches from the literature which leverage the context of the IoT data in order to execute IoT applications efficiently, are discussed below.

Mahmud et al. [147] propose an application placement policy for fog computing systems which aim at facilitating industrial applications. This policy considers the context of the IoT data for coordinating the workload of the IoT devices with the compute capacities of the nodes in order to minimize the service delivery time. However, the authors do not take into account that the context of the IoT data can be further leveraged for routing the data to compute nodes with available computational resources, which is the goal of our approach.

Mononen et al. [159] discuss a system for executing applications on distributed compute nodes. In this system, the compute nodes utilize a mechanism that leverages the context of the data to avoid sending unnecessary information to the cloud in order to reduce the network traffic. In our approach, we also allow sending data only to nearby compute nodes. However, in addition to that, we enable the IoT data to bypass nearby nodes, in case these nodes are not able to execute the required computations. This can lower the
overhead of forwarding IoT data through the various compute nodes of a fog computing system, as shown in Section 4.4.

Roy et al. [191] present an approach for storing only the context of the IoT data in nearby compute nodes, i.e., not the actual values of the data. This context is then shared among all the nearby compute nodes periodically, in order to facilitate unified IoT applications. However, the authors do not consider using the context for sending the IoT data directly to compute nodes with available computational resources, which is what we discuss in this chapter.

Akbar et al. [15] present an architecture for stream processing in the IoT. The authors propose a mechanism that utilizes the context of the IoT data, which is acquired from previous transmissions, in order to define threshold values needed by complex event processing engines. In this chapter, we also use previous transmissions for defining the context of the data. However, in contrast to the work by Akbar et al., we use the context to change the routing paths of the data, and to improve the efficiency of the system.

Wiener et al. [233] describe a conceptual architecture for context-aware stream processing in fog computing systems. In this work, the authors propose relocating the applications according to the changes in the context of the IoT data. In our approach, instead of relocating the applications, we use the context of the data to change the routing paths, and to send the data directly to compute nodes that can perform the required computations.

To summarize the discussion, there are various approaches in the literature which leverage the context of the data in order to improve the execution of applications in the IoT. However, to the best of our knowledge, none of these approaches considers a context-aware routing mechanism for fog computing systems. In this chapter, we design and implement a context-aware routing mechanism that provides reduced communication latency and improved bandwidth utilization compared to a baseline method which is based on previously discussed approaches (such as [160] and [23]).

Finally, there is related work from the field of cloud computing, which considers a centralized scheduler that maintains a global view of the system, i.e., the IP addresses of all the candidate compute nodes are known a priori [34]. This scheduler is able to perform application placement, usually based on optimization logic, so that each application is instantiated on a specific compute node, and the data is sent to that node directly (i.e., without traveling on a path of multiple nodes). Notably, such approaches have also been applied to computing systems that include both cloud and edge compute nodes [109]. In such systems however, maintaining a global view may become infeasible or prohibitively costly, e.g., due to their scale (which may be too large to be centrally coordinated) [46], or their dynamicity (which results in nodes joining/leaving concurrently) [39, 76, 115].

To avoid these issues, many approaches consider that fog computing systems are not centrally orchestrated, and no node maintains a global view of the system [160, 217]. Instead, each node maintains only few neighbors, i.e., a limited view of the system, and the data is propagated in the system on a path through these neighbors [113]. For
such systems, we design a mechanism that enables the IoT data to be sent directly to appropriate compute nodes even without a central scheduler component. To achieve that, we leverage the context of the IoT data, and a history of previous transmissions that stores which context is typically accepted by each node.

4.3 Context-Aware Routing Mechanism

To present our context-aware routing mechanism, first we discuss the utilized system model in Section 4.3.1, and then we describe in detail the proposed mechanism in Section 4.3.2. Finally, in Section 4.3.3, we discuss a predictive method based on RL that can be integrated within our mechanism in order to achieve better performance.

4.3.1 System Model

In this section, we present a system model for executing IoT applications in nearby and remote compute nodes based on fog computing principles [11]. This system model includes the arrangement of the compute nodes, the flow of the data with regard to the execution of the applications, and the type of IoT applications that we consider.

Similar to the system model of Chapter 3, in the system model of this chapter, there can be various compute nodes which span from the edge of the network where the IoT devices reside, up to the cloud. The entirety of the participating compute nodes and devices is referred to as a fog computing system (same as in Chapter 3). In contrast to Chapter 3, in this chapter we assume that all the nodes of the system are already organized (and aware of their neighbors) and form paths from the edge of the network up to the cloud, as shown in Fig. 4.1.

As shown in this figure, a fog computing system includes multiple compute nodes which reside either in the cloud or at the edge of the network [44]. The cloud compute nodes are nodes in data centers that may be located far away from the IoT devices. The edge compute nodes represent available computational resources at the network edge, e.g., access points, base stations, or specialized edge nodes offered by cloud providers (such as the edge zones by Microsoft [6]). Commonly, all of these compute nodes communicate with each other by forming hierarchies over the Internet [44] [113].

The IoT devices are located at the bottom of the hierarchy, and physically close to the edge compute nodes [101]. These devices integrate sensors and/or actuators in order to sense and/or interact with the surrounding environment [237]. The IoT devices are usually resource-constrained, and may not integrate enough computational resources to implement the necessary communication protocols for interacting with the compute nodes directly (e.g., using an application layer protocol such as HTTP) [106]. For this reason, a gateway is used [11]. This gateway provides two interfaces: One interface is for the communication with the IoT devices (commonly using low-power wireless protocols such as Zigbee or Bluetooth [29]). The other interface is used for the communication with compute nodes over the Internet [101]. The gateway may also be able to act as a
4.3. Context-Aware Routing Mechanism

A fog computing system may consist of multiple compute nodes, gateways, and IoT devices. Notably, Fig. 4.1 depicts the communication between IoT devices and one gateway, and also the gateway and compute nodes. This is done for simplicity because other devices and nodes form similar hierarchical paths towards the cloud. Each layer of the hierarchy provides additional computational resources [11]. Typically, the compute nodes close to the IoT devices provide limited computational resources with low communication latency [11]. The compute nodes which are farther along the path, on the other hand, are able to provide more or even (virtually) unlimited computational resources. Nevertheless, the communication latency to reach these compute nodes also increases due to the longer network paths [49]. Because of the virtually unlimited resources of remote cloud nodes, we assume that the last compute node on path is always a cloud compute node which is able to accept any application request [112].

Figure 4.1: A fog computing system with edge and cloud compute nodes.

compute node, in which case the second interface might not be used if only the local computational resources are utilized for executing the required computations.
4. Context-Aware Routing

**IoT Data Flow**

The flow of the data in this system model starts at the IoT devices which generate the IoT data using sensors [194]. The IoT data is then sent to the gateway which encapsulates this data into an application request. An application request consists of the necessary information to request the execution of a computation from a compute node [71]. This includes an application identifier indicating the specific application to be instantiated, and the IoT data to be used as input for this application. The application identifier is selected based on the identifier of the sensor that generated the data. The association between sensors and applications is configured in the gateway when deploying the IoT devices. This configuration can be altered when new IoT devices and/or new applications are deployed. Thus, the gateway receives the IoT data, creates an application request by adding an application identifier, and sends this request to a compute node in proximity.

Each compute node in a fog computing system is able to receive application requests through an interface. Upon the arrival of a request, the node examines the utilization of its local computational resources. Based on this, the node decides whether to instantiate an application (e.g., using software containers or VMs) and process the data, or to forward the application request to the next compute node upwards the hierarchy (in a depth-first manner [217]). Thus, similar to related work (e.g., [23, 113, 217]), we assume that each compute node (and also the gateway) is able to communicate with a node in the higher hierarchical layer. For simplicity, in this chapter we focus only on hierarchical paths of compute nodes from the edge to the cloud.

When a compute node accepts an application request and the required files to instantiate the application are not available locally, the node downloads these files (which are located using the application identifier) from a repository which contains all the applications (e.g., Docker hub). We assume that every node needs to download the files only once at the beginning of each application [170]. For this reason, we consider that the download overhead is negligible.

If a node cannot download the requested application (e.g., due to limited permissions), the application request is forwarded to the next node. For example, applications related to monitoring and storing may be eligible for execution only in the cloud, because the cloud can act as a point for central monitoring [223]. The application which handles the final processing of the data, sends the final output to the gateway (for actuating commands) and/or to the cloud (for monitoring and storing purposes).

The output of executing an application request can be: raw information to be stored in the cloud (e.g., for monitoring), an actuation command to be sent to an IoT device, or a new application request. This depends on the logic and function of each application [113]. In case the output is an application request, this request is first examined locally, and if there are not enough available computational resources, it is forwarded to the next node upwards the hierarchy. The reason that the request is forwarded upwards (i.e., towards the cloud) is that since a compute node has received an application request, it is expected that the compute nodes lower in the hierarchy are less likely to accept the new
4.3. Context-Aware Routing Mechanism

The core of the previously discussed system model (i.e., with compute nodes that form paths of edge and cloud compute nodes) has been deemed suitable for addressing various use cases such as: IoT analytics, wearable cognitive assistance, augmented reality, image processing, intelligent transportation systems, interactive networked gaming, and industrial manufacturing [23, 31, 50, 160, 217]. To represent such use cases, we consider an application model that consists of multiple applications which can be executed in a distributed manner, and work towards a common goal [49], as shown in Fig. 4.2. Notably, in Chapter 3 we further divide an application into latency-sensitive and latency-tolerant processes. Doing this allows us to evaluate the proposed algorithms (of Chapter 3) with regard to creating a system structure that may need to host applications with and without strict low latency requirements. In this chapter however, we aim at lowering the communication latency of any given application. For this reason, the distinction of processes is no longer necessary.

An example of the application model is discussed as follows: Initially, a sensor, e.g., a gas meter in the case of an IoT smart energy application, generates measurements. These measurements are then processed by various applications which may be related to anomaly detection, cost optimization, etc. The output of these applications can be a command such as: to store information in the cloud, to turn on/off the heating, to send a smartphone notification (e.g., regarding leaks), etc. In our system model, the measurements are transmitted to the gateway which sends this data on a path towards the cloud. When an actuation command is generated (by an application), it is sent back to the gateway and subsequently, to the target IoT device. In this manner, a wide range of use cases that involve input from sensors, processing from different applications, and output related to commands (e.g., send notification, store, or actuate), can be represented by our application model.

Notably, in this system model the application provider is responsible for providing, maintaining, and configuring the IoT devices, the gateway, and the compute nodes. The same model can also be applicable to a system which uses only a remote cloud compute node, i.e., by following the cloud computing paradigm. Nevertheless, there are various aspects of this model which favor a system that utilizes distributed nearby and remote compute nodes. For example: the way that application execution is requested (i.e., using...
the input data along with an application identifier), the use of multiple applications which can be executed on different compute nodes, the permissions of the compute nodes to download and execute different applications, the use of a repository which hosts all the files of the applications, and is accessible from the compute nodes. Thus, when using such a system model, an application provider is expected to be owning/leasing/renting and employing various distributed compute nodes. The benefit for the application provider can be based on a fog computing pricing model [12].

4.3.2 A Routing Mechanism For Fog Computing

Based on the system model discussed in Section 4.3.1, in the following we present a mechanism which considers the context of the IoT data in order to route application requests to the compute nodes of a fog computing system. To this end, first we discuss the conceptual foundation of the proposed mechanism and after that, we describe how the proposed mechanism can be implemented. Finally, we discuss a concrete predictive method that can be used within our mechanism in order to select the closest compute node with available resources.

Conceptual Foundation

Unlike alternative approaches from the literature (discussed in Section 4.2), which route the IoT data on a path from the edge towards the cloud as shown in Fig. 4.3a, our approach aims at sending the IoT data directly to the closest compute node with available computational resources. To achieve that, we design a routing mechanism for the gateway. According to this mechanism, the gateway determines which compute node is more likely to accept an application request. Then, based on this information, the gateway sends the application request to that compute node directly. Thus, as shown in Fig. 4.3b, the gateway of the proposed approach is able to send data directly to each compute node of the system.

In case the selected node cannot accept the application request (e.g., due to dynamic factors such as the availability of computational resources), the proposed mechanism falls back to the traditional routing approach. This means that after the initial transmission of an application request, i.e., from the gateway to a selected compute node, this compute node either accepts the request, or forwards it to the next node upwards the hierarchy, as discussed in Section 4.3.1.

The aim of this logic aligns with two prime goals of fog computing [20, 115]: i) To reduce the latency of sending IoT data. ii) To improve the utilization of the available network bandwidth. Considering these two goals, we make the following key observations regarding traditional routing (i.e., when routing the data on a path towards the cloud):

- When the IoT data is sent towards the cloud, it is expected that a compute node close to the gateway accepts and processes the data. However, considering that the compute nodes at the edge of the network have limited computational
4.3. Context-Aware Routing Mechanism

Figure 4.3: Routing approaches in fog computing systems: (a) shows a gateway which sends data through compute nodes, whereas (b) shows a gateway which is able to send data directly to each compute node.

resources \[149\], and that the amount of data and computations from the IoT keeps increasing significantly \[234\], it is likely that only a fraction of the IoT data is actually processed at the edge of the network. The rest of the data, trying to find the closest compute node with available computational resources, is forwarded closer to the cloud. Thus, in traditional routing, a big part of the IoT data is processed by compute nodes with communication latency that has been increased by all the previous nodes that examined and forwarded the application requests due to not having available computational resources.

- When an application request is sent, e.g., from a gateway A to a compute node D, this request is sent based on a routing algorithm, e.g., using the border gateway protocol that finds the best routing path from A to D \[14\]. However, when a request is routed through other nodes, e.g., from A to B to C to D, even though each transmission (e.g., A to B, B to C, and C to D) is routed through the best path,
the transmission to the final node (i.e., from A to D) may include detours in case B and C do not exist within the best path from A to D. Thus, if the utilized network path to a compute node that accepts an application request includes detours, it is likely that both the communication latency and the bandwidth utilization are increased because the best path has not been followed.

Context-Aware Routing

To design a routing mechanism for fog computing according to the conceptual foundation presented above, we leverage the context of the IoT data. To define context, we build upon the literature of context-aware computing, in which the applications are able to use information gleaned from different parts of the system in order to adapt their behavior [89]. Thus, by leveraging such information for routing, which is the goal of our work, we aim at adapting the routing paths of the data, in order to improve the network performance of fog computing systems [89].

In context-aware computing systems, it is usually the sensors that provide information which is considered as context [190]. Despite that, it has been observed that the notion of context in a system may consist of different information that is correlated and interdependent. Therefore, in some cases (such as for learning and predicting purposes, or for decision making) the context should consider information from different parts of the system [192]. In our system, the context should represent the amount of computational resources which are needed for accepting an application request. This way, there can be a relation between the context of an application request, and the compute node that typically accepts it, i.e., the node that usually has enough available resources to accept it.

For this reason, in this work we define as context the combination of the sensor identifier (of the sensor that generated the data), and the application identifier (of the application that is needed for processing the data). These are both common parameters in IoT systems [205]. We define the context in this manner because the sensor data alone may not hold enough information to indicate the required resources, since this also depends on the tasks of the application (e.g., for the same input data, an application that applies a filter may need less resources than an application that trains an artificial neural network). Similarly, an application identifier alone may also not be enough to indicate the required resources, because this also depends on the input data. The amount of input data used in each application request is assumed to be consistent for each application, e.g., an application that applies a filter on an image, is expected to always receive the same number of images per application request. We make this assumption so that the amount of data per application request, does not affect (significantly) the required resources to accept this request.

This way, by combining the sensor and application identifiers, we consider that the context can be representative of the amount of computational resources required for executing an application request. According to this context definition, we design a mechanism for the gateway, which keeps a history of the compute nodes that accept application
4.3. Context-Aware Routing Mechanism

requests, along with the context that each compute node accepts. This way, every time the gateway is about to send a new application request, the history is examined in order to find the compute node that usually accepts requests of the same context, i.e., the compute node that usually has sufficient available computational resources for executing an application request of the same context.

Presumably, the proposed approach relies on the assumption that a compute node which accepts data of a particular context, is likely to accept such data again in the future. We claim that this is a reasonable assumption in fog computing systems because the farther along the path the IoT data travels, compute nodes with additional computational resources are found. Therefore, the probability that an application request is accepted, increases while the data is forwarded along the path. Notably, this probability depends also on the current load of each compute node, and the amount of computational resources needed for accepting an application request. Since the load of the compute nodes is not known at the gateway, we rely on the latter. Thus, we leverage the context of the IoT data, because it is considered representative of the computational resources needed for accepting an application request. This is discussed further in Section 4.4.2 with results which support that in a path with an increasing probability of accepting the data, it is the same compute nodes that tend to accept application requests of the same context.

Fig. 4.4 shows the high-level architecture of the components that we place in the gateway in order to implement the proposed mechanism. The application identifiers component stores the identifiers of the applications that are needed for processing the data from each sensor. This component is assumed to be preconfigured, as discussed in Section 4.3.1. The compute nodes component stores the IP addresses of compute nodes along with the context of the application requests that each compute node has accepted in the past. Initially, this component is assumed to have the IP address of a compute node in proximity, as discussed in Section 4.3.1.

The functionality of the proposed mechanism is triggered by sending a set of data that requires processing, along with the sensor identifier, to the application request creator, as shown in Fig. 4.4. It is also possible that the set of data consists of subsets of data that have been generated by more than one sensor. In this case, each subset is accompanied by a sensor identifier. The application request creator compares the sensor identifier(s) of the received data with the sensor identifiers of the application identifiers component, and creates an application request with the match (by adding the application identifier, as discussed in Section 4.3.1). The application request is then sent to the data analysis component. At the same time, the context of this request (i.e., sensor and application identifiers) is sent to the compute nodes component. If an application request of the same context has not been transmitted before, the data analysis component pulls the IP address of a node in proximity from the compute nodes component, and sends the application request to that node.

Then, the application request travels through the compute nodes of the system until a node with available computational resources accepts it. Upon acceptance, the compute node instantiates an application as discussed in Section 4.3.1, and responds with its IP address.
address (i.e., the address of the node that accepted the request) which we refer to as the application response. Subsequently, the application response is sent back to the gateway, as shown in Fig. 4.4. The gateway stores the application response in the compute nodes component, as the IP address of the compute node that accepted the context that was stored in the compute nodes component when the application request was created. Thus, the gateway stores the context of each application request before sending it, and adds the IP address of the compute node that accepts it, as soon as this node sends back the application response.

In case an application request of a specific context has been transmitted before, the data analysis component sends the request to one of the compute nodes that have accepted requests of the same context in the past. The decision on which one of these nodes should be selected can be made based on various strategies such as: most recently used, most frequently used, or using predictive methods. Notably, current machine learning approaches from the literature which aim at selecting appropriate compute nodes for deploying applications may also make suitable strategies [65]. In particular, we consider RL to be an intuitive choice for the logic of the data analysis component, because the application response contains the node that accepted each application request. This information can then be used for determining the reward/penalty in a RL-based model [203]. We elaborate further on this strategy in Section 4.3.3 which provides the concrete logic of an RL-based data analysis component for our mechanism.

Interestingly, our mechanism stores the minimum amount of information needed to be able to send application requests to compute nodes based on context. This information includes only the address of candidate compute nodes, and the context of the application requests. While alternative approaches (discussed in Section 4.2) may also use additional
properties such as the location of the nodes, or their resource capacities, we store less information in order to make our mechanism more practical for gateways which do not have ample memory resources. However, such properties are in fact considered implicitly in our mechanism. For example, the locations of nodes are taken into account by using latency as a proximity measure, and by preferring nodes which respond with less latency (i.e., nearby nodes). Also, the resource capacities of the nodes are considered because only the nodes with sufficient resources are stored in the gateway.

In our approach, the data analysis component may also cross the information of different contexts. For instance, when there are many candidate nodes which grade equally for a specific context, the ones that have been used recently for other contexts may be excluded to avoid potentially busy nodes. In case the selected node is unresponsive (e.g., due to temporary/permanent disconnection or failure), another node can be selected based on the same strategy as long as one exists. Otherwise, the proposed mechanism falls back to the traditional approach, and sends the application request to the compute node in proximity.

Since fog computing systems can be volatile, and compute nodes may be added/removed temporarily or permanently at any time, we use a system parameter to reset the compute nodes component to its initial state periodically (e.g., based on time, or number of transmissions). This ensures that even though the compute nodes in proximity may be bypassed because they are likely to be busy, their availability is examined frequently. The exact value of this parameter depends on the reliability and volatility of the system, i.e., fog computing systems that are expected to change frequently, should perform a reset more often than stable systems. This parameter may also be adaptive, and change dynamically based on the performance of the system. For example, if it is observed that when resetting the compute nodes component, a compute node in closer proximity is found, it may be beneficial to start resetting more regularly. On the other hand, if resetting the compute nodes component results in selecting the same node as before, then resetting should be performed less frequently. This way, the frequency of the reset can converge towards a suitable value.

Finally, we note the applicability of our approach for fog computing systems with multiple IoT devices, gateways, and compute nodes. As discussed in Section 5.3.1, in this paper we focus on the communication of one gateway (with multiple IoT devices and compute nodes), because other gateways of the system form very similar paths to the cloud. Nevertheless, our system model and the proposed mechanism apply to systems with many gateways as well. Since each gateway of the system stores only few compute nodes and selects the most suitable among these nodes (rather than selecting one among all the nodes of the system), our mechanism follows a decentralized approach. In contrast to centralized approaches in which a global view of the system is assumed for selecting the most suitable node, decentralized approaches tend to scale better while causing less overhead.
4. Context-Aware Routing

4.3.3 Reinforcement Learning for the Data Analysis Component

In this section, we propose a concrete strategy for the data analysis component, which is based on RL [214]. In the following, first we adapt the RL model to the proposed context-aware mechanism and after that, we present the learning algorithm.

Reinforcement Learning Model

RL relies on agents that interact with their environment and receive feedback for their actions in the form of reward or penalty. Based on this feedback, the aim of the agents is to learn a policy in terms of selecting actions given the perceived state of the environment, so that the expected cumulative reward is maximized. In our case, the agent is installed at the gateway and at each time step, i.e., upon the arrival of an application request, selects the appropriate compute node to forward the request to, based on the environment’s current state. Then, the agent inspects the latency experienced to handle this request, which constitutes the reward signal. This signal is considered as the feedback which is used in a process towards learning to select appropriate actions.

The agent has a set \( A \) of available admissible actions, with \( A_t \) denoting the action of the agent at time \( t \). In our case, the actions available to the agent are the different compute nodes that can receive an application request from the gateway. Therefore, \( A_t = i \) represents the action to forward the application request to node \( i \) at time \( t \). Each action brings the environment to a potentially different state, out of a (finite and discrete, in our case) state space \( S \): When at state \( S_t = s \), selecting action \( A_t = a \) will lead to state \( S_{t+1} = s' \) with a specific transition probability \( p(s'|s,a) \), which is however unknown to the agent. Finally, the immediate reward the agent receives when performing action \( a \) and the system is at state \( s \) at time step \( t \), is given by a reward function \( R_t(s,a) \). Since our agent can only directly observe the latency experienced for each request, we use its normalized value as the reward, i.e., \( R_t(s,a) = 1 - \frac{l_t(s,a)}{l_{\text{max}}} \), where \( l_t(s,a) \) is the latency measured by the gateway for the \( t \)th application request when from state \( s \) compute node \( a \) was selected, and \( l_{\text{max}} \) is a reasonably large maximum latency value.

In RL, the environment is typically modeled as a Markov Decision Process (MDP), defined as the 4-tuple \((S,A,p,R)\) [214]. The key characteristic of an MDP is the Markov property, namely that the probability to transit to a state depends only on the current state and the action taken. Otherwise put, a state should encode all the information necessary to drive an agent’s decision, and the reward received should only depend on this state-action pair.

However, this assumption may not hold in some practical use cases for our system model. Importantly, the actual state of the environment might not be able to be fully observed. In our case, for example, the state in terms of the current load of each compute node, its reliability, and in general, its probability to accept an application request, which, intuitively, are critical for driving the agent’s decisions, are not visible by the agent. Instead, the agent is only aware of some environment observations, such as the actual node that responded to a request. We are therefore forced to operate only with an
4.3. Context-Aware Routing Mechanism

approximate view of the state of the environment. The agent however, can record a history of observations as a response of the environment to the agent's actions. Following the terminology and approach of Sutton and Barto [214, Section 17.3], what we need is a compact representation of a state as a summary of the history of observations and actions that have led the environment to a given state. The state thus becomes a function of history, i.e., \( s_t = f(H_t) \), where \( H_t \) is the sequence of observations and actions up to time step \( t \).

In our model, a state represents the history of how the \( k \) most recent requests were handled by the system. This is in line with what Sutton and Barto term as the \( k \)th order history [214]. In particular, \( S_t = \{n_{t-k}, n_{t-k-1}, \ldots, n_{t-1}\} \), where \( n_i \) is the identity of a compute node that processed a request; namely, the state of the system is maintained as an ordered list of the last \( k \) compute nodes that processed the respective application requests. In order to control the number of states, \( k \) is set to a small constant and can be configured based on the amount of memory available to the agent. Furthermore, this representation of the state as a function of history only includes observations and not the actions that led to them. This is a simplification that is also driven by the need to keep the state space low.

Importantly, such a state representation is not guaranteed to come with the Markov property. For example, this may be the case if the probability of a node to accept a request depends on whether the node has recently been used for a prior request, as is the case for the compute node model that we assume in our evaluation (in Section 4.4.3). In this work, the agent is considered to be agnostic of the actual behavior of the compute nodes, and is forced to operate on minimal observed information regarding their status, namely the IP address of the compute node that accepted an application request, and the respective latency. The system in question may not be able to be represented as an MDP due to how nodes operate, and because of the inability of an agent to fully observe the state of the environment. We apply Q-learning [232], which is a generic model-free RL mechanism, to derive the agent’s policy. However, due to the above-mentioned fact, convergence guarantees cannot be provided for general settings. Interestingly, there are families of decision processes beyond MDPs for which Q-learning has been shown to come with convergence guarantees [148].

As a final note, we treat each context individually. Namely, there is a dedicated agent for each context which acts upon this particular context’s application requests, and which maintains its own state of the environment (that is not shared across agents). This decision is motivated by the following observations:

- Whether a request for a specific application is served by a node is considered to be unaffected by whether this node would accept requests for other applications. For example, a node might have enough resources to serve a specific application request, but not enough for another.

- By assuming a different decision process (and, thus, a dedicated agent), per context, and given our state model, we limit the number of states handled by each agent to
$N^k$, where $N$ is the number of compute nodes and $k$ is a small constant (representing the history). The total number of states of all $M$ agents is then $MN^k$. Had we opted for factoring the application context in a single global state representation for all applications, e.g., where the elements of a state would be application instance-node address pairs, we would have come up with a single agent with $(MN)^k$ states, that is considerably larger.

Q-Learning Algorithm

We apply Q-learning [232] to learn an appropriate action selection policy in our system. A Q-learning algorithm aims at deriving a value function $Q: S \times A \rightarrow \mathbb{R}$, which represents the expected cumulative discounted future reward if the agent selects action $a$ at state $s$, and continues by following the optimal policy. $Q$ can be represented as a table, which is updated each time an agent takes an action, using the following rule:

$$Q_{t+1}(S_t, A_t) = (1 - \alpha_t(S_t, A_t))Q_t(S_t, A_t) + \alpha_t(S_t, A_t)(R_t(S_t, A_t) + \gamma \max_a Q_{t+1}(S_{t+1}, a))$$

where $\alpha_t(s, a) \in [0, 1]$ is the learning rate at time step $t$ and $\gamma \in [0, 1]$ is a factor that is used to discount future rewards. The higher the value of $\gamma$, the higher the importance of future rewards compared with immediate ones.

As it is used at runtime and starts with an unknown Q function, the Q-learning algorithm needs to make decisions while learning. Therefore, the agent alternates between exploration and exploitation steps. There are various mechanisms to perform this. A widely used mechanism is the $\epsilon$-greedy strategy, where the agent chooses a random action (thus exploring the environment) with probability $\epsilon$, while with probability $1 - \epsilon$ it follows the action with the highest Q value (thus exploiting the acquired knowledge of the environment). In order to favor exploration at the beginning, we start with $\epsilon = 1$ and gradually reduce it as rounds progress. The learning rate for each state-action pair is initially set to 1 and also decays the more the pair is visited [69].

A point that requires further attention is that an implicit assumption is made that the state and action spaces are static, as is the case for the data structure which stores the Q values. However, according to the proposed mechanism described in Section 4.3.2, the compute nodes component which stores compute node IP addresses (these correspond to actions, while histories composed of these addresses correspond to states), starts with only the IP address of a single node in proximity. When the IP addresses of the compute nodes on path are not known in advance, potential methods to get around this can be applied. Such methods can be:

- To start with a discovery phase whereby traditional routing is executed for a sufficient number of rounds until the nodes on path are discovered [107,162], and then to switch to Q-learning. The duration of the discovery phase depends on the
4.3. Context-Aware Routing Mechanism

load of application requests, and the number of nodes that accept these requests. For example, when application requests are sent sparsely, the discovery phase may take longer than if application requests are sent frequently. Also, the fewer the compute nodes which accept the requests, the faster the discovery phase finishes. For these reasons, this method is preferred when the gateway produces a high load of application requests and/or the number of nodes that accept these requests is bound.

- To build the Q table incrementally. This means adding new states and admissible actions as new nodes are being discovered, each time maintaining the existing Q values for the already known state-action pairs and learning the values of new ones by appropriately adjusting the exploration and learning rate parameters. Thus, this method requires additional updating of the Q table every time a new node is discovered. When using this method, the acquired knowledge is utilized from the beginning because there is no discovery phase (in contrast to the former method). Nevertheless, the additional updating of the Q table might introduce extra overhead. Hence, this method is preferred when new nodes are expected to be discovered sparingly.

Since these methods are executed until sufficient knowledge is acquired (i.e., not constantly), in our evaluation in Section 4.4, we assume that the compute nodes on path are known (e.g., due to a discovery phase).

It is worth noting that the proposed mechanism operates on minimal context and feedback: i) the latency observed per request, and ii) the compute nodes that accepted the request. For this reason, our mechanism does not utilize an excessive amount of resources (this is also discussed in Section 4.4.3). Each time a request is handled, the overhead of selecting the best action and updating the Q table depends on the number of possible actions. The actions correspond to the stored compute nodes, i.e., the nodes that accept application requests, which are only a subset of the compute nodes of the system. Thus, by storing only a subset of the compute nodes of the system, the number of actions remains low which limits the computational overhead (and latency) of handling each request. Our model and mechanism are also agnostic to the behavior and actual state of the compute nodes on path. If we assume further knowledge about the environment (e.g., information about the runtime of each compute node such as: the current load, device capabilities, probability to accept new application requests, etc.), a more accurate view of the state can be acquired, and more efficient mechanisms may be possible. However, this could potentially increase the utilization of computational resources. We defer such mechanisms, as well as other approaches for dealing with an uncertain environment, to future work.
4.4 Evaluation

In this section, we present an implementation of the proposed context-aware routing mechanism. Furthermore, we conduct a series of experiments, and we compare the proposed context-aware routing to a baseline routing approach. The baseline is based on [23] and [160], which propose that each compute node on path examines an application request and either accepts it or forwards it to the next node. Hereinafter, we refer to this approach, which is also shown in Fig. 4.3a, as traditional routing.

In order to show the differences compared to traditional routing, we evaluate the proposed mechanism in the following manner: First, we establish the potential benefits of performing context-aware routing on a real-world setup with nearby and remote compute nodes, and we show that even a simplistic strategy for the data analysis component has the potential to outperform the traditional routing approach. Then, we turn our attention to the performance of the RL-based strategy, and we conduct extensive simulations that show benefits in various scenarios.

4.4.1 Evaluation Environment and Prototype

To create an evaluation environment, we implement a prototype of a compute node which either accepts an application request or forwards it to the next node on path, as discussed in Section 4.3.1. Moreover, we implement a prototype of a gateway which integrates the proposed mechanism for context-aware routing, as discussed in Section 4.3.2. Both prototypes are developed in Java 11 using the Spring Framework, and implement the required functionality to perform experiments and take measurements related to the communication latency and hop count of the utilized network paths.

To emulate a fog computing system, we assume the following scenario. An IoT application provider based in Los Angeles has successfully commercialized smart energy solutions (e.g., for detecting gas leaks, reducing cost, etc.). This provider uses cloud computing resources stationed in a data center in Los Angeles, but due to the popularity of the provided IoT applications, the provider decided to expand this business to the rest of the US and to Europe. Since such IoT applications can be related to safety (e.g., due to detecting gas leaks), and may require low latency that the centralized cloud might not be able to support [139], the provider decided to follow the fog computing paradigm. Thus, the provider acquired access to various geographically distributed compute nodes in the cloud (offered by a cloud provider such as Google or Microsoft), and at the edge of the network (e.g., using the edge zones offered by Microsoft [6]).

For this evaluation, we examine the case of a client in central Europe (i.e., in Vienna, Austria), and we consider that various compute nodes exist on the path towards the cloud in Los Angeles, as shown in Fig. 4.5. Notably, we do not distinguish between edge and cloud compute nodes because: i) Both cloud and edge nodes are able to provide the same services [6], and can therefore be considered similar. ii) In this setup, we use nearby and remote compute nodes using cloud services because even though edge nodes
4.4. Evaluation

Figure 4.5: Location of the compute nodes used in the evaluation.

have been announced by cloud providers [6], their availability is still limited. Thus, similar to Fig. 4.1 which shows our system model, we examine a fog computing system with a gateway which sends application requests that are forwarded by various compute nodes towards the cloud. The specific compute nodes we use (as shown in Fig. 4.5), are provisioned using the [Google Cloud Platform (GCP)]. The type of the compute nodes is *e2-standard-2*, i.e., standard general-purpose compute nodes with two [virtual Central Processing Units (vCPUs)] and eight [Gigabytes (GB)] of RAM (although the resource capacities of the compute nodes do not affect the presented results significantly because this evaluation focuses on network-related metrics). Hence, for this evaluation we create an actual computing system with nearby and remote compute nodes that span a large geographical area. Interestingly, the remote compute nodes in the US can provide insights on the communication over high-latency network links. This is useful because high-latency links represent cases when remote compute nodes are employed due to the nearby nodes being busy. In addition, high-latency links may be considered representative of scenarios that include network links which induce high latency for other reasons, e.g., due to congestion.

In this system, the gateway receives gas volume measurements from a smart meter. Then, the gateway creates the application request of an application that detects gas leaks. In the event that a gas leak is detected, the application sends an actuation command to shut down all sources of ignition, e.g., cooktops, toasters, etc. Therefore, for such an application, reducing the communication latency of sending the gas measurements to a compute node, aids in processing the data faster, and reduces the overall latency of responding to fire hazards. For this reason, we apply the proposed approach which aims at reducing the communication latency of sending data to nearby and remote compute nodes, in order to examine the potential benefits.

To emulate the smart meter, we use real gas volume measurements from a smart home, which have been collected periodically every 30 minutes during the course of four days, i.e., 200 measurements. This dataset is part of the data provided by the Loughborough University, which has been gathered in the context of the REFIT project that monitored 20 smart homes in the United Kingdom [8].
4. Context-Aware Routing

4.4.2 Prototype-based Results

In this section, we use our prototype implementation to create fog computing systems based on the traditional and context-aware routing approaches, and we report on our findings. Specifically, in the following we provide a comprehensive view on the results of the communication latency to reach every compute node of the system based on each routing approach. For these results, we assume that the data analysis component of the context-aware routing is able to determine the compute node that accepts the application requests. Moreover, we also present results from the runtime of a specified scenario with dynamic load from application requests. For these results, the data analysis component utilizes a rather simplistic strategy: Send each application request to the most recent node that has accepted requests of the same context, and reset the compute nodes component with every second transmission. This experiment aims at showing that even in cases which do not allow for great benefits (e.g., with dynamic load and a very simplistic data analysis component), the proposed context-aware routing mechanism still manages to show improvements.

Since the proposed approach aims at reducing the communication latency (which is independent of potential processing delays), this evaluation reflects on that by focusing on network-related metrics (rather than application execution metrics). Nevertheless, the additional utilization of computational resources in the gateway, which is required to execute the context-aware routing, is considered as the overhead of the proposed approach. For this reason, in this section we also discuss resource utilization aspects. For all the presented results, we have repeated each experiment 200 times with the values of our dataset in order to capture the general behavior of each examined approach.

Context-Aware Routing Results

In order to acquire a comprehensive view of the two examined approaches considering that context-aware routing is able to determine the node that accepts each application request, we perform the following experiment. The gateway sends application requests to each compute node of the system two times. One time the data is sent according to traditional routing (i.e., on a path towards the cloud), and the other time according to the context-aware routing (i.e., directly to the selected compute node). Each time, we measure the communication latency to reach the compute node, and the hop count (using Traceroute which is a network diagnostics tool also used for the evaluation of Chapter 3 in Section 3.4.1). In the communication latency, we also include message parsing and making the data available to the application hosted in the destination node (but exclude any further processing by that application). Table 4.1 shows these measurements for each compute node, along with the percentages of reduction when using the proposed context-aware routing approach.

Notably, the hop count to reach each node of the system is always the same in the 200 iterations, because the underlying network connectivity is not affected by our experiments. For this reason, in Table 4.1 we do not specify the average and standard deviation of
4.4. Evaluation

Table 4.1: Communication latency (in ms) and hop count to reach each compute node of the system based on the two examined approaches, and the percentages of reduction.

<table>
<thead>
<tr>
<th></th>
<th>Traditional routing</th>
<th>Context-aware routing</th>
<th>Reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hops (average / st. dev.)</td>
<td>Hops (average / st. dev.)</td>
<td>Hops (average)</td>
</tr>
<tr>
<td>Zurich</td>
<td>15 / 93</td>
<td>15 / 93</td>
<td>0 / 0</td>
</tr>
<tr>
<td>Frankfurt</td>
<td>21 / 104</td>
<td>18 / 96</td>
<td>14 / 8</td>
</tr>
<tr>
<td>Belgium</td>
<td>30 / 118</td>
<td>22 / 100</td>
<td>27 / 15</td>
</tr>
<tr>
<td>Netherlands</td>
<td>43 / 127</td>
<td>23 / 101</td>
<td>47 / 20</td>
</tr>
<tr>
<td>London</td>
<td>53 / 140</td>
<td>18 / 108</td>
<td>66 / 23</td>
</tr>
<tr>
<td>Montreal</td>
<td>58 / 264</td>
<td>21 / 224</td>
<td>64 / 15</td>
</tr>
<tr>
<td>Virginia</td>
<td>65 / 282</td>
<td>23 / 226</td>
<td>65 / 20</td>
</tr>
<tr>
<td>Iowa</td>
<td>80 / 321</td>
<td>22 / 253</td>
<td>73 / 21</td>
</tr>
<tr>
<td>Salt Lake City</td>
<td>89 / 362</td>
<td>25 / 283</td>
<td>72 / 22</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>95 / 385</td>
<td>26 / 296</td>
<td>73 / 23</td>
</tr>
</tbody>
</table>

The hop count. The reason we count hops is that network hops can be considered as an indicator of bandwidth availability since nodes that reside many hops away from each other, tend to communicate with low bandwidth [196]. Therefore, a lower number of hops can be associated with higher bandwidth [43, 112]. This is also discussed in Chapter 3.

In traditional routing, whereby each application request is forwarded to the compute node in closest proximity on a path towards the cloud (as shown in Fig. 4.5), the average latency increases the farther away from the gateway the compute node is located. In context-aware routing, whereby the gateway sends the application request to each compute node directly, the average latency also increases the farther away the compute node is located. However, the rate of increase is significantly lower in context-aware routing.

Both approaches reach the first compute node with 15 hops in 93 ms because both approaches send the application request to the closest compute node directly. After that, the difference in both the communication latency and the hop count becomes evident. Specifically, the context-aware routing approach reduces the average communication latency by up to 23%, and the average number of hops by up to 73%. Notably, even though the difference in the communication latency becomes larger when the distance of the compute node becomes longer, the percentages of reduction become significant even with compute nodes that reside nearby. For example, to reach the compute nodes in the Netherlands and in London (from Vienna), context-aware routing reduces the average communication latency by 20% and 23%, respectively. This shows that our approach shows benefits for both inter- and intra-continent communication. The former can be representative of large-scale applications that operate worldwide, while the latter may be considered indicative of results for medium-scale systems (e.g., that operate within Europe).

To further analyze the rate of increasing communication latency for each routing approach,
4. Context-Aware Routing

![Figure 4.6: Average communication latency to reach each compute node of the system based on the two examined routing approaches (values acquired from Table 4.1).](image)

we plot the average values of Table 4.1 in Fig. 4.6. Since the values shown in this figure exhibit the pattern of straight lines, we also show the linear trendlines which are created using the Least Squares method. Even though the communication latency increases monotonically (which is expected since the physical distance of the compute nodes increases), there is a steep rise in the latency of compute node 6, i.e., the node in Montreal. This derives from the increased latency to reach the remote compute nodes in America, due to the significantly longer distance from the gateway (which can also be observed in Fig. 4.5). Notably, this rise prevents the values of each routing approach from following the pattern of a straight line holistically. However, it is visually evident that the values of the group of nodes before the rise (i.e., the latency of the nodes in Europe), and the values of the group of nodes after the rise (i.e., the latency of the nodes in America), both exhibit a linear pattern. For this reason, Fig. 4.6 shows a different trendline for each group of nodes.

For each trendline, we also present its linear function in the form of $y = \text{Slope} \cdot x + \text{Intercept}$. We do this because the slope of a linear function shows the rate of increase of the output values, as discussed in the evaluation of Chapter 3 in Section 3.4.3. Thus, since the $y$ values of Fig. 4.6 show communication latency, and the $x$ values show the nodes in the order of increasing distance, the slope indicates the rate of increased latency based on the proximity of the gateway (for each routing approach). In addition, we present the coefficient of determination $R^2$ for each trendline. $R^2$ is a widely-used statistical measure which indicates how close are the actual values to the values of the trendline [113]. The value of $R^2$ is always between 0 and 1 [132]. Small values indicate that the model (or the trendline in our case) does not represent the data well, while high values show that the model can be considered representative for the data [119]. Thus, we use $R^2$ to advocate
that the presented trendlines, and consequently the slopes which indicate the rate of increase in the communication latency, bear statistical significance.

For the compute nodes in Europe, the function of the trendline of the traditional approach is \( y = 11.7 \cdot x + 81.3 \), and has a coefficient of determination \( R^2 = 0.997 \). This coefficient of determination shows that the trendline represents the data very well. In addition, we note that the slope of this trendline is 11.7. For the same nodes, the function of the trendline of the proposed approach is \( y = 3.5 \cdot x + 89.1 \), and has a coefficient of determination \( R^2 = 0.948 \). While the coefficient of determination of this trendline is also very high, the slope is significantly lower, i.e., 3.5. This means that with context-aware routing, the communication latency grows by a factor of 3.5 which is about 70% lower than traditional routing.

For the compute nodes in America, we note that the function of the trendline of the traditional approach is \( y = 32.2 \cdot x + 65.2 \), and has a coefficient of determination \( R^2 = 0.985 \) which shows that the trendline exhibits statistical significance. We also note that the value of the slope of this trendline is 32.2. For the same nodes, the function of the trendline of the proposed approach is \( y = 20.1 \cdot x + 95.6 \), and has a coefficient of determination \( R^2 = 0.948 \). While \( R^2 \) is also very high in the proposed approach, the slope of the trendline is much lower, i.e., 20.1. Specifically, this slope shows that the communication latency in the context-aware routing grows by a factor of 20.1 which is about 38% lower than in traditional routing.

Therefore, we conclude that the communication latency of sending IoT data to compute nodes of fog computing systems increases for both routing approaches, when the nodes reside farther away from the gateway. However, the rate of increase is significantly lower in the context-aware routing approach. As a result, context-aware routing is able to send the IoT data with lower communication latency than the traditional approach.

Results From a Scenario with Dynamic Load

In this section, we aim at showing that the proposed context-aware routing mechanism may be able to outperform the traditional routing approach, even in cases which do not allow for great benefits. To this end, we measure the communication latency of the context-aware routing when using a very simplistic strategy in the data analysis component. In the following, first we describe the details of the scenario for this experiment, and then we present the produced results.

**Scenario:** The strategy we use in the data analysis component of the context-aware approach is the following: First, send an application request using the traditional approach, and store the compute node that accepted this request. Then, send the next request directly to the node that accepted the data before. After that, the compute nodes component is reset and the process is repeated.

In addition, we consider the potential load from other application requests and other gateways, along with the potentially limited resources of the compute nodes on path.
4. Context-Aware Routing

This is emulated in our system by configuring the compute nodes on path to have 7% probability of accepting an application request, while the final cloud compute node has 100% probability (due to integrating virtually unlimited resources, as discussed in Section 4.3.1). Since there are nine compute nodes (apart from the final cloud node), the probability of all these nodes rejecting an application request (and thus the request reaching the final cloud node) is \((1 - 0.07)^9 = 0.52\), i.e., approximately 50%. We do this to emulate a realistic environment for our experiments (as discussed in Section 4.3.2), whereby approximately half of the application requests are executed by nodes on path, and the other half in the cloud.

After executing an application, the probability of accepting another application request increases from 7% to 50% for every compute node. This is done to emulate that when one of the hosted applications finishes, a compute node has more available computational resources. This increase is temporary, and lasts until the next application request is sent from the gateway (after that, the probability is reset to 7%). The reason that the probability increase is temporary, is to emulate that other gateways in the area may send application requests, and occupy the newly available computational resources.

Results: For this experiment, the gateway sends each one of the 200 application requests two times according to the traditional and context-aware routing approaches. For the context-aware routing, the gateway uses the mechanism discussed in Section 4.3.2, and the aforementioned strategy for the data analysis component. The produced results, which are related to the communication latency and the hop count of the network paths of each routing approach, are shown in Fig. 4.7. Similar to the values of Table 4.1, for the communication latency we include message parsing but exclude any further processing by the application in the destination node.

Fig. 4.7a shows the distribution of the hop count of the application requests for both approaches. The traditional routing measurements have an average value of 72 hops, and a standard deviation of 33 hops. The context-aware routing measurements have an average value of 51 hops, and a standard deviation of 32 hops. Thus, the proposed approach reduces the average hop count by 29%. Notably, the box plot of the traditional routing does not have an upper whisker, and the median overlaps with the upper quartile. The former indicates that 25% of the maximum values are equal, and the latter that 50% of the maximum values are equal.

This happens because approximately half of the application requests are sent to the final cloud node which bears the maximum hop count. Thus, as Fig. 4.7a indicates, half of the values of the hop count in traditional routing equal the maximum value. The same applies to context-aware routing, i.e., approximately half of the application requests are sent to the final cloud node. However, since context-aware routing sends some of these requests to that node directly, the hop count is significantly lower (as shown in Table 4.1). For this reason, the box plot of the hop count in context-aware routing has lower values, and does not exhibit similar behavior.

In addition, we note that the maximum hop count in this experiment, which represents
4.4. Evaluation

(a) The hop count of the application requests.

(b) The latency of the application requests.

Figure 4.7: Network path measurements of the application requests based on traditional and context-aware routing.

the number of hops to reach the cloud node in Los Angeles is 99 hops. This is slightly different to the hop count of Los Angeles shown in Table 4.1, which is 95 hops. The reason that this happens, is that we use the external IP addresses for the communication between compute nodes. The GCP allows communication between GCP compute nodes using either internal IP addresses (that can be used only between GCP nodes), or external IP addresses that are addressable by any node on the Internet. The network paths when using internal IP addresses may be more stable, which could prevent having a slightly different hop count between experiments. However, a setup that utilizes external IP addresses, is more representative of computing systems which consist of compute nodes that may belong to different providers (which is possible in fog computing systems). Nevertheless, when using external IP addresses, slightly different hop count values between experiments might be observed, due to the dynamic nature of the Internet. We do not consider that this compromises the presented results. On the contrary, we believe that results which include slight dynamic changes are more representative of real-world setups, in which such phenomena are imminent.

Fig. 4.7b shows the distribution of the communication latency of the application requests. The measurements of the traditional routing have an average value of 302 ms and a standard deviation of 140 ms. The measurements of the context-aware routing have an average value of 258 ms and a standard deviation of 113 ms. Therefore, the context-aware routing reduces the average latency by 15%. Notably, the median of traditional routing is higher than the upper quartile of the context-aware routing. This means that 75% of the values of the context-aware routing are lower than 50% of the values of the traditional routing. This further advocates that the proposed context-aware routing mechanism is...
able to reduce the communication latency of sending application requests in fog computing systems.

Resource Utilization

As discussed above, context-aware routing is able to provide latency and bandwidth benefits due to using the mechanism of Section 4.3.2. The implementation of this mechanism however, may require the utilization of additional computational resources in the gateway, which can be regarded as the overhead that accompanies the presented benefits.

In our experiments, the gateway is implemented on a Raspberry Pi 4 single-board computer (which is a popular choice for an IoT gateway [101]). Using Top (a command-line tool also used in the evaluation of Chapter 3 in Section 3.4.3), we examine the resource utilization of the process that routes the application requests according to each routing approach. Notably, both approaches exhibit similar resource utilization with CPU that has occasional spikes which do not surpass 9%, and RAM that does not exceed 10%. The resource utilization of the compute nodes which execute the IoT applications, is not affected by our approach since the context-aware routing mechanism is only executed in the gateway.

In addition, we note that the time required to execute the proposed mechanism in our experiments, i.e., the delay needed for finding a compute node that has accepted data of certain context in the past is negligible. However, this delay may increase if a complex data analysis component is used, or if the number of different contexts stored in the compute nodes component grows significantly (due to the delay of searching). Therefore, the presented results apply to fog computing systems with a gateway which makes application requests of a reasonable number of different contexts (i.e., when a search in the compute nodes component does not incur significant delay).

The application responses used in the proposed approach might also be considered as overhead since the traditional routing does not make use of such information. Nevertheless, when a reliable communication protocol is employed for the communication between compute nodes (e.g., using the request/response HTTP which we use in our experiments), the IP address of a compute node can be encapsulated within the HTTP responses. This way, the application response can be sent without transmitting additional messages to the gateway. Thus, we conclude that the context-aware routing approach using a rather simple data analysis strategy is able to provide solid benefits without inducing significant additional overhead.

4.4.3 Simulation-based Evaluation

In this section, we turn our attention to the performance of our mechanism when using the RL-based strategy for the data analysis component (presented in Section 4.3.3) under a wide range of settings and scenarios. To evaluate this strategy in a comprehensive manner, we build a Python-based simulator which allows us to increase the scale of our
4.4. Evaluation

Experiments substantially (compared to using the setup of Section 4.4.1). In addition, we model our simulator to consider the network parameters of our real-world setup (discussed in Section 4.4.1) in order to ensure that the simulations imitate a realistic environment. The results we report in this section are based on the average latency achieved from 100,000 time steps, i.e., for the first 100,000 application requests. Each experiment is repeated 20 times, and the average values are reported with 95% confidence intervals.

When running our experiments, we compare three routing approaches: i) The traditional routing approach. ii) The simple context-aware routing approach discussed in Section 4.4.2. iii) The RL-based approach with Q-learning discussed in Section 4.3.3 which is referred to as Context-Aware with Q-Learning (CA-QL).

Scenarios

To examine the performance of the three routing approaches, our simulator considers three deployment scenarios with inherently different characteristics, as shown below:

- **Small-scale deployment**: In this scenario, there are two compute nodes: a local edge node which is assumed to be very close to the gateway (average latency of 10 ms), and a remote cloud node which is located far away (with average latency corresponding to the node in Los Angeles, as shown in Table 4.1). This can be considered representative of small-scale setups with one nearby edge node, and one remote cloud node. Notably, this scenario resembles setups which have transitioned from the cloud computing paradigm by adding a local edge node.

- **Medium-scale deployment**: This scenario includes a chain of five compute nodes deployed in the same continent. The communication of these compute nodes corresponds to the nodes in Europe (from Section 4.4.1), and is based on the latency values of Table 4.1. This scenario is considered representative for setups of medium-scale fog computing systems which operate in a wide area within the same continent.

- **Large-scale deployment**: This scenario corresponds to the real-world setup discussed in Section 4.4.1 and includes a chain of 10 compute nodes which communicate with the average latency values reported in Table 4.1. We consider this as a representative scenario for large-scale setups of fog computing systems which may need to operate worldwide.

In addition, we experiment with different cases of nodes accepting new application requests, in each of the three target deployment scenarios. In particular, we consider that each compute node accepts an application request with a probability, as discussed in Section 4.4.2. However, in this experiment, we vary this probability in order to acquire results which cover a wider range of fog computing systems. After a compute node has accepted a request, the probability of accepting the subsequent one increases to a higher value (i.e., to 50%, as discussed in Section 4.4.2). In line with our prototype-based
4. Context-Aware Routing

![Figure 4.8: Average communication latency achieved by the three examined routing approaches for each deployment scenario, as the probability that a compute node accepts an application request grows.](image)

In each deployment scenario of Fig. 4.8, we can observe that all routing approaches tend to converge towards the same minimum latency value when the initial acceptance probability increases. This happens because when this probability becomes high, it is the first compute node on path that accepts most of the application requests. Thus, the context-aware routing approaches are not able to provide benefits by bypassing busy nodes, since most nodes tend to have enough spare resources to accept new requests. This observation shows that when the proposed approach is not able to provide benefits, the performance of the system does not degrade significantly, but rather produces similar results to the traditional approach.

In our target environments, such as IoT systems that process huge amounts of data, the probability of nearby nodes accepting new application requests is rather low, as
discussed in Section 4.3.2. For this reason, we consider the results with low initial acceptance probability more relevant for this evaluation. According to these results, in Fig. 4.8a which shows the average latency in small-scale deployments, we note that the context-aware approaches, i.e., the simple context-aware and the CA-QL, both outperform the traditional routing consistently, while the RL-based CA-QL also outperforms the simple context-aware approach. Similar behavior can also be observed in Fig. 4.8b which shows the average latency of medium-scale deployments. Notably, the latency reduction of CA-QL compared to the traditional approach (in Fig. 4.8b), reaches 19.5%. Based on the latency values of the nodes in Europe which are used in this experiment, this reduction approximates the maximum possible reduction in the average latency (shown in Table 4.1). This indicates that the results of Table 4.1, which are produced assuming that the proposed context-aware mechanism is able to select the most appropriate compute node, are actually achievable when implementing predictive methods (such as the CA-QL) in the data analysis component.

Finally, for Fig. 4.8c which shows the average latency of large-scale deployments, we note that the context-aware approaches show benefits for a range of small probabilities, but when the probability increases, the results are mixed. This happens due to the high latency to reach the remote compute nodes on another continent. When most of the application requests are accepted by the final cloud node (i.e., small acceptance probabilities), the traditional approach has the highest latency due to the forwarding by the nodes on path. In this case, the context-aware approaches provide the benefits shown at the left-hand side of Fig. 4.8c until the intersection point (at approximately 4% probability), due to sending some of the application requests directly to the final cloud node. Notably, when an application request does not reach the cloud, but is accepted by a remote compute node (i.e., on another continent), the context-aware mechanism favors that remote node for the subsequent request even though the latency may be high. The traditional routing on the other hand, does not favor any nodes, which means that the subsequent request is sent on a path and might be accepted by a nearby node. For this reason, when there are compute nodes in different continents, and the acceptance probability increases, traditional routing tends to have lower latency than the context-aware approach. Hence, after the intersection point (in Fig. 4.8c), traditional routing performs better.

Notably, similar behavior may also occur before the intersection point. However, the benefits from sending some of the application requests to the cloud directly, which are achieved by the context-aware approaches, are large enough to overshadow the shortcomings. This is also the reason that our prototype shows the benefits of Fig. 4.7b using 7% probabilities. The results of Fig. 4.8c are not completely in alignment with Fig. 4.7b because the latency between the nodes in our simulations is based on Table 4.1 which has slightly different values than the values of Fig. 4.7b (due to using actual Internet measurements and external IP addresses, as discussed in Section 4.4.2). Nevertheless, while Fig. 4.7b shows that our prototype can provide benefits in a large-scale deployment, Fig. 4.8c shows that there is actually a range of small probabilities for which the context-
aware approaches can provide benefits. Outside this range, our simulations show that the traditional approach is slightly better than the simple context-aware, while CA-QL exhibits high variance, but does not perform better than the traditional.

Discussion: In general, we observe that the exact latency achieved by each routing approach is a function of the specific environment, i.e., the latencies in the underlying network, and the node acceptance probabilities. Nevertheless, we note that for low acceptance probabilities (which is our target environment), the context-aware approaches consistently outperform the traditional routing in all our experiments (both prototype- and simulation-based). This happens because the traditional approach takes time to send the application requests on a path of compute nodes towards the cloud, until a node eventually accepts it. Simple context-aware routing, on the other hand, takes advantage of the awareness of the last node that accepted the request, which may be a node farther along the path, and bypasses all previous nodes, thereby saving time. CA-QL performs even better than simple context-aware routing in most settings we examine, with performance gains that are more pronounced in the small- and medium-scale deployment scenarios. This occurs because CA-QL learns to direct application requests to cloud nodes when other intermediate compute nodes are likely to be busy. Notably, it can be observed that CA-QL strikes a good balance across the range of parameters explored: When it pays off to bypass intermediate nodes, it usually performs better than the alternatives, while as soon as intermediate nodes become more eager to accept requests, its performance converges to that of traditional routing.

When the compute node acceptance probabilities are low, the gains of CA-QL over the traditional approach in terms of latency reach up to 31% and 19.5%, for the small- and medium-scale deployment scenarios, respectively. Even though in the large-scale scenario, we observe a range of parameters for which the simple context-aware and the traditional routing perform better, we note that CA-QL has more tangible benefits overall. Notably, in our experiments we assume that the exact behavior of the compute nodes, is not known in advance (e.g., through monitoring the available resources). Therefore, even if it may not be entirely possible to create a mechanism that always selects the closest compute node with available resources, we consider that CA-QL offers a very good compromise.

Memory Consumption

An important aspect of CA-QL is the amount of maintained state. As discussed in Section 4.3.3, for every stored context, an agent needs $O(N^{k+1})$ space to store state information, where $N$ is the number of nodes and $k$ is the number of the most recent responses used in our state representation (history size). The Q-table has $N^k$ states, and for each state, there are $N$ available actions. The value of $k$ can be selected with space limitations in mind. A large $k$ may make it infeasible to operate the agent, especially on a resource-limited device, such as a Raspberry Pi which could be used as the gateway. Performance-wise, in the small- and medium-scale deployment scenarios, CA-QL is not very sensitive to the choice of $k$. However, in the large-scale scenario, low values of $k$ (e.g., $k = 1$) might result in increased latency. For this reason, larger values may be
more desirable. To appropriately tune the state space size given a specific maximum memory threshold $M$ to be used by an application’s Q-learning agent at the gateway, $k$ can be set to $k \leq \lfloor \log_N (M/c) - 1 \rfloor$, where $c$ is the space necessary per state-action pair. In fact, this is the method we use to tune $k$ in our experiments, with setting $M = 256$ MiB. In general, values for $k$ between 5 and 10 seem to work well without any notable impact on performance. Finally, we should recall that for each different context, the gateway operates a separate Q-learning agent. This is something the system operator needs to keep in mind in order to configure appropriately the amount of memory resources dedicated to each agent process. In our implementation, $c = 8$ bytes: one 32-bit floating-point number to store the current estimate of the Q function value, and a 32-bit integer to store the counter of how many times the action has been selected given that state.

4.5 Summary

In this chapter, we propose a routing mechanism which considers the context of the IoT data in order to route application requests to compute nodes of fog computing systems. To achieve that, a history of previous transmissions is kept, so that new application requests can be sent directly to compute nodes which usually accept requests of the same context. We evaluate this approach using a prototype implementation with distributed nearby and remote compute nodes, and by performing extensive simulations. The results from our prototype show that compared to an alternative method, context-aware routing reduces the latency of sending IoT data to compute nodes of fog computing systems by up to 23%, and lowers the hop count by up to 73% (which indicates lower bandwidth utilization). Furthermore, we show that when using a simple strategy for selecting compute nodes, our approach is able to reduce the latency and the hop count of the transmissions, without inducing significant overhead. In addition, we perform simulations on a wealth of settings, and we show sound benefits when combining the proposed context-aware mechanism with an RL-based predictive method. Based on these results, we deduce that such a routing mechanism can further advance fog computing systems.
In this chapter, we present the third contribution of this thesis, which is based on [114]. Similar to Contribution II, Contribution III is motivated by RQ II, and addresses the problem of routing IoT data to the compute nodes of a fog computing system. However, the routing approach we present in this chapter targets fog computing systems which include a placement algorithm that deploys the applications on the available compute nodes. As a result, the routing approach of this chapter assumes that the applications are already running on distributed compute nodes, and aims at sending the input data from the IoT devices to their corresponding applications in an efficient manner, i.e., with low communication latency. Similar to Contribution II, Contribution III can be applied to a group of nodes organized using the algorithms of Chapter 3, or to any set of distributed compute nodes used for the execution of IoT applications.

To describe Contribution III comprehensively, first we provide an overview in Section 5.1 and a discussion of related work from the literature in Section 5.2. Afterwards, we present the proposed routing approach in Section 5.3 and an extensive evaluation, using a real-world setup of distributed compute nodes, in Section 5.4. Finally, we conclude this chapter with a summary of our findings in Section 5.5.

5.1 Overview

To exploit the available computational resources at the edge of the network in an efficient manner, various approaches have been proposed for sending IoT data to compute nodes in proximity [183]. In many cases, this includes a placement algorithm which considers network-related metrics such as end-to-end propagation delay and bandwidth, to decide which one of the available compute nodes at the edge and in the cloud should process the data [109, 213]. Based on the decisions of such an algorithm, the data is sent to a suitable compute node for processing.
Even though such approaches may reduce the communication latency compared to using only remote cloud nodes, considering end-to-end metrics might hide that different parts of the underlying network may have very different capacities. For instance, when sending data from an IoT device to an edge or cloud compute node of a cloud provider, the bandwidth of the transmission is limited by an Internet provider (e.g., AT&T or Telefonica), and is commonly a few Megabits per second (Mbps) [141, 171]. However, when sending data among edge and cloud compute nodes operated by cloud providers, such low limits may not apply, thereby allowing transmissions with bandwidth that can be orders of magnitude higher [1, 99]. Nevertheless, this bandwidth is not utilized when the data is sent directly from an IoT device to the compute node that performs the processing.

To avoid this, in this chapter we analyze the latency of sending data to edge and cloud compute nodes of cloud providers. Based on this analysis, we propose edgeRouting which routes the data using a detour through the closest edge compute node. By doing that, edgeRouting exploits both the low propagation delay of nodes at the edge, and the high bandwidth among edge and cloud compute nodes of cloud providers. To evaluate this approach, we perform experiments on a real-world setup with nearby and remote compute nodes of a cloud provider, and we show that edgeRouting reduces the communication latency by up to 55% compared to alternative methods.

5.2 Related Work

We identify related work in approaches from the literature which route IoT data to edge and cloud compute nodes of cloud providers. To present such approaches, we divide them into three categories: direct routing, multihop routing, and overlay routing. Specifically, direct routing is discussed in Section 5.2.1, multihop routing is discussed in Section 5.2.2, and overlay routing is presented in Section 5.2.3. Finally, in Section 5.2.4 we provide a discussion of the existing approaches compared to the proposed edgeRouting.

5.2.1 Direct Routing Approaches

Many related approaches from the literature present a placement algorithm that distributes applications on the available compute nodes, and then route the data from the data source to a suitable compute node directly. For this reason, we refer to such approaches as direct routing. For example, Samanta et al. [193] design a framework for executing applications in a system with edge and cloud compute nodes, according to an auction-based placement algorithm. Each compute node in this system is assumed to be addressable from every access point in the network, so that data can be routed directly to the compute node which hosts a specific application. Bellavista et al. [37] present a middleware for fog computing. In this work, each IoT device utilizes a local agent which finds a suitable compute node, and sends data to this node directly. Yao and Ansari [238] propose a system whereby the IoT data is sent directly to suitable compute nodes which use a scheduler for assigning the necessary applications on VMs based on
an optimization problem. Mansouri and Wong \[201\] design a Quality of Experience optimization framework in which the IoT devices send data directly to either a nearby or a remote compute node. Du et al. \[64\] create a decision-maker component which selects a user device, a nearby compute node or a remote node for deploying an application. Then, a user device communicates with the selected node directly.

### 5.2.2 Multihop Routing Approaches

In addition to direct routing, there are other approaches which also implement a placement algorithm, but route the data on a multihop path from the data source to a suitable compute node. We refer to these as *multihop routing*. In such approaches, the data travels on a path of various compute nodes, until a suitable node accepts the data for processing. Notably, the routing path followed by multihop routing is similar to the traditional routing of Chapter 4. However, traditional routing refers to the routing of application requests. In contrast, multihop routing preconditions a placement algorithm, and focuses on the routing of the input data. Therefore, the analysis of the multihop routing in this chapter does not apply to the traditional routing of Chapter 4. Nevertheless, some of the related works of Chapter 4 (i.e., in Section 4.2) can still be considered related to multihop routing, such as the works of Tong et al. \[217\], Ascigil et al. \[23\], and Mortazavi et al. \[160\].

In addition to these approaches, Martinez et al. \[150\] design a computing infrastructure in which the data is routed through various compute nodes based on optimization logic that considers transmission delay and network congestion. Okay and Ozdemir \[174\] propose an approach based on software-defined networks, for processing IoT data on nearby and remote compute nodes. In this approach, there are various distributed controllers which manage the local traffic (and lower the latency) while the data is routed through various nodes to a suitable compute node for processing.

### 5.2.3 Overlay Routing Approaches

There are also related approaches that do not stem from the edge computing literature, but do aim at improving the communication between sender and destination by routing the data through intermediate nodes. For example, Brennan and Rabinovich \[42\] discuss the potential benefits of overlay routing compared to native IP routing, and propose a multipath TCP approach which creates routing detours that improve throughput. Lee et al. \[128\] investigate the utilization of bandwidth as the main metric for creating overlay paths which are shown to have lower latency than native Internet routing. Lumezanu et al. \[142\] present a mechanism for finding mutually beneficial overlay detours for peer nodes, so that the communication latency is reduced. Gummadi and Madhyastha \[84\] explore overlay detours for creating resilient routing paths that are less likely to fail than native Internet paths. Cai et al. \[45\] conduct an experimental study of the communication between clouds, and show improved throughput when using detours.
5.2.4 Discussion

To sum up the discussion of related approaches from the literature, most current edge computing works route the data either directly or on a multihop path, to a suitable compute node for processing. Notably, both of these routing approaches typically consider end-to-end communication, and disregard alternative network paths with detours that may lower the communication latency due to providing higher bandwidth.

The overlay detour approaches on the other hand, which consider alternative network paths, manage to show benefits compared to native Internet routing. However, such approaches have not been designed in the context of edge computing, and therefore, do not take into account edge compute nodes. In addition, most overlay detour approaches stem from the field of overlay networks consisting of peers, and may suffer from privacy concerns related to sharing data among peer nodes [142].

In this chapter, we propose using a detour when routing IoT data to edge and cloud compute nodes operated by cloud providers. We consider such environments as an ideal case for applying detours because: i) The IoT data is not routed through other (peer) users, but rather through the available edge compute nodes which can actually improve privacy [163, 173, 198]. ii) The difference in the capacity of paths with detours can be so large (e.g., due to the different bandwidth limits between Internet and cloud providers), that utilizing such paths may lead to significant benefits (as shown in Section 5.4).

5.3 edgeRouting

To present the proposed approach, first we provide a discussion of the utilized system model in Section 5.3.1. In Section 5.3.2, we analyze the induced latency of sending IoT data to nearby and remote compute nodes based on existing routing approaches, and we introduce edgeRouting.

5.3.1 System Model

For the utilized system model, in the following we describe the architecture of the target systems for processing IoT data using edge and cloud compute nodes. After that, we discuss the typical behavior of various system parameters that affect the communication latency in such systems. Notably, in the target systems of this chapter, the data is routed on a path of compute nodes from the edge to the cloud, similar to the system model of Chapter 4 which is discussed in Section 4.3.1. Such a path can be formed in either hierarchical or flat structures using, e.g., the algorithms of Chapter 3.

System Architecture

Our system includes distributed compute nodes in data centers and at the edge of the network, which can be used for processing data from IoT devices [204]. Such nodes can be provisioned on-demand by cloud providers. For example, Fig. 5.1 shows the regions...
with (existing or announced) computational resources from two cloud providers: Google and Microsoft [3, 5]. Notably, these regions, i.e., from two cloud providers alone, may provide compute nodes in the proximity of IoT devices in various areas around the world (e.g., in Europe, India, eastern Asia, Australia, and the US). Thus, based on Fig. 5.1 and by considering that other commercial cloud providers may support additional regions, we assume that compute nodes from cloud providers can be provisioned both far away, and close to IoT devices [91].

In such environments, we consider that application providers deploy their applications on compute nodes in order to process data from IoT devices. These applications can be, e.g., for detecting anomalies in the energy consumption (such as gas leaks) using smart meters, for monitoring the appearance of house visitors using smart doorbells (which are also discussed in Chapter 3), etc. [178]. For such cases, utilizing a single compute node with enough computational resources to process the data from all the IoT devices (i.e., a centralized cloud node) may result in high communication latency for data coming from regions outside the proximity of this node [109, 208]. Utilizing multiple compute nodes in the proximity of all the IoT devices, each one with enough resources to process all the data, lowers the latency, but may increase the monetary cost due to maintaining many nodes with high computational resources. Thus, utilizing few compute nodes with high computational resources, and many compute nodes with lower resources but distributed around the regions of the IoT devices, can provide a balance between delay and cost [166].

Therefore, in this chapter we target environments with a system architecture that includes multiple geographically distributed edge and cloud compute nodes and IoT devices, as shown in Fig. 5.2. The IoT devices in such systems represent devices with sensors, which act as data sources. The edge and cloud compute nodes can be created on-demand in any region which is supported by a cloud provider (e.g., Google, Microsoft, Amazon, Oracle, IBM, Alibaba, etc). Edge compute nodes offered by cloud providers are similar to cloud compute nodes in that they are both able to provide the same services [6]. However, since edge compute nodes reside in the proximity of the data sources, the communication
Figure 5.2: The target system architecture with distributed edge and cloud compute nodes and IoT devices. The links show the communication between compute nodes (facilitated by cloud providers), and between IoT devices and compute nodes (facilitated by Internet providers).

Communication Latency

The latency of sending data to a compute node can be considered as the sum of the following delays: propagation delay noted as \( \text{Pro} \) (calculated as \( \frac{\text{linkDistance}}{\text{dataSpeed}} \)), transmission delay noted as \( \text{Tra} \) (calculated as \( \frac{\text{dataSize}}{\text{bandwidth}} \)), queuing delay noted as \( \text{Que} \) (amount of time the data wait in queues, e.g., of network devices) \[123\], and processing delay (time needed to process the headers) \[131\] \[244\]. The processing delay is
5.3. edgeRouting

Table 5.1: The utilized notation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s = {n_1, n_2, \ldots, n_N}$</td>
<td>Set including all the nodes of the system.</td>
</tr>
<tr>
<td>$n_1$</td>
<td>The IoT device (i.e., a data source).</td>
</tr>
<tr>
<td>$n_2$</td>
<td>Edge compute node closest to $n_1$.</td>
</tr>
<tr>
<td>$n_X$</td>
<td>Compute node that processes the data.</td>
</tr>
<tr>
<td>$Pro_{n_i,n_j}$</td>
<td>Propagation delay from $n_i$ to $n_j$.</td>
</tr>
<tr>
<td>$Tra_{n_i,n_j}$</td>
<td>Transmission delay from $n_i$ to $n_j$.</td>
</tr>
<tr>
<td>$Que_{n_i,n_j}$</td>
<td>Queuing delay from $n_i$ to $n_j$.</td>
</tr>
<tr>
<td>$DLat_{n_1,n_X}$</td>
<td>Direct routing latency from $n_1$ to $n_X$.</td>
</tr>
<tr>
<td>$MLat_{n_1,n_X}$</td>
<td>Multihop routing latency from $n_1$ to $n_X$.</td>
</tr>
<tr>
<td>$ELat_{n_1,n_X}$</td>
<td>edgeRouting latency from $n_1$ to $n_X$.</td>
</tr>
</tbody>
</table>

These delays apply when sending a packet over a link, but for simplicity, we consider that similar delays apply when sending IoT data between nodes. Thus, all the factors that contribute to the communication latency are considered by taking into account the aforementioned delays. Notably, each one of these delays consists of all the time periods that are required for sending the data between various nodes in the underlying network, until the data reach the destination compute node. This means that, e.g., the $Pro$ of sending data from an IoT device to a cloud compute node includes all the propagation delays of the logical link that connects sender and destination. The same applies for $Tra$, and $Que$.

Such delays apply regardless of the utilized communication protocol, e.g., TCP, User Datagram Protocol (UDP), HTTP, etc. Each protocol may have a different message structure which can affect the discussed delays, e.g., an HTTP message may be larger than a UDP message with the same payload, which affects the $Tra$. Similarly, the specific characteristics of the Internet connection, and the underlying network, e.g., cable, or 4G, can affect these delays. However, when the same communication protocol is used over the same Internet connection, we consider that measuring the communication latency in this manner provides a good approximation for comparing different routing approaches in systems with edge and cloud compute nodes.

The nodes in our model are represented by a set $s = \{n_1, n_2, \ldots, n_N\}$ which includes $n_1$ being the IoT device, $n_2$ being the edge compute node in closest proximity of $n_1$, and all the other compute nodes of the system. Thus, to refer to, e.g., the $Pro$ between an IoT device and the compute node in closest proximity, we use $Pro_{n_1,n_2}$ (similarly: $Tra_{n_1,n_2}$, $Que_{n_1,n_2}$, etc.). $n_1$ may send data to any of the other nodes of $s$ for processing. The destination compute node is denoted as $n_X$. To make our system model more comprehensible, we summarize the utilized notation in Table 5.1.

As discussed above in the system architecture, this work targets systems with edge and cloud compute nodes offered by cloud providers, which are used for processing IoT data...
over the Internet. Based on the literature of similar systems, we deduce the following formulations regarding the typical behavior of the communication latency when sending data to nearby and remote compute nodes [90, 100].

- **Pro** is defined as \( \frac{\text{linkDistance}}{\text{dataSpeed}} \), which means that this delay is affected by the speed that the data travels on the network links, and the traveled distance between sender and destination. Since the speed can be considered a constant (which is usually around \( 2 \cdot 10^8 \) meters per second based on the signal speed on copper cables) [70]. **Pro** depends on the distance of the network path that connects sender and destination. Based on this definition, we can also say that since the denominator is a very high constant number, **Pro** approaches zero, when the numerator becomes small. Thus, when \( n_1 \) is very close to \( n_2 \), \( \text{Pro}_{n_1,n_2} \) approaches a very small number which does not contribute considerably to the communication latency, and may therefore be considered insignificant [90]. To denote such quantity which does not contribute significantly to the latency –but is nevertheless never zero– we use \( \varepsilon \) with \( \varepsilon \approx 0 \) and \( \varepsilon > 0 \). Hence:

\[
\text{Pro}_{n_1,n_2} \approx \varepsilon \tag{5.1}
\]

- **Tra** is defined as \( \frac{\text{dataSize}}{\text{bandwidth}} \), which means that for the same data size, this delay depends on the available bandwidth. We can also say that when the bandwidth becomes abundant, **Tra** approximates zero. Since edge compute nodes of cloud providers belong to the same network of the provider’s data centers, the bandwidth limit is the same for both edge and cloud compute nodes [6]. This limit may vary according to each cloud provider and the current network load, but is usually very high. For example, Google allows seven Gigabits per second (Gbps) of egress bandwidth to external IP addresses, while the ingress bandwidth can be more or even unlimited [1]. Microsoft also allows unlimited ingress/egress bandwidth [4]. Notably, even though the bandwidth may be abundant, a compute node needs to have sufficient processing resources (e.g., CPU and RAM) to process the ingress/egress data. In addition, the utilized bandwidth may be subject to a pricing model. When such restrictions do not pose significant concerns, the compute nodes of cloud providers are able to utilize as much network bandwidth as they need. Therefore, **Tra** does not contribute considerably to the communication latency, and can be considered insignificant, when the sender and the destination are both compute nodes which communicate with very high bandwidth [93]. Hence applies that:

\[
\forall n_i, n_j \in s - \{n_1\}, \quad \text{Tra}_{n_i,n_j} \approx \varepsilon \tag{5.2}
\]

- The queuing delay between compute nodes of cloud providers can also be considered insignificant in the calculation of the communication latency. This may happen when there is abundant bandwidth, as previously discussed, that makes it unlikely for the data to have to wait in queues [100]. Hence, **Que** does not contribute considerably
to the communication latency when sender and destination are compute nodes of cloud providers, i.e.:

\[
\forall n_i, n_j \in s - \{n_1\}, \quad Que_{n_i,n_j} \approx \varepsilon
\]  

(5.3)

We use Equations (5.1), (5.2), and (5.3) for comparing the communication latency of different routing approaches in Section 5.3.2. After that, we revisit these equations in Section 5.4.4, and we discuss them from a practical point of view based on our experiments, also considering cases when these approximations deviate from zero.

The overall communication delay of executing an application on a compute node of a cloud provider always includes the delay of sending the data from the sender to the destination node, i.e., the upload latency. In case the application produces a response that needs to be sent back to the sender, the overall communication delay also includes the download latency. For simplicity, in this work we focus on reducing the communication latency of sending the data, i.e., the upload latency, because this latency constitutes the minimum delay to start executing an application. Nevertheless, in our evaluation in Section 5.4.3, we also discuss how the download latency is affected by the different routing approaches.

To further elaborate on how the communication latency of sending data to distributed compute nodes is modeled in this chapter, we depict a flowchart in Fig. 5.3. Initially, the input data is generated at \( n_1 \) and transmitted to a target node \( n \in s - \{n_1\} \). This transmission is subject to propagation, transmission, and queuing delays, until the data arrives at the target node. Upon arrival, this node decides to either accept the data for processing, or to forward the data to a new target node. The former results in the end of the transmission which means that there is no more communication latency. The latter triggers a loop that results in another transmission until the data is accepted by a node. Thus, the communication latency of sending data to distributed compute nodes is considered as the period from the time the data is sent from \( n_1 \) until the data is accepted by a node.

5.3.2 Sending IoT Data to Compute Nodes

In this section, we examine the latency of sending IoT data to edge and cloud compute nodes, based on different routing approaches. To this end, first we analyze the latency of common routing approaches from the literature (discussed in Section 5.2). Afterwards, we take into account this analysis in order to design edgeRouting which is a routing approach that lowers the latency by utilizing the underlying network resources more efficiently.

Direct Routing

As discussed in Section 5.2, direct routing can be used for sending IoT data to edge and cloud compute nodes. According to this approach, the IoT devices are configured to send
5. edgeRouting

![Flowchart with the communication latency of sending data to a compute node.](image)

Figure 5.3: Flowchart with the communication latency of sending data to a compute node.

the data directly to the compute nodes which perform the required processing [193]. In the flowchart of Fig. 5.3, this means that when using direct routing the data is accepted at the target node after one transmission. Thus, the target node accepts the data the first time, and the loop that starts when a node does not accept the data (in Fig. 5.3) is not triggered. To facilitate this, a placement algorithm is utilized for placing the applications on suitable compute nodes based on various resource-related (e.g., CPU and RAM) and/or network-related (e.g., transmission and propagation delay) metrics [67]. After the applications are placed, the data is commonly routed from an IoT device (through a local gateway [101]) to a suitable compute node directly, using the connection of an Internet provider, as shown in Fig. 5.4a.

Thus, assuming that data is sent from an IoT device $n_1$ to a suitable compute node $n_X$ with $n_X \in s - \{n_1\}$, the communication latency using direct routing $DLat_{n_1,n_X}$ can be defined as the sum of the propagation, transmission, and queuing delays, i.e.:

$$DLat_{n_1,n_X} = Pron_{n_1,n_X} + Tran_{n_1,n_X} + Que_{n_1,n_X}$$  \hspace{1cm} (5.4)

$Pron_{n_1,n_X}$ depends on the speed that the data travels on the network links, and the distance between $n_1$ and $n_X$. Since the speed can be considered a constant (as discussed in Section 5.3.1), this delay is affected by the distance of the network path that connects $n_1$ and $n_X$. This path is usually selected based on a gateway protocol that implements a best path algorithm [14].
5.3. edgeRouting

Figure 5.4: Routing approaches for sending data to edge and cloud compute nodes. The links show the provider that facilitates the communication (either cloud provider or Internet provider).

$T_{ra_{n_1,n_X}}$ depends on the upload bandwidth limit of the Internet provider, and the data size. The data size is a variable which depends on the specific application. For example, applications that perform analytics based on sensor measurements may utilize smaller data sizes than image processing applications.

$Q_{eu_{n_1,n_X}}$ depends on the time that the data waits in queues, e.g., the buffers of the network devices that perform the routing. Notably, if the buffers are full, e.g., due to congestion (which may occur in IoT environments), packets are dropped and need to be retransmitted. This also increases the communication latency [16].

Therefore, while the propagation delay is not affected by the bandwidth limit of the Internet provider, the transmission delay can increase when the network bandwidth is low. Moreover, low bandwidth may result in the data waiting in queues longer, which might lead to congestions, dropped packets, and retransmissions [16]. Thus, the bandwidth of the Internet provider plays a critical role in the communication latency of direct routing.

Multihop Routing

Similar to direct routing, in multihop routing there is also a placement algorithm which distributes the applications on the compute nodes (as discussed in Section 5.2). However, the flow of the data is different. In multihop routing, the data is sent from an IoT device...
n₁ to an edge compute node n₂. If this node hosts the required application, and has adequate available computational resources to perform the required processing, the data is accepted. Otherwise, the data is rerouted to the next node in proximity \textsuperscript{217}. Similarly, that node either accepts or reroutes the data, and the process repeats. This process is bound to finish, because usually the last node on a multihop path is a cloud compute node which is assumed to always be able to perform the required processing \textsuperscript{112}.

In the flowchart of Fig. 5.3 when using multihop routing the data is accepted by the target node after one or more transmissions (one transmission if \( n_X = n_2 \), more otherwise). This means that the loop that starts when a node does not accept the data (in Fig. 5.3) may be triggered multiple times, thereby initiating more transmissions and inducing more delay. These transmissions are facilitated by different providers, and may therefore have different bandwidth limits. As shown in Fig 5.4b, the data is sent from the IoT device to the compute node in proximity using the connection of an Internet provider. Assuming that this node is not able to perform the required processing, the data is routed again to another compute node. This time however, since the is data is routed from a compute node, the transmission is facilitated by a cloud provider, i.e., with higher bandwidth.

Thus, when sending data from an IoT device \( n_1 \) to a suitable compute node \( n_X \) using multihop routing, the communication latency \( MLat_{n_1,n_X} \) consists of the propagation, transmission, and queuing delays, between each sender and destination until the data reaches \( n_X \), i.e.:

\[
MLat_{n_1,n_X} = Pro_{n_1,n_2} + Tra_{n_1,n_2} + Que_{n_1,n_2} + \\
Pro_{n_2,n_3} + Tra_{n_2,n_3} + Que_{n_2,n_3} + \ldots + \\
Pro_{n_{X-1},n_X} + Tra_{n_{X-1},n_X} + Que_{n_{X-1},n_X} \quad (5.5)
\]

The propagation delay \( Pro_{n_1,n_X} \) in multihop routing is the sum of the propagation delay of each transmission, i.e., \( Pro_{n_1,n_2} + Pro_{n_2,n_3} + \ldots + Pro_{n_{X-1},n_X} \). Thus, the routing path from \( n_1 \) to \( n_X \) includes detours (i.e., \( n_2, n_3, \ldots, n_{X-1} \)), and is not based on a best path algorithm (unless \( n_2, n_3, \ldots, n_{X-1} \) exist within the best path from \( n_1 \) to \( n_X \)). Therefore, the traveled distance of the data in multihop routing is expected to be longer than a direct transmission from \( n_1 \) to \( n_X \), which means that the propagation delay is increased.

The transmission delay \( Tra_{n_1,n_X} \) consists of the transmission delay from \( n_1 \) to \( n_2 \), and the transmission delay between compute nodes of cloud providers. The former can be increased by the bandwidth limit imposed by the Internet provider. The latter however, can be considered rather insignificant based on Equation (5.2).

The queuing delay \( Que_{n_1,n_X} \) consists of the queuing delay from \( n_1 \) to \( n_2 \), and the queuing delays between compute nodes. \( Que_{n_1,n_2} \) can be affected by low bandwidth and congestion, but this is unlikely because nodes that reside close to each other do not typically suffer from low bandwidth \textsuperscript{112, 195}, although the limit of the Internet provider still applies. The queuing delays between compute nodes are expected to be insignificant based on Equation (5.3).
5.3. edgeRouting

Based on the analysis of direct and multihop routing, we note that while direct routing has low propagation delay, the transmission and queuing delays may increase due to the bandwidth limit of the Internet provider. Multihop routing on the other hand, may have low transmission and queuing delays due to the high bandwidth between compute nodes, but the propagation delay may increase due to the detours. Thus, in order to leverage both the low propagation delay of nodes in proximity, and the high bandwidth between compute nodes, we propose edgeRouting.

According to edgeRouting, the data is routed from an IoT device $n_1$ to the edge compute node in closest proximity $n_2$, and from there, the data is routed directly to the suitable compute node $n_X$ for processing, as shown in Fig. 5.4c. In the flowchart of Fig. 5.3, when using edgeRouting the data is accepted by the target node after one or two transmissions (one transmission if $n_X = n_2$, two otherwise). This means that the loop that starts when a node does not accept the data (in Fig. 5.3) is either not executed (when $n_X = n_2$), or executed one time (when $n_X \neq n_2$) which triggers a second transmission directly to $n_X$.

The first transmission is routed by the Internet provider, and the second by the cloud provider. This way, during the first routing the data is sent with low propagation delay, and during the second routing, the data is sent with low transmission and queuing delays due to the high bandwidth.

Notably, direct routing commonly preconditions that there is a placement algorithm which distributes the applications on the compute nodes, and that the IoT devices (or the gateways) know the IP address of $n_X$ in order to initiate the transmission of the data (as discussed in the analysis of direct routing). Similarly, multihop routing usually preconditions that there is a placement algorithm, and that the IoT devices know the IP address of $n_2$ for initiating the transmission (as discussed in the analysis of multihop routing). edgeRouting preconditions a placement algorithm, that the IP address of $n_2$ is known, and that the IP address of $n_X$ is known. This enables sending the data to $n_2$ along with the IP address of $n_X$ which is used by $n_2$ for the second routing.

The communication latency in edgeRouting $ELat_{n_1,n_X}$ consists of the propagation, queuing, and transmission delays to send the data from $n_1$ to $n_2$, and from $n_2$ to $n_X$, i.e.:

$$ELat_{n_1,n_X} = Pro_{n_1,n_2} + Tra_{n_1,n_2} + Que_{n_1,n_2} + Pro_{n_2,n_X} + Tra_{n_2,n_X} + Que_{n_2,n_X}$$

The propagation delay $Pro_{n_1,n_2}$ can be insignificant based on Equation (5.1). $Pro_{n_2,n_X}$ depends on the distance between $n_2$ and $n_X$. The transmission delay includes $Tra_{n_1,n_2}$ which can be increased by the bandwidth limit of the Internet provider, and $Tra_{n_2,n_X}$ which can be considered insignificant based on Equation (5.2). The queuing delay contains $Que_{n_1,n_2}$ which is unlikely to be affected by congestion (as discussed in the analysis of multihop routing), and $Que_{n_2,n_X}$ which can be considered insignificant based on Equation (5.3). Regarding the comparison of edgeRouting with direct and multihop
routing, we deduce that edgeRouting is able to provide similar or lower communication latency based on Lemmata 1 and 2:

**Lemma 1.** The communication latency of edgeRouting is similar or lower than the communication latency of direct routing, i.e.: Equation (5.6) \( \leq \) Equation (5.4).

**Proof:** When the compute node in closest proximity performs the required processing, i.e., \( n_X = n_2 \), for Equation (5.6) applies that: \( Pro_{n_2,n_X} + Tra_{n_2,n_X} + Que_{n_2,n_X} = 0 \) because this part of the equation represents the communication latency to send the data from \( n_2 \) to \( n_X \), which does not incur any latency. Thus, Equation (5.6) = \( Pro_{n_1,n_2} + Tra_{n_1,n_2} + Que_{n_1,n_2} + 0 = Equation (5.4) \).

When \( n_X \neq n_2 \), the propagation delay of edgeRouting according to Equation (5.6), i.e., \( Pro_{n_1,n_2} + Pro_{n_2,n_X} \), is approximately equal to \( Pro_{n_1,n_X} \), because \( Pro_{n_1,n_2} \) can be considered insignificant based on Equation (5.1). \( Pro_{n_2,n_X} \) can be considered approximately equal to the propagation delay of direct routing from Equation (5.4), i.e., \( Pro_{n_1,n_X} \), because \( n_1 \) and \( n_2 \) reside in proximity, which means that the distance from \( n_1 \) to \( n_X \) is similar to the distance from \( n_2 \) to \( n_X \). It is possible that these two transmissions take different paths to \( n_X \), which may have slightly different path distances. However, the very high data speed of the transmissions (discussed in Section 5.3.1) makes Pro resilient to small distance changes. Thus, even in such cases, Pro does not change significantly. Consequently, since \( Pro_{n_2,n_X} \approx Pro_{n_1,n_X} \), and \( Pro_{n_1,n_2} \approx \varepsilon \), the propagation delay of edgeRouting is approximately equal to the propagation delay of direct routing, i.e.:\[
Pro_{n_1,n_2} + Pro_{n_2,n_X} \approx Pro_{n_1,n_X}
\] (5.7)

The transmission delay of direct routing from Equation (5.4), i.e., \( Tra_{n_1,n_X} \), depends on the bandwidth limit of the Internet provider. Notably, since the bandwidth may be shared among many users, the effective bandwidth of \( Tra_{n_1,n_X} \) can be lower than the limit, based on the overall network traffic which depends on the current network load [146]. This phenomenon can cause low bandwidth in \( Tra_{n_1,n_X} \), especially when \( n_X \) is a remote cloud compute node [171]. \( Tra_{n_1,n_2} \) is also facilitated by the Internet provider, and can be affected by the current network load, too. However, since \( n_1 \) and \( n_2 \) reside close to each other, \( Tra_{n_1,n_2} \) is unlikely to suffer from low bandwidth [172] [195]. Thus, we consider that \( Tra_{n_1,n_2} \leq Tra_{n_1,n_X} \). The equality stands when the network is not/equally loaded in both transmission delays, which means that similar bandwidth is utilized. The inequality applies when the network is loaded due to \( n_X \) being a remote node, and \( Tra_{n_1,n_X} \) has less bandwidth than \( Tra_{n_1,n_2} \). The case that the network is loaded only at the edge, i.e., between \( n_1 \) and \( n_2 \), without being loaded between \( n_1 \) and \( n_X \) is unlikely because based on our system model, \( n_X \) can either equal \( n_2 \) or be farther away. The transmission delay of edgeRouting according to Equation (5.6), includes \( Tra_{n_1,n_2} \) and \( Tra_{n_2,n_X} \). \( Tra_{n_2,n_X} \) is considered insignificant based on Equation (5.2). Thus, \( Tra_{n_2,n_X} \approx \varepsilon \) and \( Tra_{n_1,n_2} \leq Tra_{n_1,n_X} \), which means that the transmission delay of edgeRouting is similar or lower than the transmission delay of direct routing, i.e.:\[
Tra_{n_1,n_2} + Tra_{n_2,n_X} \leq Tra_{n_1,n_X} 
\] (5.8)
The queuing delay of edgeRouting according to Equation (5.6) consists of $Que_{n_1,n_2}$ which is unlikely to suffer from congestion, and $Que_{n_2,n_X}$ which can be considered insignificant (both are discussed in the analysis of multihop routing). The queuing delay of direct routing from Equation (5.4), i.e., $Que_{n_1,n_X}$, may be increased due to congestion in the routing path from $n_1$ to $n_X$ (as discussed in the analysis of direct routing). Thus, we assume that $Que_{n_1,n_2} \leq Que_{n_1,n_X}$. The equality applies if there is no congestion, whereas the inequality applies if there is congestion in the path from $n_1$ to $n_X$. Therefore, since $Que_{n_2,n_X} \approx \varepsilon$, and $Que_{n_1,n_2} \leq Que_{n_1,n_X}$, the queuing delay of edgeRouting is similar or lower than the queuing delay of direct routing, i.e.:

$$Que_{n_1,n_2} + Que_{n_2,n_X} \lesssim Que_{n_1,n_X} \quad (5.9)$$

By adding Inequalities (5.7), (5.8), and (5.9), applies that:

$$Pro_{n_1,n_2} + Pro_{n_2,n_X} + Tra_{n_1,n_2} + Tra_{n_2,n_X} +$$
$$Que_{n_1,n_2} + Que_{n_2,n_X} \lesssim Pro_{n_1,n_X} + Tra_{n_1,n_X} +$$
$$Que_{n_1,n_X} \Leftrightarrow$$

$$Equation (5.6) \lesssim Equation (5.4) \quad (5.10)$$

Hence, we note that the communication latency of edgeRouting is expected to be lower than direct routing because the transmission and queuing delays are likely to be lower, while the propagation delay is likely to be similar. This applies because edgeRouting enables the IoT devices to avoid direct communication with remote compute nodes, thereby avoiding network paths with potential congestion and low bandwidth.

**Lemma 2.** The communication latency of edgeRouting is similar or lower than the communication latency of multihop routing, i.e.: $Equation (5.6) \lesssim Equation (5.5)$.  

**Proof:** When the compute node in closest proximity performs the required processing, i.e., $n_X = n_2$, Equation (5.6) = $Pro_{n_1,n_2} + Tra_{n_1,n_2} + Que_{n_1,n_2} = Equation (5.5)$. Similarly, when $n_X = n_3$, Equation (5.6) = $Pro_{n_1,n_3} + Pro_{n_2,n_3} + Tra_{n_1,n_2} + Tra_{n_2,n_3} + Que_{n_1,n_2} + Que_{n_2,n_3} = Equation (5.5)$. When $n_X \neq n_2$ and $n_X \neq n_3$, both the propagation delay of edgeRouting according to Equation (5.6), and the propagation delay of multihop routing according to Equation (5.5) include $Pro_{n_1,n_2}$. Apart from that, edgeRouting also includes $Pro_{n_2,n_X}$, whereas multihop routing includes $Pro_{n_2,n_3} + \ldots + Pro_{n_{X-1},n_X}$. For these delays applies that the former is lower or equal, i.e., that $Pro_{n_2,n_X} \leq Pro_{n_2,n_3} + \ldots + Pro_{n_{X-1},n_X}$. The equality applies if $n_3, \ldots, n_{X-1}$ exist on the path from $n_2$ to $n_X$, i.e., the same path that is followed in $Pro_{n_2,n_X}$. The inequality applies when $n_3, \ldots, n_{X-1}$ are detours, because detours increase the distance from $n_2$ to $n_X$ and consequently, the propagation delay (as discussed in the analysis of multihop routing). Thus, by adding $Pro_{n_1,n_2}$ to both sides of the aforementioned inequality, applies that the propagation delay of edgeRouting is equal or lower than the propagation delay of multihop routing, i.e.:

$$Pro_{n_1,n_2} + Pro_{n_2,n_X} \leq Pro_{n_1,n_2} + \ldots + Pro_{n_{X-1},n_X} \quad (5.11)$$
The transmission delay of edgeRouting according to Equation (5.6) includes $T_{ra_{1,n_2}}$ which is also included in the transmission delay of multihop routing according to Equation (5.5). Except for that, edgeRouting includes $T_{ra_{n_2,n_X}}$, while multihop routing includes $T_{ra_{n_2,n_3} + \ldots + r_{a_{n_X-1,n_X}}}$. These are all transmission delays between compute nodes, which can be considered insignificant based on Equation (5.2). Thus, $T_{ra_{n_2,n_X}} \approx T_{ra_{n_2,n_3} + \ldots + r_{a_{n_X-1,n_X}}}$. By adding $T_{ra_{n_1,n_2}}$ to both sides, applies that the transmission delay of edgeRouting is similar to the transmission delay of multihop routing, i.e.:

$$T_{ra_{n_1,n_2}} + T_{ra_{n_2,n_X}} \approx T_{ra_{n_1,n_2} + \ldots + r_{a_{n_X-1,n_X}}} \quad (5.12)$$

The queuing delay of edgeRouting in Equation (5.6), and the queuing delay of multihop routing in Equation (5.5) both include queuing delays between compute nodes, which can be considered insignificant based on Equation (5.3). Thus, $Q_{ue_{n_2,n_X}} \approx Q_{ue_{n_2,n_3} + \ldots + r_{ue_{n_X-1,n_X}}}$. By adding $Q_{ue_{n_1,n_2}}$ to both sides, applies that the queuing delay of edgeRouting is similar to the queuing delay of multihop routing, i.e.:

$$Q_{ue_{n_1,n_2}} + Q_{ue_{n_2,n_X}} \approx Q_{ue_{n_1,n_2} + \ldots + r_{ue_{n_X-1,n_X}}} \quad (5.13)$$

By adding Inequalities (5.11), (5.12), and (5.13), applies that:

$$P_{ro_{n_1,n_2}} + P_{ro_{n_2,n_X}} + T_{ra_{n_1,n_2}} + T_{ra_{n_2,n_X}} + Q_{ue_{n_1,n_2}} + Q_{ue_{n_2,n_X}} \lesssim P_{ro_{n_1,n_2} + \ldots + r_{ro_{n_X-1,n_X}}} + Q_{ue_{n_1,n_2} + \ldots + r_{ue_{n_X-1,n_X}} \Leftrightarrow}$$

$$\text{Equation (5.6)} \lesssim \text{Equation (5.5)} \quad (5.14)$$

Hence, we note that the communication latency of edgeRouting is expected to be lower than multihop routing because the propagation delay is likely to be lower, while the transmission and queuing delays are likely to be similar. This applies because the routing path of multihop routing may include detours which increase the traveled distance of the data, and consequently the communication latency.

### 5.4 Evaluation

To evaluate edgeRouting, we implement a prototype for each routing approach, i.e., the proposed edgeRouting, and the direct and multihop routing. The latter two represent alternative approaches (as discussed in Section 5.2) which we use as baselines. Furthermore, we run experiments and we measure the incurred communication latency. To present our results in a comprehensive manner, first we describe the details of the evaluation environment in Section 5.4.1 and then we discuss the actual quantitative results in Section 5.4.2. Subsequently, we provide a discussion of the examined approaches based on other aspects than the communication latency in Section 5.4.3. Finally in Section 5.4.4, we elaborate on the weaknesses and limitations of our approach.
5.4. Evaluation

5.4.1 Evaluation Environment

In this section, we describe in detail the evaluation setup (including the location of the compute nodes, Internet connections, etc.), the approaches we use as baselines, and the evaluation experiments (including target use cases and input data).

Evaluation Setup

To create a system as shown in Fig. 5.2, we employ a Raspberry Pi 4 as the IoT device, and distributed compute nodes provisioned using the GCP. We position the IoT device in central Europe (in Vienna, Austria), and we provision compute nodes in the currently available regions of the GCP, i.e., Zurich (in Switzerland), Frankfurt (in Germany), Saint Ghislain (in Belgium), Eemshaven (in the Netherlands), and London (in England). We consider this to be an appropriate setup for examining the communication latency of sending IoT data to distributed compute nodes, because there are already many experimental IoT applications running in this area (e.g., in France, Spain, Serbia, Germany, and the United Kingdom) [52].

Even though edge compute nodes have been announced by cloud providers (e.g., the Edge Zones by Microsoft), their availability is still limited [6]. For this reason, we emulate the edge compute node $n_2$ using the compute node in closest proximity to $n_1$ (i.e., the compute node in Zurich). This way, our emulated edge compute node can be reached with the lowest propagation delay, while providing high bandwidth when sending data to other compute nodes (the same as the edge compute nodes discussed in our system model in Section 5.3.1). However, our edge compute node is not very close to the IoT device, e.g., in the same city, but rather in a nearby city which is approximately 600 kilometers away. This means that the propagation delay between $n_1$ and $n_2$ can be about 3 ms based on the propagation delay definition discussed in Section 5.3.1. In fact, this delay might be a bit higher because the network distance between $n_1$ and $n_2$ can be larger than the physical distance. Consequently, the propagation delay using the emulated edge compute node might not be extremely small as discussed in Section 5.3.1 but it is still small enough not to affect the presented results significantly. This is discussed further in Section 5.4.3 in which we analyze the error that occurs due to emulating the edge compute node. In brief, this error is estimated to be trivial for this evaluation.

Regarding the underlying network, the bandwidth between GCP compute nodes can be up to 7 Gbps because we use the external IP addresses of the compute nodes. Even though internal IP addresses enable higher bandwidth, we use the external IP addresses in order to make our setup representative of scenarios with compute nodes from different cloud providers, i.e., when the use of internal IP addresses is not possible. For the bandwidth between the IoT device and the compute nodes (i.e., from the Internet provider) we use a standard cable connection with upload bandwidth of up to 6 Mbps (and download bandwidth of up to 50 Mbps). Additionally, since 4G has become typical for IoT devices [17], we also use a standard 4G connection with upload bandwidth of up to 7.17 Mbps (and download bandwidth of up to 48.97 Mbps).
5. **edgeRouting**

**Evaluation Baselines**

For the implementation of the multihop routing approach, the data is sent on a path from the IoT device $n_1$ to $n_X$, i.e., $n_1, n_2, \ldots, n_{X-1}, n_X$. The compute nodes on this path are ordered based on network proximity. This means that $n_2$ is the closest compute node to $n_1$, $n_3$ is the closest to $n_2$, etc. To measure proximity, we use round trip times which consistently result in the same order of nodes. The order of the compute nodes based on network proximity is actually the same as based on physical proximity, i.e., $n_1$ is in Vienna, $n_2$ is in Zurich, $n_3$ is in Frankfurt, $n_4$ is in Saint Ghislain, $n_5$ is in Eemshaven, and $n_6$ is in London. To achieve this order, $n_1$ is configured to send the data to $n_2$, $n_2$ to $n_3$, etc., until $n_X$. For the implementation of the direct routing approach, the IoT device $n_1$ is configured to send the data directly to $n_X$. edgeRouting is implemented as discussed in Section 5.3.2.

The suitable compute node $n_X$ is commonly selected using a placement algorithm, as discussed in Sections 5.1 and 5.2. However, in order to gather information from routing data to each compute node of the system, we run separate experiments with each node being selected as $n_X$. To implement the routing approaches, we use prototypes developed in Java 11, which we deploy both on the IoT device (for sending the data), and on the compute nodes (for rerouting or accepting the data for processing). All the nodes use the Debian 10 OS (the Raspberry Pi uses the Debian 10-based Raspberry Pi OS).

**Evaluation Experiments**

Since the proposed edgeRouting aims at reducing the communication latency, we conduct various experiments, and we measure the induced latency for each routing approach. Considering that the communication latency can also be affected by the data size (as discussed in Section 5.3.1, and also in the evaluation of Chapter 3 in Section 3.4.3), we perform experiments with different payloads in order to acquire results that can be extrapolated to a wider range of use cases. To represent small data sizes, we consider sensor measurements of few bytes, while for larger data sizes we examine images of approximately 430 Kilobytes (KB). Both of these data sizes are selected to represent actual IoT applications, namely a smart home application, and an IoT image processing application (similar to the evaluation of Chapter 3).

For the small data sizes, we present the results of using as payload the measurements of a smart gas sensor. The actual values of these measurements are acquired from the REFIT dataset [8]. In fact, this is the same dataset we use in the evaluation of Chapter 4. Since these values have been collected periodically every 30 minutes, we configure our IoT device to produce periodic messages with the same frequency. Each message includes a gas sensor measurement as payload, and is sent to a compute node $n_X$ for processing. This configuration can be representative of a smart energy use case (e.g., for detecting anomalies such as leaks and malfunctions), and also for a variety of other IoT use cases that include sending periodic sensor measurements to a compute node [60].

For the large data sizes, we experiment with images that can be representative of various
5.4. Evaluation

IoT use cases that perform image processing [116]. For example, a smart doorbell that sends a notification with the image of the visitor to the house owner, or opens the door automatically if the visitor is also a resident [178]. The size of about 430 kB is selected for being a common size of images intended for face recognition in environments with edge and cloud compute nodes [164]. We also perform experiments with face recognition tasks being executed in compute nodes using the OpenCV library [7], similar to the evaluation of Chapter 3 in Section 3.4.3. This aids in interpreting the communication latency results better, and is discussed in Section 5.4.3.

Since measurements of experiments conducted using the Internet can vary over time due to potential changes in the network load, we run the three routing approaches at the same time. Thus, each generated sensor measurement is sent to \( n_\times \) three times based on each routing approach (sequentially). We do this in order to collect results which represent each routing approach under the same network conditions, thereby making the results of each routing approach comparable to the others.

For each routing approach, we measure the time needed to send the data from the IoT device \( n_1 \) to \( n_\times \). This time period represents the communication latency from \( n_1 \) to \( n_\times \) including all delays, and potential additional rerouting by compute nodes on path (i.e., detours). Since we use HTTP requests to send the data, the communication latency also includes parsing of the HTTP headers and making the data available to \( n_\times \), but excludes any other processing by \( n_\times \).

5.4.2 Evaluation Results

For this evaluation, we run 2,000 experiments for each routing approach, in order to acquire results that capture the general behavior of the system for each setting. Notably, the latency measurements of all the examined approaches include outliers. This is expected in experiments conducted over the Internet, because the resources of the underlying network may be shared among many users, which affects the network load. The outliers represent isolated occurrences of maximum values, which may affect the average values of latency, despite being rather rare. For this reason, we consider the median as a more representative statistical measure of the results, since it is not affected (as much) by the outliers. To visualize our results in a meaningful manner, in the following we present box plots which include all the latency values of our experiments, and we interpret these values based on the median, and the difference in the interquartile range.

Sensor Measurements as Payload

In Figs. 5.5 and 5.6, we plot the distribution of the latency values when sending sensor measurements via cable and 4G. These figures show the latency when each compute node of the system is selected as \( n_\times \). Specifically, Fig. 5.5a shows the latency of direct routing over cable, Fig. 5.5b shows the latency of multihop routing over cable, and Fig. 5.5c shows the latency of edgeRouting over cable. Respectively, Figs. 5.6a, 5.6b, and 5.6c show the latency over 4G.
5. edgeRouting

![Latency graphs](image)

Figure 5.5: Latency of sending sensor measurements to distributed compute nodes via cable Internet for each routing approach.

![Latency graphs](image)

Figure 5.6: Latency of sending sensor measurements to distributed compute nodes via 4G for each routing approach.

Notably, when using the same Internet connection, the latency of \( n_2 \) is always very similar, and the latency of \( n_3 \) is very similar between multihop routing and edgeRouting. This is expected due to the proofs of Lemmata 1 and 2 which say that \( n_2 \) has the same latency in all the examined routing approaches, and \( n_3 \) has the same latency in multihop routing and edgeRouting.

For direct routing, we note that based on the interquartile ranges of Figs. 5.5a and 5.6a, the latency has a tendency to increase when \( n_X \) resides farther away from \( n_1 \). This happens because the propagation delay increases due to the longer distance between \( n_1 \) and \( n_X \). Multihop routing also exhibits this behavior based on the interquartile ranges of Figs. 5.5b and 5.6b, although the rate of the increase is higher. This is expected since the propagation delay of multihop routing includes detours which are not present in direct routing. In edgeRouting as shown in Figs. 5.5c and 5.6c, this increase is not as evident. There is a clear increase in the interquartile ranges from \( n_2 \) to \( n_3 \), but after that the latency stabilizes with a slight increase for \( n_5 \) in Fig. 5.5c. This happens because even though the propagation delay increases, this increase is countered by a decrease in the transmission and queuing delays due to the transmissions between compute nodes.

To visualize the trend of each routing approach in a more obvious way, Fig. 5.7 shows
5.4. Evaluation

![Figure 5.7: Median values of latency from Fig. 5.5 for each routing approach.](image1)

![Figure 5.8: Median values of latency from Fig. 5.6 for each routing approach.](image2)

only the median values of latency when using cable Internet (from Fig. 5.5). In Fig. 5.7 we note that all approaches start with a very similar median at $n_2$ (about 59 ms). At $n_3$, we observe that the median of direct routing (64 ms) is increased only slightly, while multihop and edgeRouting are a bit higher (about 74 ms). This happens because multihop and edgeRouting follow the same path to $n_3$ which includes a detour from $n_2$.

After that, the median of multihop keeps increasing until $n_6$ (133 ms), while the median values of direct and edgeRouting are very similar, and do not increase significantly until $n_6$ (the edgeRouting median is 81 ms and the direct routing median is 86 ms). Thus, we conclude that when sending sensor measurements over cable Internet, edgeRouting is similar or better than multihop routing with latency reductions that reach up to 39% based on the median values, while direct routing and edgeRouting perform similarly.

In Fig. 5.8 we show only the median values of latency when using 4G (from Fig. 5.6). In Fig. 5.8 we can observe that all approaches start with a very similar median at $n_2$ (about 99 ms). At $n_3$, we observe that all approaches increase similarly (about 114 ms) due to the increased propagation delay. After that, each approach follows a different trend.

Direct routing increases monotonically but only slightly, with a more defined increase
Figure 5.9: Latency of sending images to distributed compute nodes via cable Internet for each routing approach.

Images as Payload

In Figs. 5.9 and 5.10, we show the distribution of the latency values when sending images using the cable and the 4G connections. These figures show the latency of each compute node of the system being selected as $n_X$. Figs. 5.9a, 5.9b, and 5.9c show the latency over cable for direct routing, multihop routing, and edgeRouting, respectively. Similarly, Figs. 5.10a, 5.10b, and 5.10c show the latency over 4G for direct routing, multihop routing, and edgeRouting, respectively. Presumably, the latency of $n_2$ for each routing approach when using the same Internet connection is very similar, and the latency of $n_3$ is similar between multihop routing and edgeRouting (as discussed in Lemmata 1 and 2).

Based on the interquartile ranges of the latency over cable in Fig. 5.9, we note that all routing approaches have a very slight tendency to increase. This tendency stems from the increased propagation delay when $n_X$ is farther away. The reason that this increase is very small is that the propagation delay does not contribute significantly to the latency when routing images (i.e., rather large data). Thus, the latency is affected more by the transmission and queuing delays. In the case of edgeRouting, the propagation delay is also increased, but this is not very obvious due to the low transmission and queuing delays.
5.4. Evaluation

(a) Direct  
(b) Multihop  
(c) edgeRouting

Figure 5.10: Latency of sending images to distributed compute nodes via 4G for each routing approach.

of the transmissions between compute nodes. Multihop routing also has low transmission and queuing delays but the latency is increased because of the detours.

When using 4G as shown in Fig. 5.10, the baselines exhibit an increase which is now more defined, while edgeRouting is rather stable. The reason that the increase is more defined, is that the latency is much lower when using 4G. Thus, the same increase in propagation delay has more potential to affect the overall latency. However, this does not affect edgeRouting significantly, because the increased propagation delay is again hidden by the low transmission and queuing delays.

Interestingly, when sending images, the cable connection results in much more variance than the 4G connection, which can be observed by looking at the interquartile ranges between Figs. 5.9 and 5.10. We believe that the prime contributing factor for this phenomenon is the higher upload bandwidth limit of the 4G connection which allows larger volumes of data to be transferred at once, thereby reducing the variance. Another factor could be a potential difference in the network load between cable and 4G. However, since this is not observed when sending sensor measurements (between Figs. 5.5 and 5.6), it is more likely the former.

To show the trend of each approach, we show the median values alone in Figs. 5.11 and 5.12. In Fig. 5.11 which shows the median values of latency over cable Internet (from Fig. 5.9), we note that the three approaches are very similar at $n_2$ (about 1,190 ms), and $n_3$ (about 1,290 ms). After that, direct routing increases the most until $n_6$ (1,639 ms). Multihop routing also increases but slightly less until $n_6$ (1,518 ms). edgeRouting remains rather stable until $n_6$ (1,293 ms). Therefore, when sending images over cable Internet, edgeRouting is similar or better than the baselines with latency reductions that reach up to 21% compared to direct routing, and up to 15% compared to multihop routing.

In Fig. 5.12, which shows the median values of latency over 4G (from Fig. 5.10), we observe that all approaches start with a very similar median at $n_2$ (about 530 ms), and $n_3$ (about 570 ms). After that, direct and multihop routing increase until $n_6$ (at 988 ms and 1,482 ms, respectively). edgeRouting on the other hand increases only slightly until $n_6$ (671 ms). Hence, when sending images over 4G, edgeRouting is similar or better than
the baselines, and the latency reduction is up to 32% compared to direct routing, and up to 55% compared to multihop routing.

**Overview of the Results**

To provide an overview of our results, Fig. 5.13a shows the distribution of all the latency values of sending sensor measurements over cable. This means that each box plot includes the latency from all compute nodes being used as $n_X$. Similarly, Fig. 5.13b shows the latency of sending sensor measurements over 4G. Fig. 5.14a shows the latency of images over cable, and Fig. 5.14b shows the latency of images over 4G. The exact values of average, median, and standard deviation of these box plots are shown in Tables 5.2 and 5.3.

In Fig. 5.13a, we note that the median of edgeRouting (73 ms) using a cable connection is very similar to direct routing (74 ms), but approximately 26% less than multihop routing (98 ms). In addition, we note that the upper quartile of edgeRouting (79 ms) is well below the median of multihop routing. This shows that 75% of the edgeRouting values are lower than 50% of the multihop routing values. Fig. 5.13a also shows that excluding outliers, multihop routing has the highest latency. This happens because the
transmission and queuing delays which are affected by the data size do not contribute a lot to the latency, since the data size is small. Thus, the latency is affected more by the propagation delay. Multihop routing has the highest propagation delay because the data travels to $n_X$ through detours, as discussed in Section 5.3.2. Direct routing and edgeRouting have similar propagation delays (as discussed in Section 5.3.2) which is why their latency is also similar.
5. edgeRouting

Table 5.2: Average, median, and standard deviation of the latency values from Fig. 5.13.

<table>
<thead>
<tr>
<th></th>
<th>Cable</th>
<th>4G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Direct</td>
<td>Multi.</td>
</tr>
<tr>
<td>Average</td>
<td>76</td>
<td>100</td>
</tr>
<tr>
<td>St. dev.</td>
<td>21</td>
<td>33</td>
</tr>
<tr>
<td>Median</td>
<td>74</td>
<td>98</td>
</tr>
</tbody>
</table>

Table 5.3: Average, median, and standard deviation of the latency values from Fig. 5.14.

<table>
<thead>
<tr>
<th></th>
<th>Cable</th>
<th>4G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Direct</td>
<td>Multi.</td>
</tr>
<tr>
<td>Average</td>
<td>1515</td>
<td>1478</td>
</tr>
<tr>
<td>St. dev.</td>
<td>677</td>
<td>587</td>
</tr>
<tr>
<td>Median</td>
<td>1356</td>
<td>1343</td>
</tr>
</tbody>
</table>

The results when using 4G shown in Fig. 5.13b exhibit the same behavior as cable (in Fig. 5.13a) with regard to the interquartile range of multihop and edgeRouting. However, the upper quartile of edgeRouting (115 ms) is now lower than the median of direct routing (116 ms). This indicates that 75% of the values of edgeRouting are also lower than 50% of the values of direct routing. This stems from the transmission and queuing delays which are affected by the bandwidth, and are lower in edgeRouting due to the higher bandwidth between compute nodes. The median of edgeRouting (105 ms) is approximately 9% less than direct routing (116 ms), and 16% less than multihop routing (125 ms).

Notably, the latency when using 4G shown in Fig. 5.13b is, overall, higher than when using cable (in Fig. 5.13a), even though the 4G connection has a higher upload bandwidth limit. This shows that the bandwidth of the Internet provider is not the only factor that affects the incurred latency of sending data to compute nodes. For this reason, routing approaches such as edgeRouting may be able to reduce the communication latency despite the utilized Internet connection. The reason that 4G has higher latency in this case, is likely due to potentially higher propagation delay of the paths over the 4G network (since small data sizes are not affected significantly by the transmission delay).

In Fig. 5.14a, which shows the latency of images over cable, the median of edgeRouting (1,252 ms) using a cable connection is approximately 8% less than direct routing (1,356 ms), and approximately 7% less than multihop routing (1,343 ms). Interestingly, the time reduction of edgeRouting when sending images over cable (about 100 ms reduction in the median compared to the baselines), is much higher than when sending sensor measurements (1 ms compared to direct routing, and 25 ms compared to multihop routing). The reason for this is that images are larger files which are more likely to be...
affected by bandwidth limitations. Thus, since edgeRouting leverages the high bandwidth between compute nodes, images are able to be sent faster.

In Fig. 5.14b, which shows the results of images over 4G, we note that the median of edgeRouting (644 ms) is approximately 32% less than direct routing (954 ms), and approximately 46% less than multihop routing (1,195 ms). Excluding outliers, the maximum value of edgeRouting (846 ms) is less than the median of both direct routing, and multihop routing. This suggests that all the values of edgeRouting (apart from the outliers) are less than 50% of the values of the baselines.

Notably, the latency of using 4G to send images (shown in Fig. 5.14b) is, overall, lower than with cable (shown in Fig. 5.14a). This can be attributed to the higher upload limit of the 4G connection. Furthermore, we note that edgeRouting performs particularly well compared to the baselines when using 4G (as shown in Fig. 5.14b). We presume that the reason for this is that the 4G network, which is primarily used by mobile devices, may be less loaded at the network edge (i.e., between \( n_1 \) and \( n_2 \)) than cable which is used primarily by stationary users who may run more bandwidth-consuming applications such as online gaming [78]. Thus, edgeRouting benefits from using a larger part of the available bandwidth between \( n_1 \) and \( n_2 \). Direct routing on the other hand, may not benefit from this, because the image is sent from \( n_1 \) to \( n_X \), and this path might be more loaded. Multihop routing can also benefit from more bandwidth at the edge, but the overall latency is increased due to the detours. This assumption is also supported by Fig. 5.10 which shows that the latency of direct routing is rather low for nearby nodes suggesting that the network is not particularly loaded. However, after \( n_3 \) there is a steep increase indicating potential bandwidth limitations for direct transmissions to remote nodes. Similarly, multihop routing has low latency for nodes \( n_2 \) and \( n_3 \), but after that the latency increases due to the accumulated propagation delay of the detours.

In our experiments, we also measure the bit rate of the transmissions based on the routed data (including payload and packet headers). These results align with the presented communication latency results, and can be interpreted in the same manner. For this reason, we do not discuss them explicitly in this evaluation. However, in order to provide a different perspective on the overall performance of each routing approach we note the following: When sending sensor measurements over cable and 4G, direct routing has an average of 3.42 KB per second (KB/sec) with a standard deviation of 0.97 KB/sec. Multihop routing has an average of 2.89 KB/sec with a standard deviation of 0.95 KB/sec. edgeRouting has an average of 3.54 KB/sec with a standard deviation of 0.87 KB/sec. When sending images over cable and 4G, direct routing has an average of 431.06 KB/sec with a standard deviation of 198.45 KB/sec. Multihop routing has an average of 400.36 KB/sec with a standard deviation of 189.04 KB/sec. edgeRouting has an average of 499.83 KB/sec with a standard deviation of 199.23 KB/sec. Presumably, the values when sending sensor measurements are much lower than images because the available network bandwidth of both Internet connections is not fully utilized when sending very small data sizes. This reduces the bit rate compared to utilizing more of the available bandwidth, which happens when sending images.
Hence, we conclude that, overall, edgeRouting provides the lowest latency and highest bit rate in our experiments. There are cases that edgeRouting has similar latency to the baselines, but the distribution of the latency values in general, is reduced. The minimum gains for edgeRouting occur when sending sensor measurements over cable Internet, in which case edgeRouting performs similar to direct routing (but better than multihop routing with 26% reduction). The maximum gains occur when sending images over 4G, in which case edgeRouting reduces the latency by 32% and 46% compared to direct and multihop routing, respectively. Notably, these outcomes comply with the analysis of Section 5.3.2, which draws similar conclusions (i.e., Lemmata 1 and 2).

5.4.3 Discussion

Since the proposed edgeRouting aims at reducing the communication latency of sending IoT data to edge and cloud compute nodes, our evaluation reflects on that by measuring this latency, and by comparing it to the baselines. However, the implementation of each routing approach utilizes a different amount of computational resources, e.g., for rerouting, which can be considered as overhead. For example, direct routing does not require additional computational resources in compute nodes for rerouting data, because the data is sent directly to $n_X$. Multihop routing, on the other hand, may employ many compute nodes for rerouting (i.e., $n_2, \ldots, n_{X-1}$) until the data reaches $n_X$, while edgeRouting employs exactly one (i.e., $n_2$). To make sure that the overhead does not compromise the operation of the system, in our experiments we monitor the resource utilization of all the compute nodes. This overhead is found to be negligible in all the examined approaches. Specifically, we use relatively small compute nodes (i.e., with two vCPUs), and the CPU utilization in all the approaches remains constantly less than 3%, with sporadic spikes which do not exceed 7%.

Furthermore, in systems with edge and cloud compute nodes, the goal is usually to achieve low latency of offloading computations. This includes both the communication latency, and the execution delay of the application [138]. The execution delay can vary based on the specific tasks of an application. For example, performing face recognition tasks on an edge node using common image files (like the images we use in this evaluation), requires about 200 ms [164]. In our experiments, using the OpenCV library and pre-trained Haar cascade classifiers, we also reach this number, with a standard deviation of about 30 ms. Notably, this is significantly lower than the average time needed for sending an image to a compute node, as discussed in Section 5.4.2. Thus, the communication latency can be a prime factor in the overall latency of offloading computations. This further advocates the importance of approaches that reduce the communication latency—such as the proposed edgeRouting. Additionally, the communication latency may include both upload and download latency, if the application produces a result that needs to be sent back (e.g., to an actuator). Even though the download bandwidth of the Internet provider may be higher than the upload bandwidth (which is the case in our setup), the difference compared to the bandwidth between compute nodes is still significant. Thus, we consider that the interpretation of our results applies to download latency as well.
As discussed in Section 5.4.1, in our experiments we use a compute node which resides in Zurich, to emulate the edge compute node \( n_2 \). Thus, \( n_2 \) is not in such close proximity to \( n_1 \) to provide close to zero propagation delay. Instead, \( \text{Pro}_{n_1,n_2} \approx 3 \text{ ms} \), as mentioned in Section 5.4.1. This may raise the question of how the presented results would differ, had we used an actual edge compute node. If we had used an edge node as \( n_2 \), the latency values of \( n_2 \) in all our experiments would be slightly lower due to a 3 ms lower propagation delay. In direct routing, the rest of the latency values (i.e., to all the compute nodes apart from \( n_2 \)), would not be affected because \( \text{Pro}_{n_1,n_2} \) is not part of the communication latency \( D\text{Lat}_{n_1,n_X} \) since the data is sent directly to \( n_X \). This also means that for direct routing, the median values of the overall latency in Figs. 5.13 and 5.14 would not be affected, because the median is not affected by a change in the lower values of the data. In multihop routing and edgeRouting, \( \text{Pro}_{n_1,n_2} \) always affects the communication latency \( M\text{Lat}_{n_1,n_X} \) and \( E\text{Lat}_{n_1,n_X} \), as discussed in Section 5.3.2. This means that for multihop routing and edgeRouting, all the latency values of our experiments would be slightly lower due to a 3 ms reduction in \( \text{Pro}_{n_1,n_2} \). The median values in this case would also be lowered by the same amount as all the other values. However, since the exact values of the medians are between 73 and 1,343 ms (as shown in Tables 5.2 and 5.3), a 3 ms reduction accounts for about 0.2–4% of these values. A slight change in such a small percentage of the values does not have the potential to create a considerable error. Thus, we consider that the emulated edge compute node does not affect the results significantly, compared to an actual edge compute node.

Based on our analysis in Section 5.3.2, edgeRouting manages to provide similar or lower communication latency than the baselines, under the condition that \( \text{Pro}_{n_1,n_2} \) approximates zero, as shown in Equation (5.1). However, the presented results show that the proposed routing approach works well even when \( \text{Pro}_{n_1,n_2} \) slightly deviates from zero. The reason that our approach works well even though \( n_2 \) is not an edge compute node, is that \( n_2 \) is significantly closer to \( n_1 \) than the other available compute nodes. Thus, we note that edgeRouting performs best when \( n_2 \) is an edge compute node in very close proximity to \( n_1 \), although benefits can also be observed as long as \( n_2 \) is significantly closer to \( n_1 \) than the other compute nodes of the system.

One additional factor that might be affecting the communication latency is context switching [35]. Context switching (e.g., from kernel mode to user mode of the nodes) is usually part of the queuing delay [127]. While measuring the context switching delay does not affect our results or the presented benefits of edgeRouting (since it is already included in our latency measurements), it could provide additional insights. For example, context switching delay might be contributing to the increased latency of multihop routing because multihop paths can be longer than the paths of the other routing approaches. Thus, while the reason for the increased latency of multihop routing is attributed mostly to increased propagation delay, increased queuing delay might also be contributing.

Finally, it is worth mentioning that even though our system model and evaluation target the specific case of IoT devices sending data to distributed compute nodes of cloud providers, edgeRouting may also be applicable to other similar environments. For
example, a recent trend is the utilization of various distributed local nodes which all work together to provide an ad hoc cloud [73]. In such environments, there is usually a cloud connector that connects the ad hoc cloud with actual cloud resources [72]. For this exact communication between a node of the ad hoc cloud, and cloud compute nodes of cloud providers, edgeRouting can be used for reducing the communication latency. In this case, the local ad hoc cloud node becomes the data source $n_1$, while the other available cloud compute nodes are $n_2, \ldots, n_N$. Therefore, our system model and the presented evaluation can also apply for the communication between an ad hoc cloud and compute nodes of cloud providers.

5.4.4 Weaknesses and Limitations

In Section 5.3.1, we formulate Equations (5.1), (5.2), and (5.3), for delays that do not contribute significantly to the communication latency, and approximate zero. While these equations can apply in the target environment, there is no guarantee that they will always hold, especially since the communication over the Internet can be unpredictable [172, 243]. Therefore, we consider this as a potential weakness in our theoretical analysis of the communication latency in systems with edge and cloud compute nodes. In our experiments, we use the standard 7 Gbps bandwidth limit with external IP addresses from Google, and an edge compute node which resides in a nearby city. These settings do not opt for a very close approximation to zero in Equations (5.1), (5.2), and (5.3), which could be pursued by utilizing an edge compute node in the same city, and higher bandwidth that can be acquired using internal IP addresses or a pay-to-use plan for increased bandwidth. Despite not aiming for a close approximation to zero, edgeRouting still exhibits the expected behavior with reduced communication latency over the baselines. This shows that even when taking into account the potential weakness in our theoretical analysis due to the unpredictability of the Internet, edgeRouting still provides significant benefits.

A potential limitation of our work can be with regard to the different routing policies of Internet and cloud providers. These providers may implement their own policies for selecting the network path of every transmission, and each policy may affect the communication latency. In our experiments, we use only the default routing of the providers, because different policies are not configurable by users.

Nevertheless, we consider that edgeRouting can exhibit benefits over direct and multihop routing, independently of the routing policy of the providers, because of the following reasons: i) In direct routing, the Internet provider is not aware of potential nearby compute nodes of cloud providers, that can be used as detours. Thus, the data cannot be sent through a compute node. For this reason, edgeRouting which preconditions an edge compute node that can be used as a detour, holds an advantage over direct routing, regardless of the routing policy of the Internet provider. ii) In multihop routing, sending data through various compute nodes, can introduce detours which inevitably increase the propagation delay. Therefore, by avoiding these detours, edgeRouting can provide benefits independently of the routing policy of the cloud provider. There is the possibility that only compute nodes which exist on the network path from $n_2$ to $n_X$ are used as...
5.5. Summary

Since current approaches for routing IoT data to edge and cloud compute nodes may not consider routing paths with increased bandwidth due to detours, in this chapter we propose edgeRouting. edgeRouting routes the data through the closest edge compute node which is commonly able to communicate with other compute nodes with very high bandwidth, thereby reducing the communication latency. To support this claim, we analyze the factors that contribute to the communication latency in edgeRouting and in alternative approaches, and we show that edgeRouting provides similar or lower latency. Furthermore, we perform an evaluation using nearby and remote compute nodes, and considering real-world use cases. Our results show that edgeRouting reduces the communication latency by up to 55% compared to the alternatives.
In the final chapter, we provide a summary of the main outcomes achieved within the context of this thesis. To this end, in Section 6.1 we highlight the main results of each contribution, and we describe how the proposed approaches advance the state of the art of fog computing systems. Subsequently, in Section 6.2 we revisit the RQs from Section 1.2 and for each one, we discuss our efforts to provide an answer. Finally, Section 6.3 provides an overview of promising directions for further research on fog computing.

6.1 Summary of Contributions

In this thesis, we aim at advancing fog computing systems by designing algorithms that enable nearby and remote compute nodes to self-organize into a predefined system structure without the need for manual user configuration. This self-organization takes into account the proximity of the nodes, so that the resulting structure can facilitate the execution of applications with low latency. In addition, we propose two routing approaches that can be implemented on top of a system structure of self-organized compute nodes—or on any set of distributed compute nodes—in order to route the data in an efficient manner. In the following, we discuss each contribution individually in more detail.

6.1.1 Contribution I

For the self-organization of a fog computing system, Chapter 3 presents the design and implementation of distributed algorithms which can be integrated into the participating compute nodes in order to enable all nodes to self-organize into a predefined system structure. Through this structure, each compute node is made aware of the IP addresses of a set of nearby neighbor nodes which can provide additional computational resources. Based on the results of the evaluation, we show that the proposed algorithms enable
6. Conclusion

fog computing systems to execute applications with reduced communication latency and increased network bandwidth.

Interestingly, the proposed algorithms can be implemented based on the architecture of Fig. 3.5 on any compute node either virtual using cloud computing resources in data centers, or physical using local computing resources such as desktops and single-board computers. In fact, such a setup of virtual and physical compute nodes is used for the experiments of the evaluation in Section 3.4 Therefore, the proposed approach of Contribution I does not have a strong dependency on infrastructure support, i.e., it is not assumed that the infrastructure provider needs to implement additional features. This makes the approach of Contribution I more easily applicable to real-world settings.

Nevertheless, certain requirements need to be met in order for the proposed algorithms to work, and to exhibit beneficial behavior. First, since we target large-scale distributed systems, we assume communication over the Internet. Thus, all nodes are assumed to be accessible through an IP address. Also, we assume that all nodes know the IP address of at least one other node which is used as contact for the initial bootstrapping.

Furthermore, since the compute nodes estimate the network proximity to each other, a proximity measure needs to be available. In our experiments, we show that using the number of network hops as a proximity measure can provide significant benefits (such as lower latency and reduced bandwidth utilization). To count this number, we use [ICMP] messages as discussed in Section 3.4 This preconditions that [ICMP] is supported by the routers of the underlying network which is in our case the Internet. Typically, [ICMP] is omnipresent in network devices because it is widely used for troubleshooting, error reporting, and testing (as discussed in Section 3.4). For very accurate measurements, [ICMP] messages should also be accepted by the firewall of the compute nodes, although this is not necessary since the number of hops until the firewall will be measured in any case. For scenarios which do not allow the use of [ICMP] latency measurements can also be utilized. However, latency measurements might exhibit fluctuations which can make some measurement unreliable. These matters are discussed in detail in Section 3.4.

There are also some additional assumptions which relate more to the correct use of the proposed algorithms. As discussed in Section 3.3.2 for scenarios with compute nodes that leave the system unexpectedly, we assume a mechanism that handles failures and maintains the system structure. Such mechanism might be essential for scenarios with high network and/or resource instability, but may induce additional computational and network overhead [115]. Thus, for scenarios which do not need to consider instability, failure handling might be redundant. Finally, we assume that all the participating compute nodes of the system implement the proposed algorithms and follow the specified procedures to the letter. This means that, for example, the computational resources of a compute node that does not follow the algorithms and functionality of Contribution I will not be visible to a fog computing system consisting of nodes that implement the proposed approach.
6.1. Summary of Contributions

6.1.2 Contribution II

Regarding the routing of data to neighbor nodes, we propose two routing approaches in Chapters 4 and 5. The routing approach of Chapter 4 targets scenarios with IoT devices that send data to a nearby compute node which may accept the data or forward it to another node. Thus, in this case the IoT devices are agnostic of the compute node that actually processes the data. For such cases, the proposed approach of Contribution II leverages the context of the IoT data and a history of previous transmissions in order to route the data directly to compute nodes which usually accept requests of the same context. According to the results of the evaluation, this approach manages to reduce the communication latency of sending data to nearby and remote compute nodes by up to 23%.

For this contribution, we rely on the assumption that a compute node which has enough available resources to accept an application request of a particular context, has a high probability to accept application requests of the same context again in the future. This assumption can be plausible in fog computing systems because when an application request is forwarded towards the cloud, and in general, towards compute nodes that integrate more computational resources, the probability that this request is accepted increases. Thus, considering that application requests of the same context require a similar amount of computational resources for being accepted, it is likely that the same node will tend to accept requests of the same context. This is discussed in detail in Section 4.3.2.

However, the probability that an application request is accepted depends on other factors as well. For example, the current availability of computational resources in the compute node which receives a request, and the amount of resources needed for accepting this application request. These factors may be more dynamic because they can be affected by the overall load of the system, and the type of the user application that is being executed. In our experiments in Section 4.4, we do examine various ranges of values for the assumed system conditions, and we do show that our assumptions are generally applicable to fog computing systems. Nevertheless, there is no guarantee that our assumptions will always hold, especially in systems that deviate from the scope of fog computing.

A big part of this contribution is the architecture of a system that is able to perform context-aware routing as shown in Fig 4.4. In order to show beneficial results, this architecture depends to a large degree on an appropriate strategy in the data analysis component (shown in Fig 4.4). The role of this strategy is to select based on context the most suitable compute node for each application request. In our experiments in Section 4.4, we show that benefits can be observed even when using a rather simplistic strategy such as: select the compute node that accepted an application request of the same context in the previous iteration. However, the fact that a simple strategy shows beneficial behavior does not guarantee that any strategy will bear benefits.

We also examine a more complex strategy that is based on RL and uses the Q-learning algorithm. As discussed in Section 4.3.3 RL is common in cases of systems that can be
modeled as a MDP. This means systems in which the transition to the next state depends on the current state and the chosen action. Thus, by modeling the problem of selecting a suitable compute node accordingly, we manage to achieve benefits using the RL method, without making additional assumptions which limit the scope of the proposed approach.

6.1.3 Contribution III

Contrarily to Contribution II, the routing approach of Contribution III in Chapter 5 targets scenarios with IoT devices that know which compute node performs the processing of the data. For such cases, we propose edgeRouting which routes the data via the closest edge compute node. The closest compute node is usually able to communicate with the other nodes of the system with very high bandwidth, thereby reducing the overall communication latency. Based on the results of the evaluation, this routing approach is shown to be able to provide significant latency benefits, especially when routing the data over a 4G connection.

The theoretical background of this routing approach relies on various assumptions which are generally applicable to the communication of compute nodes over the Internet, as discussed in Section 5.3. Specifically, the basis of our approach is that: i) The latency between a device at the edge of the network and the closest edge compute node is commonly significantly smaller than the latency to reach remote cloud compute nodes. ii) The available network bandwidth between compute nodes of cloud providers is usually larger than the available bandwidth at the edge of the network (both are discussed in detail in Section 5.3).

For the evaluation of this approach, we perform experiments on a wealth of scenarios including various geographically distributed compute nodes, and different Internet connections. Since we perform the experiments over the Internet which can be unpredictable (as discussed in Section 5.4) we are not able to verify that our assumptions always hold. However, the results of the experiments do comply with our theoretical analysis which is provided in Section 5.3. This indicates that our assumptions tend to apply for the general case.

A potential weakness of this routing approach is that we do not consider the routing policies that are utilized for selecting routing paths in the underlying network, as discussed in Section 5.4.4. Since the network operators may have their own policies for selecting the best path between two nodes, it is possible that these policies interfere with edgeRouting by selecting alternative routing paths due to considering various factors such as the overall network load. Unfortunately, in the experiments we conduct for the evaluation of edgeRouting, we do not have the means to account for such policies (or even measure the potential impact) because we cannot access the logs of the routers in the underlying networks.

Despite not accounting for the routing policies of the network operators, edgeRouting still shows superior performance compared to the alternatives. This happens because edgeRouting assumes the presence of an edge compute node in the vicinity of the user,
which is used as a detour for sending data to other nodes. The network operators have no means to know if such an available compute node exists. This means that regardless of the routing policy, a detour through an edge compute node will not be leveraged. Due to this detour, edgeRouting may be able to reduce the communication latency of sending data to compute nodes independently of the utilized routing policy.

6.2 Revisiting the Research Questions

In Section 1.2, we introduce the RQs which motivate the work of this thesis. In order to assess to what extent these questions have been answered, in this section we revisit the RQs and we provide a summary of how they have been addressed.

Research Question I:

*How to create a scalable fog computing system consisting of distributed compute nodes that are organized considering network proximity?*

To answer RQ1, we propose the use of distributed algorithms that enable the compute nodes of a fog computing system to self-organize into a predefined system structure. The use of a distributed algorithm in this case allows for the integration of network proximity measurements within the process of adding new nodes to the system. This means that each time a new compute node is added to the system, the new node and the existing nodes take network proximity measurements to each other in order to determine a suitable position in the system structure for the new node. Thus, our decision to use distributed algorithms with integrated network proximity measurements provide a partial answer to RQ1. Specifically, our distributed algorithms address the part of creating a fog computing system consisting of distributed compute nodes that are organized considering network proximity.

With regard to creating a scalable fog computing system, the proposed distributed algorithms make use of groups. A group is a set with a fixed maximum size, that includes neighbor compute nodes. Through the evaluation of the proposed distributed algorithms, we show that when the control actions of adding a new node to the system affect only the groups of the contact node, the overhead of adding new nodes becomes independent of the size of the system, and dependent on the size of the groups. Since the size of groups is also independent of the size of the system, it means that the overhead of adding new nodes to the system is independent of the size of the system. This allows a fog computing system to scale to a large degree without increasing overhead that could compromise the operation of the applications.
6. Conclusion

Research Question II:

How to enable the distributed compute nodes of a fog computing system to route data towards other nodes in an efficient manner?

To answer RQ II, we propose two routing approaches: context-aware routing and edge-Routing. Context-aware routing makes use of a history of previous transmissions combined with an analysis of this history (potentially using predictive methods), in order to route the data directly to compute nodes which accept it for processing. This allows for better use of the available resources (i.e., without forwarding by nodes on path) compared to alternative methods, and faster transmissions (23% latency reduction).

edgeRouting is based on the observation that many current approaches for routing data to edge and cloud compute nodes do not consider that edge compute nodes may be able to communicate with the other nodes of cloud providers with very high bandwidth. This bandwidth is not utilized when sending the data from an IoT device to the compute node which performs the required processing, due to the upload bandwidth limit of the Internet provider. To avoid this, edgeRouting sends the data through an edge node, thus exploiting the very high bandwidth between compute nodes. This way, edgeRouting makes better use of the available network resources, thereby reducing the communication latency by up to 55% compared to alternatives.

6.3 Future Work

Even though the contributions of this thesis directly address the RQs of Section 1.2, there are several aspects of the proposed approaches that can be investigated further.

Regarding the proposed distributed algorithms of Chapter 3, it would be interesting to investigate aspects of fog computing systems that change at runtime, i.e., while the system is running. Specifically, one promising research direction is to focus on the size of the groups, and design mechanisms that enable the compute nodes to calculate the most appropriate group size dynamically based on the proximity of the neighbors, and their resource capacities. Another promising direction would be to investigate necessary adaptations for scenarios with compute nodes that leave the system, cases with potential churn, i.e., when many nodes join/leave the system concurrently, and cases with nodes that have malicious intent and aim at causing malfunctions in the system structure. Finally, future research directions can derive from the assumptions that we make, e.g., related to mobility and transactional join requests. Therefore, future work can explore mechanisms for including also mobile compute nodes (e.g., in electric vehicles), or for facilitating multiple nodes joining at the same time.

With regard to the context-aware routing of Chapter 4, it would be interesting to analyze further the information which is stored by the proposed mechanism, i.e., the compute nodes, and the context that each compute node accepts. Specifically, we consider two promising research directions for future work: i) To explore effective ways for profiling the
6.3. Future Work

To investigate additional ways for predicting the compute node in closest proximity which may accept new application requests (e.g., by examining/combining various machine learning models). Interestingly, the latter might also aid in avoiding the dependence on various assumptions of our current approach, such as the assumption that the same compute node tends to accept application requests of the same context.

Finally, due to the encouraging results of edgeRouting in Chapter 5 which are particularly promising when routing images via a 4G connection, it would be interesting to adapt our approach to consider compute nodes in base stations for potential integration into 5G networks. This can be very useful for applications that operate over cellular networks and require a lot of bandwidth, such as IoT image processing, augmented reality, and video streaming. In addition, since this work relies on assumptions related to the communication delay over the Internet, it would be interesting to perform experiments in a controlled environment which can offer additional data, e.g., from the network routers. Such data may provide further insights about routing that can be used for confirming various assumptions, and also for devising novel routing approaches.
Bibliography


deadline and priority-aware task offloading in fog computing framework leveraging
multi-level feedback queueing. IEEE Internet of Things Journal, 7(7):5773–5782,
2020.

Khalifa, and Momoh JE Salami. Border gateway protocol to provide failover
in multihoming environment. International Journal of Information Technology,

[15] Adnan Akbar, Francois Carrez, Klaus Moessner, Juan Sancho, and Juan Rico.
Context-aware stream processing for distributed IoT applications. In World Forum

Congestion avoidance for smart devices by caching information in manets and IoT.

[17] Muhammad Mahtab Alam, Hassan Malik, Muhidul Islam Khan, Tamas Pardy,
Alar Kuusik, and Yannick Le Moullec. A survey on the roles of communication
technologies in IoT-based personalized healthcare applications. IEEE Access,

[18] Fahed Alkhabbas, Ilir Murturi, Romina Spalazzese, Paul Davidsson, and Schahram
Dustdar. A goal-driven approach for deploying self-adaptive iot systems. In
International Conference on Software Architecture (ICSAs), pages 146–156. IEEE,
2020.

[19] Bastian Alt, Markus Weckesser, Christian Becker, Matthias Hollick, Soumak Kar,
Transitions: A protocol-independent view of the future Internet. Proceedings of the

On the integration of information centric networking and fog computing for smart
home services. In The Internet of Things for Smart Urban Ecosystems, pages 75–93.

[21] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. Online user-driven
task scheduling for femtoclouds. In International Conference on Fog and Mobile

[22] Atakan Aral and Tolga Ovatman. A decentralized replica placement algorithm
for edge computing. IEEE Transactions on Network and Service Management,


[56] Albert Cohen, Xipeng Shen, Josep Torrellas, James Tuck, Yuanyuan Zhou, Sarita Adve, Ismail Akturk, Saurabh Bagchi, Rajeev Balasubramonian, Rajkishore Barik,


[87] Kirak Hong, David Lilletun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications


[184] Ahmad Rabay’a, Eduard Schleicher, and Kalman Graffi. Fog computing with P2P: Enhancing fog computing bandwidth for IoT scenarios. In International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pages 82–89. IEEE, 2019.


# List of Figures

1.1 Example of a hierarchical system structure. ........................................... 2

2.1 Example of an IoT setup with IoT devices, a gateway that implements an API and users that access this API. ......................................................... 10

2.2 Example of an IoT setup with cloud computing resources. ......................... 12

2.3 Example of an edge computing setup with compute nodes that reside close to the IoT devices and the users. ......................................................... 14

2.4 Example of a fog computing setup with IoT devices, gateways, compute nodes, and users. The light blue nodes represent nearby compute nodes, whereas the dark blue node represents the cloud. ......................................................... 15

3.1 Example of a system structure with low and high proximity logical links. 18

3.2 Examples of typical fog computing structures from the literature. ............... 20

3.3 Example of the application model based on an image processing application. 26

3.4 Examples of fog computing structures according to the proposed system model (when the compute nodes join sequentially, i.e., n₁ is the first, n₂ is the second, n₃ is the third, etc.). ................................................................. 29

3.5 High-level architecture of a compute node (dotted lines signify self-organization operations, whereas straight lines denote application execution operations) in the proposed fog computing system (lines with arrows signify interactions with neighbors and with IoT devices). ....................................... 30

3.6 Example of the process of HP1 step-by-step when using a maximum group size that equals four. ................................................................. 34

3.7 Example of the process of FP1 step-by-step when using a maximum group size that equals four. ................................................................. 38

3.8 Location of the compute nodes used in the evaluation (the cloud icon next to a location signifies a Google cloud node, and the Raspberry Pi icon signifies a Raspberry Pi node). ................................................................. 40

3.9 Communication latency to nodes in proximity. ......................................... 44

3.10 Communication latency to the rest of the nodes in the system. .................. 45

3.11 Number of control messages for each new node that joins the system. .... 52

4.1 A fog computing system with edge and cloud compute nodes. ................. 61

4.2 Example of the application model. ........................................................... 63

151
4.3 Routing approaches in fog computing systems: (a) shows a gateway which sends data through compute nodes, whereas (b) shows a gateway which is able to send data directly to each compute node. ........................................ 65
4.4 The high-level architecture of the proposed mechanism. ................................. 68
4.5 Location of the compute nodes used in the evaluation. ........................................ 75
4.6 Average communication latency to reach each compute node of the system based on the two examined routing approaches (values acquired from Table 4.1). ........................................ 78
4.7 Network path measurements of the application requests based on traditional and context-aware routing. .................................................. 81
4.8 Average communication latency achieved by the three examined routing approaches for each deployment scenario, as the probability that a compute node accepts an application request grows. ........................................ 84
5.1 Regions around the world with computational resources offered by two cloud providers. ................................................................. 93
5.2 The target system architecture with distributed edge and cloud compute nodes and IoT devices. The links show the communication between compute nodes (facilitated by cloud providers), and between IoT devices and compute nodes (facilitated by Internet providers). ........................................ 94
5.3 Flowchart with the communication latency of sending data to a compute node. 98
5.4 Routing approaches for sending data to edge and cloud compute nodes. The links show the provider that facilitates the communication (either cloud provider or Internet provider). ........................................ 99
5.5 Latency of sending sensor measurements to distributed compute nodes via cable Internet for each routing approach. ........................................ 108
5.6 Latency of sending sensor measurements to distributed compute nodes via 4G for each routing approach. ........................................ 108
5.7 Median values of latency from Fig. 5.5 for each routing approach. .................... 109
5.8 Median values of latency from Fig. 5.6 for each routing approach. .................... 109
5.9 Latency of sending images to distributed compute nodes via cable Internet for each routing approach. ........................................ 110
5.10 Latency of sending images to distributed compute nodes via 4G for each routing approach. ........................................ 111
5.11 Median values of latency from Fig. 5.9 for each routing approach. .................... 112
5.12 Median values of latency from Fig. 5.10 for each routing approach. .................... 112
5.13 Latency of sending sensor measurements to distributed compute nodes for each routing approach. ........................................ 113
5.14 Latency of sending images to distributed compute nodes for each routing approach. ........................................ 113
List of Tables

3.1 Overview of related work ........................................... 23
3.2 The basic notation of the system model ............................ 28
3.3 Communication latency (in ms) to nodes in proximity for different file sizes. 44
3.4 Communication latency (in ms) to the rest of the nodes in the system for different file sizes. 45
3.5 Network bandwidth (in MB/sec) to nodes in proximity for different file sizes. 47
3.6 Network bandwidth (in MB/sec) to the rest of the nodes in the system for different file sizes. 48
3.7 Execution delay (in ms) of image processing applications for different file sizes. 49
3.8 Regression analysis of the data in Fig. 3.11 ......................... 53
4.1 Communication latency (in ms) and hop count to reach each compute node of the system based on the two examined approaches, and the percentages of reduction ............................................ 77
5.1 The utilized notation .................................................... 95
5.2 Average, median, and standard deviation of the latency values from Fig. 5.13 114
5.3 Average, median, and standard deviation of the latency values from Fig. 5.14 114
# List of Algorithms

3.1 HPI (Hierarchical Proximity-Integrated) ........................................ 32
3.2 Perspective of existing nodes ......................................................... 33
3.3 FPI (Flat Proximity-Integrated) ...................................................... 36
List of Acronyms

4G 4th Generation broadband cellular Internet.

API Application Programming Interface.

CA-QL Context-Aware with Q-Learning.

CPU Central Processing Unit.

FPA Flat Proximity-Agnostic.

FPI Flat Proximity-Integrated.

GB Gigabytes.

Gbps Gigabits per second.

GCP Google Cloud Platform.

HPA Hierarchical Proximity-Agnostic.

HPI Hierarchical Proximity-Integrated.

HTTP Hypertext Transfer Protocol.

IaaS Infrastructure as a Service.

ICMP Internet Control Message Protocol.

IoT Internet of Things.

IP Internet Protocol.

JAR Java Archive.

JVM Java Virtual Machine.
List of Acronyms

KB  Kilobytes.
LAN  Local Area Network.
MB  Megabytes.
Mbps  Megabits per second.
MDP  Markov Decision Process.
ms  milliseconds.
OPT  Optimal latency algorithm.
OS  Operating System.
OSI  Open Systems Interconnection.
PaaS  Platform as a Service.
PDS  Peer Data Sharing.
QoS  Quality of Service.
RAM  Random Access Memory.
REST  Representational State Transfer.
RL  Reinforcement Learning.
RQ  Research Question.
SaaS  Software as a Service.
TCP  Transmission Control Protocol.
UDP  User Datagram Protocol.
vCPU  virtual Central Processing Unit.
VM  Virtual Machine.
Curriculum Vitae

Name: Vasileios Karagiannis
Location: Vienna, Austria
E-mail: basilkaragiannis@gmail.com

Education
04/2018 – present  Doctorate in Computer Science
TU Wien, Austria
10/2014 – 03/2017  MSc in Integrated Hardware and Software Systems
University of Patras, Greece
02/2008 – 07/2014  BSc in Computer Science
Technological Educational Institute of Thessaloniki, Greece

Work Experience
09/2021 – present  Software Architect, Siemens AG, Austria
04/2018 – 08/2021  Research Scientist, TU Wien, Austria
05/2019 – 08/2019  Visiting Researcher, TU Kaiserslautern, Germany
05/2017 – 02/2018  Research Assistant, Nova University of Lisbon, Portugal
09/2016 – 03/2017  Research Intern, NEC Labs, Germany
11/2013 – 04/2014  Research Intern, CTTC, Spain

Research Projects
09/2021 – present  Catena-X Automotive Network
04/2018 – 08/2021  Fog Computing for Robotics and Industrial Automation
H2020 FORA grant agreement No. 764785
05/2017 – 02/2018  Lightweight Computations for Networks at the Edge
H2020 Lightkone grant agreement No. 732505
09/2016 – 03/2017  IoT Platform
NEC Labs internal project