TU WIEN Informatics

Enabling Nomadic Applications in Fog Computing Infrastructures

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Michael Mittermayr
Registration Number 1126749

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Schahram Dustdar
Assistance: Univ.Ass. Dipl.-Ing Thomas Rausch, BSc
Proj.Ass. Dipl.-Ing. Mag. Christoph Hochreiner, BSc BSc

Vienna, 14th January, 2021

Michael Mittermayr | Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Michael Mittermayr

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, 14. Jänner 2021

Michael Mittermayr

Acknowledgements

I would like to thank my advisor Thomas Rausch for taking over advisory for this thesis. I am especially thankful to my family and friends for the all the support provided throughout my studies. A special thanks to Thomas Hiessl, Thomas Kaufmann, Stefan Haider and Andreas Taranetz for all the group works we had been in together. After all this time, when struggling over some exercise, I can still look back knowing that there had been so many other great moments and discussions we had. Last but not least, I would like to thank Thomas Kaufmann once again, for proof reading parts of this thesis and for providing the necessary support at work, allowing me to find the required time to complete this thesis.

Kurzfassung

Das steigende Interesse sowie der damit einhergehende Einzug des Internets der Dinge in die aktuelle Industrie, hat zur Entwicklung alternativer Bereitstellungsmodelle zu den bereits gut etablierten Cloud-basierten Modellen geführt. Durch die Erweiterung der Cloud durch Fog Computing entsteht eine einheitliche Plattform, welche das beste aus beiden Welten kombiniert, d.h. die scheinbare grenzenlose Skalierbarkeit der Cloud sowie die unmittelbare Nähe zu den Datenquellen in Fogs, wodurch neue Datenverarbeitungsmöglichkeiten zu Adressierung gängiger Hausforderungen des Internets der Dinge ermöglicht werden.

In jüngerer Vergangenheit hat sich in der Literatur das Konzept der Nomadic Applications, autonom agierenden Anwendungen, welche sich dynamisch innerhalb verbundener Fogs bewegen, etabliert. Da Fogs jedoch typischerweise in ihren Computer-Ressourcen und zustandsbehaftete Anwendungen auf eine einzelne Instanz beschränkt sind, kann ein faire und effiziente Anfragenabwickelung maßgeblich zur Steigerung der gesamt System Effizienz beitragen.

Im Zuge dieser Arbeit haben wir ein Framework für Nomadic Applications ausgearbeitet und implementiert, welches sich auf autonom agierende, statusbehaftete Anwendungen fokussiert, Mechanismen für die Aktualisierung und Wiederherstellung der Anwendungen vorsieht und zur Evaluierung einer neuartigen auktionsbasierten Anfragenabwickelung dient.

Zu diesem Zweck haben wir Szenarien aus dem Produktionssektor gewählt, darunter gängige Anwendungsfehler sowie Aktualisierungsszenarien und damit eine Reihe von Experimenten zur Evaluierung der neuartigen Anfragenabwickelung durchgeführt. Unsere Experimente zeigten eine signifikante Verbesserung der Metriken, unter anderem bestehend aus der Latenz bei der Abarbeitung von Anwendungsanfragen im Vergleich zur naiven FIFO Ausgangsimplementierung. Darüberhinausgehend hat sich unsere auktionsbasierte Anfragenabwickelung als besonders vorteilhaft im Zusammenspiel mit Fachwissen herausgestellt, wodurch die Ausübung feingranularer Kontrolle bei strategischen Geboten für spezifische Nomadic Applications ermöglicht wird.

Abstract

The rapidly gaining interest and adaption of the Internet of Things (IoT) in today's industry has led to the development of alternative deployment models, complementing well-established Cloud based ones. The extension of the Cloud by Fog computing establishes a unified platform combining the best of both worlds, i.e., the virtually endless scalability of the Cloud and close proximity to data sources in Fog environments, enabling new data processing possibilities to address common challenges in the IoT.

More recently, the concept of Nomadic applications has emerged in the literature, where autonomous applications dynamically travel amongst connected Fogs. As Fogs, however, are typically limited in their computational resources and stateful applications restricted to a single instance, a fair and efficient application relocation request scheduling may greatly benefit the systems overall efficiency.

In the course of this work we devised and implemented a Nomadic Application framework focusing on autonomous, stateful applications including application upgrade and recovery mechanisms in order to evaluate a novel auction-based scheduling mechanism.

To this end, we have identified scenarios from the manufacturing domain including common application failure and upgrading scenarios and performed a series of experiments for the evaluation of the proposed scheduling algorithm. Our experiments have shown significant improvement with respect to a metric indicating latencies for processing application relocation requests compared to a naïve FIFO scheduling baseline. Beyond that our auctioning-based scheduler has also proven to be particularly advantageous in combination with domain knowledge, allowing fine grained control and strategic bids on individual Nomadic Applications.

Contents

Kurzfassung **vii**

Abstract **ix**

1 Introduction **1**
1.1 Problem Statement 1
1.2 Aim of the work 2
1.3 Methodology 3
1.4 Structure 4

2 Background **5**
2.1 IoT and Industry 4.0 5
2.2 Fog Computing 6
2.3 Mobile Agents 6
2.4 Virtualization and containerization 6

3 Motivating Scenario **9**
3.1 Smart Manufacturing 9

4 Related Work **11**

5 System Design **15**
5.1 High Level Architecture 15
5.2 System Monitoring and Recovery 20
5.3 Application Evolution 23
5.4 Target Scheduling for Stateful Applications 24

6 Implementation **27**
6.1 System Components 27
6.2 Used Technologies 29
6.3 REST Everywhere 30
6.4 Implemented Services 31
6.5 Application Lifecycle 35
6.6 Target Selection and Scheduling 47

6.7 Request Auctioning Implementation 48
6.8 Technical Challenges To Solve 49
6.9 Similarities and Differences when compared to Kubernetes 50

7 Evaluation **53**
7.1 Scenarios 53
7.2 Experimental Setup 56
7.3 Simulation Master 57
7.4 Results 59
7.5 Statistical Tests 60
7.6 Summary 65

8 Conclusion **67**
8.1 Future Work 68

List of Figures **71**

List of Tables **72**

List of Algorithms **73**

Bibliography **75**

CHAPTER 1

Introduction

1.1 Problem Statement

The Internet of Things (IoT) plays a crucial role in tomorrow's smart factories, a central component of the Industry 4.0 [SOM14]. Major Cloud vendors have invested heavily towards Cloud support for various IoT workloads, fostering the integration of IoT devices and Cloud-based applications [Mic17, Ama17b]. Traditionally, IoT devices have been used only for collecting sensor data. With the increasing computational power, edge devices have become part of the operational infrastructure (e.g. routers) connecting all of our IoT devices as part of the processing to cope with the steadily increasing flood of data originating from various geographically distributed sensors. Such devices can pool their individual resources to so called Fogs [BMZA12].

Bonomi et al. describe Fog computing as an extension to the well-known Cloud computing inheriting central concepts like virtualization and resource pooling while additionally standing out by offering low latency, location awareness and data privacy [BMZA12] but without the on-demand resource scalability the Cloud offers.

The Cloud's rapid elasticity is one reason why operating software in public Clouds, like Amazon Webservices[1] or Microsoft Azure[2] or within private Clouds has become de-facto standard in today's IT landscape. However, Cloud computing is confronted with several shortcomings in the field of IoT [BMZA12]. Based on a forecast by ABI Research data generated by IoT devices is going to exceed 1.6 Zettabytes by 2020 [Kel15]. Sensor data generated at the edge has to be sent to the Cloud for further processing. Transferring the data to the Cloud is a feasible approach for a small amount of data but becomes infeasible when dealing with a large amount of IoT sensor data.

[1] https://aws.amazon.com/
[2] https://azure.microsoft.com/

Besides the high demand on bandwidth, data privacy plays another important role. In many situations, e.g., due to legal restrictions, it is impossible for data owners to let their data leave the companies premises. When the data enters the Cloud, they would lose control over data access. In such scenarios companies would, for instance, not have full control over possible backups created by the Cloud vendors.

To tackle these challenges, we introduce the concept of Nomadic applications: self-contained applications, autonomously traveling within Fogs and Clouds, built upon the concept of mobile agents. Such mobile agents operate in close proximity to the data source and therefore benefit from reduced latency and network traffic [LO99a]. It is no longer sufficient to process all the data in the centralized Cloud, instead we want to do some of the processing before the data is even transferred over the Internet by applying pre-processing steps right where the data originates.

Nomadic Applications are a promising approach for addressing many of the previously mentioned data privacy concerns as well as bandwidth limitations [Hec16]. However, implementing a platform for operating Nomadic Applications introduces new challenges such as secure and reliable data transfer between Fogs and the Cloud as well as data recovery [HVS+17]. In addition, we need to implement means to detect and resolve network partitions between Fogs and the Cloud while making sure not to lose any data or to introduce a single point of failure. Furthermore, we need to guarantee integrity of the applications as well as the associated data during application travel.

1.2 Aim of the work

The aim of this work is to develop a framework for Nomadic Applications, i.e., autonomously acting and self managing applications in Fogs, supported by the Cloud. Avoiding single points of failure, while optimizing the framework towards efficiency is among the key objectives. To that end, we will implement the conceptual framework proposed by Hochreiner et al. [HVS+17] and examine the open question concerning a fair and efficient way to schedule application requests from the various Fogs.

One of the main challenges lies within a correct handling of stateful applications. Such an application must run at most once, at all times to avoid branching of state. This is required, e.g. in the case of application license restrictions. Furthermore, we need to develop a failure detection and recovery mechanism for applications operating in Fogs. Moreover, we need to distinguish between temporary failures, which the system will recover from on its own, e.g., due to a temporary network partition, and the ones we actively have to take care of. Because resources in Fogs are often limited and available only sporadic and in combination with the previously mentioned network partitions, efficient target scheduling turns out as rather challenging. A baseline version of the scheduling is implemented in a first-come, first-served (FIFO) principle. A subsequent, more elaborate version will provide fair and efficient scheduling via an auctioning mechanism.

Bonomi et al. [BMZA12] list heterogeneity as on characteristic of Fog computing,

with Fog nodes coming in different form factors and environments. However, from an administrator's or developer's perspective we would like to have a platform that Nomadic Applications can be built upon. Moreover, it is desirable to develop an application only once, targeting only one platform. Therefore, virtualization plays an important role to provide this platform. The framework will leverage Docker containers to provide a homogeneous and reproducible environment on the various heterogeneous devices.

To evaluate the feasibility of the approach, the implemented framework is applied to a use-case from the manufacturing domain, described by Hochreiner et al. [HVS+17].

1.3 Methodology

1. **Literature review**
   We survey existing approaches and literature to elicit specific challenges for both Fog computing platforms and mobile agent frameworks.

2. **Architectural design**
   The design phase requires to design the platform services deployed at the individual node types and their interaction as well as an application framework for the Nomadic Applications. One main objective is to identify existing well established and maintained software components applicable for building various aspects of the framework. Furthermore, the design criteria include scalability of our platform and avoiding single points of failure. Therefore, we need to design scalable service communication patterns for the first criterion and possibilities for data replication for the second one.

3. **Implementation**
   The implementation of the framework, addressing the challenges discussed in the problem statement is built using *Java* and the *Spring Framework*[3]. *Docker Containers*[4] will host the Nomadic Applications as well as the framework's management components.

4. **Empirical evaluation**
   We create an exemplary application for the evaluation of the implemented system, representing a use-case from the manufacturing domain described by Hochreiner et al. [HVS+17]. The resulting platform, as well as the implemented scheduling algorithm, is qualitatively evaluated based on this scenario.

   In a quantitative evaluation, we compare the platform's performance, e.g., the time required to move a Nomadic Application, with the time required using traditional manual workloads carried out by administrators. The scheduling algorithm is compared to the suggested first-come, first-served baseline implementation [HVS+17].

[3] https://spring.io/
[4] https://www.docker.com/

1.4 Structure

The remainder of the thesis is structured as follows. Chapter 2 describes the background of this work and defines some relevant terminology. We first provide an overview of the IoT in general and later introduce the concepts of Fog computing and mobile agents, two fundamental concepts of this work. Chapter 3 introduces the motivational scenario based on the smart manufacturing domain. This motivational scenario is an essential part of the subsequently following use case definitions and the evaluation scenarios. In chapter 4 we summarize the relevant related work and the theoretical foundation of this work. In the following chapter 5 we introduce the proposed system design and the high-level architecture. Furthermore, this chapter clarifies the mapping of this work's implemented architecture and the initially proposed one by Hochreiner et al. [HVS+17]. Chapter 6 takes a closer look at the implementation of the platform. In addition to implementation specific details, we also define essential lifecycle phases, our Nomadic Applications pass through. Last but not least this chapter tries to cover parts of the framework's communication protocol, by introducing important patterns used, especially those relevant for the previously mentioned lifecycle phases. With chapter 7 we introduce the evaluation specific details including a description of the used methodology, the actual evaluation scenarios and the necessary framework extensions. Additionally, we we discuss the evaluation results and later summarize the work results in chapter 8 by giving a conclusion and providing an outlook concerning potential future work.

CHAPTER 2

Background

2.1 IoT and Industry 4.0

One prominent example for the application of Internet of Things (IoT) is the fourth industrial revolution, called Industry 4.0. Up to now, three Industrial revolutions have changed the manufacturing domain. The first one, mechanization through water and steam, the second one by mass production using assembly lines and finally the automation [TWW17]. With new market requirements and the emerging IoT an industry shift towards smart factories has started, building the ground layers of the fourth industrial revolution. Such smart factories enable individual mass production, through a highly flexible process, supporting changes on-the-fly[TWW17]. One key characteristic in Industry 4.0 is a highly flexible production volume [SOM14]. The German industry is well known for its high share in manufacturing. In order to support Germany's position as a manufacturing country, the German government has established an Industry 4.0 program [Hen13].

IoT is considered a key technology for the implementation of the Industry 4.0. In the upcoming years IoT is going to provide a bridge technology for the connection of physical objects in order to support intelligent decision making [AFGM+15]. This rapidly growing amount of newly connected Internet driven devices is going to raise new challenges. Already by the year 2010, the amount of Internet connected objects has superseded world population. Machine to Machine (M2M) is considered a key technology for the realization of IoT [Mtm15]. M2M traffic is considered responsible for up to 45 % of Internet traffic by the year 2022 [AFGM+15, Eva11, Pro13, LLM+98]. In 2011 a comparison of M2M traffic in the cellular network has shown an increase by 250 % over that year [SJL+13]. However, the authors point out that the amount of traffic generated by smart phones is way higher than the traffic generated by M2M communication. Furthermore, they point out, even-tough M2M communication relies on different communication patterns, when compared to smart phones, both smart phones and M2M traffic have significant impact

on cellular network design and respectively the amount of traffic such networks have to cope with.

2.2 Fog Computing

A new paradigm called Fog Computing, also referred to as Edge Computing, tries to cope with the amount of traffic originating from various compute devices by shifting compute resources close to physical devices. The term Fog Computing itself is not yet fully defined and mainly considered an extension to the Cloud. This new paradigm comes with some significant benefits, enumerated subsequently[BCL+16, PIUB+17]. Due to the close proximity to the end user and the resulting low latency, Fog computing enables the implementation of real time services, e.g., video streaming and gaming. The distributed nature provides a way to implement geographically distributed large sensor networks while enabling mobility and location awareness. Another key characteristic of the Fog paradigm is its inherent scalability, considering the number of connected devices and services. Considering the manufacturing domain and more specifically smart factories, Fog computing provides a way to integrate the individual factories compute resources and IoT devices with the Cloud.

2.3 Mobile Agents

The concept of Nomadic Applications combined with Fog Computing, is similar to the already well known mobile agents, concerning several aspects. Kotz et al. [KG99] describe mobiles agents as *'programs that can migrate from host to host in a network, at times and to places of their own choosing'*. Such mobile agent saves its state, transports it to the new host, where the state is restored and the program continues working.

Most of the research related to mobile agents, often also referred to as mobile code, is already more than 20 years old. In the early 1990s a prominent idea was the exchange of executable in order to perform client-server computation, popularized by researchers as a way to build intelligent network services [CHK94]. The study authors, Chess, Harrision and Kreshenbaum have concluded that there where nothing that could be done with mobiles agents, one would be unable to do with other means. However, they point out some key advantages mobile agents offer, when compared to other approaches. In their opinion, two of these advantages, amongst several other ones mentioned, are the mobile agents ability to support disconnected operation and their scalability. In addition to those two, Lange et al. [LO99b] published a list of seven advantages mobile agents provide.

2.4 Virtualization and containerization

Virtualization has been an important technology over the last years and utilized to provide isolated compute environments. Moreover, with the introduction of Cloud computing, its importance and growth rapidly gained momentum, culminating in it becoming a

key technology in Cloud computing [Sha14]. Operating Clouds the way they are today, would not be possible without the extensive use of virtualization technology. With the introduction of virtualization technology we are able to increase physical machine utilization, while at the same time, we are able to reduce the amount of work required by machine maintenance, through building a homogeneous compute environment on top of the heterogeneous infrastructure, by utilizing a software based abstraction layer.

Heterogeneity of devices is one important aspect of edge computing, as we already discussed in the previous sections. Bonomi et al. [BMZA12] list this heterogeneity of Fog computing devices as a key characteristic of the Fog paradigm. Nevertheless, it turns out to be a two-sided sword. An application executed on Fog devices relies on some platform assumptions and requirements. Therefore, creating and operating a specific, dedicated machine configuration on a varying set of devices is of high importance concerning edge computing.

With the introduction of virtualization in Fogs we gain the ability to provide such a homogeneous software platform on various highly diverse devices. However, operating virtual machines, we typically rely on in Cloud computing, is not applicable in Fogs. This is due to the tight resource constraints Fogs typically have to face.

Luckily we find a remedy by utilizing the upcoming containerization technologies, generally providing better performance and a higher density in executed applications per host when compared to traditional virtual machines [MKK15]. Unlike full virtualization, containerization uses lightweight containers to establish a homogeneous platform. Those containers usually require only a very limited amount of computational overhead, in comparison to virtual machines.

One prominent example for containerization technology is Docker[1]. Over the last year Docker has gained momentum and has been acknowledged as key technology by the industry, especially in the DevOps discipline. Docker provides means to easily develop and scale IoT applications with almost zero overhead [Ruc16].

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

CHAPTER 3

Motivating Scenario

3.1 Smart Manufacturing

The manufacturing domain has been subject to drastic developments in the advent of the IoT. In particular, smart manufacturing is a prominent example of the realization of Industry 4.0. Hochreiner et al. [HVS+17] consider Nomadic Applications a promising step in the creation of smart factories, utilizing the power of IoT.

In today's multi-site manufacturing companies, site locations are often geographically distributed. Such manufacturing site is equipped with various machines, each collecting a huge amount of sensor data. Continuous monitoring of this data is used to ensure steady quality of the manufactured products.

Maintaining a certain level of quality requires recurring recalibration of those manufacturing machines. Therefore, a dedicated recalibration application is used to analyze the collected sensor data. Later, this application reconfigures the machines, to ensure optimal quality of the manufactured product. It is desirable to improve the calibration process over time. Hence, those applications apply reinforced learning, based on the previously carried out actions and collected sensor data.

There are two ways how we can execute such applications. Either as a Cloud-based application or as a Fog-based one. In a Cloud-based application, data is sent to a centralized application operated in the Cloud. Conversely, a Fog-based application is operated in the respective Fogs. Fog- and Cloud-based applications come with individual, highly different capabilities at their disposal and respectively with individual limitations.

In some cases, the execution as a Cloud-based application is not possible, e.g., due to legal or organizational policies prohibiting data leaving the companies premise. In other cases it could simply be infeasible to transfer the huge amount of sensor data from one manufacturing site into the Cloud. Luckily, the concept of Nomadic Applications offers a

solution to both problems, by executing the application as a Fog-based one. With the Fog-based approach, it is not necessary to transfer any raw data to the Cloud. More precisely, the raw data never leaves the companies premise but is processed right at the individual Fogs.

Moreover, with the execution of the recalibration application within the Fogs, we put a way higher level of trust in the recalibration application. With this higher level of trust, we additionally gain the privilege to access internal machine interfaces. Those interface, would typically not be accessible for Cloud-based applications, due to data privacy and security concerns.

Besides the advantages we gain when executing applications at the individual manufacturing sites, we also need to enable such Nomadic Applications to execute at each of the different sites. As a result, a Nomadic Application is required to travel between the corresponding Fogs. Whenever an application has completed work at some manufacturing site, it checks whether there is another site requesting the application. In case there is one, it travels to the waiting site and continues work there.

Because of the inherent distributed architecture of the proposed system and respectively the potential introduction of a single point of failure, we do not want to introduce some centralized coordination mechanism. Hence, each manufacturing site must be able to request the application autonomously. These requests are handled and stored by the application itself, in a remote assignment list.

Due to the fact that there is no restriction on when and how often a manufacturing site may request an application we need some more advanced mechanism to determine the order sites are visited by the applications. Such a mechanism has to ensure in a fair and efficient way that every manufacturing site, i.e. its Fog, is visited, however this not necessarily implies a first-come, first-served principle.

Moreover, we would like to have a flexible scheduling algorithm capable to prevent single Fogs from occupying one application. Such an occupation could occur due to a Fog spamming the waiting queue. One may come up with the idea to ignore subsequent requests issued by the same Fog. However, this is not an option since each Fog may contain multiple machines, each requiring a recalibration carried out by the same application. Since each recalibration requires a certain amount of time, performing all the recalibrations required at one Fog at once is not an option. Hence, we do require the capability to handle multiple application requests originating from the same Fog.

CHAPTER 4

Related Work

The OpenFog Consortium[1] describes the growth in IoT as explosive and unsustainable under current architectural approaches with cloud-only models. Based on their definition, Fog Computing is a hierarchy of elements added in between the Cloud and the endpoint devices as well as between devices and gateways. Bonomi et al. [BMZA12] describe Fog computing in a very similar way, where Cloud computing is extended with resources located at the edge of the network. As a rather new field, a considerable amount of current research in Fog computing, which is often also referred to as edge computing, deals with its definition [DGCG, SD16, VRM14].

On the one hand, the Fog's heterogeneous infrastructure offers a way to tackle different requirements of data processing applications by offering, e.g., low latency or cost efficiency [BMZA12]. However, the heterogeneity of devices such Fogs are composed of is one challenging aspect. Vaquero et al. [VRM14] discusses the "Softwareeisation" of the network managements with such heterogeneous devices relying on Network Function Virtualization and Software Defined Networks. Cisco introduced with IOx[2] "an application enabled framework for the Internet of Things". IOx offers a homogeneous platform for application deployment and the required management tools and services[3], e.g., for monitoring the deployed applications. However, IOx is limited to a selection of Cisco devices and misses support for mobile agents.

The LEONORE framework proposed by Vögler et al. [MSID16] considers the diversity of Fog devices and focuses on the elastic provisioning of application components on resource constrained devices using push and pull-based deployment techniques. The deployment and scheduling of Nomadic Applications highly depends on Fog resource constraints.

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

[2] https://blogs.cisco.com/digital/cisco-iox-an-application-enablement-framework-for-the-internet-of-things

[3] IOx feature list: http://www.cisco.com/c/en/us/products/cloud-systems-management/iox/index.html

Skarlat et al. [SSB16] propose a conceptual framework for Fog resource provisioning. They optimize the utilization of available Fog-based computational resources with the introduction of the concept of Fog colonies. Fog colonies are micro data centers composed of Fog cells, with the later being single IoT devices coordinating other IoT devices and providing virtualized resources.

Recent research has led to various improvements in the area of software defined networking (SDN) [KDTR12], enabling programming of the network resources. However, SDN is limited to routing logic and does not allow generic application logic [HL13], nor an elastic resource model.

The reference architecture proposed by Dastjerdi et al. [DGCG] utilizes the computations resources at the edge of the network by serving request in the local Fog rather than in the Cloud. Before the term Fog computing was introduced a concept called Cloudlets, an intermediate layer between the Cloud and each mobile device was introduced [SBCD09]. Devices connect to the nearest Cloudlet instead of the Cloud. Cloudlet can be seen as an important special case of the later introduced Fog computing [Sto15]. Nishio et al. [NSTM13] propose an architecture and mathematical framework for heterogeneous resource sharing in Mobile Clouds. In their architecture neighboring compute nodes share their resources in local Clouds based on the idea of service-oriented utility functions [Sto15].

Nomadic Applications revisit the concept of mobile agents, introduced more than 20 years ago. However, in today's IT landscape, mobile agents have to face new challenges, e.g., the rapidly changing availability of resources in Fogs. Kotz et al. [KG99] list several technical and non-technical burdens when implementing such mobile agents. Chess et al. [CHK94] point out several severe security concerns. Arden et al. [AGL+12] propose a new architecture for secure mobile code by analyzing the impact of information flow on confidentiality and integrity within mobile code, utilizing novel constraints on information flow and authority. Unlike their approach, we utilize virtualization to isolate mobile code.

Typically, mobile agents are designed for a specific type of network due to performance optimizations which turns out as a limitation when such an application is deployed to a differently structured network. Satoh [Sat03] proposes a framework targeting this obstacle by distinguishing between navigator and task agents. With Navigator agents being designed for specific sub-networks, efficiently targeting task agents to their destinations in the sub-network. Task agents on the other hand are application specific and therefore execute the actual application logic. In their studies [CPV97, CPV07], Carzaniga et al., classify mobile systems into three categories, namely remote evaluation, code on demand and mobile agents.

Scheduling the execution of such mobile agents in a fair and efficient way is a non-trivial task. During operation, various applications are executed in our platform with resources available in the different Fog cells, depending on external criteria, e.g., applications operated beyond our framework's control. Benoit et al. [BMP+10] propose a way to schedule concurrent Bag-of-Tasks applications on a heterogeneous platform by minimizing

the maximum stretch of concurrent applications. Oprescu et al. [OK10] present a budget-constrained scheduler for scheduling large bags of tasks, across multiple Clouds, considering different CPU performance and cost. Their bag of task scheduling approach requires no a-priori information about the task execution time but learns it throughout execution. The scheduling approach we have implemented does not yet consider execution time, however, future work should consider a similar approach.

Besides fundamental research, IoT and smart factories as two emerging topics are very interesting to industry leading Cloud vendors like Microsoft Azure, Google or Amazon Web Services. With the Azure IoT Suite[4], introduced in 2015, Microsoft tried to integrate IoT devices with their Cloud software. However, with their development in Azure IoT Edge[5], their effort shifts towards edge computing by distributing intelligence across IoT devices. The Amazon AWS platform offers various services for IoT workloads with their AWS IoT-Platform[6], supported by AWS Greengrass[7] as their edge computing solution for the local execution of AWS Lambda[8] functionality. The Google Cloud IoT[9] offers a similarly extensive platform.

Furthermore, we evaluated existing container orchestration platforms which typically follow a centralized approach utilizing API Gateways for all incoming traffic. We concluded that such approach is not feasible for our system. One representative example for such a traditional orchestrator is the emerging Kubernetes [10] from Google, which we provide a more detailed comparison for in Section 6.9.

[4] https://www.microsoft.com/en-us/internet-of-things/azure-iot-suite
[5] https://azure.microsoft.com/en-us/campaigns/iot-edge
[6] https://aws.amazon.com/iot/
[7] https://aws.amazon.com/greengrass/
[8] https://aws.amazon.com/lambda/details/
[9] https://cloud.google.com/solutions/iot/
[10] https://kubernetes.io

CHAPTER 5

System Design

This chapter takes a look at the proposed and subsequently implemented architecture of the Fog framework for Nomadic Applications. Moreover, we are going to discuss some core challenges when implementing such framework.

First, we take a look at the High Level Architecture of our proposed framework, followed by some of the key characteristics of Nomadic Applications. To support a better understanding of the various parts of our framework, we are going to take a look at the lifecycle states a Nomadic Applications goes through. We describe this lifecycle and the individual phases it is composed of. Finally, we discuss the individual parts of the platform concerning their location, their purpose as well as some of the interaction patterns between them.

5.1 High Level Architecture

The high level architecture shown in Figure 5.1 is the system proposed by Hochreiner et al. [HVS+17]. Their architecture is designed to operate in a decentralized manner across the Cloud and Fogs. In their opinion, application management aspects should be handled by the Cloud, while data management-oriented ones are executed in the Fogs.

As already discussed in the introduction in Chapter 1, each type of location, namely the Cloud or a Fog cell, comes with its individual characteristics and different purposes. In the following we are going to give a short recap.

Cloud In the proposed architecture, the Cloud, as a highly elastic part of the system, referring to the available resources, is able to scale up to seemingly unlimited resources, when demanded. On the other hand, when resources are not used, the Clouds inherent scalability enables scaling down or entirely disposing the allocated resources. Such highly scalable behaviour is referred to as elasticity [HKR]. This elasticity is a very useful

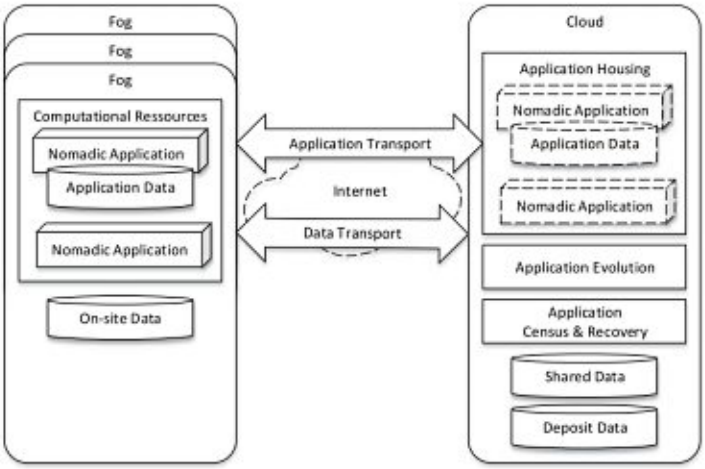


Figure 5.1: Nomadic Application Infrastructure [HVS+17]

characteristic, which we are able to utilize for operating management tasks. Those tasks typically do not profit from the key advantages of a Fog cell, e.g., low latency and close proximity to the data sources. However, in terms of the management tasks being executed in the Cloud, we can profit from high computing power, e.g., for the proposed reinforced learning or during an application upgrade. Using Cloud resources also allows us to relieve the limited resources at Fog cells, thereby balancing overall resource usage.

Additionally, the Cloud is used to operate a Nomadic Application in standby mode, whenever such application is not requested or working in a Fog. Doing so, allows us to further reduce the amount of resources occupied by the operated Nomadic Applications. Moreover, we are not only able to save resources at the Fogs, but also gain higher availability and fault tolerance for the Cloud operated parts of the system. A Cloud typically ensures high availability to the executed applications by providing redundant computing infrastructure and network connectivity, e.g., due to the utilization of independent network connections [NTK16].

Fog Cells Unlike the Cloud, Fog cells come with lower or sporadic availability and limited compute power. However, their close proximity to the data is their key advantage for data intensive applications. Compared to the Cloud, the Internet connection at a Fog cell, is highly limited in the available bandwidth or offers only intermittent Internet connectivity [CZ16]. Therefore, we do not want to, or simply cannot, transfer unprocessed unfiltered data to the Cloud. Each Fog cell can operate autonomously and as result provide the necessary services for the dependent applications. By executing the data processing logic right at the Fog cell, we are able to utilize the potentially much higher internal network bandwidth and further reduce the required Internet bandwidth. Hence, it is the ideal match for processing data originating within the Fog cell. Figure 5.1 illustrates the Fog cells as the ones providing the computational resources for the Nomadic Applications

while additionally being responsible for the storage of on-site data.

In the following we are going to take a look at the individual components deployed at the Cloud and the Fog cells, depicted in Figure 5.1. Furthermore, we are going to refer to the participating locations, i.e., the Cloud and Fog cells as compute locations.

Application Housing Application Housing is a key component of our platform and part of the Cloud. It has several responsibilities, including the storage of shadow copies and checkpoints of the Nomadic Applications. Any Nomadic Application's integrity is validated at the Application Housing using cryptographic hash functions. These integrity checks are required to ensure applications have not been modified or tampered with. As a centralized component, Application Housing supports applications with their travels between individual Fogs, or between the Cloud and a Fog. Additionally, it offers Nomadic Applications a way to update their data, before moving to another Fog. Finally, it is responsible for a Nomadic Application in standby mode, i.e., when there are no Fogs requesting the application. Any Nomadic Application executed in standby mode, resides in Application Housing, waiting for new requests. We are going to take a more precise look at the standby mode in Section 5.1.2.

Application Evolution Application Evolution is responsible for updating and evolving the Nomadic Applications. Any Nomadic Application is required to regularly check for updates. Updates are only applied to applications when they are in standby mode, i.e., any application working at a Fog cell will not be interrupted by an application upgrade. Whenever a Nomadic Application decides to apply an update, it first moves to Application Housing in the Cloud where subsequently the required updates will be applied. During such an update the applications transfer their individually stored state to the updated version. Section 5.3 will take a look at the updating mechanism and outline some of the challenges.

Application Census & Recovery Application Census & Recovery as the third service operating in the Cloud, keeps track of all Nomadic Applications by providing information about their current location as well as their lifecycle state. The concept of lifecycle states is explained in Section 5.1.2.

In addition to managing and monitoring the platforms and applications metadata, the service is responsible for the Nomadic Application's recovery, in case of an error or application misbehavior. Section 5.2 will take a look at some situations requiring recovery and the steps taken by Application Census & Recovery to re-establish the desired system state. Furthermore, it is going to depict some challenges we have to face when considering stateful applications operating in an unreliable network.

Besides the three services the platform distinguishes between four types of data, each coming with its individual requirements and characteristics.

Application Data The proposed system intends to store only a very small amount of Application Data as part of an individual Nomadic Application. The tight coupling between a Nomadic Application and application data ensures maximum availability for the application data, from a Nomadic Applications perspective. The application data itself is transfered as part of the Nomadic Application. As a result, it can be used for the low level configuration of the system, i.e., for the storage of application settings, e.g, connection strings and services URLs, both potentially required by any application to work.

Shared Data Shared data on the other hand is stored in a centralized way offering high availability as well as fast and redundant access, similar to Amazon S3, providing instant data storage and access. Each Nomadic Applications is able to manage its own shared data, with this data being isolated from another applications data. Additionally, the platform's meatadata, stored by the individual platform services, i.e., Application Housing, Evolution and Census & Recovery, is considered shared data.

Deposit Data Unlike shared data, deposit data is not necessarily stored instant, but persistence rather takes some time. Not only storing data requires more time, but also accessing the previously stored data does. While shared data can be compared to Amazon S3, this type of data should follow the principles of Amazon Glacier [Ama17a].

The higher response times typically result in lower storage costs, as this is the case with S3 and Glacier.

On-site Data On-site data is stored at the individual Fog cells and is characterized by the requirement not to leave the individual Fog cell. Hochreiner et al. propose methods from information rights management to ensure any data potentially leaking is useless due to the information rights management restrictions [HVS+17]. There is no further limitation on the form of data itself, it is fully up to the individual Nomadic Application.

5.1.1 Stateful and Stateless Applications

One key aspect of the Nomadic Applications is their ability to travel between the Cloud and the Fog cells, as well as between two different Fog cells. Whenever an application travels from one location to another, it should be able to preserve its state. Therefore, we would like to enable Nomadic Applications to transfer their own state when moving between the different locations. What we consider the state of an application is entirely up to the individual Nomadic Application itself. That includes the possibility of not having any state at all. Such an application is referred to as stateless.

Stateless applications can be considered a subset of the stateful applications, namely those with an empty state. This immediately leads to the insight that any framework capable of managing stateful applications, is also capable of managing stateless ones.

However, for stateless applications we could implement various additional optimizations. In the remainder of the chapter we describe the architecture required to handle stateful applications but still try to stress some of the possible optimizations for stateless ones.

5.1.2 Application Lifecycle

Before we can take a look at the individual components our framework is composed of, we first need to clarify a Nomadic Application's lifecycle.

The lifecycle shown in Figure 5.2, depicts the states every Nomadic Application potentially goes through. In general, we can distinguish between healthy and defective states. In Figure 5.2 all the defective states are distilled in a single one, crashed.

For a better understanding we are going to analyze the lifecycle regarding some exemplary Nomadic Application. The figure illustrates the two types of locations any application can reside at. The initial deployment of an application is done in the Cloud where it stays in standby until some Fog requests it. Triggered through such a request the application initiates the moving process to the requesting Fog. Both, the order and the target selection for the request queue, required when handling multiple concurrent requests, is explained in Section 5.4.

In case of our exemplary application, we would first deploy it in the Cloud, where it runs in standby till some Fog, let's call this specific one Fog A, requests our Nomadic Application. In our example, there is no other request for this application, it immediately initiates the transition to Fog A upon request arrival.

After successful startup at the requesting Fog, i.e., Fog A, the Nomadic Application starts working. Finally, when finished, the application initiates leaving Fog A. Doing so, we need to distinguish between three possible situations.

First, there is no pending request, the application travels back to the Cloud where it again resides in standby. Doing so allows to release valuable resources at the Fog cell. Alternatively, in case there was some pending request, any Nomadic Application always prefers direct travel between the participating Fogs. However, whenever there is an upgrade scheduled for the application, it first moves back to the Cloud, performs the requested upgrade and later on continues working at the next Fog in the request queue. Besides application upgrades the system offers maintenance tasks, which again are executed in the Cloud. One example for such a maintenance task could be reinforcement learning, based on the data collected throughout application execution at the individual Fogs. Typically, such a maintenance tasks would not profit from the Fog's key advantages, it rather profits from a high amount of on demand computational resources, one of the key advantages of the Cloud.

Being able to move between the different locations not only enables applications to utilize the individual location's advantages but also introduces a wide range of possible failure scenarios, including the well known eight fallacies of distributed systems, by L. Peter Deutsch [TV07]. The previously mentioned rather general error cases are only a subset

of the possible ones we need to take care of. For an easier illustration and understanding of Figure 5.2, the sum of those faulty or defective states, is depicted in the state crashed. An application in this state is not capable of resolving the error on its own.
A dedicated recovery service is going to tackle those problems, by monitoring the executed Nomadic Applications and providing a mechanism to detect and recover faulty or misbehaving applications. Section 5.2 summarizes some potential fixes applied by this recovery service. However, in some cases the resolution of one faulty Application state again results in another faulty state. For instance, restarting a Nomadic Application at a different location, can result in an inconsistent system state, with multiple concurrent instances, of the same Nomadic Application, being spawned, at different locations.
With two different versions of the same application running we need some mechanism to identify and distinguish the individual version, as such situation conflicts with our requirement for single instance deployments of stateful services. For a better identification of the deployed instances, we are going to introduce an instance id. This instance id is generated during the initial application deployment and typically kept static while the application travels amongst the system. We will take a look at the exceptions when an instance id might change, in the subsequent sections considering upgrade and recovery scenarios. The instance id is part of the metadata stored by the platform. Besides the instance id this metadata contains information about each instance's deployment location, its application version as well as checkpoints created. Each application movement results in a new checkpoint stored at the Cloud. Such a high frequency for checkpoints, efficiently reduces the risk of lost state and eases recovery for an individual application. We are going to take a closer look at the recovery mechanism and the associated problems in the next section.

5.2 System Monitoring and Recovery

In the Cloud failure is norm rather than the exception [GGP17]. This fully applies to Fogs as well, making failure detection and recovery essential for any resilient distributed system. Only after the successful detection of misbehaving or non-responding applications and services a recovery mechanism can restore valid system state.

The proposed architecture therefore includes a dedicated application monitoring and recovery service, called Application Census & Recovery [HVS+17]. The Census and Recovery component is required to keep track of the executed Nomadic Applications. One of its responsibilities is to ensure that stateful applications will not be replicated, i.e., executed more than once at the same time. Preventing multiple current instances for stateful applications is crucial to avoid application state branching and respectively to prevent the occurrence of data conflicts.

Additionally, Application Census & Recovery provides a mechanism to ensure correct execution of the various Nomadic Applications. This mechanism has to trigger a recovery process in case an application starts to differ from the intended behavior or is not responding at all. We can use the existing metadata managed by the platform, in order

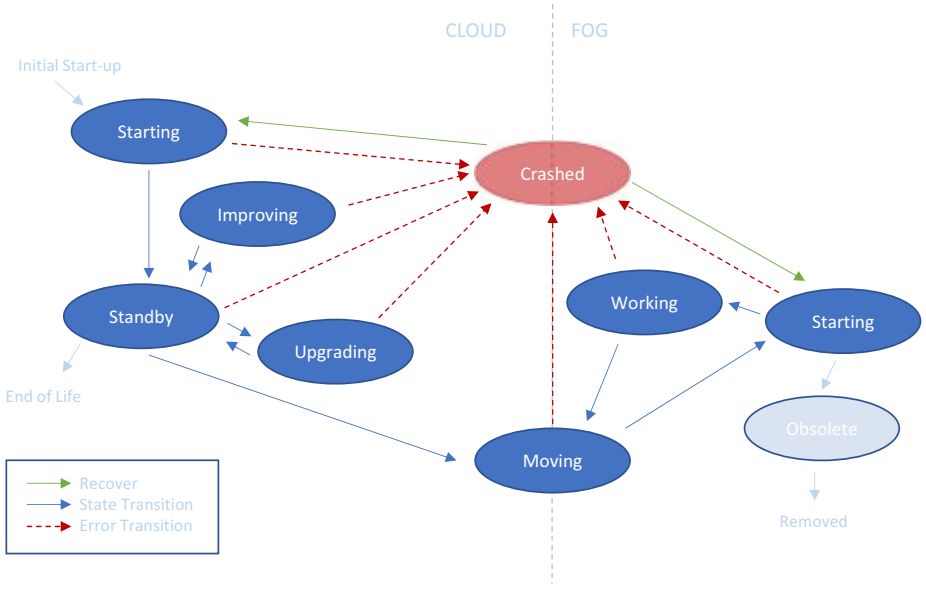


Figure 5.2

to correctly identify which instance of the Nomadic Applications should be executed and vice versa, identify those we have to terminate. Each Nomadic Application is by design responsible for its own lifecycle. As a result, it should be responsible for the resolution of such conflicts, i.e., its own termination. A Nomadic Application is required to detect its redundancy and subsequently it is required to initiate its shutdown and termination.

Obviously recovering such a faulty state can fail and eventually culminate in a Nomadic Application being unable to resolve the error on its own. The following list contains typical errors, we explicitly consider during system design.

- **An application crash during execution or standby mode.**
  A simple restart of the affected application potentially resolves the failure condition.
- **An application crash during travel.**
  In case an application fails during travel, it is restored at its most recent location.
- **A network partition between the application's current location and the rest of the system.**
  Any network partition detected by the monitoring system eventually results in a trade-off between triggering recovery or waiting for network connectivity being restored. However, due to the nature of such partition we are typically unable to decide on how long it is going to last. Additionally, a Nomadic Application might

have failed anyways, due to not being able to contact platform services. To cope with such situations, we would like to propose timeouts. Whenever such timeout is reached, we trigger an application recovery in the Cloud, based on the most recently stored checkpoint.

- **A missing application.**
  It might be the case an application gets lost, i.e., such an application is not running anymore and not even deployed to a connected Fog or the Cloud. This case can be considered a more general case of the previous network partition; however, it might originate due to different reasons. To recover from such state, we can rely on the same strategy we apply for network partitions.

- **Duplicate or obsolete application instances.**
  One prominent cause for obsolete application instances is recovery, e.g., due to the previously mentioned network partitions. When network connectivity for the affected Fog is restored, the original instance in that Fog might be in an operating state. In case recovery has already restored a new instance based on a checkpoint, we eventually end up with two instances of the same application.

  The solution we are proposing relies on two components. First, we are introducing an instance id, a random UUID generated and assigned during initial application deployment. This instance id is preserved throughout application lifetime, including application travels. The second component we are introducing is a platform service responsible for the storage and management of the instance ids. This service keeps track of the assigned instances ids, both valid and obsolete ones. An instance id becomes obsolete when an application is removed from the system. This is happening when an application reaches its end of life, but it is also the case when recovery restores the application, by creating a new instance, based on a checkpoint.

However, in any of those error cases, we potentially get inconsistent or incomplete metadata, partially finished work or even lost data. Hence, the system is required to expect any of these problems, whenever recovery intervenes.

Application integrity

Ensuring integrity of transferred data is a central concern and common task in today's IT. Especially when data is transferred using an untrusted, commonly accessible medium like the Internet. These integrity checks include both, author verification and verifying that the stored and subsequently transferred data has not been altered.

In case of Nomadic Applications, data and application image transfer is an essential part of the system. To fill these requirements, we need to include some mechanism to ensure integrity of the deployed applications. It is desirable to integrate those security mechanisms into the existing transport layer in a transparent way. This transport layer is responsible for both data and application transfer between the participating Fogs and

the Cloud. Relying on a transparent implementation allows to reduce the complexity for a single Nomadic Applications.

Without further integrity checks Nomadic Applications would offer a powerful attack vector. Such an attack could deploy malicious code through altering the used base images. The solution we propose applies cryptographic hash functions after each moving operation to verify application and data integrity [BPSN95]. This approach ensures a transparent implementation within the application transport layer, with negligible overhead, while it meats the previously discussed requirements.

5.3 Application Evolution

One key aspect of the proposed Nomadic Applications is their ability for continuous evolution. With Application Evolution we provide a mechanism to update Nomadic Applications. Such updates may range from application improvements, bug fixes to security related updates. As for application improvements we further distinguish between, programming related changes, e.g., application updates and those originating from reinforcement learning, based on data collected during each application's work at the Fog cells. In the proposed system we do not want to interrupt applications during their work at the Fogs. Hence, updates must be applied when an application has finished working or is in standby mode. Additionally, we would like these updates to be carried out as soon as possible, i.e., it should be the first thing we do after an application finishes its currently assigned work.

This leads to four different updating scenarios.

1. An application is in standby mode, residing at Application Housing in the Cloud.
2. An application finished working at some Fog.
    - There are no new requests for the application.
    - There are requests, i.e., Fogs waiting for the application.
3. The application is a stateless application.

With the first scenario, updating is a straightforward process without any major obstacles. However, most of the time an application will be working somewhere in the Fogs when a new update gets available. Therefore, we decided to make each Nomadic Application check prior to moving if there was any update available.

In case there was such an update, the application itself initiates the updating process and continues moving to the next target, only after finishing the update at the Cloud.

Since Application Evolution is part of the Cloud only, we need each Nomadic Application to travel to the Cloud where updates are applied. During those upgrades we want to

preserve application state. Moreover, we want to copy any application specific data from the old version of the application to the new one.

Such copied data might require some additional processing, e. g., due to evolved data structures used by the new application. This processing is part of the applications responsibilities. Due to the fact, that Application Evolution is part of the Cloud, we can utilize the ability to perform data modifications after such an update, utilizing the scalability and resource elasticity of the Cloud, as discussed in Section 5.1.

After a successful application and data update, the Nomadic Application continues working at the next requesting Fog or keeps running in standby mode at the Application Housing, in case there were no open requests. Request Scheduling and Target Selection will be discussed in Section 5.4.

Upgrading stateless applications is easier. The upgrade process can be applied for new instances without the need to actually update any existing applications. We simply start new application instances using the updated application's shadow copy. Existing applications on the other hand do not require any effort due to our decision not to interrupt working applications. This results in updates being used automatically at the next compute location, i.e. the next time such an application has to move.

5.4 Target Scheduling for Stateful Applications

A Nomadic Application within the platform is identified using an arbitrary freely selectable unique name, which is stored as part of the application metadata. This name is independent of the application's version and stays constant throughout a Nomadic Application's lifetime. In order to perform requests the Fogs have to be aware of the available applications and their names. The scheduling we discuss in this section does not depend on a Fogs' reasons for requesting an application. Moreover, there are various possible approaches a Fog could perform the requesting, e.g., using a fixed time schedule or some metric threshold. Most important, the requests for a specific application are not limited to a single Fog at a time, which implies different challenges we need to address.

With stateful applications being limited to one instance at a time we have to find a way to coordinate the applications execution order among the requesting Fogs. As already mentioned before, it is very desirable to have a system capable of dealing with concurrent, non-coordinated requests for such a Nomadic Application. This means any of the connected Fogs is allowed and respectively able to request any Nomadic Application at any time. Additionally, we also need to allow any Fog to request some application multiple times. Being able to request the same application multiple times is essential to support the individual Nomadic Applications capacity. This is best illustrated using some exemplary use case.

Let us again consider the manufacturing domain. At a manufacturing site, we typically have multiple machines. Each of those requires recurring calibration, as we already discussed in Chapter 3. Since we would like to calibrate each machine individually, we

do require some way to split up the calibration tasks. This individual calibration again is important, since otherwise one Fog with many machines, i.e., multiple machines we would like to recalibrate at the same time, might occupy an application for a very long time. Since we would like to queue a visiting request the exact moment its respective demand arises, we need some way to enable our applications to visit one site multiple times and therefor queue up the same Fog in the applications visiting queue, multiple times.

5.4.1 First In, First Out Principle

A very basic implementation could simply put all requests into a queue, which we later on refer to as travel requests. That approach implements a FIFO scheduling for all the travel requests affecting one specific application. This is particularly interesting due to the widespread use of FIFO scheduling, its inherent simplicity and the wide range of proven implementations, especially in the area of message queuing systems. Such systems are typically designed with scalability, availability and recovery in mind. Message queuing systems such as Amazon Simple Queue Service [1], RabbitMQ[2] or ZeroMQ[3], offer an easy to use and reliable storage for the requests.

However, we do not consider it a desirable solution since a single Fog could still occupy some application for a very long time.

A FIFO approach would introduce a fixed processing order travel requests are handled and allow travel requests for one Fog to be queued up multiple times, and potentially lead to starvation of the other Fogs. This results from the possibility to request an application multiple times due to the lack of any restriction concerning the amount and frequency such requests may occur.

Furthermore, a simple queue would not allow any control and adjustments in the visiting order. In some factories, one machine might be more important than another one. Hence, we would like to have some ways to contribute to the actual visiting order by utilizing application domain specific knowledge.

5.4.2 Auction-based algorithm

We propose an auction-based algorithm to tackle the previously mentioned limitations and to offer a way to control the visiting order by incorporating both, system's state and domain knowledge in the auctioning. When utilizing an auction-based algorithm, a Fog is capable to prioritize its application requests independently from a centralized queuing mechanism.

The algorithm we propose is built using an iterative procurement auction. Chandrashekar et al. [CNR+07] see several advantages with iterative auctioning mechanisms

[1] https://aws.amazon.com/de/sqs/
[2] https://www.rabbitmq.com/
[3] https://zeromq.org/

in comparison to one shot mechanisms. They point out that such algorithm enables bidders to continuously apply corrections to their bids. In our platform, we would like to utilize this possibility by allowing Fogs to increase their bid on some application over time.

In the remainder of this section we discuss the system we have implemented. In our system, we continuously auction visits for each application. Each Fog cell has an individual amount of credits it is able to spend across all the requested applications. Those credits are refilled at discrete intervals. The amount of credits a Fog spends for a specific application request, is entirely at the individual Fogs control. With an application and fog specific utility function we optimize the credits spent for the at individual application request. This potentially allows the application of an optimization problem at the individual Fog.

Whenever an application is ready to move, another round of the auction closes and the highest bidder is the winner, i.e., the Fog the application is going to visit next. One adjustment we made, different from a classic auctioning, is the automatic reinvestment of the current bids for all the losing bidders. Therefore, existing requests get preserved across multiple auctioning rounds.

Due to the ability to increase bids over time, those previously losing bids can get raised later and eventually every request can become the winning one. Another important difference we have implemented comparing to classic bidding systems is that credits are used when bidding, not only when winning. In other words, a Fog can spend the amount of credits available, but no more than that. This approach prevents possible speculation, since it is not possible to place multiple high bids on various applications. The amount of credits at a Fog's disposal during each time frame offers us a way to control application distribution across all Fogs.

These means of control prevent single Fogs from occupying all the applications and additionally ensures that less wealthy Fogs, i.e., those with lower amount of credits, eventually, e.g., after multiple times raising the bid, get an application. More important Fogs spending a higher amount of credits at some specific request will result in getting that application faster. Moreover, we can influence the individual Fog's waiting time by increasing or decreasing its spendable credits.

CHAPTER 6

Implementation

This chapter presents the implemented components the Fog framework is composed of, based on the high level architecture we discussed in Section 5.1 of the previous chapter. Furthermore, it provides a mapping between the high level system components and the actual services.

6.1 System Components

Our implementation is mainly composed of three core platform services (Figure 6.1) in the Cloud and a single one running at each Fog cell (Figure 6.2). Additionally, we run some infrastructure services in the Cloud. Although Figure 6.1 may lead to the impression that those services have to be located at a single host, this is not case. Each of the following services hosted in the Cloud can be distributed amongst several machines, without any restriction.

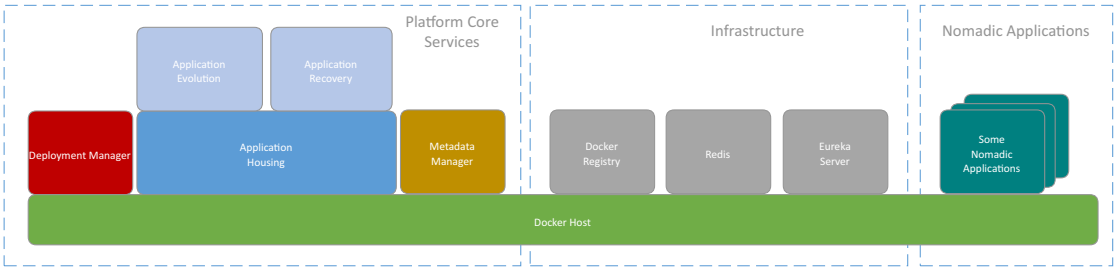


Figure 6.1: Services operated in the Cloud.

In the following we would like to introduce the implemented services our framework is composed of and how they relate to the system design described in Section 5.1. A detailed description of the implementation of each individual service can be found in the respective subsection in Section 6.4.

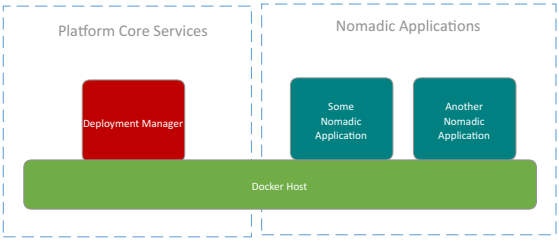


Figure 6.2: Services operated in the Fogs.

Deployment Manager The Deployment Manager is the key component responsible for the management of the running Nomadic Applications by exposing REST interfaces for the necessary container operations. These operations include the creation, transfer and removal of Nomadic Applications, but also provides interfaces required for the implementation of Application Census and Recovery Section 5.2 and Application Evolution Section 5.3. Each compute location runs at least one instance of the Deployment Manager, while each instance of the Deployment Manager is responsible for a single Docker host where it executes the required Docker commands.

Application Housing The Application Housing serves two purposes and is located in the Cloud only. First, it performs monitoring and recovery tasks for the deployed Nomadic Applications, as described in Section 5.2. Second, it takes care of Application Evolution, described in the following Section 5.3.

Metadata Manager The Metadata Manager is responsible for the storage and retrieval of the platform's required metadata about base images, deployed applications, running containers as well as used and retired application instance ids. The typically deployment location is in the Cloud, but does not have to be. Owing to performance reasons, it is a feasible approach to run an instance of the Metadata Manager at the individual compute locations or just at some of them.

Infrastructure Services In addition to the already mentioned platform core services we require three additional services. These three are implemented using existing open source software. The checkpoint and images storage is implemented using a private Docker Registry. Second, we use Redis for the metadata storage, accessed by the Metadata Manager. Moreover, Redis is used for the implementation of each Nomadic Application's request queue. Finally, each application stores application specific metadata. This metadata is in the sole responsibility of the individual Nomadic Application. The third component is Eureka Server, responsible for the service location and part of the Application Census and Recovery by collection heartbeat metadata for each application.

6.2 Used Technologies

The implementation is built using Java 8[1] and Spring Boot[2]. In combination with the Spring Boot stack offers a well-integrated ecosystem, especially in the area of Cloud and Internet technology, utilizing frameworks form the Netflix OSS[3] stack. Spring Boot as one of the industry leading Cloud application frameworks in the Java ecosystem incorporates technology from the Netflix OSS in the Spring Cloud Netflix[4] project. Our implementation uses Eureka for the service discovery and Feign for the service calls, both part of the Netflix OSS stack and considered as industry hardened components in the Spring ecosystem.

For the storage and access of the metadata we use Redis[5] and Redisson[6]. Redis is a simple key value storage offering both high performance and scalability. Moreover, Redis supports cluster deployments[7] capable of resisting certain kinds of failures without human intervention. Redisson, being built up on Redis as a data storage, offers easy to use distributed objects and collections.

While heterogeneity is supportive for the different computing needs, it is a problem when it comes to any administrative operation like monitoring or even deploying applications. With the deployed applications we typically face an additional level of heterogeneity. In the system we propose, a computing device, not only hosts a single application but it rather contains various ones, each application requiring different dependencies. This set of dependencies potentially culminate in version conflicts, rendering our system unable to operate some of these applications.

Emerging container technologies like Docker offer an easily usable way to tackle those problems by providing reproducible conditions while ensuring that every application gets the desired environment [NBM+]. Any application running on top of our Fog infrastructure requires a specific environment, which we can provide using a custom Docker image. For the ease of development not only the Nomadic Applications are hosted within Docker containers but also the platform services.

The Nomadic Applications themselves as well as any checkpoint created, owing to an application traveling amongst Fogs or the Cloud, is pushed to a private Docker Registry[8] hosted in the Cloud.

During development, we additionally used Spring Boot Admin[9] for the debugging and the administration of the deployed service and Nomadic Applications. Log aggregation

[1] https://java.com/
[2] https://projects.spring.io/spring-boot/
[3] https://netflix.github.io/
[4] https://spring.io/projects/spring-cloud-netflix
[5] https://redis.io/
[6] https://github.com/redisson/redisson
[7] https://redis.io/topics/sentinel
[8] https://docs.docker.com/registry/
[9] https://github.com/codecentric/spring-boot-admin

and visualization is implemented using Elasticsearch, Kibana and Filebeat, all three of them part of the Elastic[10] stack.

For the management of the various Docker hosts we rely on Portainer[11]. It provides us with a web-interfaces for the Docker daemon.

6.3 REST Everywhere

In today's web development, REST interfaces are a very popular approach when it comes to service interfaces. Even though RESTful services are considered easier to handle in comparison to those relying on WS-* standards, they do come with certain challenges. With an increasing number of services, those challenges become more severe. In the following we are going to discuss two main challenges, service discovery and communication interfaces by taking a look at our REST everywhere strategy. This includes details about our implementation for the service location and the way we implemented REST calls utilizing Spring Boot and the Netflix OSS stack. The Netflix OSS stack offers industry hardened implementations for the most

Service Discovery In the field of micro services, one recurring problem is the successful and efficient service location [MW16]. Whenever a service wants to call another one, we are required to identify the target service's URL. Configuring those URLs manually or even hard coding them is infeasible for complex, dynamic systems.

With the introduction of Eureka, a service registry developed by Netflix, we can reduce the number of URLs we need to configure to a single one, namely the URL of the Eureka service registry itself. Each of our applications then registers with the service registry during startup. Whenever an application needs to call another one, it uses the service registry to dynamically resolve the required service URL as shown in Figure 6.3.

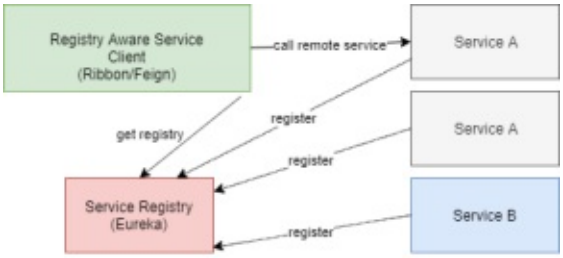


Figure 6.3: Service discovery

Service Client Implementation On the client-side on the other hand, we utilize Feign, a framework for the generation of strongly typed Java client objects based on the

[10] https://www.elastic.co/
[11] https://portainer.io/

API interfaces. The Feign client objects will take care of the request creation, execution and response parsing, in a way, similar to SOAP service calls. The clients generated using Feign provide useful features, including exception serialization and a transparent request retry logic.Additionally, these client objects integrate with the previously mentioned Eureka server for the service location. The Feign clients use both, Ribbon and Eureka internally, in order to enable client side load balancing [MW16] for the deployed services. Based on a service descriptor, a configurable, service type specific string, used during service registration at the Eureka server the Feign clients select a matching service instance.

6.4 Implemented Services

In this section we are going to discuss implementation details and their relation to the high level architecture described in Chapter 5. We are taking a closer look at the three implemented core services, namely the Deployment Manager, the Application Evolution Service and the Metadata Manager. Even tough each one of those three platform services is responsible for specific framework aspects, we still require the combination of them working together, in order to implement the desired platform scenarios. A more elaborated description of the implemented scenarios can be found in Chapter 7.

In the remainder of this section we first take a look at each platform service, followed by a description of the communication patterns and the steps required for application lifecycle management in Section 6.5.

6.4.1 Deployment Manager

The implemented Fog platform uses Docker containers for both providing platform service as well as executing Nomadic Application. Such containers offer an elegant way to create a reproducible software environment [BZ17] and therefore ease the development and operation effort.

One key aspect throughout our system design is the Nomadic Applications' autonomous behavior and their self-responsibility. Based on our system design we want each of the Nomadic Applications to be fully responsible for its own actions. When it comes to certain use cases, an application is responsible for the actions taken, however, such application may require platform support. The most common use case such support is required, is application travel. An application may still decide where it wants to travel next, however, it is unable to perform the moving operation between two different compute locations, without support from some platform component. This platform support is provided by the Deployment Manager. In order to ensure this, we operate at least one Deployment Manager at each compute location. These Deployment Managers act as orchestrators for the underlying Docker hosts.

However, the management of the deployed Nomadic Applications and their corresponding containers, requires additional metadata, stored at the Metadata Manager, described

in Section 6.4.2. This metadata contains information about the currently operated container, including references to the images used and checkpoints created and exists for any Nomadic Application deployed to our system. Even-though there exists a direct relationship between a specific container and the Nomadic Application, this metadata is not managed by the Nomadic Application itself. Rather more, the compute location's Deployment Managers are responsible for it. Doing so not only removes complexity from the individual Nomadic Application but also provides some additional security, by centralizing the metadata management. Due to this centralized management, platform services are not required to trust metadata and respectively information provided by an individual Nomadic Application. Without metadata originating from the Nomadic Applications and respectively the trust in such data provided by the Nomadic Applications, a prominent potential security flaw has been prevented.

Nevertheless, we require some way to identify a specific container's metadata. In order to do so, we utilize the generated Docker container id. This id is generated during the container creation as a UUID for each container, rendering it an ideal match for our requirement [12]. Unlike other settings the container id is not configured in advanced. This results in a Nomadic Application requiring some way to retrieve its own container id. We achieve this by utilizing the Container's hostname, which is equal to the assigned short container id. The short container id is not necessarily unique in the global system; however, it is unique within the individual compute location. Therefore, the combined information of compute location and the short id enables us to retrieve the full id. With this, we can easily access and store container specific metadata.

As already elaborated in the previous sections, failure is rather the norm than the exception when dealing with distributed systems. This results in recovery playing an important role throughout our platform. The monitoring and error detection, based on Census and Recovery in Section 5.2, is implemented in the Application Evolution Service, which we are going to elaborate on in Section 6.4.3. However, this Application Evolution Service requires support by the individual Deployment Managers. A Deployment Manager supports Application Evolution by providing a health endpoint, exposing Snapshot information about its own compute location and the currently deployed Nomadic Applications. Furthermore, a Deployment Manager is responsible for the execution of certain recovery measures, when instructed by the Application Evolution Service. These instructions range from restarting applications to deploying Nomadic Applications based on checkpoints and will be explained in Section 6.5.5.

Like Census and Recovery, Application Evolution, is again part of the platform provided Application Evolution service and orchestrates a Nomadic Application's upgrade utilizing the Deployment Managers to actual perform the upgrade.

The communication patterns and the steps required in order to execute the scenarios mentioned before as well as additional application lifecycle management concerns, are covered in Section 6.5.

[12] https://docs.docker.com/engine/reference/run/#name—name

6.4.2 Metadata Manager

Most of the data required by the platform is managed by the Nomadic Applications themselves. The Metadata Manager's main responsibility is the storage of application lifecycle related data about the created containers and existing images show in Table 6.1. This data is either managed by the application itself or used by the Deployment Manager, as part of the lifecycle phases. The Metadata Manager itself mainly operates as a CRUD services for the mentioned metadata. It is implemented as a stateless service, allowing a multi instance deployment among the various compute locations in our framework. A multi instance deployment and respectively the required data replication is enabled by the Metadata Manager utilizing Redis as the data storage, which provides us with effective means to scale out [TF17].

The implemented platform requires keeping track of various ids, either generated by external tools, e.g., Docker container ids, or by the platform itself. One of those self-generated ids is the instance id, a unique id, required to identify a specific instance of a deployed Nomadic Application. Unlike the previously mentioned container id, the instance id is assigned by the Deployment Manager during initial application deployment and kept constant throughout an application's travel. It is stored as part of the application metadata at the Metadata Manager. Exceptions when an instance id might change, will be dealt with in the later sections, considering upgrade and recovery scenarios.

Besides the container and images metadata required by the platform, the Metadata Manager is also used for the storage of Nomadic Application specific metadata. The individual application uses the Metadata Manager for the centralized and reliable storage of the travel requests. Even though the requests are stored externally, the next moving target selection depends on the implemented scheduling strategy depicted in Section 6.6 and is up to the individual Nomadic Application itself.

6.4.3 Application Housing

The third and last component, Application Housing combines two essential aspects of our framework. First, it is responsible for Application Evolution, described in Section 5.3. With Application Evolution, Hochreiner et al. [HVS+17] enable constant evolution, for Nomadic Application, by providing a mechanism to upgrade and respectively improve existing applications. Second, Application Housing is responsible for Census and Recovery [HVS+17], described in Section 5.2. Unlike the Deployment Manager, Application Housing is not performing the actual actions required for either of the two tasks. It rather acts as an orchestrator for both upgrade and recovery scenarios by delegating work to the Nomadic Applications themselves or to other platform services. While Application Housing is responsible for the collection and management of any runtime specific metadata, including information about each Nomadic Application's current state, it is not responsible for the storage of such data. The storage aspect is again handled by the previously mentioned Metadata Manager.

Category	Name	Description
Image Metadata	Id	Internal unique identifier
	Name	Docker image name
	Tag	Docker image tag
	BaseImageId	ID of the base image metadata record. It is used to store the relation between Nomadic Application checkpoints and their baseline image.
	Application Name	Logical name of the application within the system.
	IsStateless	Control information used by application moving.
	Ports	Application specific ports mapped by Docker engine.
	Environment	Application specific environment variables for the container.
Container Metadata	ContainerId	Docker generated container id. Uniquely identifies the container at the deployed compute location.
	ImageMetadataId	Metadata record ID of the container's underlying image.
	InstanceId	System generated unique identify for an application throughout it's entire lifetime.
	FogId	Unique identifier of the compute location the container is deployed at.

Table 6.1: Container and Image Metadata

First, we are going to take a closer look at Application Evolution. As already discussed in the previous sections, Nomadic Applications are considered self responsible, i.e., it is up to the Nomadic Application itself to trigger an upgrade. However, in order to do so, a Nomadic Application requires some mechanism to retrieve information about possible, existing upgrades. Each Nomadic Application automatically checks for the existence of such upgrades prior to performing an application move. In case an upgrade is available, Application Housing provides the necessary information, including the image id of the new application version. Application Housing does not perform the actual application upgrade itself, but rather stores all the information required for an application to request and detect whether an update is available or not. Subsequently, the Nomadic Application itself is responsible for the upgrade, which is performed at the Cloud only. Hence, a Nomadic Application has to travel to the Cloud, where it requests the upgrade at the Deployment Manager. An in detail description of the upgrade process can be found in Section 6.5.4.

The second component, being part of the Application Housing, namely Census and Recovery, is required to maintain an up to date view of the system's state, by continuously checking all the deployed Nomadic Application's health status. These checks are performed using a health endpoint, each Nomadic Application exposes. Application Housing applies transparent retrying when querying those health endpoints. In case it is unable to retrieve an up to date health status for a Nomadic Application, within a configured grace time, the recovery component triggers a recovery process.

During such a recovery a new instance id might be generated, depending on the actual recovery steps taken. In case a new instance id was generated, the previously used one is marked as deprecated. This deprecation information is managed by Application Housing using a history table and required for the detection of obsolete instances. The detection of an obsolete instance is again within the responsibility of a Nomadic Application. Again, each Nomadic Application automatically and repeatedly checks for its deprecation and triggers its own teardown, in case it was marked as deprecated. A more elaborated description of possible causes for a recovery and respectively an instance deprecation as well as the steps executed and platform services involved, is described in Section 6.5.5.

6.5 Application Lifecycle

In the lifecycle of every Nomadic Application we have to deal with a set of mandatory and optional phases. Almost all the required work during those phases is part of the platform's implementation. Even though, the steps required during such phases are implemented in the platform, each Nomadic Application is still responsible for correctly triggering the required steps.

In the following we take a look at the phases each application potentially faces throughout its lifetime.

6.5.1 Application Start

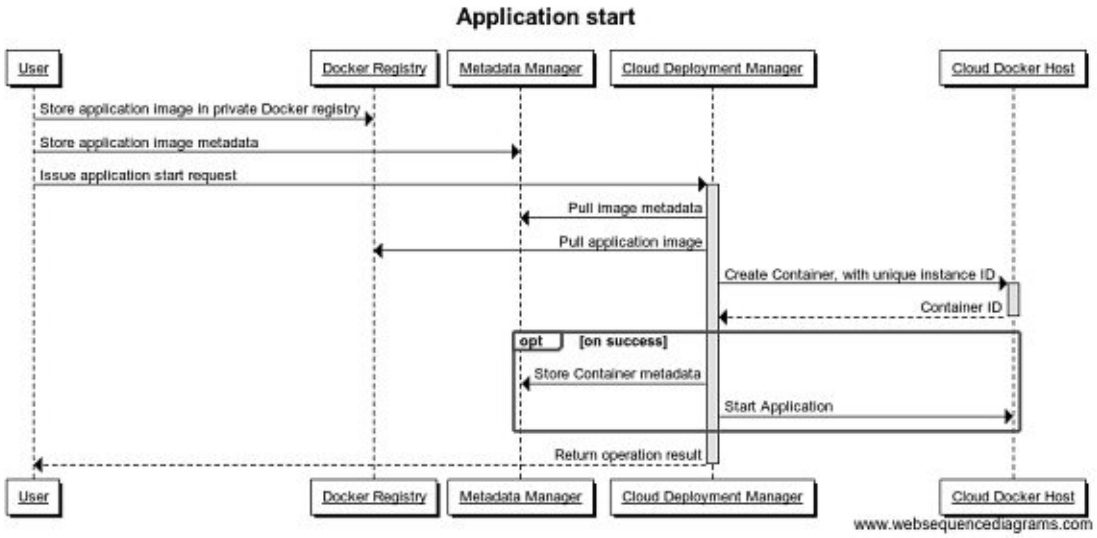


Figure 6.4: Starting a new Nomadic Application

The first phase any Nomadic Application has to go through is the initial deployment. In the framework we have implemented, an application resides in the Cloud, i.e., in the Application Housing, in case there is nothing to do for it. Since the application is at the very beginning of its lifecycle, there obviously is nothing to do yet. Therefore, one simply has to start the application in the Cloud. We can achieve that by issuing a start application request at the Deployment Manager. The work required during the initial deployment is identical with a common application start, which we are going to require very often throughout a Nomadic Application's lifetime.

Figure 6.4 depicts the actions required to deploy a new Nomadic Application. First of all, the application needs to be stored as a Docker image in our private repository. We are going to take a look at the Docker base image as well as the repository configuration later. Additionally, our system needs metadata for each application supported by the platform. That metadata is stored at and retrieved from the Metadata Manager and contains the required configuration for the Deployment Manager, in order to create a new container.

When executing a new start request, the metadata for an application is retrieved from the Metadata Manager and the Docker image pulled from the private repository. After successfully fetching both, the metadata and the required image, the Deployment Manager creates a new container, based on the configuration stored in the image metadata. On successful creation the Deployment Manager stores container specific metadata at the Metadata Manager. Due to the storage of this metadata we do not require to keep any additional state at the Deployment Manager. The container specific metadata contains information about the used Docker image, the deployment location and the instance id

and is required by subsequent operations performed by the Deployment Manager.

Since there is no centralized configuration server, we need to supply several parameters during container creation. Those parameters range from the server port, the service registry URL, to platform specific configuration values and the internally used unique application instance id. We are going to take a look at the parameter passing logic in section 6.8.1.

The bootstrapping logic for each application is part of the platform's implementation. Each application registers with the Eureka service discovery. After successfully bootstrapping the application, it resides in the Cloud in standby mode until it transitions into another lifecycle phase, e.g. by being request from some Fog, which we are going to discuss next.

6.5.2 Application Move

The Nomadic Application's ability to move between compute locations is one of their key characteristics. Such application move transfers the application including application state to another compute location. There are three possible reasons for such an application move.

- The application moves to some Fog due to a request by that Fog
- It moves to the Cloud for standby
- It moves to the Cloud for an upgrade

In the remainder of this section, we are going to take a look at each of the three possibilities. First, we start with the implementation of the Nomadic Application requesting, followed by a discussion of the target selection. Last but not least we deal with the actual application moving, including the application checkpointing and packaging.

Application request

Whenever an application moves to a Fog cell, it has been requested to do so.

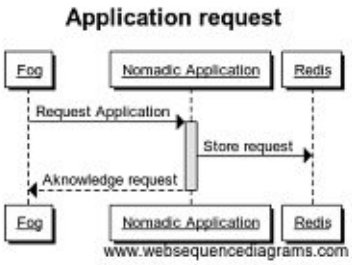


Figure 6.5: Fog requests a Nomadic Application

In the framework we have implemented, an application request is issued by a Fog cell, as shown in Figure 6.5. In the system design, Section 5.1, we already discussed the high importance of preventing a single point of failure. One such potential single point of failure could be introduced by relying on a centralized coordination mechanism for application requests. Our proposed framework rather pushes such responsibility to the Nomadic Applications. We want an application to take care of the management of its requests by its own, only utilizing some platform services. However, responsibility and control keep relying with the applications.

A potential solution could use a remote assignment list to store application requests. When an application receives a new request, that request should be stored in the remote assignment list and subsequently handled by the application itself. The assignment list can be implemented using Redisson and stored in a Redis server in the Cloud. By doing so, we can utilize both, the high performance of Redis and its cluster and replication feature to ensure data safety. Using a remote assignment list enables us to prevent the loss of requests up on application recovery using previously created checkpoints, as described in Section 6.5.5.

However, such an approach comes with a major drawback. A Nomadic Application is not able to handle requests at any time, it is actually rather restricted when an application is able to do so. The time span an application is able to accept and store new requests is limited to lifecycle phases where it is executed in some Fog cell or in standby mode in the Cloud. For instance, whenever an application is moving, upgrading or somehow not reachable, we would not be able to request the application. Due to this limitation, we do not consider it a feasible solution. One possible workaround could implement some retry logic for the Fogs issuing those requests, by repeating the requests until they eventually get processed by the application. Yet, doing so would require additional work by the Fogs and furthermore, introduce further complexity.

Nevertheless, because we are using Redis as storage for those requests, we are able to implement a different solution, without threatening availability nor scalability. We introduce a service handling those requests and storing them. Each Nomadic Application still has its own remote assignment list, but it is no longer the application only, accessing that list. Rather, we would like the service to support the applications by managing the list for them. An application no longer has to take care about storing the Fog requests but only has to ask the service for its requests. This newly introduced service again can be implemented stateless and deployed as multiple instances.

With the introduction of the proposed requesting service, we have the possibility to handle requests at any time, without respect to the application's current lifecycle state. Additionally, such requesting service would be able to leverage extended knowledge about the platform's internal state, including the state of other Nomadic Applications. Such knowledge is of high value for a more advanced scheduling mechanism.

Selecting a target

Immediately after finishing work at a Fog, a Nomadic Application has to move, either to another Fog, or to the Cloud. Such a moving process starts with an application selecting the target compute location. For this target selection step, we have to consider the following four possible starting points.

- **The application was requested by one or more Fogs:** In that case, the target is selected from the remote request list. The actual selection algorithm is based on the target scheduling we discussed in chapter 5, section 5.4.
- **The application is located at some Fog and the remote request list is empty:** Since we decided to free resources in Fogs as soon as possible, the application has to move to Application Housing in the Cloud.
- **The application is located at the Cloud and there is no target in the remote request list:** In this case the application stays in standby mode at the Cloud. The moving operation is canceled. Such Nomadic Application periodically checks whether a move is required, by re-executing the target selection.
- **Moving to another Fog is not possible:** This is typically the case when the target Fog is not reachable, or out of resources. We list this situation as part of the target selection, since a Nomadic Application restarts target selection at its current computing location upon a failed moving attempt. A Nomadic Application is capable to detect such failure when restarted. In case the application is located at some Fog, it decides to move to the Cloud instead. When located at the Cloud it starts a new attempt.

Moving an application

Figure 6.6 shows a more in detailed description of the actual steps carried out in order for a Nomadic Application to move to some remote location. We assume the application has already decided on some target compute location to move to.

The application itself simply requests moving from its associated Deployment Manager. This Deployment Manager first asks the target Deployment Manager whether a move was possible. The target Deployment Manager checks if there were enough free resources available at the target location. If this was not the case, the Deployment Manger cancels the moving process and respectively informs the application about the lack of resources at the target location. Upon this, the Nomadic Application selects a different target, as described in the previous section.

However, assuming there were enough resources at the target, the Nomadic Applications continues with the shutdown procedure. During this shutdown, the Nomadic Application cancels its registration with the Eureka service registry. After some timeout the application has either stopped on its own, or the Deployment Manager actively stops the

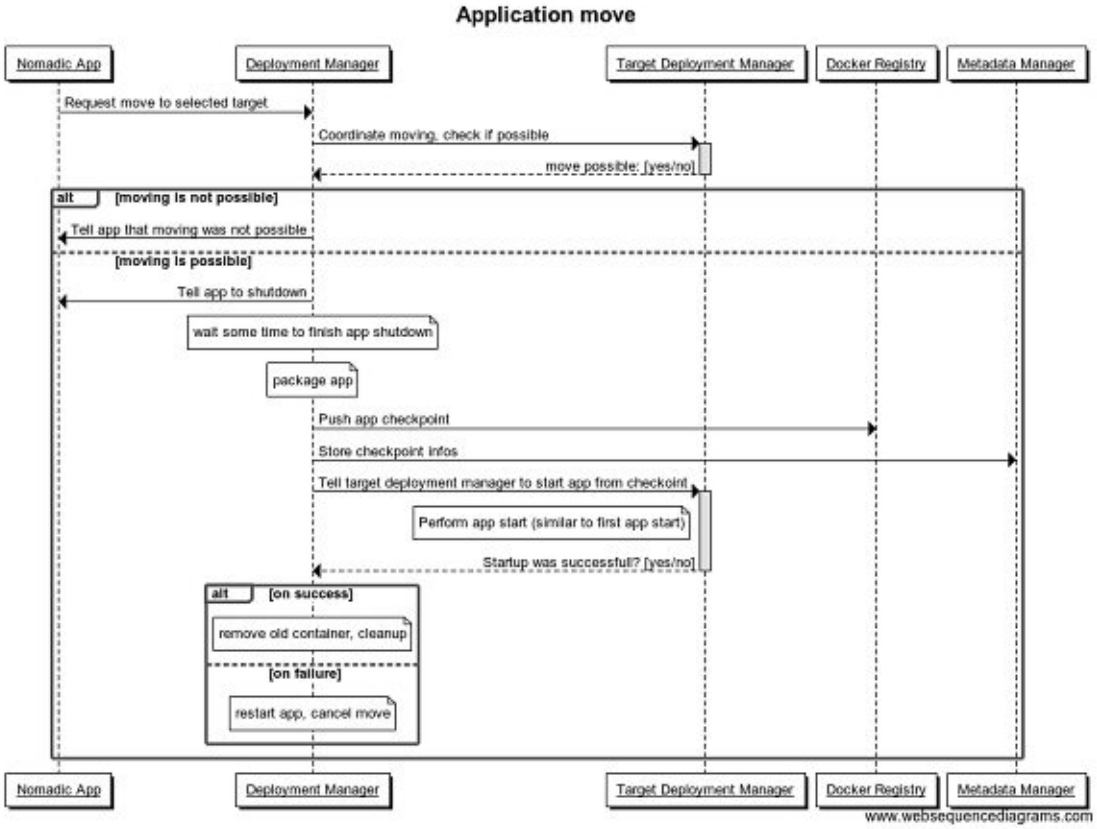


Figure 6.6: Steps executed during a Nomadic Application move

Nomadic Application's container. After container shutdown, the Deployment Manager starts the packaging process. We are going to take a more detailed look at the packaging step, including the creation of checkpoints, in the next section.

When both, the data and the metadata is successfully transferred, the Deployment Manager asks the target Deployment Manger to start a new application.The application start at the target Deployment Manager works as described in section 6.5.1, with one exception, we do not generate a new instance id. Instead, we preserve the existing one and reuse it at the target Fog. In case the application start at the target Deployment Manager was successful, the source Deployment Manager removes the existing source container and performs additional cleanup tasks. In case it was not successful, the original source application's container and respectively the application itself gets restarted. After restart, the Nomadic Application is able to select a new target and the moving process starts again.

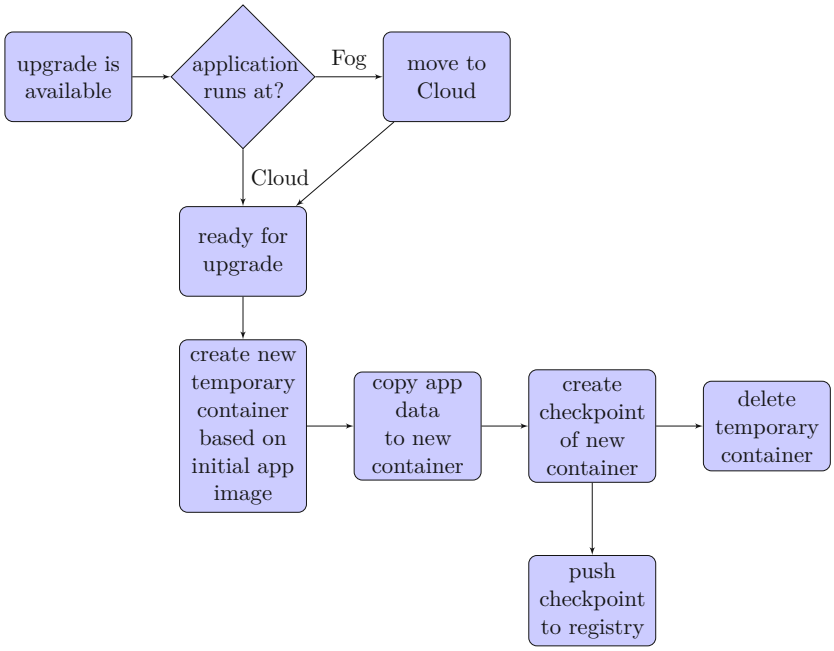


Figure 6.7: Steps executed at packaging process

Package Application for moving

To explain the application packaging, we first take a look at the implemented checkpointing mechanism. Our implementation is related to the approach proposed by Goiri et al. [GJGT10], utilizing Another Union File System to backup virtual machines in read-only and read-write layers. With checkpointing we create Nomadic Application snapshots, called checkpoints. Such a checkpoint contains the application as well as the application state and is an essential aspect used by both the application moving and the recovery. Whenever we decide to move, we create a checkpoint at the source location and push that checkpoint to the remote location utilizing the private Docker registry, located in the Cloud. Such a Nomadic Application checkpoint is implemented using the Docker commit functionality, which we are going to look at later in this section. After successfully pushing the checkpoint to the Docker registry, it is pulled by the target location where the application is started using that checkpoint. In addition to the checkpoint data, we again need to store checkpoint specific metadata at the Metadata Manager. This checkpoint specific metadata is an updated version of the initially created application image metadata and is used by any subsequent request referring to the associated checkpoint, as a replacement for the initial application image metadata.

Our platform relies on Docker images as mechanism to transfer both, the application image and the associated application state side by side. Each Nomadic Application persists its state to the container file system and loads state during startup. By utilizing the Docker commit functionality we can transfer the exact state of our application and

data. Before we get to look at the actual packaging implementation, we first need to understand how Docker manages its images.

Docker images are built using multiple layers, with each layer containing just the differences to the underlying layers [BZ17]. One advantage of this layering system lies within the possibility to reuse layers. In our case, we build Docker images based a custom base image. This base image itself, is built from a public Docker image, containing the Java JRE. Since each of our applications and services uses the same custom base image, most of the images data, i.e. the layers, are reused among all the Nomadic Applications as well as each of the platform's services. Therefore, we only need to pull the layers on top of the base image, whenever we want to run a new application at some Fog or the Cloud. With that layering system we can save both, storage and network traffic. However, there is some limitation on the maximum number of layers an image can use.

Since we would like to create a checkpoint on every move of our Nomadic Applications and checkpoints are implemented as commits, with each commit basically creating a new layer on top of the existing ones, we eventually reach the point where we are no longer able to create new layers and respectively no longer able to create checkpoints. Hence, we had to find some way to circumvent that limitation, in order to utilize the Docker commit function.

In our platform each Nomadic Application has one dedicated data folder where all the application specific data should be stored. Our solution for the limited number of layers utilizes this data folder at the implemented packaging process. Although this step is shown in Figure 6.6 as a single step only, namely 'package app' it involves quite some effort. In Figure 6.7 we can see the actual work required.

As a first step in the packaging process we create a new container based on the Nomadic Application's base image. This newly created container comes without any application state. Therefore, we copy the content from the Nomadic Application's data folder to the newly created container. Due to this copy step, the new base image container contains all the application's data including the most recently persisted state. Moreover, this allows us to ensure that the underlying platforms state always stays clean by ensuring potential changes outside the application data folder, e.g., temporary files, are ignored during copying. The checkpoint itself is then created for this new container. The resulting, new image is built using just one additional file system layer, in comparison to the initial application image. This additional layer contains the application's state and the application specific stored data.

In consequence of repeating this process on every move, the number of layers will not increase. Hence, we never have more than one layer on top of the initial application image, no matter how often a Nomadic Application travels. Additionally, it allows us to reduce the amount of data transfered to the state layer only, since the application layers are immutable and do not require pushing to the Cloud, a solution comparable to the read-only and read-write layers proposed by Goiri et al. [GJGT10].

6.5.3 Application Standby

Whenever an application is not required to work, it has to reside in standby mode, waiting for new requests. We consider Fog resources both, limited and more valuable than Cloud ones mainly because adding additional resources in the Cloud is typically easier compared to the Fogs. As a result standby mode is handled by Application Housing in the Cloud. Furthermore, we want applications to release resources at Fogs as soon as they are not required anymore. Hence, an application immediately triggers the moving process when finished working.

In case there is no other application request, or the requesting target is not available or capable of running the application at that point in time, the application decides to move to the Cloud. In the Cloud, an application is always executed in standby mode.

In standby mode, the Nomadic Application repeatedly checks for application upgrades or open requests. In case such an upgrade is available, it immediately triggers the upgrade process. The repeated checks are implemented using a Spring scheduled method and trigged every 30 seconds after the last call has completed.

As soon as there is a valid request from some Fog, the application initiates the moving process to that Fog.

6.5.4 Application Upgrade

The ability of continuous evolution for Nomadic Applications is a key part of our framework. A Nomadic Application may upgrade multiple times throughout its lifetime. Since our framework and any Nomadic Application relies on multiple other frameworks and software tools, we eventually have to update those platform dependencies as well. There are various reasons behind such updates, ranging from security related ones to simple bug fixes or improvements. Updates should be carried out as part of the application moves. Additionally, they should be applied immediately when a Nomadic Application is in standby mode. However, we do not want to lose application state as a result of such an update. Due to that, we cannot simply replace the running application by the new version. Rather, we have to preserve the state while updating both, the environment, i.e., the underlying Docker base image and the application.

Our application packaging process, we already discussed in the previous sections, equips us with the necessary tooling for upgrades. In our solution, the Nomadic Application itself is responsible for all the application's state modifications required. Those modifications should be applied, e.g., as part of the first Nomadic Application's startup after the upgrade. Since such an application upgrade might require higher computing power, we limit upgrades to the Cloud. Therefore, any application has to move to the Cloud, in order to be upgraded.

Figure 6.8 depicts the steps executed during such an application upgrade. The first step is to ensure that the application is executed in the Cloud. In case the affected application is running in a Fog, it first has to move to the Cloud. For the upgrade itself, we create

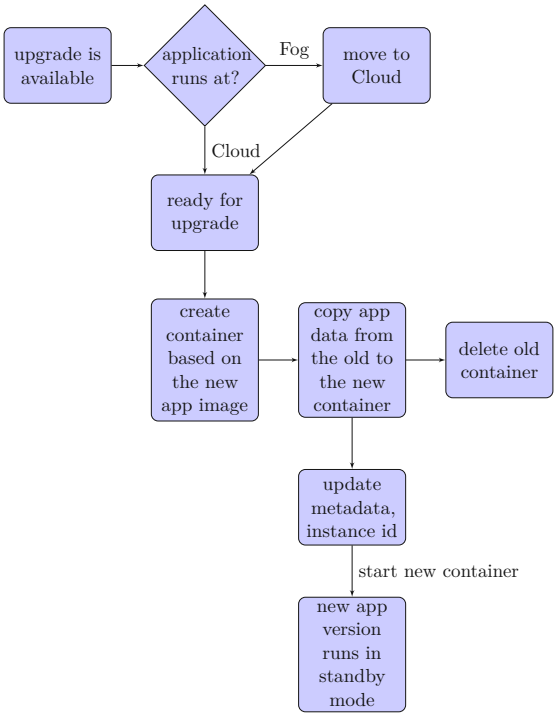


Figure 6.8: Steps executed at an application upgrade

a new container in the Cloud, using the updated base image. This container also gets a new instance id, to make sure we can easily identify the old and the new application version. The old instance id is later marked as deprecated at the Metadata Manager. Like the application packaging, we need to copy the application data from the old version, i.e., the old application image, to the new one. Finally, we only need to delete the old container and start the new one. This startup process is based on the steps described in the application start phase. A detailed insight concerning the necessary steps for an application upgrade can be found in Figure 6.8.

6.5.5 Application Recovery

Considering the Fallacies of Distributed Computing from L. Peter Deutsch from Sun Microsystems in 1994, recovery is an essential part of our platform to ensure successful application execution.

Recovery tries to minimize the effects of an unreliable network on our Nomadic Applications, while the recovery service itself struggles with these Fallacies of Distributed Computing. Bala et al. [BC12] identified checkpointing as a reactive recovery method which allows to limit dataloss caused by recover to the unprotected data created since the most recent checkpoint rather than all data collected from the initial startup.

Topics like high latency, limited bandwidth as well as a changing network topology make

things a lot more difficult for the recovery service. One example for such a critical situation is an application traveling between Fogs. We have empirical data on how long such travel should take, however due to, e.g., a temporary network partition, the Nomadic Application will not start within the recovery services configured timeout. In case of a timeout, we consider the application as a potentially crashed one, which requires recovery.

Due to the nature of our system and requirement for autonomously acting applications, we are unable to identify crashed Nomadic Applications, without the use of pre-configured timeouts. In case our platform would have a centralized coordination component we would always know about the exact state of our system. Since there is no such component, the recovery has to rely on the existing metadata as well as the empirically estimated timeouts, to identify failed applications. Correctly identifying the best fitting value for such timeout is a challenging task which is beyond the scope of this work.

Coming back to our example, there still is the chance that the moving operation is completed successfully, even after a very long time. From the Recovery Manager's perspective, we just know that the application is not running or responding correctly. Due to this, the Recovery Manager has to initiate the Nomadic Application's recovery at some point. Even though recovery is a multi-step process, the Recovery Manager eventually has to decide whether it should spawn a new instance somewhere else or not.

In general, we need to distinguish between two different situations, when a recovery is required, depending on the application's location, either Cloud or Fog recovery. We first take a look at those two situations and their individual characteristics and later describe the actual steps taken during their individual corresponding recovery processes.

Cloud recovery

A Nomadic Application running in the Cloud is the easier one to recover. In such a situation it is very likely that simply restarting the affected application, i.e., the affected container restores healthy operation. Restarting the application would preserve the file system stored application state and only discard in-memory state. This may resolve potential problems, however, if not, we can simply spawn a new instance of the application and remove the old. Such instance would cloud be based on a previously created checkpoint. Using a checkpoint, we not only discard the in-memory state but also revert state changes made after the most recently created, healthy checkpoint. In case we have not created any checkpoints yet, we use the initial Nomadic Application's image to create a new container.

Fog recovery

Recovery is more difficult for Nomadic Application residing at some Fog. Furthermore, it is reasonable to assume that most recovery operations will be caused by unhealthy Nomadic Applications located in the Fogs. With Nomadic Applications executed at the Fogs, we have to deal with all the error situations we already considered for the Cloud. Restarting an application in the Fog may resolve some of the problems which have been

caused by an internal program error, nevertheless, we might face a similar situation in future program runs. However, there are a set of additional characteristic error scenarios, which we need to tackle at the Fogs. For instance, with the recovery service located the Cloud, we would not be able to restart the service in case there was a network partition between the Fog and Cloud. In such case, our recovery mechanism would eventually decide to spawn a new instance in the Cloud. This newly spawned instance uses the most recently created checkpoint, as its base image. However, application state and data collected after creating the checkpoint is lost.

Spawning a new instance

Recovery typically does not know whether an application reconnects within some timeout, nor does it provide any mechanism to detect the current state of a running but disconnected, application. As a result, recovery of a running instance is not possible and spawning a new one somewhere else is the last possibility. Doing so results in state and data loss, affecting any changes made after the creation of the checkpoint, which we used for recovery. One of our implementation targets is to minimize effects of recovery. On every Nomadic Application's relocation, we create a checkpoint and push it to the private Docker registry at the Cloud [BC12]. That is why we never lose more than the changes made to the data and state at the most recent compute location.

While restarting an instance has no effects on the platform's metadata, this is not the case with newly spawned instances. We may not assume that a newly spawned instance is the only existing one. Therefore, we want to assign a new instance id to the newly spawned Nomadic Application. Furthermore, similar to application upgrades, the original instance id is marked as deprecated at the Application Evolution. The combination of those two steps makes it possible to distinguish the new and the original instance. Every Nomadic Application repeatedly checks its instance id for deprecation. Due to that, the original instance is able to detect its deprecation and initiate a tear-down, in case it eventually reconnects with the network.

6.5.6 Detecting An Obsolete Application Instance

During application startup, each Nomadic Application is required to check whether it still is the most recent instance of an application. Such an instance might not be the most recent version anymore, in case recovery spawned a new instance.

Therefore, the application calls Application Evolution using its instance id. Application Evolution than is able to check if the supplied instance id was still valid or already marked as deprecated. In case the instance id was deprecated, we know that there must be a newer one. Even for the short possible timeframe there was no newer one, recovery would spawn one. In any case, the checking application has to be destroyed. As a result, the application triggers a tear-down at its Deployment Manager.

6.5.7 Application Tear-Down

An application tear-down happens at a Nomadic Application's end of life but is also an important functionality in combination with the recovery mechanism. Recovery may decide to spawn a new instance in case the original one does not respond or is not reachable anymore. After spawning that new instance, it may be the case that the originally not responding application reconnects with the system, and we eventually end up with two versions of the same application.

Hence, we need some way to detect such situations and means to resolve. In section 6.5.6 we are going to discuss how such a detection can be implemented. However, the detection is just the first step, a Nomadic Application still needs some mechanism to tear itself down. Nomadic Applications are responsible for their own lifecycle and designed to act autonomously. This autonomy includes application tear-down. For executing some steps from the application tear-down process, a Nomadic Application requires support by requesting tear-down from its associated Deployment Manager. The Deployment Manager then takes over, stops the application, frees up resources, i.e., removes the container, and triggers a platform metadata update.

Each of the previously mentioned steps in a Nomadic Applications lifecycle, is essential for enabling them to travel. In the next section we will take a look at how scheduling is implemented and to what extent it involves and influences platform service.

6.6 Target Selection and Scheduling

Algorithm 6.1: Bid on requests at each Fog

Result: Each Fogs available credits have been spent on pending requests

```
1 while simulation is active do
2     if there new or pending requests then
3         creditsPerRequest = CreditsAvailableAtFogPerIteration /
           numberOfNewOrPendingRequests;
4         foreach request do
5             request.increaseCredits(creditsPerRequest);
6         end
7     end
8 end
```

Request scheduling heavily impacts a Nomadic Application's lifecycle, by determining when an application transitions to a different compute location and which location it is going to be. The platform we have implemented supports two different scheduling approaches. A high level introduction to both approaches, the baseline FIFO approach, as well as the auctioning mechanism we proposed, can be found in Section 5.4.

Scheduling implemented at a single Nomadic Application not only impacts the application itself, but also interferes with other applications in several ways. The first and most important aspect such interference happens, is resource usage, at the different compute locations. Fogs are restricted in their capacity, i.e., the amount of Nomadic Applications they are able to operate concurrently is limited. Due to this restriction we would like to achieve a high resource utilization at each compute location. With a higher utilization we can serve more requests, considering the overall system.

While the baseline FIFO approach ignores resources limits at the individual compute locations, we are able to consider those limits using the auctioning, by individually controlling the amount of credits each compute location is able to spend on requests.

Both approaches are implemented using the same platform services and remote assignment lists, stored at the Redis cluster and maintained by the Metadata Manager, Section 6.4.2. However, depending on the scheduling approach, we implemented different queuing logic. While the FIFO implementation simply queues the requests the auction approach is required to continuously reorder the requests depending on the amount of credits spent for the individual one. A higher amount of credits results in an early execution of the request, similar to a priority queue.

6.7 Request Auctioning Implementation

The auctioning implementation is built on top of the baseline FIFO implementation. First we extended the basic application request object by an additional credits counter. This counter is used to track the amount of credits a Fog spends on specific application requests. Whenever a Fog bids for an application either a new request is created and added to the request queue, or an existing one is updated. Either way, the credits counter for the concerned request object is increased by the amount spent. The second part we changed is the actual queue implementation. We replaced the FIFO queue by a priority queue utilizing the credits spent counter as the ordering criterion. Whenever an application wants to move, it picks the highest priority target, i.e., the requests with the highest credits spent counter, from the queue. In case there were multiple requests with equal credits spent counter values, we choose an arbitrary one. In our bidding implementation a Fog is able to bid every minute using its individual amount of credits. A fog is not able to save up any credits, hence it is desirable to spend all available every minute. In our implementation a Fog simply divides its credits by the amount of open application requests and therefore evenly distributes them across all of them. During development we tried different ways to distribute the amount of credits, nevertheless, this rather simplistic approach turned out to be amongst the most promising ones. However, future work could apply more elaborated approaches by incorporating additional knowledge.

6.8 Technical Challenges To Solve

The technology stack we have chosen, while being state-of-the-art and production-grade, incurred several challenges and limitations, when implementing the platform, the Nomadic Applications and the scheduling logics. This section reports on both technical and architectural challenges we faced during the development.

6.8.1 Passing Parameters to Nomadic Applications

One of the key characteristics in this system is the autonomous nature of Nomadic Applications. There is no centralized coordination or configuration available. The only centralized part of the system is the metadata stored at the Metadata Manager and the registration with the Eureka service registry. Therefore, every application requires some basic knowledge to be able to access that metadata and other parts of the system.

The solution we have implemented uses environment variables to pass parameters such as the Nomadic Application's instance id or the eureka service URL. This is possible because Docker containers are isolated environments with configuration applied to one of the containers not affecting any other container. In a nutshell, environment variables set at one container are available only within that one specific container and won't affect another one. Since each application runs within its own Docker container, they will not affect each other. When using Spring, accessing those environment variables from an application is an easy and straight forward task. Spring encourages the use of environment variables to configure parts of the framework.

In the remainder of this section we are going to discuss some of the most important configuration values used in our system. The first two are essential to every application in our system, while the later are used by Nomadic Applications only.

Eureka Service URL: One very important configuration value each of our framework's application requires, is the Eureka service URL. Each of the applications uses this URL to register with the service registry. Additionally, this registration provides them with the necessary means to locate other services.

Eureka Client IP: Besides the service URL, we also need the client's IP address. This client IP is the IP address other applications can use to access it.

An attentive reader may analyze why such a setting is required. This is justified by Docker providing internal IP addresses for the individual containers. In the default Eureka client configuration, we would pass those internal IP addresses to the Eureka server. The use of internal IP addresses would render the service locator useless, since applications running at different Docker hosts, would not be able to access each other.

Metadata and Instance Id: The metadata and instance id are important configuration values, required for the identification of Nomadic Applications. Identifying containers

is another technical challenge we consider in the next section. An in details description of the instance id can be found in Section 6.4.2.

6.8.2 Identifying the Host Container

When an application calls its responsible Deployment Manager, the container id is very important. Without the container id, the Deployment Manager would not be able to know which container requests its help.

While passing parameters using environment variables is effective for values we know when creating the containers, it is of no use for values we do not know the during container creation. One example is the previously mentioned, auto-generated Docker container id. Our solution to that problem is a rather easy one. Using default configuration, the Docker container's hostname is equal to the beginning of its id. Hence, a Nomadic Application uses its hostname for identification at the Deployment Manager.

6.9 Similarities and Differences when compared to Kubernetes

The described architecture is in many aspects similar to well-established container orchestrators like Kubernetes[13] from Google. Figure 6.9 shows an exemplary Kubernetes cluster with two worker nodes. In Kubernetes the master node is responsible for the node and pod deployment orchestration. The term pod describes the smallest deployable and scalable unit of work and can consist out of a single or multiple containers. In the case of Nomadic Applications, a single Nomadic Application would be deployed as a pod with a single container inside that pod.

The master node contains two essential components we can also find in the Nomadic Applications's architecture, namely the controller and scheduler. In terms of the Nomadic Applications the controller is reponsible for the tasks of the Census and Recovery (Section 6.4.3), while the scheduler decides where a pod is deployed. Additionally, the mater node utilizes etcd, a key value store, similar to what we implemented with the Metadata Manager (Section 6.4.2). The kublet container is part of every node and responsible for the management of the deployed containers, similar to our Deployment Manager (Section 6.4.1). The kube-proxy is responsible for establishing and forwarding connections to the deployed containers within the pods. In the solution we have implemented we replaced such proxy system by utilizing the Eureka services discovery and communicating directly with the deployed applications.

[13] https://kubernetes.io

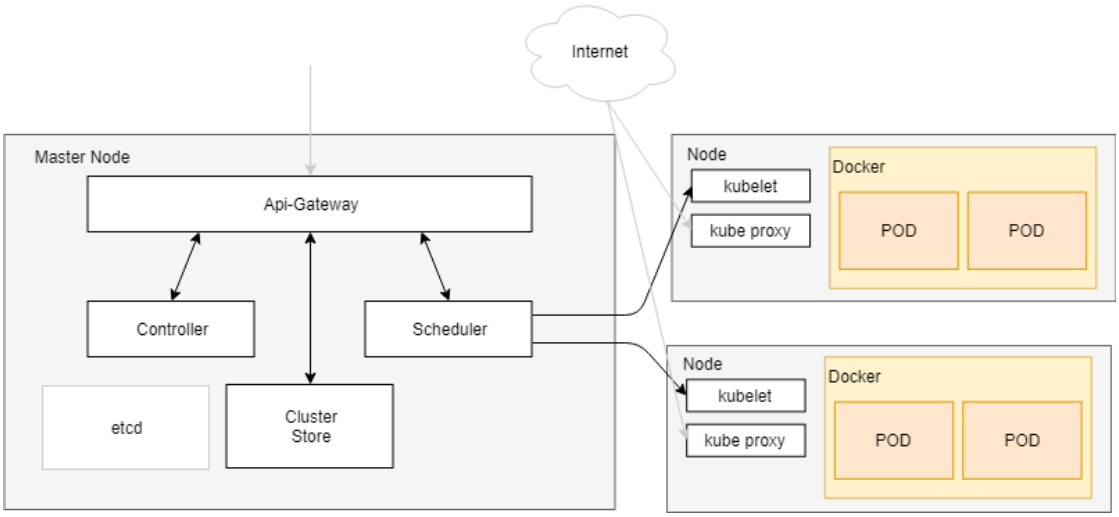


Figure 6.9: Kubernetes architecture

CHAPTER 7

Evaluation

In this chapter we present the evaluation setup. For the evaluation of the implemented system we have prepared three different scenarios. Each scenario is executed using the originally proposed FIFO scheduling and the more advanced auction based one. Furthermore, we run each scenario simulation three times to foster insights concerning potentially varying evaluation results. The goal of the evaluation is to examine the performance of the auctioning based scheduling in comparison to FIFO one.

First we are going to take a look at the scenarios in Section 7.1, followed by a description of the evaluation environment in Section 7.2 and the required software changes as well as newly introduced components in Section 7.3. Finally, in section 7.4, we are going to present the evaluation results.

7.1 Scenarios

The scenarios we have chosen for evaluation are built upon common use-cases based on the motivational scenario from Section 3.1. We implement the following three, namely:

- Basic Travel,
- Application Upgrade and
- Network Partition and Recovery.

These three scenarios try to cover different use-cases, while at the same time, we keep them comparable by implementing the Application Upgrade and the Network Partition and Recovery scenarios as extension to the Basic Travel scenario. Although we try to cover several edge cases, the implemented scenarios perform valid actions only and will not send any malicious requests or spam the application request queues.

The first one, Basic Travel, focuses on efficiency in a perfectly working network, without major incidents or edge cases to consider. We will take a more precise look at this scenario in Section 7.1.1. The second scenario introduces two application upgrades. Finally, the third scenario requires recovery to intervene, due to two network partitions. When designing a scenario suitable for the manufacturing domain, with multiple factories and each factory being represented by a Fog cell, we have to consider temporary network partitions. Such network partitions originate for instance from Internet connectivity problems or temporary internal network infrastructure problems at some manufacturing site and potential results in connectivity problems with a Fog being not reachable from the rest of the network. These network partitions may range from very short ones, resulting in a few dropped packages only, to longer lasting ones, with up to multiple hours to restore connectivity.

Each of our scenario consist of 10 Nomadic Applications, traveling amongst 5 Fog cells, i.e., 5 manufacturing sites in reference to the manufacturing domain, and the Cloud. The requests will be generated throughout an overall duration of 120 minutes. After the 120 minutes we will not generate any new requests, however, existing ones will be processed till all requests have been served. This results in an overall execution time beyond the 120 minutes. Additionally, we include the cleanup time as part of the scenario. This is the time required for the applications to move back to the Cloud and successfully complete the teardown procedure.

7.1.1 Basic Travel

The first scenario, referred to as Basic Travel scenario, simulates a perfectly stable and reliable network. This scenario enables an easier comparison of the two implemented scheduling algorithms, including the scheduling algorithm's impact on application requests throughout a simulation run. We minimize external effects for all simulation runs by providing a clean environment for each run. In order to do so, we recreate all virtual machines from a previously created snapshot. Furthermore, specific to the Basic Travel scenario, we ensure a deterministic execution behavior, by avoiding possible error states. Such error states will be dealt with in the other two scenarios.

All three implemented scenarios are based on the same application requesting pattern, based on the matrix shown in Table 7.1. The requesting pattern has been chosen with the intention to include several scenario relevant difficulties, ranging from different Fog capacities over request peaks due to concurrent requests. These peaks concern both the Fogs due to their capacity limits and the Nomadic Applications due to their inherent single instance nature. Each row in the table represents a Fog, with the cells in the 'Nomadic Application' columns containing the number of minutes between distinct requests, for the respective Nomadic Application. The last row contains the total amount of requests per hour for the individual Nomadic Applications. Based on that row we can conclude that Nomadic Application 1 has to face the highest workload with approximately 1 request every 2 minutes, on average. Additionally, this application is requested with a high frequency by Fog A, which has the lowest capacity amongst our Fog cells. This

Fog	Capacity	Nomadic Application										Requests / h (aprox.)
		0	1	2	3	4	5	6	7	8	9	
A	1	-	5	10	-	-	-	20	5	-	30	35
B	2	20	30	10	8	-	10	-	-	8	-	31
C	3	-	5	-	-	8	-	20	-	-	7	31
D	4	15	30	10	-	-	5	10	15	-	9	40
E	5	5	-	-	5	5	-	-	-	5	15	52
Request / h		19	28	18	19.5	19.5	18	12	16	19.5	21.2	

Table 7.1: Nomadic Application requests every t minutes

combination might result in higher wait times for Nomadic Application 1 during our simulation runs. On the other hand, there is the Nomadic Application 6 with only 1 request every 5 minutes, however this application is requested at the same times as it is the case with the previously discussed application 1. The request matrix has been designed to create concurrent requests at certain times throughout the simulation. These concurrent requests lead to potential bottlenecks concerning the individual Fogs capacity. Due to these bottlenecks scheduling will have a higher impact on efficiency. Furthermore, with individual request patterns for the applications, we gain different peaks in workload our system has to cope with. With only 18 requests per hour, Nomadic Application 2 is part of the less active ones, yet requests for it are designed to happen concurrently. The smallest Fogs A and B, in reference to their computational resources, are amongst the competitors for requesting Nomadic Application 2. This is especially important for Fog A as it is computational resources support a single application execution at a time only. The Fog concurrently requests at least 2 but up to 5 applications every 5 minutes which eventually results in request queue congestion.

7.1.2 Application Upgrade

The Application Upgrade scenario is based on the previously mentioned Basic Travel scenario, but additionally introduces three application upgrades. The scenario performs upgrades on the Nomadic Applications 0 and 2.

The first upgrade for Nomadic Application 0 is deployed 4 minutes after simulation start. The second upgrade for both applications, i.e., Nomadic Application 0 and 2, is deployed after an additional 8 minutes. In this second upgrade phase, the previously upgraded Nomadic Application 0 is upgraded again.

Based on the request matrix, it is safe to assume that the first upgrade for Nomadic Application 0 is not going to intervene with any application requests, due to it being requested 5 minutes after simulation start, first. However, this is not going to be the case anymore at the second upgrade phase, i.e., approximately 12 minutes after simulation start. At that time, it is safe to assume that the applications has already been requested several times, based on the implemented request matrix. Any open request at that time is

potentially affected by the scheduled upgrade and potentially results in a delayed request processing.

7.1.3 Network Partitions

Like the previous two, the Network Partitions scenario implements the same request matrix. Specific to this scenario we introduced two network partitions. The first one occurs 3 minutes after simulation start and separates Fog E from the rest of the network. This network partition lasts only 2 minutes and potentially affects the applications 0, 3, 4 and 8. Due to the very short time after simulation start and the rather small amount of time it lasts, this network partition will not affect simulation much. An additional 10 minutes later, i.e., about 15 minutes after simulation start, we separate Fog B from the rest of the network. This second network partition lasts 6 minutes and potentially result in a delay for the applications 0, 2, 5 and 8. Application 1 should not be directly affected as there are no requests for this application from the affected Fogs. Nevertheless, due to Fog resource limits these lists of affected applications is not an exclusive one. Other applications not operating on the disconnected Fog are likely affected by the network partition, due to delays propagating in our network.

7.2 Experimental Setup

The experimental environment is built using two physical hosts and several virtual machines running Ubuntu 16.04. Each of the physical hosts is equipped with 32 GB of main memory, backed by an SSD drive and a current 5th generation quad core Intel CPU.

The virtual machines are executed using Hyper-V with the dynamic memory feature enabled. This allows dynamic provisioning of memory to the virtual machines, a very useful feature, due to the changing demand in memory. Owing to the moving behavior of our Nomadic Applications, we typically need a fixed amount of memory distributed amongst the virtual machines, linearly increasing depending on the amount of Nomadic Applications deployed. When an application moves from one virtual machine to another, the memory demand for the affected application on the origin machine is release and can get assigned to the target one where demand typically increases similar to the previously freed amount. This might culminate in a single virtual machine operating all the applications and as a result requiring most of the memory while the others require very little memory at that time. Due to our scenario limiting the amount of concurrent applications running in each Fog, this can only be the case for the virtual machine simulating the Cloud.

Additionally, we use a third machine for the infrastructure services, e.g., Redis, Mongo DB, Eureka and the ELK stack. This infrastructure machine is running at some remote location. Although, upload bandwidth is limited with that machines, this location is mostly used for the storage of simulation results and logging output during development.

	Machine	min. memory	max. memory	CPU cores	Network
Cloud	A	4 GB	13 GB	3	1 Gbit
Fog A	A	2 GB	4 GB	2	10 Mbit
Fog B	A	2 GB	4 GB	2	10 Mbit
Fog C	A	2 GB	4 GB	2	10 Mbit
Fog D	B	2 GB	7 GB	2	10 Mbit
Fog E	B	2 GB	7 GB	2	10 Mbit
Cloud Services	B	4 GB	8 GB	4	1 Gbit
Infrastructure	C	10 GB	20 GB	2	5 Mbit

Table 7.2: Evaluation environment

Another very useful feature provided by Hyper-V is the ability to individually limit network bandwidth for virtual machine interfaces. This feature offers a very easy way to control the network bandwidth for Fog cells.

Table 7.2 provides an overview of the system configuration. In addition to the Cloud virtual machine on physical machine A, we run some services on machine B due to main memory reasons.

7.3 Simulation Master

In addition to the infrastructure setup we need an orchestrating service, required by the simulation only and subsequently referred to as Simulation Master. The Simulation Master is responsible for executing our scenarios. It controls the environmental conditions including network connectivity, resource availability and the application requests.

Unlike the other services implemented, the Simulation Master is a stateful service. Nevertheless, the implementation itself is similar to the other platform services. Like the platform services, it uses all the provided core platform functions as well as the Eureka service registry. Due to this, we can easily rely on the existing infrastructure, as for instance, the Feign client modules. These Feign clients enable access to other services, i.e., any platform service or Nomadic Application, without additional implementation effort. Another difference, when comparing the Simulation Master to the rest of the system, attributes to the Simulation Master's nature as an orchestrator for the entire system, with full knowledge about the simulation state. In order to implement such an orchestrator, this service is connected to all applications and is able to access all platform services and Nomadic Applications. The Simulation Master also provides public interfaces for the Nomadic Applications as well as the Deployment Managers for accessing current scenario control information. This control information is for instance used to control the current network connectivity state. Besides that, it also provides information about the currently available amount of resources at each Fog cell. Controlling such information through the Simulation Master eases the scenario implementation without restricting generality. Moreover, the control interfaces exposed by the Simulation Master, provide an API for

publishing application feedback. This is especially useful to collect all the generated results from the various services involved in our systems implementation. The results are stored in a Mongo Db instance, for later evaluation and possible additional processing.

In the remainder of this section we take a look at Simulation Master's implementation, including the ability to manage multiple concurrent simulation tracks, the Fog request generation and the result collection and aggregation.

7.3.1 Tracks and Tasks

For the scenario execution we decided to implement a system using multiple concurrent tracks, with each track consisting of a list of subsequently executed tasks, similar to a workflow engine. The subsequently executed tasks offer a way to handle some internal state within the tracks, these tasks belong to. One prominent example for such a state is the platform generated, automatically assigned instance id, each Nomadic Application has. It is retrieved during container startup and stored for subsequent usage by other tasks. Additionally, we require knowledge about the ids for efficient evaluation of the generated data.

The implemented task execution engine enables retrying tasks till they have been executed successfully. This is required due to tasks typically depending on their previous executed ancestors. A task is typically delayed till its predecessor task is executed successfully. With the ability to handle concurrent tracks, we gain parallel task execution. This parallelism is utilized by implementing a dedicated track for each Nomadic Application. Additionally, we have tracks for the implementation of incidents, e.g., the network partition or the application upgrades. In some cases, we are required to synchronize state within two or more tracks at certain points in time. This can be done using commonly accessed objects, in a way similar to the well-known concept of shared memory. For the creation of the tracks themselves, we have created a builder, exposing a fluent API.

7.3.2 Request Generation

The second part of the Simulation Master, namely the request generation, is implemented as a scheduled task. During scenario creation we define time intervals requests will be generated in, i.e., the request execution matrix shown in table 7.1. A dedicated track will continuously check whether we need to perform new application requests. Moreover, this track includes the required bidding logic, simulating the Fog's behavior based on the proposed system in Section 5.4.2. After an initial request, we will subsequently increase the Fogs' bids for incomplete requests. In case an entire Fog cell is marked as disconnected it obviously will not be able to increase its bids or send any requests at all.

7.3.3 Collecting Execution Results

An important aspect of the Simulation Master is its ability to collect simulation results. The server offers an API for other applications to submit simulation feedback. The

Action	Description
Start	The Nomadic Application's container has been started.
Move	The reporting Deployment Manager has packaged the Nomadic Application and handed over to the target compute location's Deployment Manager.
Upgrade	A Nomadic Application has been upgraded to a newer version. It is able to process requests again.
Recover	A Nomadic Application has been recovered. Either restarted or redeployed. Possibly existing previous versions have been marked as obsolete.
Remove	A Nomadic Application has requested removal at its currently associated Deployment Manager.

Table 7.3: Feedback collected by the Deployment Manager

resulting feedback, collected at the Simulation Master is aggregated, pre-processed and later stored in the Mongo DB. Table 7.3 depicts the feedback generated and collected by the platform, namely the Deployment Managers. Every feedback event refers to a unique Nomadic Application using the instance id and additionally contains information about the generating event compute location, the action performed and event specific parameters such as the target compute location for a move action. Nomadic applications themselves can not provide any additional insight, platform services have not been already aware of. This is because every Nomadic Application requires the support of various platform services, in order to perform any operation, relevant for the simulation run. With the platform services already sending feedback to the Simulation Master we collect all the information we are able to, i.e., the information provided by the Nomadic Applications has already been provided through platform services. Feedback can be submitted fully independent from simulated network partitions.

To minimize the impact of side effects such as, memory and disk fragmentation, existing pre-fetched docker images or increased database storage, we restart all platform services and clear data storage after each simulation run. Doing so should help to avoid possible impacts of previous runs. During simulation the involved physical machines as well as the network infrastructure is used for the simulation only.

7.4 Results

The criterion used for the comparison of the simulation runs is the amount of time required to fulfill an application request by a Fog. In the remainder of this work we refer to this as wait time. Requests with lower wait time have been served faster than their counterparts with higher wait times. Therefore, it is desirable to minimize wait time for the individual request. We define wait time as the duration between a Fog requesting a Nomadic Application and the moment the requested application initiates its moving towards the requesting Fog.

Hence every request additionally includes a constant platform management overhead, composed of the checkpoint creation, push and pull (section 6.5.2) and the time required for the application start at the new location. This additional time is not included in the measurements and numbers discussed in this section, nevertheless we still want to give a short summary on the expected additional overhead. Due to us relying on Docker commits we only transfer one delta layer to the registry. This results in approximately 1-2 seconds overhead, based on the measurements taken in our simulation runs. Transfer time for the created checkpoints is tightly coupled with the size of the Nomadic Application's state. Any change in state size is especially visible due to the rather low configured network bandwidth at the Fog cells. Besides transfer time, the individual Nomadic Applications startup time is also considered part of application work, at the respective Fog and therefore neglected during the subsequent evaluation.

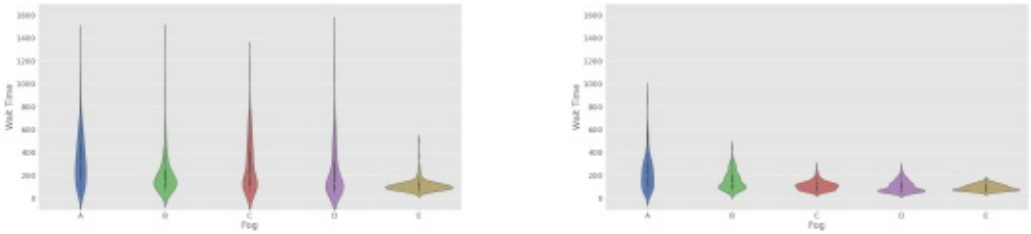
Every Nomadic Application is pulled during first time startup at the location and kept for the entire simulation run. Such initial application pull requires between 30 seconds and 1 minute and consists of the application layer only, since the base image layers are shared with the Deployment Manager and therefore already stored at the compute locations. Any subsequent request for the same location requires the application state layer only. Our evaluation Nomadic Applications is implemented using Spring Boot and requires an additional 15-20 seconds for application startup, again considered part of the applications work at the Fog cells. However, startup time may effect the wait time due to application upgrades which happen prior to any work in the Fogs.

During each simulation run a total of 1068 distinct requests have been processed and measured. These collected measurements have been analyzed individually for each scenario type and visualized using three different charts. To support a more stable comparison we combine the three individual simulation runs per simulation type and therefore reduce the impact of individual single measurement when compared to the overall simulation. The Figures 7.1, 7.4 and 7.7 depict a violin plot of the wait time concerning the Fog compute locations, while Figures 7.2, 7.5 and 7.8 visualize the distribution of wait times for the individual Nomadic Applications. The scatter plots in Figures 7.3, 7.6 and 7.9 illustrate the trend in request wait time over simulation run time. Additionally, a regression line per Fog location has been added to support visualization. This regression line is intended as support line only and is not suitable for any conclusion concerning future development of wait times.

Table 7.4 shows basic statistical metrics in order to support comparison in the following chapter.

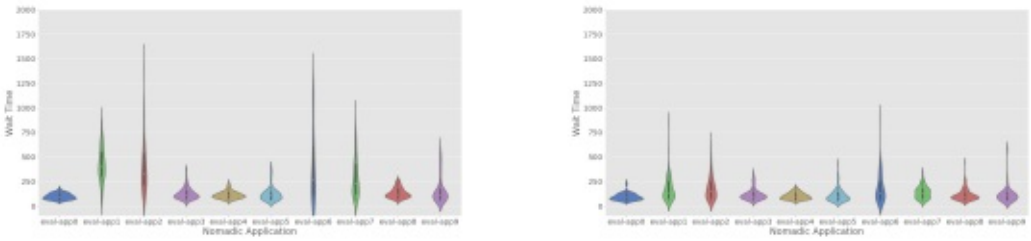
7.5 Statistical Tests

As part of our evaluation we want to ensure simulation runs for each scenario type are statistically significantly different when comparing FIFO and Auction based scheduling. In order to verify this, we apply a statistical hypothesis test. Due to the lack of knowledge concerning the measurement data's underlying distribution we decided to apply



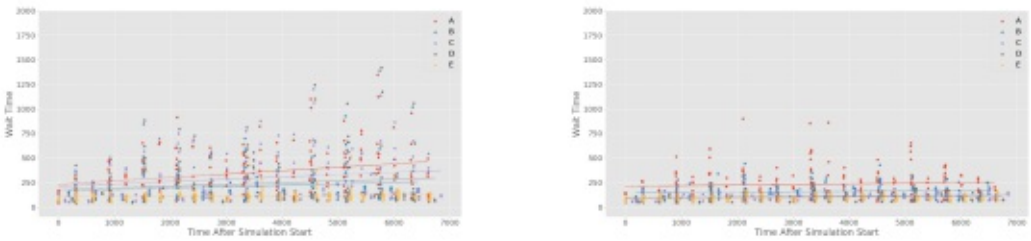
FIFO based scheduling | Auction based scheduling

Figure 7.1: Basic Travel: wait time per Fog



FIFO based scheduling | Auction based scheduling

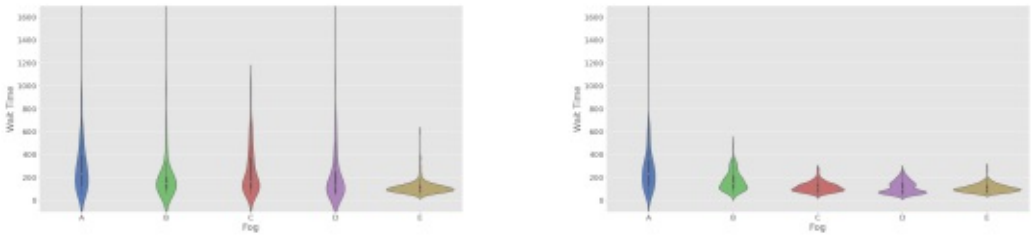
Figure 7.2: Basic Travel: wait time per Instance



FIFO based scheduling | Auction based scheduling

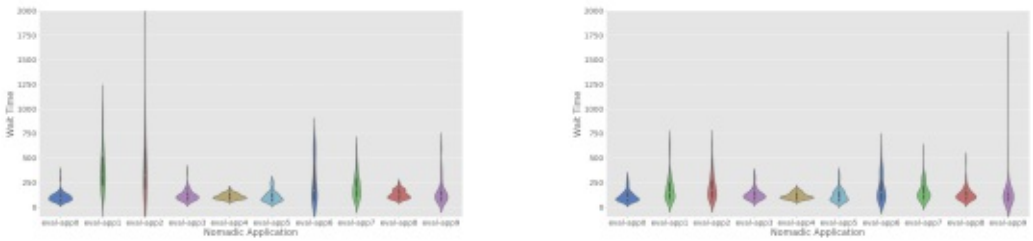
Figure 7.3: Basic Travel: scatter plot with wait times and time after simulation start

a nonparametric test. Hence, we applied the Mann–Whitney U test , a nonparametric test to evaluate whether two independent samples are taken from populations with different median values [She07]. Unlike a t-test this test does not require normal distribution but only requires the samples to be independent, which is the case for our measurements. A significant Mann–Whitney U test indicates significant difference between two sample medians. Due to this, we can conclude a high likelihood that the sample represented populations have different median values.



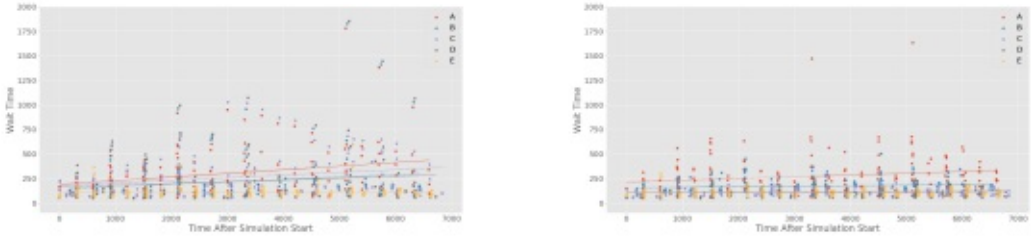
FIFO based scheduling | Auction based scheduling

Figure 7.4: Application Upgrade: wait time per Fog



FIFO based scheduling | Auction based scheduling

Figure 7.5: Application Upgrade: wait time per Instance

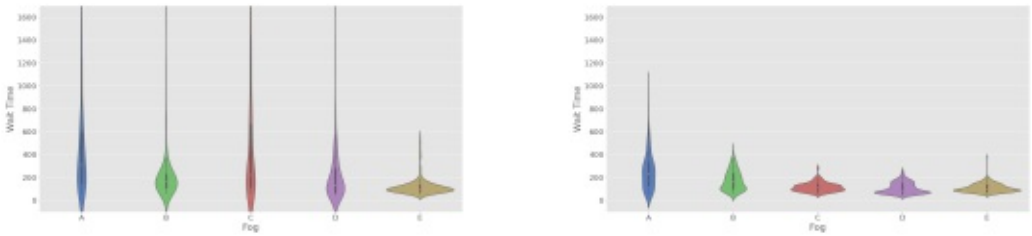


FIFO based scheduling | Auction based scheduling

Figure 7.6: Application Upgrade: scatter plot with wait times and time after simulation start

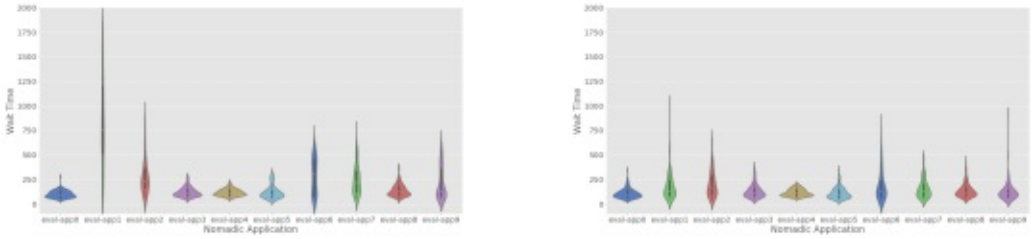
Evaluation Hypothesis

- *H_0: The measurements of wait times for both FIFO and Auction based scheduling are the taken from the same distribution.*
- *H_1: The average wait time of requests scheduled using auction-based scheduling is smaller than those scheduled using FIFO.*



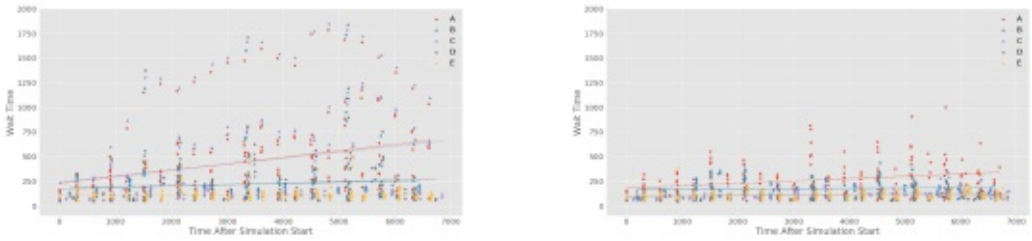
FIFO based scheduling | Auction based scheduling

Figure 7.7: Application Recovery: wait time per Fog



FIFO based scheduling | Auction based scheduling

Figure 7.8: Application Recovery: wait time per Instance



FIFO based scheduling | Auction based scheduling

Figure 7.9: Application Recovery: scatter plot with wait times and time after simulation start

- n = 1068 (3 x 356)
- $\alpha = 0.05$

	min	max	median	mean	σ
Basic (FIFO)	45	515	103	114	57.6
Basic (Auction)	46	171	88	94	26.8
Upgrade (FIFO)	52	604	101	104.4	57.6
Upgrade (Auction)	56	301	101	106	34.7
Recovery (FIFO)	53	561	98	117.7	69.7
Recovery (Auction)	53	379	99	108.5	36.6

Table 7.4: Evaluation metrics

	U	p	Result
Basic	408871.5	4.7920151974969055e-30	reject H_0
Upgrade	409459.0	7.352632074079756e-07	reject H_0
Recovery	452851.0	8.489849471813285e-17	reject H_0

Table 7.5: Test statistic evaluation

7.5.1 Evaluation Result

As we can see in Table 7.4 measurements taken from a scenario with Auction based scheduling have a much lower standard deviation than their FIFO based counterparts. This is also the case for the maximum wait time throughout the simulation runs. However, depending on the scenario the Auction based approach is not necessarily faster when considering the average wait time. The test statistic results depicted in Table 7.5 led to the rejection of H_0 for all three scenarios. Hence we can conclude that the distributions of the FIFO based and the auctioning based scheduling measurements are different.

Basic Travel The Basic Travel scenario as described in Section 7.1.1 has had the highest performance gain, out of the three scenarios. It is especially noticeable that not only the maximum wait time has been reduced from 515 to 171 seconds but also the average wait time from 114 to 94 seconds. This can be explained due to the possible out of order execution for requests when using an Auction based approach. With such an out of order execution we can reach a higher degree of concurrency and respectively a higher utilization of Fog resources in our system. This is mainly caused by the applications doing work as soon as possible. Such an approach keeps the request queues shorter, however it not necessarily results in lower wait time for all the requests.

Application Upgrade Similar to the Basic Travel scenario, we can see an improved maximum wait time for the Application Upgrade scenario, Section 7.1.2. Unlike the Basic Travel scenario, it was not possible to improve the average/median wait time. When comparing the Basic and the Upgrade scenario we can see that performing an application upgrade results in a lower average wait time, even though the maximum wait time increases.

Recovery In the Recovery scenario, Section 7.1.3, we are again able to reduce maximum wait times but cannot improve median wait time. However, this scenario profits from a lower average wait time.

Due to the drastic improvement in the maximum wait time we still consider the Auction based approach better than the FIFO one. Owing to the Mann–Whitney U test we know distribution is significantly different when compared to the FIFO implementation.

Outliers in measurements The amount of outliers in the Auction based measurements has been reduced throughout all three scenarios as we can see from the scatter plots in Figures 7.3, 7.6 and 7.9. While FIFO scheduling results in existing outliers for all Fogs except E, Auctioning is able to reduce the existence of outliers to Fog A. The non-existent outliers for Fog E may be explained by both, the high capacity of 5 concurrent applications and due to the participating applications not being requested by the rather busy Fog A. Additionally, we can see from the scatter plots that wait times increases stronger using FIFO scheduling, throughout the simulation runs. This is especially noticeable for the Recovery scenario.

The Auction based approach results in a different, more efficient scheduling order. Due to this changed scheduling order we have a higher resource utilization at the Fog cells, which results in lower wait times for the individual requests and a higher degree of concurrency.

7.6 Summary

One common observation we can make from all three scenarios is a lower wait time for the average application requests when using auction based scheduling. In general, we can see that any extremes, i.e., especially long wait times, have been drastically reduced in occurrence. Moreover, the overall simulation time has been reduced as there is less congestion running upstream at the end of the simulations. This is particularly true for the lower capacity Fogs, in combination with the more frequently requested applications. An auction based simulation therefore enables a higher resource utilization at the participating Fogs and respectively the requested applications.

7.6.1 Using Microservices over Kubernetes

When comparing our platform with Kubernetes we can find many similarities (Section 6.9), however, some specific aspects are done differently. One prominent example can be found in the higher incorporation of domain specific knowledge, both within the Nomadic Applications but also within the Fog cells themselves. The implementation we proposed allows the application of detailed domain knowledge, when bidding for requests. Due to this additional knowledge, Fogs are capable of applying strategic bids for applications, allowing them to individually maximize their benefit. By utilizing such an auctioning approach in combination with the Cloud deployed request queues, we are able to combine

extensive domain knowledge with full system knowledge about compute locations and surrounding applications.

CHAPTER 8

Conclusion

With the progresses in the implementation of IoT and its ubiquitous application in Industry 4.0, an overwhelming amount of data requires processing. More recently, a frequently applied approach tries to pool a broad range of distributed computational resources into logical units, so called Fogs, enabling scalability while preserving close proximity to data sources. Due to the rising prevalence of Fogs in the IoT landscape, they have been established as deployment model for novel Nomadic Applications, where computational resources are provided to self-aware applications operating close to data. Such environments allow to deploy applications to locations with demand, while at the same time they provide the flexibility to shift resources between the Cloud at high load times and Fogs when close proximity is beneficial.

In the context of such systems, Hochreiner et al. [HVS+17] identified stateful applications as particularly challenging, since they require a consistent global view to prevent concurrent executions as well as special means to recover state in case of faults. Apart from that, they specifically identified the request scheduling as an open research topic. They suggested to apply an auctioning based scheduling approach, where Fogs bid on application requests in a roundly manner, preventing a single Fog from occupying an application by spamming the request queue.

To this end, we conceptualized an auctioning based mechanism, which ensures a fair and efficient request scheduling across all Fogs by incorporating information such as capacities as well as application request patterns. With our approach a Fog is able to place strategic bids on certain applications which allows them to prioritize work by increasing bids on highly requested applications while at the same time placing lower bids on less frequently requested ones. To effectively evaluate the scheduling approach, we designed and implemented a generalized framework to study the runtime behaviour of stateful Nomadic Applications in simulated real world scenarios. These scenarios are taken from the manufacturing domain where applications travel independently amongst compute locations in Fog cells and the Cloud, incorporating common challenges such as

application upgrades and failures. Most importantly we provide means to recover from defective states based on a check-pointing mechanism which snapshot the application's state.

While such platform offers various aspects subject to optimisations, such as the time required to detect faulty applications in need for recovery, this work focuses on application request efficiency. To this end our evaluation compares the request wait times, i.e., the amount of time between a Fog's initial request and the processing. In a series of experiments, covered in Section 7.1, we studied the behaviour of our auctioning technique compared to the baseline FIFO implementation, where 10 Nomadic Applications travel amongst 5 different Fogs with certain recurring application requirements. The results of the simulations, depicted in Section 7.4, show that the auction-based approach generally performs better compared to the FIFO approach. We were able to significantly reduce the number of outliers in request wait times, while at the same time we accomplished to reduce their average. Especially in Fog environments with limited resources, the auction-based scheduling performs particularly well, as it achieves higher resource utilization and is able to serve overall more requests than the baseline approach, given the same amount of time. The higher utilization attributes to the level of control a Fog has on the bidding processes by varying the amount of credits spent on individual application bids. We found that Fogs can utilize domain knowledge to effectively perform strategic bids while at the same time we restrain from adding complexity to the individual Nomadic Application.

Finally, our work on Nomadic Applications has shown several additional less quantifiable benefits. First, a Nomadic Application platform can manage the complexity inherent to diverse edge computing environments by utilizing established virtualization technologies such as Docker and implementing communication through platform agnostic REST interfaces. Second, it has proven beneficial to shift large parts of the individual Nomadic Application's complexity to the platform in a service oriented architecture, as for instance the request queue management or the recovery mechanism. Such platform capabilities enable rapid implementation of simple but powerful applications focusing on business logic rather than infrastructure aspects.

Summarizing we can say, such a platform allows a very diverse compute environment with applications of various kinds. It not only enables developers to choose from a less restricted range of technologies, with the opportunity to choose the appropriate technology for the individual problem to solve, but also allows applications to decide where and when to move.

8.1 Future Work

While this work has successfully demonstrated advantages of an auction based scheduling approach in a Nomadic Application platform it has compiled a list of topics potential future work could focus on. As part of the evaluation we have performed only basic parameter tuning for the auctioning mechanism and the used decision models thus, one

stream of future work could focus on further improving the auctioning by either varying the amount of credits each Fog is capable to spend or refining the features included in decision making. Even more, scheduling should have a more elaborated model for considering resource limits at the individual Fog cell. Such limits typically vary over time and are influenced by other Nomadic Applications. Common optimisation techniques like Mixed-Integer Linear programming could be used to optimise an individual Fog cells utilization of resources. We suspect that such an approach should also consider the estimated execution time for a Nomadic Application. In addition to the model and bidding logic improvements, forecasting models could be applied to predict runtime behaviour and resource demands. Such predictions can be used as additional features to fine tune the discussed bidding logic.

In the scope of this work we decided to use some centralized components in the auctioning implementation while at the same time the actual bidding is performed decentralized. Potential future work could elaborate on the possible performance impacts, trade-offs and opportunities when implementing a fully centralized scheduling.

Besides all the possible advances in the scheduling aspects of this work, future work could try to improve the recovery process, since recovery behaviour has a significant performance impact on the overall performance, either caused by the delay in error detection or the amount of time spent during the actual recovery. One could start by refining the error model by tuning the heartbeat timeout in order to improve the error detection. Selecting a reasonable timeout for application travel not only depends on the application itself but is also influenced by both, the network infrastructure between the Fog cells, the Cloud and the hardware used at the compute locations.

List of Figures

5.1 Nomadic Application Infrastructure 16
5.2 21

6.1 Services operated in the Cloud. 27
6.2 Services operated in the Fogs. 28
6.3 Service discovery 30
6.4 Starting a new Nomadic Application 36
6.5 Fog requests a Nomadic Application 37
6.6 Steps executed during a Nomadic Application move 40
6.7 Steps executed at packaging process 41
6.8 Steps executed at an application upgrade 44
6.9 Kubernetes architecture 51

7.1 Basic Travel: wait time per Fog 61
7.2 Basic Travel: wait time per Instance 61
7.3 Basic Travel: scatter plot with wait times and time after simulation start . 61
7.4 Application Upgrade: wait time per Fog 62
7.5 Application Upgrade: wait time per Instance 62
7.6 Application Upgrade: scatter plot with wait times and time after simulation start 62
7.7 Application Recovery: wait time per Fog 63
7.8 Application Recovery: wait time per Instance 63
7.9 Application Recovery: scatter plot with wait times and time after simulation start 63

List of Tables

6.1 Container and Image Metadata 34

7.1 Nomadic Application requests every t minutes 55
7.2 Evaluation environment 57
7.3 Feedback collected by the Deployment Manager 59
7.4 Evaluation metrics 64
7.5 Test statistic evaluation 64

List of Algorithms

6.1 Bid on requests at each Fog 47

Bibliography

[AFGM+15] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys and Tutorials*, 17(4):2347–2376, 2015.

[AGL+12] Owen Arden, Michael D George, Jed Liu, K Vikram, Aslan Askarov, and Andrew C Myers. Sharing Mobile Code Securely with Information Flow Control. *S{&}P*, pages 191–205, 2012.

[Ama17a] Amazon. Amazon glacier – aws, June 2017. Last accessed: 14.07.2018.

[Ama17b] Amazon. Aws iot – amazon web services, June 2017. Last accessed: 21.06.2017.

[BC12] Anju Bala and Inderveer Chana. Fault Tolerance-Challenges, Techniques and Implementation in Cloud Computing. *International Journal of Computer Science Issues*, 9(1):288–293, 2012.

[BCL+16] Marc Barcelo, Alejandro Correa, Jaime Llorca, Antonia M. Tulino, Jose Lopez Vicario, and Antoni Morell. IoT-Cloud Service Optimization in Next Generation Smart Environments. *IEEE Journal on Selected Areas in Communications*, 34(12):4077–4090, 2016.

[BMP+10] Anne Benoit, Loris Marchal, Jean François Pineau, Yves Robert, and Frédéric Vivien. Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. *IEEE Transactions on Computers*, 59(2):202–217, 2010.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and Its Role in the Internet of Things. *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

[BPSN95] S Bakhtiari, J Pieprzyk, and R Safavi-Naini. Cryptographic hash functions: A survey. *Centre for Computer Security ...*, pages 1–26, 1995.

[BZ17] Paolo Bellavista and Alessandro Zanni. Feasibility of fog computing deployment based on docker containerization over RaspberryPi. *ACM International Conference Proceeding Series*, 2017.

[CHK94] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile agents: Are they a good idea? *Mobile Object Systems Towards the Programmable Internet*, pages 25–45, 1994.

[CNR+07] T. S. Chandrashekar, Y. Narahari, Charles H. Rosa, Devadatta M. Kulkarni, Jeffrey D. Tew, and Pankaj Dayama. Auction-based mechanisms for electronic procurement. *IEEE Transactions on Automation Science and Engineering*, 4(3):297–321, 2007.

[CPV97] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with mobile code paradigms. *Proceedings of the 19th international conference on Software engineering - ICSE '97*, pages 22–32, 1997.

[CPV07] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Is code still moving around? Looking back at a decade of code mobility. *Proceedings - International Conference on Software Engineering*, pages 9–18, 2007.

[CZ16] Mung Chiang and Tao Zhang. Fog and IoT: An Overview of Research Opportunities. *IEEE Internet of Things Journal*, 4662(c):1–1, 2016.

[DGCG] Amir Vahid Dastjerdi, Harshit Gupta, Rodrigo N Calheiros, and Soumya K Ghosh. Fog Computing : Principles , Architectures , and Applications. pages 1–26.

[Eva11] Dave Evans. The Internet of Things - How the Next Evolution of the Internet is Changing Everything. *CISCO white paper*, (April):1–11, 2011.

[GGP17] Mohit Kumar Gokhroo, Mahesh Chandra Govil, and Emmanuel S. Pilli. Detecting and mitigating faults in cloud computing environment. *3rd IEEE International Conference on*, 2017.

[GJGT10] Íñigo Goiri, Ferran Julià, Jordi Guitart, and Jordi Torres. Checkpoint-based fault-tolerant infrastructure for virtualized service providers. *Proceedings of the 2010 IEEE/IFIP Network Operations and Management Symposium, NOMS 2010*, pages 455–462, 2010.

[Hec16] Jeff Hecht. The bandwidth bottleneck that is throttling the internet. *Nature*, 536(7615):139–142, 2016.

[Hen13] Johannes Helbig Henning Kargermann, Wolfgang Wahlster. Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0. *Bmbf.De*, (April):1–116, 2013.

[HKR] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not.

[HL13] Kirak Hong and David Lillethun. Mobile fog: a programming model for large-scale applications on the internet of things. *Proceedings of the second ACM SIGCOMM Workshop on Mobile Cloud Computing*, pages 15–20, 2013.

[HVS+17] Christoph Hochreiner, Michael Vögler, Johannes M. Schleicher, Christian Inzinger, Stefan Schulte, and Schahram Dustdar. Nomadic applications traveling in the fog. 2017. (accepted for publication) In 2nd EAI International Conference on Cloud, Networking for IoT Systems, pages NN-NN, Brindisi, Italy.

[KDTR12] Boris Koldehofe, Frank Dürr, Muhammad Adnan Tariq, and Kurt Rothermel. The Power of Software-defined Networking: Line-rate Content-based Routing Using OpenFlow. *Acm Mw4Ng*, (December):1–6, 2012.

[Kel15] Rhea Kelly. Internet of things data to top 1.6 zettabytes by 2020, April 2015. Last accessed: 02.07.2017.

[KG99] David Kotz and Robert S Gray. Mobile Agents and the Future of the Internet. *Search*, 33(3):7–13, 1999.

[LLM+98] D. Lee, D. Lough, S. Midkiff, N. Davis, and P. Benchoff. The next generation of the internet: aspects of the ipv6. *IEEE Network*, 12(April):28–33, 1998.

[LO99a] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Commun. ACM*, 42(3):88–89, 1999.

[LO99b] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.

[Mic17] Microsoft. Azure iot suite—iot cloud solution | microsoft, June 2017. Last accessed: 21.06.2017.

[MKK15] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: A performance comparison. *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, (March):386–393, 2015.

[MSID16] V Michael, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. A Scalable Framework for Provisioning Large-scale IoT Deployments. *ACM Transactions on Internet Technology*, 16(2):1–20, 2016.

[Mtm15] Abstract Machine-to machine. Cognitive Machine-to-Machine Communications for Internet-of-Things : A Protocol Stack Perspective. 2(2):103–112, 2015.

[MW16] Fabrizio Montesi and Janine Weber. Circuit Breakers, Discovery, and API Gateways in Microservices. 2016.

[NBM+] R Nagler, D L Bruhwiler, P Moeller, S D Webb, Radiasoft Llc, Radiabeam Technologies Llc, and Santa Monica. Sustainability and Reproducibility via Containerized Computing * Bivio Software Inc ., Boulder , CO 80303 , USA Containerized C omputing A cknowledgment s R eferences. pages 10–11.

[NSTM13] Takayuki Nishio, Ryoichi Shinkuma, Tatsuro Takahashi, and Narayan B. Mandayam. Service-oriented heterogeneous resource sharing for optimizing service latency in mobile cloud. *Proceedings of the first international workshop on Mobile cloud computing & networking - MobileCloud '13*, page 19, 2013.

[NTK16] Mina Nabi, Maria Toeroe, and Ferhat Khendek. Availability in the cloud: State of the art. *Journal of Network and Computer Applications*, 60:54–67, 2016.

[OK10] Ana Maria Oprescu and Thilo Kielmann. Bag-of-tasks scheduling under budget constraints. *Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010*, pages 351–359, 2010.

[PIUB+17] Goiuri Peralta, Markel Iglesias-Urkia, Marc Barcelo, Raul Gomez, Adrian Moran, and Josu Bilbao. Fog computing based efficient IoT scheme for the Industry 4.0. *2017 IEEE International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM)*, pages 1–6, 2017.

[Pro13] United States Profile. THE DIGITAL UNIVERSE IN 2020: BigData , Bigger Digital Shadows , and Biggest Growth in the Far East — United States. pages 1–7, 2013.

[Ruc16] Ruchika. Evaluation of Docker for IoT Application. *International Journal on Recent and Innovtion Trends in Computing and Communication*, 4(6):624–628, 2016.

[Sat03] Ichiro Satoh. Building reusable mobile agents for network management. *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, 33(3):350–357, 2003.

[SBCD09] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cáceres, and Nigel Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[SD16] Weisong Shi and Schahram Dustdar. The Promise of Edge Computing. *Computer*, 49(5):78–81, 2016.

[Sha14] R M Sharma. The Impact of Virtualization in Cloud Computing. *International Journal of Recent Development in Engineering and Technology*, 3(1):197–202, 2014.

[She07] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edition, 2007.

[SJL+13] M. Zubair Shafiq, Lusheng Ji, Alex X. Liu, Jeffrey Pang, and Jia Wang. Large-scale measurement and characterization of cellular machine-to-machine traffic. *IEEE/ACM Transactions on Networking*, 21(6):1960–1973, 2013.

[SOM14] F. Shrouf, J. Ordieres, and G. Miragliotta. Smart factories in Industry 4.0: A review of the concept and of energy management approached in production based on the Internet of Things paradigm. *IEEE International Conference on Industrial Engineering and Engineering Management*, 2015-January:697–701, 2014.

[SSB16] Olena Skarlat, Stefan Schulte, and Michael Borkowski. Resource Provisioning for IoT Services in the Fog Resource Provisioning for IoT Services in the Fog. *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications Resource*, (November), 2016.

[Sto15] Ivan Stojmenovic. Fog computing: A cloud to the ground support for smart things and machine-to-machine networks. In *2014 Australasian Telecommunication Networks and Applications Conference, ATNAC 2014*, pages 117–122, 2015.

[TF17] Enqing Tang and Yushun Fan. Performance comparison between five NoSQL databases. *Proceedings - 2016 7th International Conference on Cloud Computing and Big Data, CCBD 2016*, pages 105–109, 2017.

[TV07] Andrew S Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms, 2/E*. 2007.

[TWW17] Klaus-Dieter Thoben, Stefan Wiesner, and Thorsten Wuest. "Industrie 4.0" and Smart Manufacturing – A Review of Research Issues and Application Examples. *Internantional Journal of Automation Technology*, 11(1):4–19, 2017.

[VRM14] Luis M. Vaquero and Luis Rodero-Merino. Finding your Way in the Fog. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.