47 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Development and Analysis of Middleware for Fog Computing On Energy Efficient Devices Suiting the Internet of Things

Marcel Großmann

University of Bamberg Press

47 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Band 47

University of Bamberg Press
2025

Development and Analysis of Middleware for Fog Computing On Energy Efficient Devices Suiting the Internet of Things

Marcel Großmann

University
of Bamberg
Press
2025

Bibliografische Informationen der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Diese Arbeit hat der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg als Dissertation vorgelegen.
1. Gutachter: Prof. Dr. Udo R. Krieger
2. Gutachter: Prof. Dr. rer. nat. Guido Wirtz
3. Gutachter: Prof. Dr. Sven Overhage

Tag der mündlichen Prüfung: 29.09.2025

Herstellung und Druck: docupoint, Magdeburg
Umschlaggestaltung: University of Bamberg Press
Umschlaggraphik: Graph, © Marcel Großmann

University of Bamberg Press, ubp@uni-bamberg.de, Bamberg 2025
ROR https://ror.org/004fa0x02

ISSN: 1867-6197 (Print)
ISBN: 978-3-98989-088-6 (Print)
eISSN: 2750-8560 (Online)
eISBN: 978-3-98989-089-3 (Online)

URN: urn:nbn:de:bvb:473-irb-111125x
DOI: https://doi.org/10.20378/irb-111125

Acknowledgements

During the last thirteen years at the university, I experienced numerous memorable encounters with a diverse group of individuals, for which I am profoundly thankful. Regrettably, I am unable to name each individual; thus, I would want to express my deepest appreciation to all who have accompanied me on this intriguing journey.

I would like to express my gratitude to my supervisor, Prof. Dr. Udo R. Krieger, for granting me the opportunity to evolve through the challenges of research and teaching. The courses on the "Foundations of Internet Communication (GIK)" and "Multimedia Communication on High-speed Networks (MMK)", which covered networking mechanisms, significantly refined my cognitive approach. I extend my gratitude to the other members of my dissertation committee, Prof. Dr. Guido Wirtz and Prof. Dr. Sven Overhage, for their invaluable criticism during the colloquium.

A remarkable workplace is unattainable without exceptional individuals. Primarily, all present and past members of the Computer Networks Group (KTR), including Philipp Eittenberger and Duy Thanh Le. You were consistently there to provide support when we worked together and always helped realizing new ideas. I cannot conceive of a superior workplace! Without Cornelia Schecher's assistance in administrative affairs, I would be unable to compose these lines and would still be engaged in completing forms. Moreover, I would like to express my gratitude to the university library, particularly Steffen Illig, for facilitating an environment conducive to a project aimed at preserving the cultural heritage and for his continuous efforts to effectively publish and sustain it. Furthermore, I extend my ap-

preciation to the university's IT service, particularly Martin Mai, for consistently facilitating the necessary computing infrastructure.

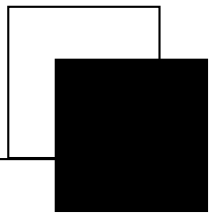
I am particularly appreciative of my close collaboration with the Hypriot Group, especially Andreas Eiermann and Matthias Renner, who laid the foundation for my dissertation by facilitating container virtualization on single-board computers and contributing valuable material to several papers. We successfully initiated a community gathering format, Docker Meetups, at the university, resulting in significant impact.

In recent years, I would like to express my gratitude to Prof. Dr. Tobias Hoßfeld of the University of Würzburg, head of Informatik III, for annually organizing workshops and providing helpful comments on our efforts.

My PhD role also involved lecturing, which I thoroughly enjoyed. I encountered numerous pupils that challenged me, and I reciprocated. I take particular pride in the collaborative papers we authored, notably with Hendrik Cech, Manuel Dworzak, Tobias Homeyer, Christos Ioannidis, Andreas Keiper, Clemens Klug, Cornelius Matějka, Chandan Sarkar, Christian Schenk, Lukas Schwöbel, Stephan Schuberth, Hannes Stadler, Noëlle Weinmann, and Michael Ziegler. Many thanks to Christos, Manuel, and Noëlle, who also contributed to the research prototypes.

I extend my heartfelt gratitude to my family, particularly my parents, for their unwavering support in all my endeavors. I regret any inconvenience this may have given you.

Danksagungen



Während meiner dreizehnjährigen Zeit an der Universität habe ich viele unvergessliche Begegnungen mit einer vielfältigen Gruppe von Menschen erlebt, wofür ich zutiefst dankbar bin. Es ist mir leider nicht möglich, jede Person einzeln beim Namen zu nennen. Deshalb möchte ich allen, die mich auf diesem spannenden Abenteuer begleitet haben, meinen aufrichtigsten Dank aussprechen.

Ich möchte meinem Betreuer, Prof. Dr. Udo R. Krieger, meinen Dank aussprechen, dass er mir die Möglichkeit gegeben hat, mich durch die Herausforderungen in Forschung und Lehre weiterzuentwickeln. Die Kurse über die "Grundbausteine der Internet-Kommunikation (GIK)" und die "Multimedia-Kommunikation in Hochgeschwindigkeitsnetzen (MMK)", die Netzwerktechniken abdeckten, haben meinen kognitiven Ansatz erheblich erweitert. Ich danke den anderen Mitgliedern meines Promotionsausschusses, Prof. Dr. Guido Wirtz und Prof. Dr. Sven Overhage, für ihre unschätzbare Kritik während des Kolloquiums.

Ein herausragender Arbeitsplatz ist ohne außergewöhnliche Individuen undenkbar. Dazu zählen alle gegenwärtigen und früheren Angehörigen der "Professur für Informatik - insbesondere Kommunikationsdienste, Telekommunikationssysteme und Rechnernetze (KTR)", insbesondere Philipp Eittenberger und Duy Thanh Le. Ihr wart während unsere gemeinsamen Zeit immer da, um Unterstützung für neue Aufgaben zu bieten und mit Rat und Tat zur Seite zu stehen. Ich kann mir keinen besseren Arbeitsplatz vorstellen! Ohne die Unterstützung von Cornelia Schecher in administrativen Angelegenheiten wäre ich nicht in der Lage, diese Zeilen zu schreiben und würde immer noch damit beschäftigt sein, For-

mulare auszufüllen. Darüber hinaus möchte ich der Universitätsbibliothek, insbesondere Steffen Illig, meinen Dank aussprechen, dass er ein Umfeld geschaffen hat, das ein Projekt zur Erhaltung des kulturellen Erbes fördert, und für seine kontinuierlichen Bemühungen, es effektiv zu veröffentlichen und zu pflegen. Darüber hinaus möchte ich dem IT-Service der Universität, insbesondere Martin Mai, für die kontinuierliche Bereitstellung der notwendigen Computerinfrastruktur meinen Dank aussprechen.

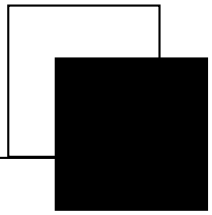
Ich bin besonders dankbar für die enge Zusammenarbeit mit der Hypriot-Gruppe, insbesondere mit Andreas Eiermann und Matthias Renner, die die Grundlage für meine Dissertation gelegt haben, indem sie die Container-Virtualisierung auf Einplatinencomputern ermöglicht und wertvolle Materialien zu mehreren Arbeiten beigetragen haben. Zusätzlich haben wir erfolgreich mehrere Events, sog. Docker Meetups, an der Universität initiiert, die eine bleibenden Eindruck bei den Teilnehmern hinterlassen haben.

In den letzten Jahren möchte ich meine Dankbarkeit gegenüber Prof. Dr. Tobias Hoßfeld von der Universität Würzburg, Leiter von Informatik III, ausdrücken, der jährlich Workshops organisiert und mit hilfreichen Ratschlägen zu unseren Forschungen beiträgt.

Meine Rolle als Doktorand umfasste auch die Lehre, die mir sehr viel Freude bereitet hat. Ich traf zahlreiche Studierende, deren Herausforderungen ich gerne annahm. Ich bin besonders stolz auf die gemeinsamen Arbeiten, die wir verfasst haben, insbesondere mit Hendrik Cech, Manuel Dworzak, Tobias Homeyer, Christos Ioannidis, Andreas Keiper, Clemens Klug, Cornelius Matějka, Chandan Sarkar, Christian Schenk, Lukas Schwöbel, Stephan Schuberth, Hannes Stadler, Noëlle und Michael Ziegler. Vielen Dank an Christos, Manuel und Noëlle, die auch zu den Forschungsprototypen beigetragen haben.

Ich spreche meiner Familie, insbesondere meinen Eltern, meinen herzlichen Dank für ihre unerschütterliche Unterstützung in all meinen Bestrebungen aus. Ich bedaure jegliche Unannehmlichkeiten, die Ihnen dadurch entstanden sein könnten.

Abstract



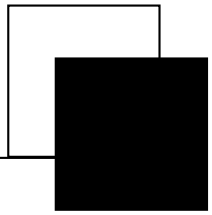
The emergence of next-generation infrastructures is facilitating the transition from centralized cloud computing to the cloud-to-fog continuum, wherein services are disseminated over many layers, ranging from centralized clouds to decentralized edge devices. This model facilitates more efficient and responsive service provisioning, especially in latency-sensitive and resource-limited contexts. In these situations, middleware is essential for managing the distributed characteristics of services, aiding energy-efficient devices at the network periphery, and enabling seamless service orchestration.

These frameworks facilitate the monitoring of small-scale services and nodes, hence maintaining performance consistency. Subsequently, distributions can be refined by integrating insights from real-time measurements and network topology data, facilitating dynamic resource allocation and improving system efficiency.

Additionally, emulation frameworks are essential for testing diverse networking scenarios and finding nearly perfect service deployments in a sandboxed environment. By determining the most appropriate emulator for application development, we provide an easy-to-use platform to researchers and developers, such that they can evaluate their services' needs. Moreover, novel network paradigms and protocols are easily adaptable on given emulations.

This dissertation examines the design of middleware solutions within the cloud-to-fog continuum and analyzes the application of emulation tools to enhance service delivery at the network edge.

Kurzfassung



Die Entwicklung von IT-Infrastrukturen der nächsten Generation erleichtert den Übergang vom zentralisiertem Cloud-Computing hin zum Cloud-to-Fog-Kontinuum, bei dem Dienste über viele Schichten verteilt werden, die von zentralisierten Cloud-Rechnern bis hin zu dezentralen Edge-Geräten reichen. Dieses Modell ermöglicht eine effizientere und reaktionsschnellere Bereitstellung von Dienstleistungen, insbesondere in latenzsensiblen und ressourcenlimitierten Kontexten. In diesen Situationen ist Middleware unerlässlich, um die verteilten Eigenschaften von Diensten zu verwalten, energieeffiziente Geräte am Netzwerkrand zu unterstützen und eine nahtlose Dienstorchestrierung zu ermöglichen.

Diese Frameworks erleichtern die Überwachung von kleineren Diensten und Knoten ohne die Leistungsfähigkeit zu beeinträchtigen. Anschließend können Verteilungen verbessert werden, indem Auswertungen aus Echtzeitmessungen und Netzwerk-Topologiedaten in Verteilungsstrategien integriert werden, was eine dynamische Ressourcenallokation ermöglicht und die Systemeffizienz verbessert.

Durch die Bestimmung des am besten geeigneten Emulators für die Anwendungsentwicklung bieten wir Forschern und Entwicklern eine benutzerfreundliche Plattform, damit sie die Bedürfnisse ihrer Dienste bewerten können. Darüber hinaus sind neuartige Netzwerkparadigmen und -protokolle leicht in bereits exisitierenden Emulationen umsetzbar.

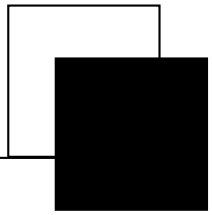
Diese Dissertation untersucht das Design von Middleware-Lösungen im Cloud-to-Fog-Kontinuum und analysiert die Anwendung von Emulationswerkzeugen zur Verbesserung der Servicebereitstellung am Netzwerkrand.

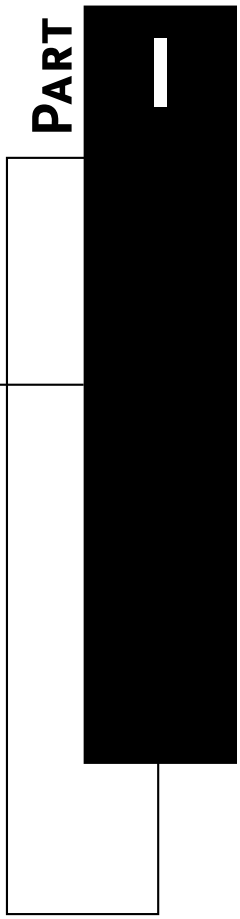
Contents

List of Figures

List of Tables

Listings





Introduction

Fairy tales are more than true: not because they tell us that dragons exist, but because they tell us dragons can be beaten.

N. Gaiman

Parts of this chapter are taken from [65], [74]

The Cloud-to-Fog-Continuum is classified as a fundamental factor affecting service operations in next-generation infrastructures. Service providers must now establish frameworks for service deployment that account for multiple factors, including the geographical dispersion of compute nodes. Nevertheless, the swift advancement of next-generation networks and virtualization methodologies promotes the emergence of a middleware for the Cloud-to-Fog-Continuum. This chapter presents an incentive for a fresh approach to providing services within these dispersed infrastructures, as explored in Chapter 1. Subsequently, previously published related work is presented and assessed in Chapter 3, which serves to delineate the study objectives in Chapter 2.

Chapter 1
Motivation for Service Orchestration

The swift expansion of data generation, together the rising utilization of Internet of Things (IoT) devices, Artificial Intelligence (AI), and Edge Computing (EC), has posed considerable hurdles to conventional Cloud Computing (CC). CC, despite its capabilities, frequently experiences latency, bandwidth constraints, and scalability challenges as the demand for real-time, distributed services escalates. These issues have catalyzed the development of Fog Computing (FC), which enhances cloud functionalities nearer to the network's periphery, hence diminishing latency and streamlining data processing.

However, managing services in a Cloud-to-Fog-Continuum, where resources are allocated across many layers – from centralized cloud data centers to local fog and edge nodes – introduces complexity. The development of middleware is essential in this context and must take communication paradigms for next-generation applications into account [24].

Worthless to say, communication infrastructures are the long-term backbone for all Machine-to-Machine (M2M) connections and are the per-default bottleneck for all distributed applications. Since there are still not enough programmers and applications to overload the hardware prowess of the available computing infrastructures, the network hinders faster operations with its characteristics, which are mainly described as latency, bandwidth and jitter.

In recent years, virtualization offered a first possibility to fully utilize computing resources by providing Virtual Machines (VMs) for different services running on the same machine. With the increasing investments providers established to offer contracts for VMs, which are deployed in their datacenters – the birth hour of

CC, where everyone can outsource applications based on their Operating Systems (OSs) to providers, which make full use of the economies of scale.

Nevertheless, with new architectural concepts like Service-Oriented Architectures (SOAs) arising, developers are being organized in SCRUM teams, which forces them to faster deployment cycles that are hardly satisfiable by utilizing only VMs. Currently, the virtualization paradigm shifts in this service or even microservice world to a lightweight model, called container virtualization.

Additionally, a contrary effect is visible. Besides nearly centralized or intelligently placed cloud datacenters, new energy-efficient devices arrive and generate a new dimension to the big data collections and processing. This uprising of the so called IoT, which seems contradictionary to CC, needs a thorough reconsideration of network capabilities, such that new services can substitute existing ones without overwhelming the underlying network infrastructure.

In this regard, virtualization already establishes as key factor again by decoupling forwarding configurations from the hardware. Therefore, the idea of a Software Defined Network (SDN) is the default approach, where a "centralized", virtualized control logic supervises a fleet of switches and deploys forwarding rules on them. Ideally, those rules are influenced by the applications and consider Quality of Service (QoS) requirements. Last but not least, the extension with Network Function Virtualization (NFV) includes microservices into the network, by providing them on machines that are additionally optimized to be geographically located near to requesters.

The bargain of the ability to shift services from the powerful cloud datacenters to every place in the network enables new QoS levels and should be considered as the enabler for the era of "cloudless computing" [74].

Moreover, current communication infrastructures must comply to the changes necessitated by the growing popularity of mobile devices and the rise of the IoT with enhanced solutions that maximize their capabilities. For numerous use cases involving substantial data transmissions, such as autonomous vehicles and services that are extremely latency-sensitive, the path to data centers and the core network may impede the performance of contemporary applications. As depicted in Figure 1.1, "ubiquitous computing" will face the need to distribute microservices from powerful CC nodes to energy-efficient edge devices in order to ensure their high availability and, ideally, prevent them from overwhelming the core network.

The evolving demands involve heightened levels of traffic volume, acceleration of transmission speeds, and improved performance in computation and storage. In addition, the expanding computing capacity must be allocated to a network that is exceptionally dynamic and adaptable.

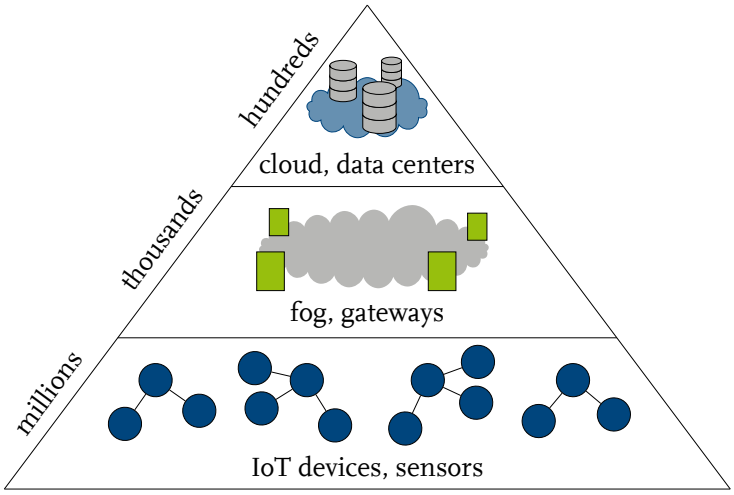


Figure 1.1: Conceptual delineation of the Cloud-to-Fog-Continuum, illustrating the exponential growth of devices from hundreds within cloud data centers to millions across the edge network [cf. 3].

The initial application we built is titled SensIoT [65], which establishes an IoT platform based on virtualized services for environmental monitoring. It can operate in multiple locations within an organization and currently monitors the conditions of the book storage rooms of the university's library, which are situated in ancient buildings and are sadly vulnerable to mold. Following its deployment, it clearly lacks the required support for widespread deployment, which is needed for, e.g., smart cities. Through the exploration of several paradigms such as serverless computing [66] to improve SensIoT for extensive deployments, we observed that the advantages are negligible. This motivated us to acquire knowledge on designing an orchestration system capable of dynamically routing traffic in diverse, distributed computing environments.

Chapter 2
Research Objectives

From the conceptual idea that Al-Fuqaha *et al.* [3] provided for the Cloud-to-Fog-Continuum, new research issues arise. First, we take a look at the service deployment and the questions that arise in Section 2.1. To cope with the issues, the need for sandboxed systems and their value are shown in Section 2.2. Finally, the idea of an orchestration in such an environment is described in Section 2.3.

2.1 Service Deployment in the Cloud-to-Fog-Continuum

With the omni-presence and complexity of digital services, there has been a significant demand for low-latency, high-performance, and energy-efficient computing solutions. Existing cloud infrastructures, despite their significant scalability, frequently fail to satisfy the rigorous demands of latency-sensitive applications, including autonomous systems, real-time analytics, and smart city services [42]. To mitigate these limitations, the Cloud-to-Fog-Continuum concept arose, which integrates centralized cloud resources with distributed edge computing nodes in proximity to end users. In this concept the processing capabilities of cloud data centers are combined with the immediacy and responsiveness of edge devices, creating a continuum of resources from the core to the network edge as shown in Figure 1.1.

The deployment of services over this continuum presents various essential factors that influence infrastructure architecture and service orchestration. This encompasses latency reduction, as edge devices provide instantaneous responses to local requests; resource optimization, wherein available bandwidth and computational power are effectively utilized across layers; and energy efficiency, which is

especially crucial for edge devices with constrained power capacity. Furthermore, scalability and fault tolerance are crucial when services are disseminated across several and potentially unreliable nodes.

Research Question 1 seeks to identify the essential motivations behind this distributed computing paradigm. By examining its characteristics, we may gain a clearer understanding of the technical, operational, and strategic elements that affect service placement, middleware development, and infrastructure management. Therefore, virtualization models need to be evaluated, especially in the context of an IoT ecosystem, which benefits from lightweight, adaptable virtualization models that facilitate optimal resource utilization, multi-tenancy, and prompt deployment across diverse edge nodes. The underlying network is impacted by those services, which affect bandwidth consumption, latency, and network robustness, necessitating optimization solutions to avert bottlenecks and uphold service quality across distributed nodes. Comprehending the interaction between virtualization and network architecture is essential for guaranteeing uninterrupted service delivery in the Cloud-to-Fog-Continuum.

Research Question 1: Cloud-to-Fog-Continuum

What are the key considerations that motivate the deployment of services in the **Cloud-to-Fog-Continuum**?

1. Which virtualization models offer advantages to the **IoT**?
2. How do virtualization approaches impact the underlying **network**?

2.2 Network Emulation for Virtualized Services

Emerging technologies such as 5G, 6G, and IoT increase the complexity of next-generation infrastructures, requiring robust and scalable service deployments. Distributed designs offer minimal latency, superior performance, and energy efficiency. Emulation technologies are essential for evaluating and analyzing service deployment strategies [150].

Research Question 2 investigates the function of emulation in evaluating and enhancing the performance of distributed systems with a special focus to networking paradigms. Emulation technologies offer a regulated, scalable environment to replicate intricate networking conditions, enabling researchers and developers to

assess service performance under diverse scenarios. These technologies facilitate the testing of middleware and service orchestration mechanisms by emulating real-world situations, thereby mitigating the risks and expenses linked to full-scale deployments. Moreover, emulation can offer a versatile environment to test novel transmission protocols like, e.g., multi-path transmissions. The capability to replicate failure scenarios, such as node failures, network congestion, or cyber-attacks helps to evaluate the fault tolerance, redundancy, and self-healing properties of services and their underlying infrastructure in a sandboxed environment.

Research Question 2: Emulation Tools

How can **emulation tools** help to run deployments with next-generation networks?

1. Are there possibilities to develop and verify **alternative transmission** paradigms?
2. Can they assist in evaluating the **resilience** of a service and its underlying infrastructure?

2.3 Orchestration of Services in the Cloud-to-Fog-Continuum

Combining Research Question 1 and Research Question 2 reveals that CC and FC have revolutionized service infrastructures, distributing applications across centralized data centers and decentralized fog nodes. The main goal is to reduce latency, enhance responsiveness, and optimizes network resources, but achieving it faces technical and operational challenges, which can be overcome in a sandboxed environment with the right choice of emulation tools.

Research Question 3 pinpoints the primary problem in distributing services across the Cloud-to-Fog-Continuum on diverse infrastructure layers. This encompasses resource management, latency optimization, scalability, and the assurance of energy efficiency in resource-constrained nodes. Moreover, ensuring data consistency, security, and fault tolerance across distributed nodes is essential for robust and reliable application performance. To support developers, the DevOps principles may optimize service deployment by prioritizing Continuous Integration (CI), automation, and collaboration, which can effectively operate with heterogeneous hardware across many layers, facilitating expedited and uniform de-

ployments. This leads to designing a framework for service provisioning in the Cloud-to-Fog-Continuum, which requires capabilities to offer dynamic resource allocation, real-time monitoring, service orchestration, and load balancing across diverse infrastructures.

Research Question 3: Requirements

Which **requirements** need to be solved to run applications on a service infrastructure distributed **from cloud to fog nodes**?

1. How can DevOps principles **ease service development** in networks with divergent hardware?
2. What is the **optimal design** for a framework that supports service deployment in cloud to fog environments?

Chapter 3
Scientific Categorisation

To assess the extent to which we can address the research questions outlined in Chapter 2, we examine the scientific contributions presented in Section 3.1. In the subsequent phase, the existing literature is classified based on its pertinence to this dissertation in Section 3.2.

3.1 Contributions to the Scientific Community

First we investigate already published work and point out their contribution to the scientific community. We can divide the contributions into several parts:

C1. According to Al-Fuqaha *et al.* [3], a plethora of varied devices are incorporated within the IoT. Many of them are referred to as Single Board Computers (SBCs), providing limited functionality for executing virtualized programs. In collaboration with the Hypriot team[1], we successfully executed Docker on Raspberry Pis (RPis) and incorporated our modifications into the official source code. Furthermore, we established the foundations for a distributed computing cluster utilizing several Docker-enabled nodes in [69], [70], prior to the inception of Docker Swarm.

[1] https://blog.hypriot.com/crew

C2. A team from the University of Rome[2] developed Netkit and its successor, Kathará [17], which we rigorously adapted to fulfill our emulation requirements. Our research introduced innovative network paradigms [71] and assessments of attack scenarios [68] to their environment. Furthermore, we introduced a user-friendly Graphical UI (GUI) for Kathará in [76].

C3. By recognizing the trend of distributed computing in container based frameworks like Kubernetes[3] and Docker Swarm[4], we added contributions for resource and application monitoring that are capable to run on SBCs again. Several approaches [75], [77] led to a stable monitoring system to gather service statistics and run on a cluster of diverse computing nodes. With the final approach [108], the monitoring system is integrated into a SDN, where a module of the SDN control plane collects the statistics.

C4. The synthesis of the previous contributions results in an orchestration middleware capable of operating in a distributed manner along the Cloud-to-Fog-Continuum. It adheres to the theoretical Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) concept and incorporates a Federated Machine Learning (FL) strategy to facilitate intelligent service placement and dynamic routing in the Cloud-to-Fog-Continuum [43], [44].

C5. Numerous contributions have been made about apps that may operate on SBCs and provide advantages to the IoT when deployed in a virtualized manner over several heterogeneous nodes. Environmental monitoring approaches like SensIoT [65] might benefit from the orchestration framework for optimal service allocations. Distributing blockchain algorithms [28], [121], [164] facilitates novel applications in the IoT, provided that the applications are allocated based on their computational requirements.

[2]https://compunet.ing.uniroma3.it
[3]https://kubernetes.io
[4]https://docs.docker.com/engine/swarm

3.2 Published Research

Individual aspects of this dissertation have already been published. Table 3.1 lists all publications that were submitted to peer-reviewed conferences, journals, and workshops.

	Year	Type	Where		Year	Type	Where
[79]	2024	Workshop	WueWoWAS	[28]	2019	Conference	ICFC
[68]	2024	Conference	I4CS	[73]	2019	Conference	NetSoft
[59]	2024	Conference	I4CS	[164]	2019	Conference	ICBC
[67]	2023	Conference	I4CS	[65]	2019	Journal	Hindawi
[121]	2023	Conference	I4CS	[6]	2019	Journal	JUCS
[71]	2023	Workshop	WueWoWAS	[77]	2018	Conference	IINTEC
[76]	2023	Workshop	WueWoWAS	[72]	2018	Journal	ABI Technik
[44]	2023	Workshop	WueWoWAS	[75]	2017	Conference	ITC
[43]	2023	Conference	ICICT	[64]	2017	Conference	TPDL
[92]	2022	Conference	I4CS	[46]	2017	Conference	I4CS
[108]	2022	Workshop	WueWoWAS	[70]	2016	Conference	ICT
[74]	2020	Conference	ICOIN	[147]	2016	Conference	ICEBE
[66]	2019	Conference	UCC	[69]	2016	Conference	ITC

Table 3.1: Publications by type and year

Now, let us take a closer look, how the publications are relevant for the contributions of Section 3.1.

Current virtualization techniques are studied for *C1*, and container virtualization has emerged as a predominant technology for operation on SBCs. We established a distributed architecture for providing containerized services on heterogeneous nodes with “Hypriot Cluster Lab (HCL)” [70]. Additionally, a traffic-secured version utilizing a Virtual Private Network (VPN) with *tinc*[5] was developed. These concepts ultimately prompted the Docker community to create Docker Swarm, which incorporated much of our published research. In both publications, I supported the design of the concept and realization on Advanced RISC Machine (ARM) devices in my role as a leading author.

[5]https://www.tinc-vpn.org/

[70]	Großmann *et al.*	"Hypriot Cluster Lab : An ARM-Powered Cloud Solution Utilizing Docker"
[69]	Großmann and Eiermann	"Automated Establishment of a Secured Network for Providing a Distributed Container Cluster"

Table 3.2: Contributions for cluster computing on SBC - C1

In continuation with *C2,* we employed several network emulators throughout our endeavor and also experimented with Mininet[6] to simulate real-world network topologies [78]. Nevertheless, the capabilities of Mininet provided diminished flexibility in comparison to Kathará. With our current emphasis on utilizing Kathará in our publications, we can seamlessly incorporate virtualized services into an emulated network and evaluate its performance. For instance, Distributed DoS (DDoS) attacks can be conducted, and their impact on SDN is examined [68], [79]. Moreover, novel network transmission paradigms on Programming Protocol-Independent Packet Processor (P4) compatible switches can be implemented without the necessity of acquiring costly hardware. We delineated various implementations of multi-path transmissions directly at the data link layer [71]. Primarily, I focused on all Kathará code and experiment related issues, as well as, taking the lead as the main author.

[71]	Großmann and Homeyer	"Emulation of Multipath Transmissions in P4 Networks with Kathará"
[76]	Großmann and Le	"Visualization of Network Emulation Enabled by Kathará"
[68]	Großmann and Weinmann	"Emulation of Denial-of-Service Attacks for Software Defined Networks"
[79]	Großmann and Weinmann	"Real-Time Monitoring of Software-Defined Networks in Kathará for Denial-of-Service Attacks"

Table 3.3: Contributions for Kathará - C2

[6]https://mininet.org/

Monitoring inside distributed computing encounters novel obstacles in the heterogeneous landscape of the IoT, where many architectures are employed to implement services. Consequently, to address *C3*, we evaluated two established node monitoring systems capable of retrieving container metrics. Consequently, *monit*[7] was the primary selection, facilitating deployment across a Docker Swarm cluster [75], while *cadvisor*[8] represented the subsequent iteration, enhancing the statistics collection framework through the incorporation of a Time-Series Database (TSDB) [77]. Furthermore, the incorporation of a SDN resulted in a system capable of monitoring the behavior of services operating on a virtualized network [108]. In [75], [77], I took the lead as a main author and supported the statistical analysis of the monitored data, while in [108] I supervised the concept and ideas for the monitoring system and supported the paper submission as the second author.

[75]	Großmann and Klug	"Monitoring Container Services at the Network Edge"
[77]	Großmann and Schenk	"A Comparison of Monitoring Approaches for Virtualized Services at the Network Edge"
[108]	Le *et al.*	"Cloudless Resource Monitoring in a Fog Computing System Enabled by an SDN/NFV Infrastructure"

Table 3.4: Monitoring approaches for SBC - C3

By integrating the prior research, we can delineate a system capable of distributing services across a SDN enabled infrastructure within the Cloud-to-Fog-Continuum. The concepts for *C4* were effectively executed using a FL framework to consistently disseminate containerized services that can operate on heterogeneous devices [43], [44]. In both publications, I was the leading author and supported the ideas of the FL framework while it was developed.

[43]	Dworzak *et al.*	"Federated Autonomous Orchestration in Fog Computing Systems"
[44]	Dworzak *et al.*	"Federated Learning for Service Placement in Fog and Edge Computing"

Table 3.5: Cloud-to-Fog-Continuum framework to deploy services - C4

[7] https://mmonit.com/monit
[8] https://github.com/google/cadvisor

Concepts for novel applications in the Cloud-to-Fog-Continuum have been examined for *C5*. We conducted multiple iterations to create an environmental monitoring system, which is presently operational at the libraries of our university [64], [65], [72], [91]. For those papers, I worked on all Docker related issues and took the role of the main author, except [72] where I reviewed before submission.

Furthermore, healthcare systems gain advantages from deploying apps at the network edge to facilitate telesurgery [59], although multi-path transmissions are essential to mitigate potential risks to patient safety [92]. For the e-health publications, my main responsibility was the development of the "digital twins" and guidance with network related configurations. Here, I used the role as a second author before the submission.

Data gathering at the edge nodes presents issues in securing these nodes and establishing authorized requester protocols for data access. We examined the algorithms of several Blockchain concepts, virtualized them, and delineated the concepts for execution on heterogeneous nodes [28], [121], [164]. In the blockchain publications, I lead the containerization of blockchain applications for SBC on different architectures and supported the publications as a second author, except [28], where I was in the role of the main author.

Ultimately, video transmission via decentralized camera nodes is another application scenario in which energy-efficient devices provide advantages, particularly for surveillance systems [67], [147]. For the real-time communication, I was concerned about the performance of SBCs to deliver videostreams and containerized appropriate modules. While I was reviewing [147] as a second author, I did the lead for the surveillance publication [67].

[64]	Großmann *et al.*	"Environmental Monitoring of Libraries with MonTreAL"
[72]	Großmann *et al.*	"Bestandsmonitoring in kulturellen Einrichtungen"
[91]	Illig and Großmann	"Überwachung von Bibliotheksbeständen mit Hilfe von Raspberry PIs"
[65]	Großmann *et al.*	"SensIoT: An Extensible and General Internet of Things Monitoring Framework"
[92]	Inamdar *et al.*	"A Web Architecture for E-Health Applications Supporting the Efficient Multipath Transport of Medical Images"
[59]	Gerwien *et al.*	"The System Architecture of a Reliable Telesurgery Service and its Performance Analysis"
[28]	Cech *et al.*	"A Fog Computing Architecture to Share Sensor Data by Means of Blockchain Functionality"
[164]	Ziegler *et al.*	"Integration of Fog Computing and Blockchain Technology Using the Plasma Framework"
[121]	Mullick *et al.*	"A Feasibility Study of a Lightweight Fog Computing Architecture Integrating Blockchain Technology for Smart E-Health Applications"
[147]	Stadler *et al.*	"Design of a Secure Mobile Business Communication Platform Utilizing Next Generation Web Technologies"
[67]	Großmann *et al.*	"Efficient Internet of Things Surveillance Systems in Edge Computing Environments"

Table 3.6: Application concepts for the Cloud-to-Fog-Continuum - C5

Chapter 4

Outline

Finally, we will provide a concise overview of this work, which is segmented into six sections. Excluding the introduction section in Part I and the conclusion in Part VI, the primary material is delineated in the following way.

The foundations for this work are presented in Part II. Commencing with various virtualization approaches in Chapter 5, which provide the groundwork for CC, we juxtapose the renowned ideas of VMs and containers. We additionally classify the service provisioning models in Chapter 6. Opening with the notion of CC in Section 6.1, we progress to EC in Section 6.2 and subsequently address FC in Section 6.3; this culminates with the extension at the network edge known as the IoT in Section 6.4.

Additionally, the underlying infrastructure for service delivery is illustrated in Chapter 7, which presents a next-generation networking framework that enables virtualization approaches. After introducing the notion of SDN in Section 7.1, we further explore and augment it by transitioning network functions through NFV in Section 7.2. To ensure the proper functioning of all these ideas, a central control entity is required, as illustrated in Section 7.3. This entity will communicate via a standardized protocol, OpenFlow, detailed in Section 7.4, with compatible switches such as Open vSwitch (OvS), as demonstrated in Section 7.5.

In Chapter 8, a brief overview is provided on how future service deployments might be improved by the application of DevOps principles, as illustrated in Section 8.1. The integration of CI tools for service deployments, particularly within the DevOps framework, is of paramount relevance and is detailed in Section 8.2.

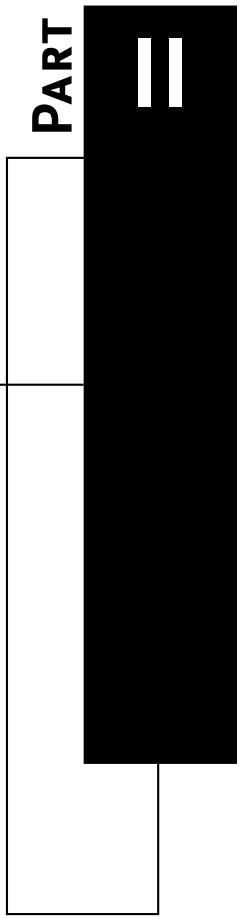
Subsequent to Part III, the prevailing distributed models are presented, and existing implementations are evaluated for their usability within the Cloud-to-Fog-Continuum environment. The initial theoretical models are presented in Chapter 9, where autonomic computing serves as a reference for more specific models such as the Observe Decide Act (ODA) loop, detailed in Section 9.1, and MAPE-K, elucidated in Section 9.2.

Current implementations of cluster computing for containerized services are detailed in Chapter 10, with a discussion of the lightweight Docker Swarm in Section 10.1, followed by an examination of its more robust counterpart in Section 10.2. Additionally, these implementation services can be disseminated using a Function as a Service (FaaS) framework, as illustrated in Section 10.3.

Furthermore, emulation middleware is assessed in Chapter 11. The content compares various container-based emulation techniques that facilitate dynamic next-generation networking paradigms. Our implementation [76] for deploying a user-friendly system is detailed in Section 11.1.

A framework is built in Part IV to manage the delivery of services in a Cloud-to-Fog-Continuum environment. Consequently, several steps are required, starting at the foundational level; a Docker network plugin is elucidated in Chapter 13 to directly link a new container to an OvS instance. This serves as the basis for establishing a cloudless infrastructure as outlined in Chapter 14. In addition to these infrastructures, service monitoring programs are essential for tracking resource utilization in a heterogeneous system, as demonstrated in Chapter 15. Intelligent service placement is constructed using the collected statistics in Chapter 16.

Finally, Part V enumerates potential applications for the Cloud-to-Fog-Continuum, which are assessed for their feasibility on energy-efficient devices.



Foundations

The scientific man does not aim at an immediate result. He does not expect that his advanced ideas will be readily taken up. His work is like that of the planter - for the future. His duty is to lay the foundation for those who are to come, and point the way.

N. Tesla

Parts of this chapter are taken from [46], [73]

The current virtualization concepts are presented in Chapter 5. Commencing with traditional virtualization via VMs in Section 5.1, we examine the microservice-oriented variant of container virtualization in Section 5.2.

From the provider's perspective, Chapter 6 presents an overview of several service providing methods.

Beginning with the perspective of CC in Section 6.1, we transition to EC in Section 6.2. Furthermore, the devices in between cloud and edge are included in a concept called FC as delineated in Section 6.3. To address the intricacies of contemporary infrastructures, the IoT is examined in Section 6.4.

One tier beneath all service provisioning models, the communication infrastructure is augmented, warranting a detailed examination of networking ideas in Chapter 7. Upon gaining insights into the concept of SDN in Section 7.1, we acquire an understanding of service provisioning with NFV in Section 7.2. Nonetheless, the choice of a suitable SDN controller and its functionalities are elucidated in Section 7.3. The standardized protocol for communication between switches and the controller is OpenFlow, as detailed in Section Section 7.4. Ultimately, a virtualized switch, OvS, utilized throughout the dissertation, is elucidated in Section 7.5.

A novel concept for the development and deployment of services in a swiftly evolving microservices environment is presented in Chapter 8. Commencing with the DevOps lifecycle in Section 8.1, we concentrate on the segment of CI in Section 8.2.

CHAPTER 5

Evolution of Virtualization

Virtualization involves the process of abstracting hardware or software elements. The notion of virtualization is not a recent innovation and has been in use for many decades, as indicated by the publication "Survey of Virtual Machine Research" [61]. The emergence of VMs can be traced back to the early 1960s when input/output (I/O) processors and multiprogramming, a precursor to parallel computing, were first introduced [25]. During that time, computers were large, costly and predominantly used by research institutions,universities, and major enterprises. IBM prevailed in the computing sector with its mainframe computers, such as the IBM 704 and IBM System/360, where resources were scarce, necessitating the efficient execution of various tasks. To optimize hardware consumption, early mainframe users operated a singular OS and applications per machine, resulting in under-utilization until IBM invented the concept of time-sharing, which enabled numerous users to concurrently utilize the same system and interact with it as though each had a dedicated machine [155].

Afterwards, IBM commenced the development of virtualization technologies and aimed towards separating and securing environments on a single machine. The CP-40 project was among the earliest computers to achieve complete virtualization by implementing a VM Monitor (VMM), enabling the concurrent operation of several instances of various OSs on a single hardware platform. The successor CP-67 integrated a seminal early VM OS [118] that was groundbreaking since it enabled a single physical mainframe to be divided into several "virtual machines", each operating its own version of an OS. This capability significantly enhanced efficiency and established the foundation for modern virtualiza-

tion. IBM's achievements resulted in the first commercial OS for virtualization, named VM/370, which reinforced IBM's supremacy in virtualization throughout the 1970s, especially among corporations requiring workload consolidation or multiple user access to system resources.

However, the emergence of Personal Computers (PCs) in the 1980s diminished the prominence of virtualization of mainframes. Compact, specialized systems became economically viable and available to enterprises and even people, decreasing the need for costly mainframe virtualization solutions. Additionally, the ascendance of UNIX based systems and the proliferation of client-server designs reduced the necessity for virtualization in enterprise settings, as organizations progressively utilized several physical servers instead. Nonetheless, IBM persisted and advanced its VM systems for mainframe computers to support organizations that remained on their systems. By the 1990s, VMs had largely faded from mainstream computer discourse, as the focus shifted towards networking, desktop computing, and server proliferation.

A novel wave of virtualization arose again in the late 1990s, where organizations aimed to optimize their extensive infrastructure and minimize hardware expenditures, propelled by advancements beyond the mainframe domain. VMWare launched its inaugural product VMWare Workstation in 1999, which pioneered virtualization for x86-based PCs, a feat previously deemed overly complex due to hardware limitations. VMWare workstation enabled users to run multiple OSs concurrently on a single x86 machine, imitating the VM concept initially developed by IBM decades prior, but tailored for smaller, commodity hardware. In 2001, VMware ESX Server was launched, signifying the company's foray into server virtualization. It gained immense popularity by enabling businesses to consolidate servers, decrease expenses, and enhance resource efficiency, which rekindled interest into virtualization within the industry again.

By the 2010s, virtualization had emerged as a fundamental technology for data centers, CC, and enterprise Information Technology (IT). VMs enabled service providers to deliver cloud-based infrastructures, permitting customers to lease virtual servers on-demand instead of managing actual hardware. Cloud-based VMs offer scalability, cost efficiency, and flexibility, enabling enterprises to adjust resources according to their need. Sophisticated orchestration tools were developed, enhancing the accessibility of VM management.

Concurrently, containerization surfaced as a substitute for VMs for specific workloads. Docker, launched in 2013, popularized the notion of containers, which are lightweight, isolated environments that utilize the host OS's Kernel. Containers significantly improved resource consumption efficiency, facilitating consistent application deployment across many settings.

5.1 Virtual Machines as Widely Adopted Commodity

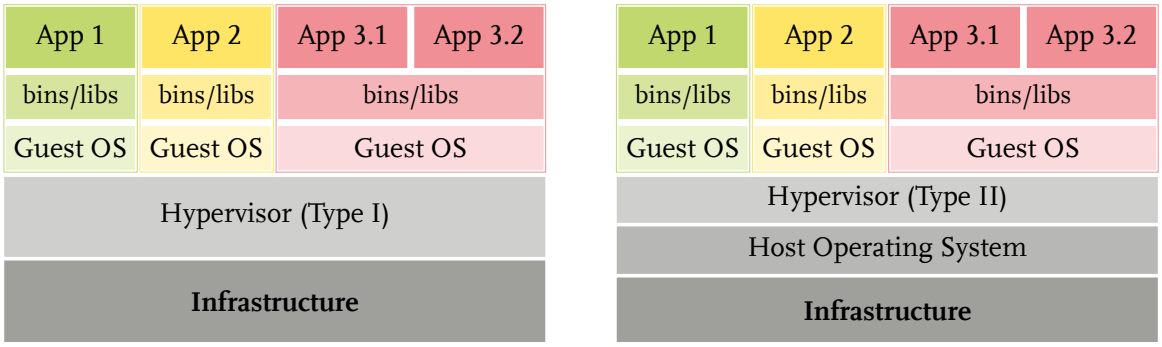


Figure 5.1: Hardware to software layers on a single machine using VMs. Hypervisors can be deployed on top of the infrastructure (Type I) or on top of a host OS (Type II) [cf. 46].

Definition 5.1: Hypervisor

A hypervisor abstracts from the hardware layer it runs on, such that:

1. VMs share resources instead of using them one after another, which results in higher efficiency
2. every VM is an exact copy of the original hardware, such that you can run any OS on every machine
3. every user has it's own OS - more reliable system

VMs are the virtualization innovation, which changed the application infrastructure and, therefore, influenced developers, consumers, industries and created new business models. Coming from bare metal machines, which were either hosted in-house or already on premise did not promise the economies-of-scale to support every use case and were hardly customizable. It changed with the invention of OS level virtualization, where a strict separation between hardware and OS was enabled by a hypervisor. Several VMs can be started on one physical machine and allow a full utilization.

The emergence of this issue gave rise to the notion of the VM as depicted in Figure 5.1, wherein a supervisory layer, referred to as the VMM, currently recognized as *Hypervisor*, regulates the access of the guest OS to the physical layer of the host system.

The previously mentioned methodical approach gave rise to a new predicament: It became unfeasible to execute the same improved software Kernel several times on a single computer, introducing immense challenges in its maintenance, because any attempts by a maintainer to update the system impede all system activities.

The issue of controlling various system processes was resolved by the advancement of VMs. According to this historical inquiry, it is evident that VMs have existed for a considerable time. Nevertheless, VMs and virtualization have gained increasing importance and visibility in recent years.

Due to the enhanced memory and Central Processing Unit (CPU) capabilities of modern computers, even common PCs can execute VMs without any discernible degradation in performance. Virtualization of servers has become a typical practice in firms that operate many servers. This is because it allows for consolidation rates ranging from 1:5 to 1:100 [148]. This consolidation optimizes cost-efficiency by reducing energy usage, space utilization, and air conditioning demands, as well as administrative and maintenance costs. VMs are employed, for example, in university data centers due to these characteristics. Nevertheless, there are complementary methods of virtualization that can get a greater consolidation rate, as elucidated in Section 5.2.

5.2 Container Virtualization

Containers serve as a more efficient and less resource-intensive substitute for VMs. As depicted in Figure 5.2, Docker eliminates the requirement of an OS for each container, contrary to VMs. Therefore, it has the ability to launch at a significantly accelerated rate and requires a reduced amount of hard disk space.

The operation of a container in Linux relies on the system architecture of a conventional Linux system, wherein the `init` process is executed upon machine startup. Each subsequent process represents a duplication of the original `init` process. Every succeeding fork inherits the exact same environment variables from the `init` process and is granted access to the system resources. Namespaces

encapsulate and combine processes logically, allocate a specific fraction of system resources to related processes, and ensure that processes from other namespaces do not affect the processes inside their namespace. With this paradigm, it becomes feasible to generate a branch of the `init` process within a namespace that covers an additional `init` process. This results in the formation of an entirely unenclosed system called *container*. Every container must be compatible with the host's Kernel for the initialization process to function properly [110]. Despite the availability of containerization through the advent of Linux' `cgroups` (formerly called container but deemed too broad), which was released in Kernel version 2.6.24 in January 2008, this form of virtualization received less attention. However, this scenario changed when Linux Containers (LXC) was officially released, which subsequently paved the way for the development of products such as *Docker* and *Rocket*[9].

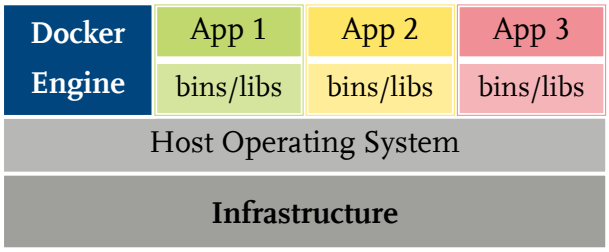


Figure 5.2: Hardware to software layers on a single machine using container virtualization by Docker [cf. 46].

Definition 5.2: Container

A container is a lightweight, portable software unit that encapsulates an application and its dependencies, enabling consistent execution across diverse computing environments. Containers utilize the host system's OS Kernel while isolating the application's processes, files, and network resources to guarantee independent operation from other containers or system components. Containers are more efficient than conventional VMs because they utilize the host OS while maintaining segregated execution contexts.

When discussing the historical progression of containers, we can begin with the introduction of `chroot`, which functioned as an early method of sandboxing engineered to isolate a process. A `chroot` environment restricts the filesystem to a specific scope, allowing only processes initiated within it to access it. However, the

[9] Rocket was discontinued in 2020 (see https://github.com/rkt/rkt).

process is not restricted to itself and has visibility of all other processes operating on the same host. Subsequent advancements in achieving greater isolation and segregation of processes included the introduction of openVZ and subsequently LXC. Upon its development in 2005, openVZ[10] missed essential features such as namespaces and control groups, which were not yet incorporated into the mainstream Kernel. Therefore, openVZ required a specific Kernel on the host system in order to adequately separate and isolate processes. For each container it establishes distinct entities, which include a separate filesystem, users and groups, network stack, devices, and a process tree.

LXC differs from openVZ as it utilizes the unmodified vanilla Linux Kernel. Established in 2008, it leverages the latest Kernel functionalities to provide isolation. Currently, containers utilize vanilla Linux Kernel features such as namespaces and control groups to isolate processes [129]. LXC is a Linux container implementation at a basic level for which Docker and LXD are software tools that provide administration capabilities. LXD, developed by Cannonical, promotes the usage of LXC and additionally provides the capability to manage system containers. While LXD provides a way to access LXC, Docker surpasses it by creating an entire ecosystem focused on containers.

A drawback of containers is that they can only run on hosts with the same Linux Kernel. That implies that you can initiate many Linux distributions, but not a BSD, OSX, or Windows container on a Linux host. Containers merely segregate processes in contrast to VMs that establish fully independent virtual computers.

There exist two primary categories of containers. *Application containers*, which may run a database, have the characteristic of running only one process. On the contrary, *system containers* initiate several processes and function as lightweight VM. Each type of container serves a unique purpose. Application containers serve as independent entities and simplify the isolation and expansion of certain components within a microservice. Conversely, system containers ease the testing and execution of software in an isolated environment that requires several processes. For instance, if the software expects the utilization of both *syslog* and *cronjobs* in order to function efficiently and avoid excessive resource consumption.

[10] https://openvz.org

Finally, here is a concise enumeration of the benefits of containers [100], [144]:

- Quick initialization without the requirement for an initialization process, unlike a VM. (If utilizing application containers)
- Containers use fewer memory resources compared to VMs, as VMs allocate and reserve RAM in advance.
- Require reduced hard disk storage by reusing layers across several containers.
- The security of processes is enhanced by "sandboxing" them within containers, as opposed to running them directly on the host.
- It is necessary to expressly establish a one-to-one correspondence between devices.
- Resolve environment inconsistencies by establishing separate virtual environments for each service.
- Computing units that adhere to the 12-factor paradigm and do not maintain any state.
- The failure of one service does not impact any other function.
- Simplified process for both updating and reverting a service that is currently executing within a container.

5.2.1 Leaders in Terms of Realizing Linux Container Technology

Nowadays, a lot of environments to run containers exist, such that we investigate the most famous ones with a detailed investigation shown in Table 5.1. LXC/Linux Container Hypervisor (LXD) are Linux native container-based technologies that construct applications within a singular container, whereas Docker[11], Podman[12], and CRI-O[13] utilize applications in layers. Both methodologies adhere to the principle of being "lightweight VMs" where isolation is achieved with Linux namespaces. When an extra service is incorporated, a new container is positioned adjacent to an existing one and atop the container engines, thereby aligning all containers on the same layer. This process is referred to as "scaling" and can be categorized into "horizontal" and "vertical" scaling. LXC/LXD exhibit horizontal scalability, but Docker, Podman, and CRI-O demonstrate vertical scalability, utilizing binaries and libraries as "containers with applications" that need the under-

[11] https://www.docker.com
[12] https://podman.io
[13] https://cri-o.io

lying container for execution. Docker and Podman minimize storage utilization by reusing existing file system layers and interlinking them, facilitating a layered architecture of file system differences. This methodology, termed Layered Application Composition (LAC), meets minimal resource demands more efficiently compared to LXC/LXD. Conversely, CRI-O is a lightweight orchestration engine tailored for Kubernetes, intended to run large-scale containerized workloads. It integrates effortlessly with Kubernetes' orchestration engine and guarantees efficient scaling and resource utilization. In contrast to Docker and Podman, CRI-O does not directly manage image creation, establishing it as an essential element for scaling containerized workloads in cloud-native settings [50], [100].

The Open Container Initiative (OCI)[14] is a vital organization for defining container image standards, facilitating seamless interoperability among container runtimes such as Docker, Podman, and CRI-O and therefore, enhances the ecosystem's modularity and vendor neutrality. CRI-O, Docker, and Podman comply to OCI standards to ensure compatibility and flexibility in managing containerized applications across many systems and tools.

[14]https://opencontainers.org/

Property / Solution	**LXC/LXD**	**Docker**	**Podman**	**CRI-O**
Development lead	Canonical Ltd.	Docker, Inc.	Red Hat	OpenSource
Main characteristics	VM-like containers.	Efficient containers through application specific design	Application centric with advanced security features.	Kubernetes specific runtime.
Platform compatibility	Linux	Linux, MacOS and Windows[15]	Linux OSs, MacOS, and Windows	Linux
Rootless executions	Needs additional effort	Limited rootless support	Rootless by default	Rootless by default
CVE handling	Community driven	CVE scans on Docker's registry	Patching community	Enterprise-grade management in Kubernetes
Scalability included	No	By enabling Swarm or Kubernetes	No	Designed to work with Kubernetes
Standardization	LXD images[16]	OCI/Docker images	OCI/Docker images	OCI/Docker images
Networking Capabilities	To be configured via Linux	Integrated networking stack	Offers a few networking options	Relies on Kubernetes networking
Use Cases	Capable to run multiple processes like a VM	Intended to run only one process	Secured version, capable to run multiple processes	Intended to run single Kubernetes workloads
Image build support	Yes, with external tools	Dockerfile and strong CI integrations	Podmanfile/Dockerfile and strong CI integrations	Needs other tools to build images like Docker

Table 5.1: Comparison of the most promising implementations of container runtimes.

[15] with additional support for Windows containers
[16] converter needed to run OCI images

5.2.2 The Prominent Container Technology Enabler Docker

We favor Docker over LXD, Podman, and CRI-O for several reasons. From the beginning, Docker is a freely available software that was publicly released in June 2014 with the initial version being 1.0. The firm *dotCloud* was established in 2008 with the objective of providing a Platform as a Service (PaaS) solution that is not tied to any one programming language. In March 2013, the company *dotCloud* made the decision to release their project called "Docker" to the public [120]. In October 2013, *dotCloud* re-branded itself as *Docker.Inc* and then sold its service to the company *cloudControl* in 2014 [110]. Prior iterations of Docker were built upon LXC, however, this paradigm shifted starting from version 1.0. Subsequently, Docker implemented *libcontainer*, which eliminated the reliance on LXC. Around Docker a rich ecosystem was established, featuring integrated tools like "Docker Compose[17]" for multi-container applications, "Docker Swarm[18]" for orchestration, and "Docker Hub[19]" for sharing pre-built images. Docker is renowned for its user-friendliness compared to LXC/LXD, offering a straightforward and clear interface for developers. Its extensive utilization in the industry and its fostered community provide it as a secure alternative by increasing stability, long-term support, and cross-platform compatibility. Docker offers a comprehensive suite of features encompassing image development, container management, and orchestration tools under a single platform, rendering it a holistic solution for containerized applications. Moreover, Docker's portability, compliant with OCI standards, guarantees that images created using Docker are compatible across various container runtimes and environments. Although rivals such as Podman and CRI-O have distinct benefits, Docker continues to be the most comprehensive, feature-laden, and extensively supported container technology, rendering it the preferred choice for several teams engaged with varied application architectures and necessitating extensive ecosystem support.

[17]https://docs.docker.com/compose/
[18]https://docs.docker.com/engine/swarm
[19]https://hub.docker.com/

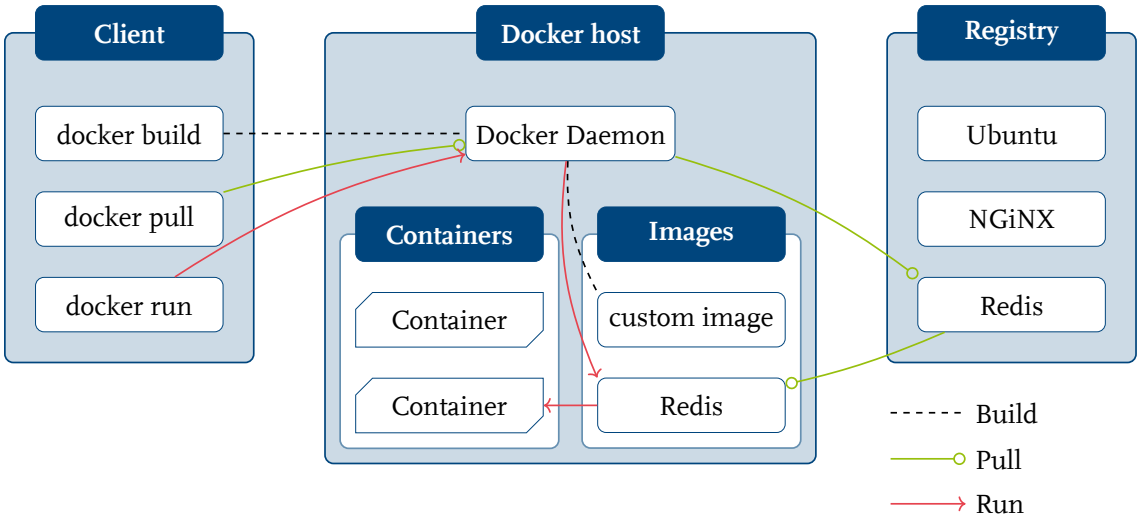


Figure 5.3: Docker comprises three primary components. A client provides each user access to the container ecosystem by invoking functions on the daemon that manages the containerization framework, supported by a registry for sharing ephemeral images.

Docker's main modules, which are depicted in Figure 5.3, are based on a client-server principles consisting of three components the Docker Engine, the Docker Client, and the Docker Registry offers a solution to manage and store images.

The **Docker Client** is a Command Line Interface (CLI) tool that enables users to interact with the Docker daemon. Docker client and Docker Host can run on the same machine or separately on different hosts. The Docker Client communicates with the Docker engine and allows users to issue commands for building, running, and pulling containerized applications. The **Docker Daemon** is the core component that provides an ecosystem to handle "images", "containers", "networks", and "volumes". It offers an Representational State Transfer (REST) Application Programming Interface (API) to provide communication to e.g. the Docker Client. "Images" are read-only files used as a foundation to run containers and include metadata that specifics the container's needs and capabilities. "Containers" are running instances of an "image" that can be created, started, stopped, moved, or deleted. Finally, the **Docker Registry** provides a service for sharing "Images" among developers and Docker's community.

Due to it's architecture and features, Docker has significantly influenced contemporary software development. It offers essential elements that assist developers in establishing an application development lifecycle, resulting in optimized

time utilization and enhanced cooperation. Docker offers an efficient and standardized platform for the rapid construction and deployment of applications and transforms the perspective on software development, testing, deployment, and delivery through containers. Containerization is a technology that facilitates the management and construction of complicated applications and software [101].

Chapter 6
Service Provisioning

In the context of service provisioning, we commence with the elder CC model referenced in Section 6.1. Followed by Section 6.2, currently available enhancements through the EC paradigm are explained and finally, we reach the stage of service distribution in the FC paradigm in Section 6.3. The integration of IoT, explained in Section 6.4, into FC ultimately yields the complete range of the Cloud-to-Fog-Continuum.

6.1 The Cloud Computing Paradigm

The procurement, establishment, and operation of dedicated computer infrastructure has consistently posed a substantial financial burden, particularly for small- to medium-sized enterprises. The effective use of this particular technology necessitates cautious deliberation and strategic investment planning. The convergence of computational, storage, and network technologies, coupled with the evolution of the Internet, has given rise to a novel computing paradigm known as CC. This model enables users to access and utilize resources such as CPU and storage as omnipresent utilities, which can be leased and released on-demand via the Internet [161]. The utilization of such technology is rendered far more attainable, since additional resources may be seamlessly incorporated into an existing framework without demanding significant financial commitments.

The concept of CC received significant attention, leading to National Institute of Standards and Technology (NIST)'s publication of a formal definition for this phenomenon. This definition finds some features that can be ascribed to such services: The automated provisioning of resources offered by the service provider

can be conveniently accessed through a network by a variety of devices with different characteristics, utilizing widely adopted technologies. The utilization of resources is typically indistinguishable by the consumer and stabilized to simultaneously provide services to multiple clients. Another essential characteristic of this computing model involves rapid provisioning and reallocation of resources. This feature facilitates efficient and rapid expansion of a service without requiring manual intervention. Ultimately, the use is carefully assessed for several objectives. On one side, the optimization of resource allocation to services and their refinement is necessary. On the other side, the process of invoicing clients for their consumption typically occurs through the quantification of the provider's infrastructure usage. For example, the pricing structure for the Google Cloud Compute Engine is calculated either on an hourly or monthly basis, as stated in the official price list for the Google Cloud[20]. The price for AWS Lambda is determined by the number of requests made, as outlined in the information provided on the official AWS Lambda website[21]. The price models mentioned are cited in the works of Mell, Grance, *et al.* [117] and Armbrust *et al.* [8].

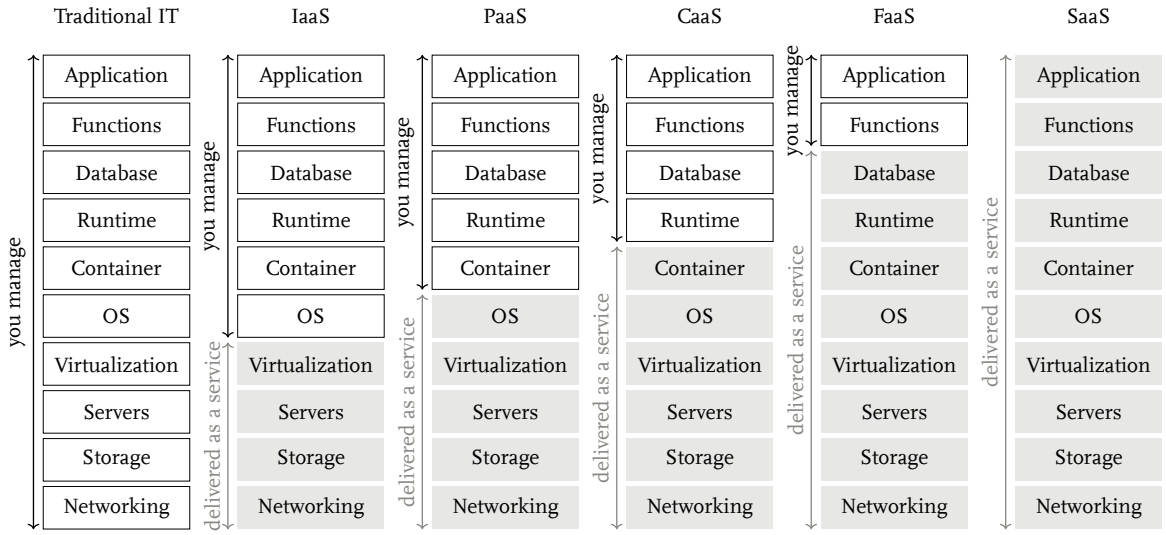


Figure 6.1: Service models in the CC domain. Colored in white are the layers, which the application provider/consumer needs to manage per service model. Transferring layers to cloud operators simplifies application deployment through various service models by limiting configuration options [cf. 93].

[20]https://cloud.google.com/compute/all-pricing
[21]https://aws.amazon.com/de/lambda/pricing

Additional categorization of the CC paradigm can be achieved by considering the service model, which delineates the specific capabilities that the customer is utilizing. The multilayered character of a typical environment is depicted in Figure 6.1. The abstractions, ranging from the hardware layer to the application layer, are horizontally arranged. Meanwhile, the vertical axis allows for the identification of the layers' correspondence to the given services. Several examples are provided on the right. The concept of Software as a Service (SaaS) encompasses the utilization of pre-existing, operational programs via the Internet. The way people interact with these apps might occur through diverse interfaces, such as a basic web client or a programming interface. The utilization of rental services situated on the platform layer grants clients the opportunity to leverage preinstalled and preset OSs and software frameworks, simplifying the deployment of applications to the cloud. The potential for customization is inherent in the strategic administration of applications and the deliberate layout of the surrounding environment. The Infrastructure as a Service (IaaS) concept encompasses the provision of lower-level functionalities, as well as the customization of OSs, storage, applications, and occasionally limited capabilities related to networking components [8], [34], [93], [117], [161].

The physical components of this system are commonly located within designated data centers, which serve as the infrastructure for housing the computational power, storage, and interconnection devices. Isolation of those entities from their surrounding environment is of utmost importance in order to ensure uninterrupted functionality. Therefore, the isolation concept encompasses a range of disadvantageous conditions, including natural phenomena such as floods, hurricanes, earthquakes, and so forth. Furthermore, it is necessary to enforce preventive measures in order to effectively tackle and reduce occurrences of sabotage or terrorism, including both physical and digital realms. Securing cloud infrastructures is crucial in order to guarantee the continuous availability of cloud services without any disruption [102].

The building of large-scale data centers has historically been preferred due to the significant expenses associated with design, setup, and maintenance. This approach helps to minimize the necessity for repetitive and costly activities. In addition to the evident benefits associated with adhering to this strategy, there exist various consequences that, based on the specific application context, may

render a CC service inappropriate. As the concentration of functionality increases at a certain site, the potential for a single failure to significantly damage customer services also rises. Moreover, the construction of a smaller number of larger data centers leads to a higher average proximity between clients and their closest data center. The implication of this aspect is twofold: At first, the route by which the data is transmitted may cross international borders, potentially exposing sensitive information to various legal regimes. Additionally, the geographical distance and higher average proximity may result in an undesirable delay for certain applications that for example require strict real-time capabilities [151].

Common denominator of the following edge paradigms is to shift the functionalities of CC capabilities to the edge of the network [134]. They all provide support for some type of multi-tenant virtualization infrastructure

6.2 Improvements through Edge Computing

To recap, EC is defined as "the enabling technologies allowing computation to be performed at the edge of the network, on downstream data on behalf of Cloud services and upstream data on behalf of IoT services" [142]. In order for EC to fully realize its potential, it is necessary for computation to occur in close proximity to data sources, as stated by Shi *et al.* [142].

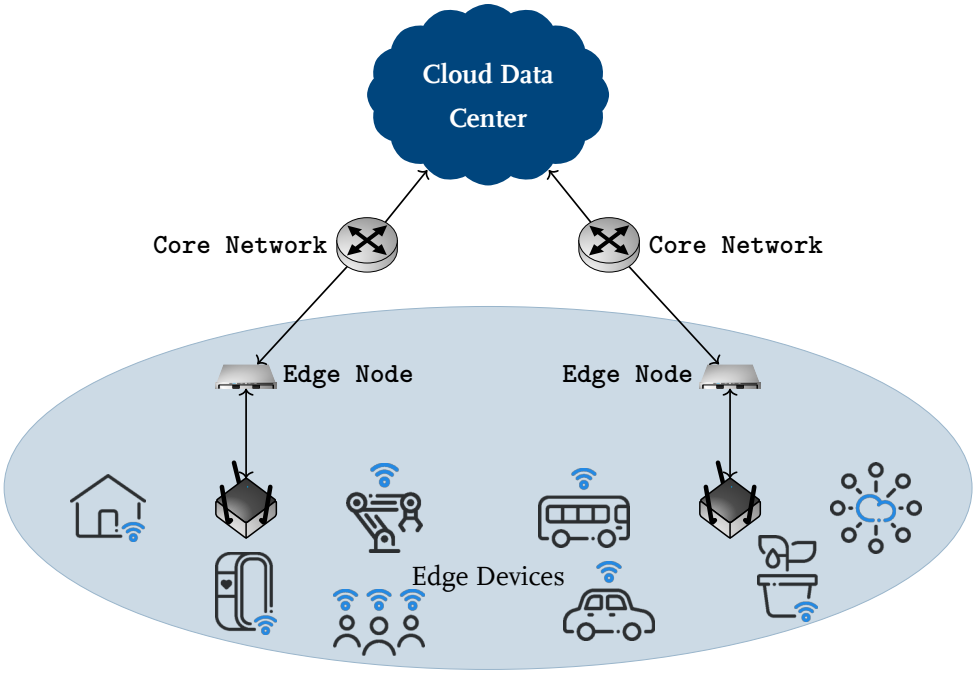


Figure 6.2: The overall picture of the Edge Computing paradigm [cf. 90].

In Figure 6.2, we demonstrate that EC provides processing and storage capabilities in close proximity to the end-users [90], [156]. Mobile phones, smart gadgets, and IoT devices have the ability to communicate with edge gadgets and can even function as EC devices by running their own protocols and services. Given the advancements in computer vision and AI, it is now possible to see modern smart automobiles as nodes for EC. The article by Arena *et al.* [7] provides an overview of the present state and future prospects of intelligent automobiles.

Khan *et al.* [102] present a list of characteristics of EC. Referring to the previous example of mobile phones and smart automobiles, it is evident that numerous computer nodes are already in existence. By observing apartment complexes and roadways crowded with automobiles, it becomes evident that computer devices are omnipresent and cover a vast geographical expanse, encompassing numerous interconnected gadgets, which leads to a concentrated geographical distribution. In order to accommodate the movement of wearable or handheld devices, EC must additionally take into account the provision of additional computing and network capabilities and therefore, support mobility. Scientists create novel protocols, such as the Locator/ID Separation Protocol [52], to facilitate this attribute. As previously said, computation should occur in close proximity to the data sources to be location-aware. If, for example, smartphones are running certain services that generate data, our objective is to do computations on this data nearby to the smartphone. In order to enhance performance by reducing the distance between them, both producer and consumer devices must possess knowledge of their respective locations. Global Positioning System (GPS) is the predominant method for determining the location of a gadget. Contextual judgments rely on the proximity factor, along with other considerations, to determine the utilization or offloading of services. In addition, proximity enables the analysis of user behavior and facilitates actions such as adjusting the allocation of resources and services among the edge nodes. The short distances between consumer and producer devices, along with high-speed network connections for both wired and wireless connections, principally facilitated by 5G technology, result in minimal network delays. The article by Hassan *et al.* [84] explores the influence of 5G technology on EC. Mobile gadgets, particularly smartphones, feature an intrinsic ability to see and understand their surroundings. Smartphones possess a multitude of sensors that enable them to accurately discern our activities at specific moments. EC may utilize this informa-

tion to make informed decisions throughout the network. The article by Han *et al.* [81] provides an illustration of how including context awareness may enhance the effectiveness of an authentication system. As previously mentioned in this section, end devices of EC encompass smartphones, smart automobiles, and IoT gadgets. A prevalent motif across these gadgets is their distinct hardware capabilities and software. Similar to end devices, EC servers might vary greatly as a result of multiple suppliers providing distinct APIs, regulations, and platforms.

6.3 Full Utilization through Fog Computing

With the advent of CC, there is now a growing interest in transitioning into the realm of FC. Once again, we examine the definition as published by the NIST: FC is a resource paradigm that exists between smart end-devices and conventional cloud or data centers, encompassing both physical and virtual components. This paradigm facilitates the operation of vertically isolated, latency-sensitive applications through the provision of ubiquitous, scalable, layered, federated, and distributed computing, storage, and network connections [94]. In this study, we provide a graphical depiction of FC as illustrated in Figure 6.3. It showcases the hierarchical relationship among Edge devices, which are smart end devices utilized by users, Fog nodes, and Cloud data centers.

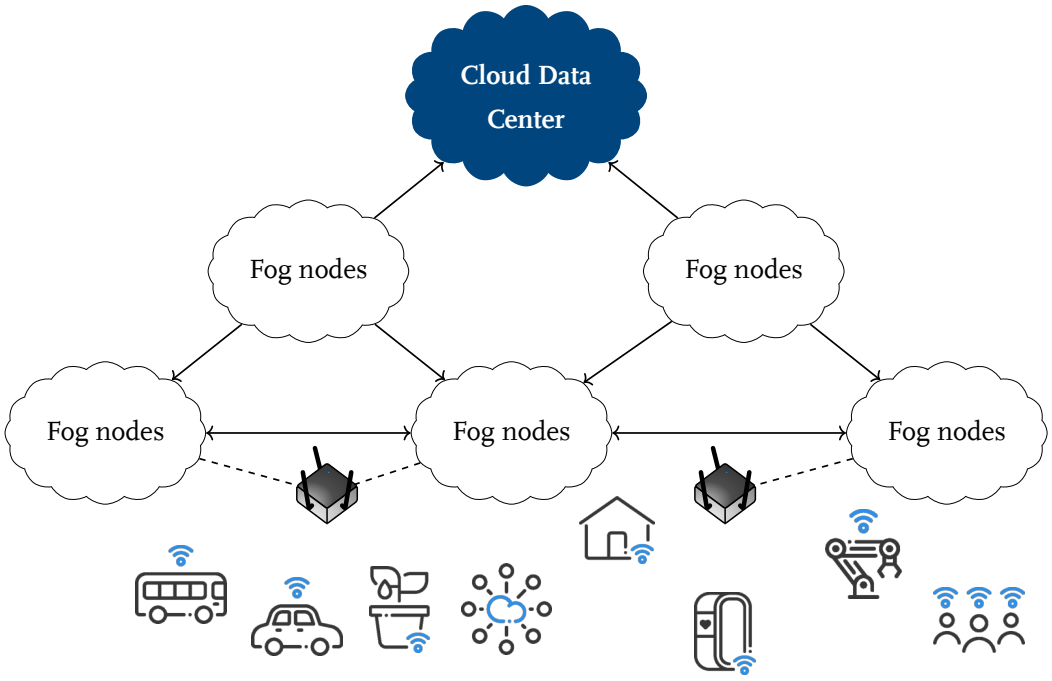


Figure 6.3: The overall picture of the Fog Computing paradigm [cf. 90].

NIST has also established a set of specified features for FC [94]. In terms of contextual location awareness and low latency, it is worth noting that fog nodes exhibit closer proximity to the user's geographical position in comparison to traditional CC servers. As a consequence of this particular circumstance, fog nodes have the capability to leverage the user's geographical coordinates in order to diminish latency, predominantly by benefiting from the reduced physical proximity between nodes and users. The geographical spread of fog nodes is significantly wider compared to the limited locations of centralized CC data centers. The utilization of FC enables the establishment of a large-scale sensor network that is capable of monitoring the environment, thanks to its geographical dispersion. Moreover, due to the geographical dissemination, a tremendous amount of nodes are available in FC networks. Facing mobility, FC enables seamless connections for mobile endpoint devices. Real-time interactions are prioritized by FC services due to the minimal delay reaching end devices, as opposed to batch processing. The prevalence of wireless connectivity is a notable characteristic of FC, as it offers strong support for mobility and geographical dispersion. Consequently, wireless IoT devices are highly compatible with this computing paradigm.

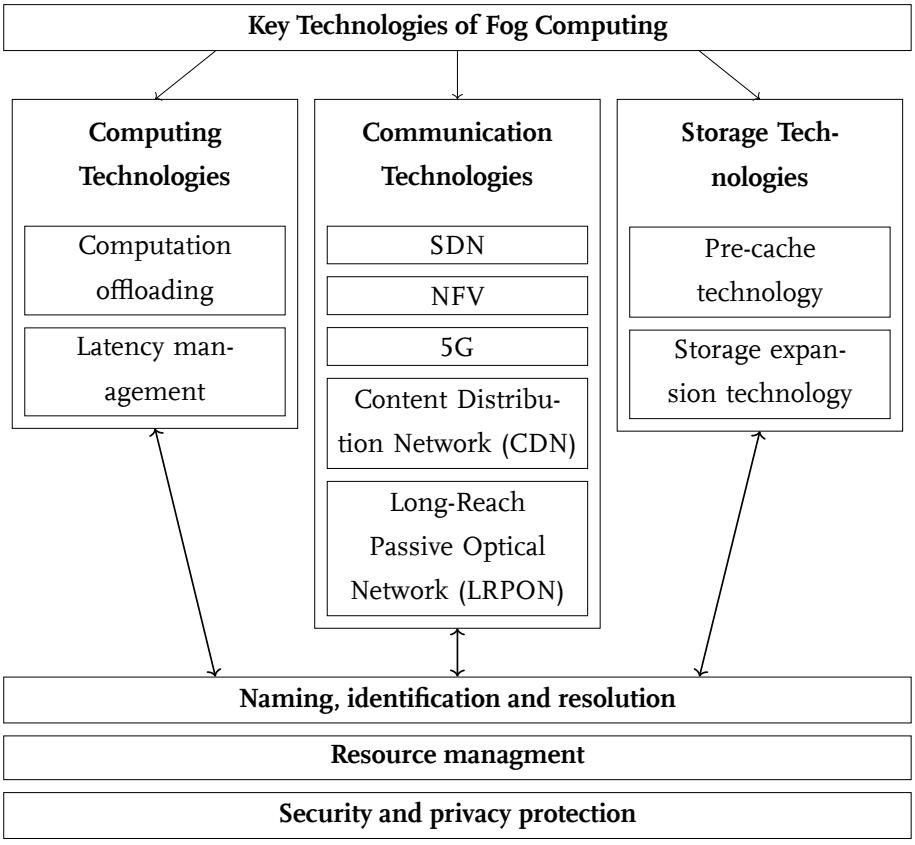


Figure 6.4: Technology stack for Fog Computing [90]

Heterogeneity refers to the presence of diverse resources, surroundings, and network capacity among the devices utilized in FC. Interoperability and federation are essential in ensuring the provision of steady services by several fog providers, particularly in light of their geographical dispersion. Analytics are provided in real time and are easily integrated into cloud infrastructures. By positioning the fog layer between end devices and the cloud, FC facilitates the collection of real-time analytics from these devices, which can then be processed in batches inside a cloud environment.

Essential technologies in the FC paradigm encompass computing, communication, and storage technologies, as well as naming, resource management, security, and privacy protection as shown in Figure 6.4. Computation offloading is a strategy that alleviates resource limitations on edge devices, enhancing performance and conserving battery life. For example, a CDN is an Internet-based caching system that disseminates related content to consumers, thereby minimizing download latency and enhancing response speed. Supported by the LRPON, the network reach is increased to 100 km while accommodating latency-sensitive and bandwidth-intensive applications. FC has been introduced with innovative naming techniques like Named Data Network (NDN) and mobility first to enhance scalability, efficiency, security, and robustness in the present Internet landscape. NDN packets utilize hierarchically structured data names in lieu of source and destination addresses, with the objective of enhancing the scalability, efficiency, security, and robustness [90].

The service models employed in FC bear resemblance to the service models utilized in CC, namely SaaS, PaaS, and IaaS.

	CC	FC	EC
Location	centralized	distributed	at the network edge
Infrastructure	remote data centers	edge to cloud	near devices
Latency	high	moderate	low
Heterogeneity	low	high	medium to high
Scalability	very high	moderate to high	low to moderate
Deployment	fixed, static	dynamic, opportunistic	fixed, opportunistic
Virtualization	heavyweight (e.g. VMs)	mixed	lightweight (e.g. container)

	CC	FC	EC
Processing Power	large-scale, powerful	medium (distributed nodes)	limited (local devices)
Key Advantage	massive compute power	balance between latency and scalability	minimal latency and local data processing
Challenges	Network-dependent, higher latency	Complex management, security	limited resources, device management
Use Cases	big data analytics, SaaS	Industrial IoT (IIoT), smart cities	real-time applications (e.g., IoT, autonomous vehicles)

Table 6.1: Comparison between CC, FC, and EC [cf. 58]

6.4 The Internet of Things

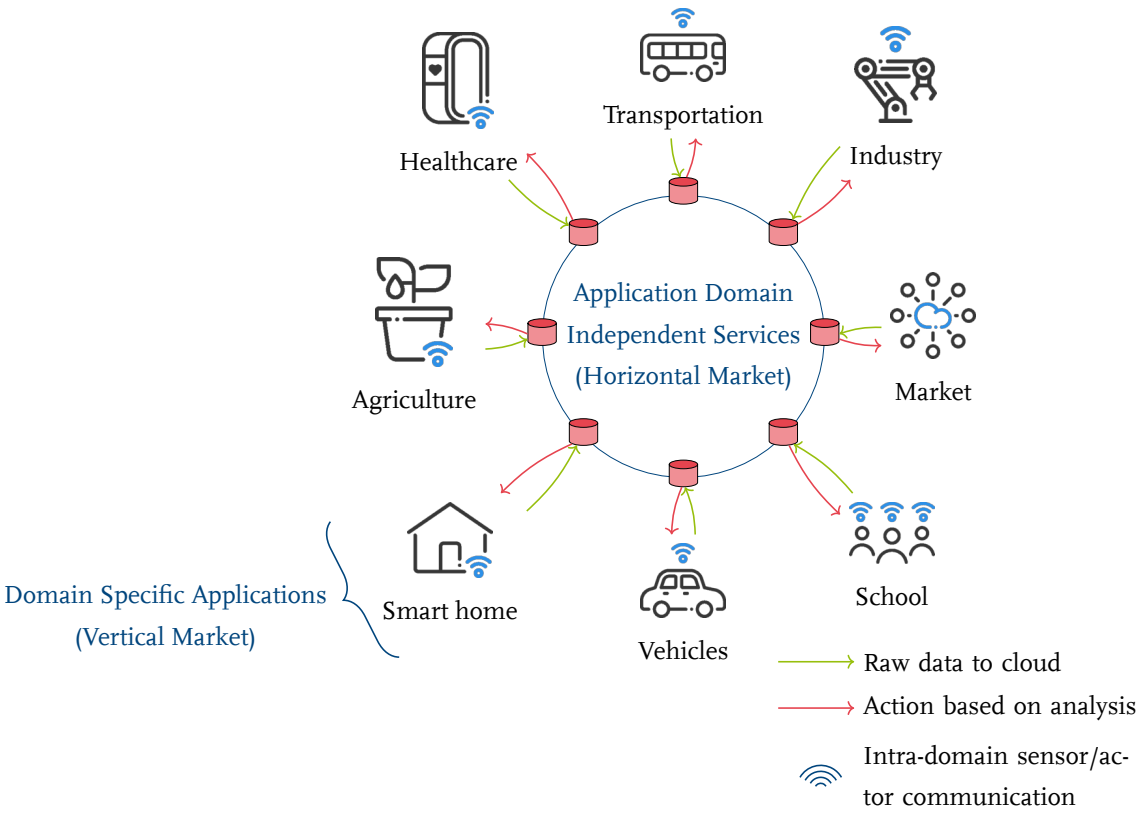


Figure 6.5: The overall picture of IoT [cf. 3].

In 1999 the expression "Internet of Things" was originally applied by Kevin Ashton, a British technological innovator and co-founder of the Auto-ID Center at the Massachusetts Institute of Technology (MIT). The author's initial aim was to employ Radio-Frequency Identification (RFID) as a method to automate the management of all things. The name "Auto-ID" encompassed several identifying technologies such as barcodes, smart cards, sensors, speech recognition, and bio-informatics. Since 2003, the notion of the IoT began to emerge outside the boundaries of the Auto-ID center. Throughout the course, other authors submitted broad elements, such as technological phrases like *machine of the future*, which explores the concept of machines acquiring the ability to *perceive*, and commercial components like *the world of mainstream commerce* [152].

The IoT involves the interconnection of all objects to the Internet, and its outcomes are not entirely foreseeable. The IoT is the result of both technology advancements and human demand. The IoT experience is a novel technology approach that will have a significant influence on how product developers and company owners design their products using software, hardware, data analytics, open standards, innovation, and interoperability. The convergence of technical advancements and human requirements, together with the rising interconnectedness with the environment, renders the IoT very disruptive. This disruption will profoundly alter both individuals' daily routines and commercial frameworks [105].

Determining the range of the interconnected "things" is a complex and abstract task, since their significance varies in different settings. Instead of attempting to provide specific details, consider the implications of a fully interconnected system. This will assist in evaluating the magnitude of the influence that can be noticed inside a community. The application of connection enables the real-time tracking and consolidation of data. The collected data can be leveraged to enhance the process of creating smart decisions. Conversely, every component of a product life cycle, including the product itself, the equipment, and transportation, may be seen as an entity. In the context of the environment, this encompasses elements such as trees, buildings, and instruments used for measuring conditions. In the context of society, it refers to all the technologies that are part of our daily lives. Such aggregation is seen in Figure 6.5.

The fundamental distinction between the structural arrangement of protocols that drive the Internet, which are often referred to as the Open Systems Interconnection (OSI) model (see Chapter 7), and the IoT protocol stack is situated in the physical layer. Wireless connection is the primary method of communication in IoT, utilizing technologies such as RFID, ZigBee, WiFi, Local Area Network (LAN), 4G, 5G, and others. A modification at the physical layer denotes an alteration at the infrastructural level, known as the *technological push*. In addition, to enhance this technology advancement, it is imperative to have a corresponding response from the application level, as well as, widespread human acceptance. Ultimately, the application layer establishes the business logic that mirrors the user's requirements.

The rate at which the IoT is adopted can be significantly higher because of the novel problems it presents. Working with complex data streams generated by diverse infrastructure can be quite demanding. Occasionally, machines with limited resources need to do fast calculations and use communication channels with minimal delay in order to make efficient judgments. A conventional host-cloud architectural approach is inadequate in properly addressing the difficulties it presents. In order to address these issues efficiently, it is crucial that we possess a thorough comprehension of the situation. Chiang and Zhang [31] thoroughly examined these problems and following are some of the significant challenges mentioned.

Real-time communication between machines (M2M) is greatly sought after in the industrial sector. In order to achieve precise control and thorough monitoring in industrial control systems or IoT applications such as autonomous driving, vehicle-to-vehicle communications, advanced traffic control systems, virtual reality, gaming, or real-time financial trading, it may be necessary to possess a very low latency. In addition to facilitating various applications, reduced latency is crucial for attaining a superior QoS in real-time audio and video applications. This is an exceptionally demanding need in any situation.

The volume of data generated by an IoT system might be immense. The notion of a smart city is based on the aggregation of data from various municipal assets, including citizen mobility information, smart vehicle transportation systems, utility management, trash management, power plant operations, crime detection, and citizen services. Connected components will produce a substantial volume of data; an autonomous car can create one GB of data each second. The United States is

projected to produce 100 Petabytes of data annually through the implementation of a smart grid. Transferring all of this data to the cloud necessitates a network connection with exceptionally fast speeds.

Frequently, the end devices exhibit significant limitations in terms of resources, such as sensors, actuators, controllers, and so on. Performing computationally demanding activities on those devices is quite tough. Hence, performing tasks like a software update or increasingly demanding cryptographic computations to ensure secure communication with the server pose a challenge.

Cloud providers may have challenges in delivering uninterrupted services to a remote location. For instance, consider an oil rig located in the middle of the ocean or in a distant area where the only means of accessing the Internet is by satellite connection, which might be unstable and have sporadic availability.

The current approach to Internet security, which relies on Intrusion Detection and Prevention Systems located behind the firewall, is not an effective answer for IoT devices. The current infrastructures lack the capacity to safeguard resource-constrained devices through the storage of several security credentials or the regular updating of software.

In addition to the aforementioned issues, there may be other obstacles encountered throughout the management of the software life cycle, such as safeguarding security credentials and effectively responding to security breaches without causing significant harm.

The proliferation of IoT devices connected to the cloud is steadily growing. Additionally, it has become evident that cloud-based solutions are not universally applicable for all applications. The cloud is unable to fulfill the objective of low latency applications and establishing an efficient communication channel between computers. Hence, the imperative to discover a more optimal resolution is evident.

Chapter 7
Emerging Network Concepts

Prior to exploring the new virtualized network paradigms, we will briefly examine the theoretical OSI model, a conceptual framework utilized to comprehend and execute network communications among disparate systems. It standardizes the operations of telecommunication or computing systems into seven levels, each fulfilling a distinct role and communicating with the adjacent layers above and below. The model was established by the International Organization for Standardization (ISO) in 1984 and assists in the design and troubleshooting of networking systems.

The following seven layers – bottom to top – exist in the OSI model [cf. 4]:

Layer 1: **Physical Layer**

Function: Concerns the physical linkage between devices, including the transfer of unprocessed bit streams across a physical media (cables, radio waves, etc.).

Examples: IEEE 802.3 PHY (Ethernet), IEEE 802.11 PHY (WiFi), Bluetooth PHY.

Layer 2: **Data Link Layer**

Function: Accountable for node-to-node data transmission and error detection/correction inside the physical layer. It also delineates the formatting of data for transmission.

Divisions: This layer is often divided into two sublayers:

- Medium Access Control (MAC): Handles how devices on the same network share the medium.

- Logical Link Control (LLC): Manages frame synchronization, flow control, and error checking.

Examples: MAC, IEEE 802.3 MAC, IEEE 802.11 MAC, Point-to-Point Protocol (PPP), Address Resolution Protocol (ARP).

Layer 3: **Network Layer**

Function: Manages the routing of data between different networks and handles packet forwarding. It determines the best paths for data to travel from sender to receiver.

Key Concepts: Logical addressing, routing, packet fragmentation.

Examples: Internet Protocol (IP), IP Security (IPSec), Internet Control Message Protocol (ICMP), Open Shortest Path First (OSPF), Routing Information Protocol (RIP).

Layer 4: **Transport Layer**

Function: Ensures reliable data transfer between systems, managing error correction, retransmission, and flow control. It can provide either reliable or unreliable service depending on the protocol.

Key Concepts: Segmentation, flow control, error correction, connection-oriented and connectionless communication.

Examples: Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Stream Control Transmission Protocol (SCTP), Datagram Congestion Control Protocol (DCCP), QUIC.

Layer 5: **Session Layer**

Function: Manages sessions or connections between applications. It establishes, manages, and terminates communication sessions between two systems.

Key Concepts: Session establishment, maintenance, and termination.

Examples: Sockets, Real-time Transport Protocol (RTP), Point-to-Point Tunneling Protocol (PPTP), Remote Procedure Call (RPC), Structured Query Language (SQL), Network File System (NFS).

Layer 6: **Presentation Layer**

Function: Deals with data translation, encryption, and compression. It ensures that the data sent from one system is readable by another, regardless of differences in data format.

Key Concepts: Data encoding, encryption, and data conversion.

Examples: Encryption (Secure Sockets Layer (SSL)/Transport Layer Security (TLS)), Multipurpose Internet Mail Extensions (MIME) types.

Layer 7: **Application Layer**

Function: Provides services directly to user applications and is closest to the end user. It ensures communication with the lower layers via protocols that interact with network resources.

Key Concepts: End-user applications and their interaction with the network.

Examples: Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP).

The OSI model simplifies networking design, troubleshooting, and protocol development by breaking down the complex process of network communication into manageable layers. Nevertheless, the TCP/IP model with four layers, developed by the U.S. Department of Defense (DoD), is more practical in real-world Internet and networking as it combines layers 5 to 7 into one, called the application layer, and does not consider the physical layer. Therefore, it is simpler and directly tied to the suite of protocols that underpin the Internet. To avoid confusion, we will always refer to the OSI model, if we point to a certain layer throughout this dissertation.

7.1 Software Defined Network as Virtualization Enabler

The examination of network technology becomes pertinent following the exploration of the transition from constructing internal, private data centers to the use of public cloud services, and the subsequent decentralization of some functionalities towards the network edge. The adaptation of the supporting networks and their configuration is necessary in response to the emerging computer paradigms. Requirements arising from the paradigm shift include increased traffic volumes, enhanced transmission rates, and improved processing and storage capabilities.

For an effective management of the growing computing capabilities, future networks need to increase their degree of flexibility and dynamic behavior.

In addition to the proliferation of CC, the emergence of distributed architectural patterns, including web services and microservices, has significantly contributed to the increased dynamism, adaptability, and perpetual evolution of communication processes. Considerable traffic is generated among multiple workstations, particularly in a design that follows a microservice-oriented approach, prior to delivering a suitable outcome to the end-user. In addition, it is worth noting that end users have the ability to utilize a wide range of services on various devices, regardless of their location or the time of day [127]. The seamless operation of the increasingly interconnected applications developed within this extensively distributed paradigm is contingent upon the accessibility and functionality of all constituent components. The integrity of these two components is jeopardized by the presence of faults and the variability of load conditions. The existing networks lack the capability to alter policies that regulate access to services as required automatically. Automated response mechanisms are necessary in the domain of network reconfiguration in the event of faults [106]. Both of these solutions are deficient in terms of support and hence require manual configuration, wherein the desired modifications are implemented on "[...] every single hardware instance [...]" [78]. The resulting intricacy of these networks leads to a state of near immobility, as the required maintenance, upgrade efforts, and associated risks of service disruption experience a significant increase. The phenomenon of impeding innovation and hindering the progress of the Internet and network technologies is commonly referred to as "ossification" [126].

The traditional network architecture involves the integration of both the control and data plane within individual devices, while the SDN concept separates the devices' focus on traffic forwarding from the control aspect, which is separated into a distinct layer. The presence of SDN controllers (see Section 7.3) at this layer enables a conceptually centralized approach to managing and programming interconnected switches, routers, and other middleboxes via a standardized protocol. The process of decoupling significantly decreases the level of effort required when there is a need to modify the network structure or policies. Additionally, numerous SDN controllers offer support for the incorporation of individual applications. This functionality allows to extend the controller to suit the requirements

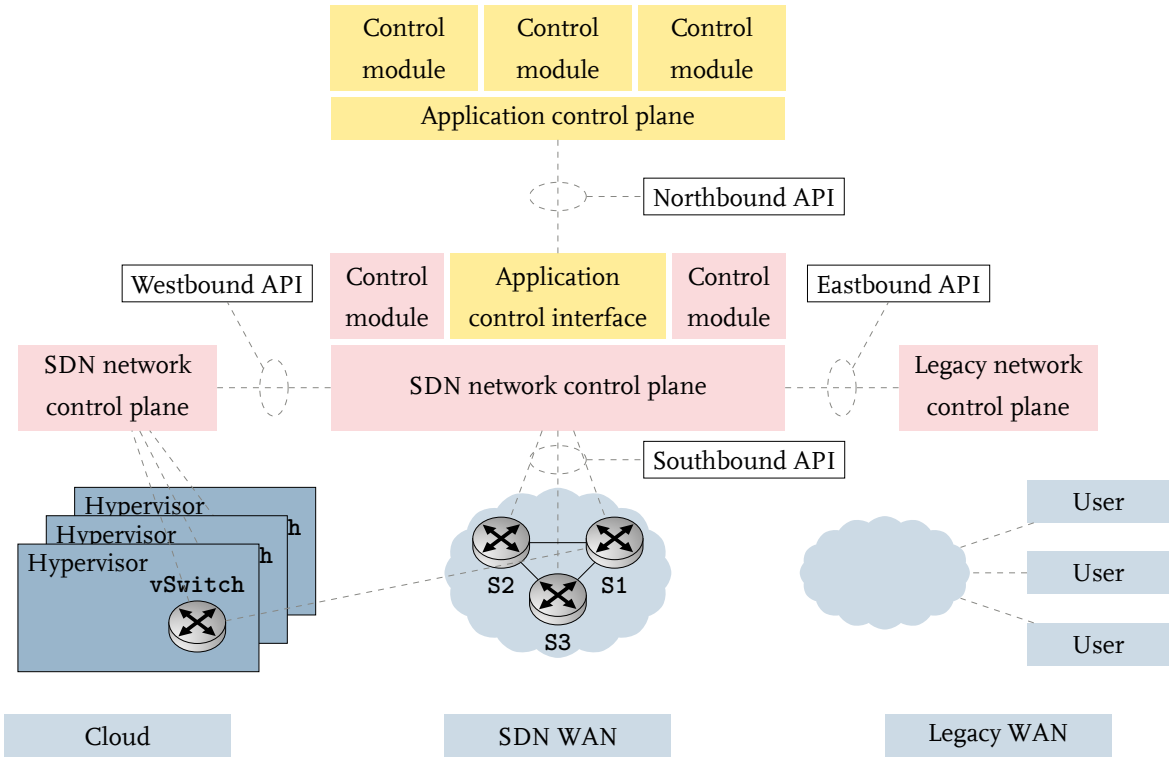


Figure 7.1: Architecture and integration of the SDN concept [cf. 97]

of a given network, eliminating the need to rely on a single vendor to integrate the desired feature in a timely manner.

For the programming of network hardware a standardized communication protocol is utilized, which is determined based on the specific requirements of the controller and the controlled switches. OpenFlow is a protocol that is utilized for this particular objective, as further described in Section 7.4. The API responsible for establishing a connection between the controller and the network infrastructure using a specific protocol is referred to as the southbound API, as depicted in Figure 7.1. The SDN controller software is housed within the middlebox. Multiple instances of the controller can establish communication with each other, with their spatial orientation predominantly towards the left. The westbound API, commonly referred to as such, facilitates the decentralization of network management responsibilities across many controllers, which ensures increased availability, fault tolerance, and scalability. Furthermore, separation of concerns can be achieved by assigning the responsibility of managing distinct and logically cohesive aspects of the network architecture and its associated devices to separate controller instances. The eastbound API, positioned on the right side, permits communication with conventional networks that are not based on SDN, hence

ensuring compatibility with legacy systems. Their effective interaction relies on the utilization of particular protocols and frameworks inside the related legacy networks.

The distinction between the direction of travel (eastbound and westbound) in the API is not consistently specified in architectural requirements. This is demonstrated by its absence in the research conducted by Sezer *et al.* [139]. Their solution entails the utilization of both eastbound and westbound APIs to facilitate inter-cluster communication across various instances of controllers. The diagram additionally extends vertically through the utilization of the northbound API, which establishes a connection between an application context and the SDN controller. This exemplifies the potential for growth through the integration of customized applications that can utilize the fundamental services offered by the SDN controller [89], [96], [97], [126], [139], [143].

7.2 Deploying Services by Network Function Virtualization

The enhancement of performance in a conventional network has historically been accomplished through the utilization of middleboxes that fulfill highly specialized functions. The construction of a device with such a high degree of specialization presents opportunities for optimizing power consumption, efficiency, and output that may be anticipated. Sherry and Ratnasamy [141] conducted a study in 2012 to analyze the distribution of hardware throughout 57 industry and academic networks. Their approach employed in this study was soliciting feedback from operators on their experiences with specialized network equipment. Specifically, operators were asked to provide information on the amount of these appliances, as well as any associated issues, expenses, and the level of comfort in managing them. The results provide a concise overview of the disadvantages associated with the dependence on the implementation of these intermediate devices, known as middleboxes, in the construction and administration of a network. The numerical value is significantly elevated, perhaps reaching a level comparable to that of critical networking equipment such as layer 3 switches. The implementation of a substantial quantity of physical and highly specialized equipment necessitates significant initial expenses and investments. This issue mostly affects enterprise and Internet Service Provider (ISP) networks, since they tend to house vertically integrated and highly specialized middleboxes. Common responsibilities encom-

pass the enhancement of the Wide Area Network (WAN), identification of unauthorized access attempts, implementation of protective barriers such as firewalls, and several other duties.

In addition to the initial acquisition cost, there are additional expenses associated with the installation and maintenance of the boxes, resulting in administrative and operational costs. Given the private nature of these devices at both the hardware and software levels, it is imperative for employee training to account for the unique configurations provided by the vendors. The choice to invest in a particular implementation of middleboxes might be influenced by this factor, since lacking the necessary skills may pose a significant risk to prospective profitability. The improvement of scalability and fault tolerance is frequently achieved by the incorporation of supplementary and redundant devices into the network, hence increasing operating expenses. Ultimately, when fresh features become available in the market, there is little potential for upgrading outdated middleboxes. The sole method of utilizing contemporary functionalities is to substitute the relevant hardware [11], [116], [141], [158].

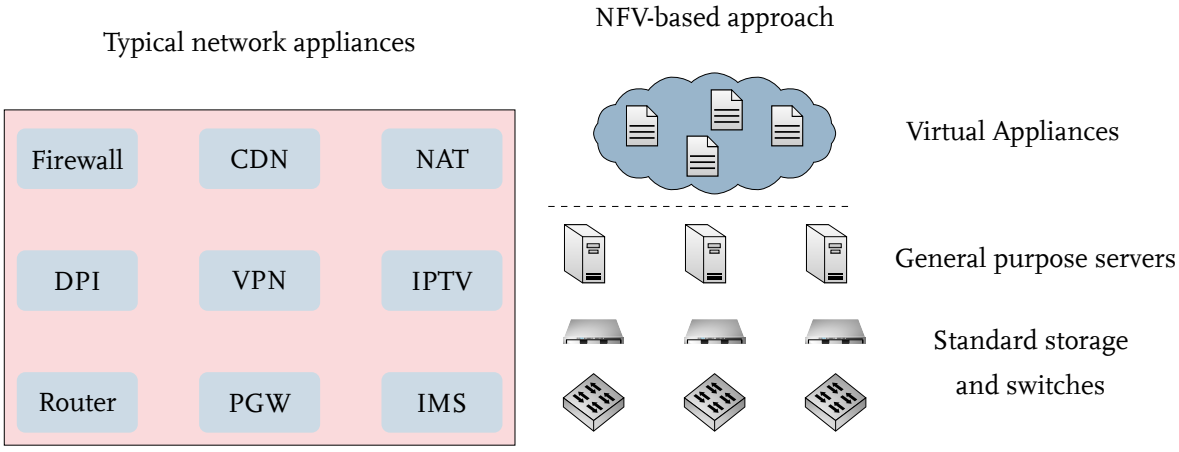


Figure 7.2: Virtualization of traditional network appliances by NFV [cf. 82]

The core principle of NFV is to shift focus from application-specific integrated circuits to software-based solutions that are executed on general-purpose hardware. Initially, the notion of NFV was introduced in a white paper published by a specialized interest group affiliated with the European Telecommunications Standards Institute (ETSI) in 2012 [123]. Subsequently, the aforementioned approach has gained significant recognition and is being properly considered in the strategic development of future networks [128]. The fundamental objective, as derived

from the introduction white paper, is to dismantle the strong interconnection between functionality and tangible device. The diagram depicted in Figure 7.2 illustrates the progression from a collection of closely linked hardware middleboxes to a system in which the needed services are delivered in virtualized form by autonomous software providers (located in the top right corner). The foundational hardware comprises equipment that are now present in data centers, such as conventional high-volume servers, storage units, and Ethernet switches. The process of deploying virtual appliances has evolved from a laborious and manual setup on a per device basis to a more streamlined and automated approach that can be executed remotely through orchestration. The definition of these services has been separated from hardware development, leading to the growth of a more expansive software ecosystem. Within this particular context, the availability of potential network functions experiences an increase, hence allowing for the acquisition of updated or novel functionality through remote means. The distribution and utilization of network functions is greatly enhanced when implemented inside a network environment that adheres to the concepts of SDN. The whitepaper [123] explains the core concepts, including a chapter that explicitly outlines the symbiotic association between NFV and SDN. SDN has the potential to enhance performance, simplify compatibility with existing systems, and ease operational and maintenance processes, where NFV plays a crucial role in supporting SDN by providing the necessary infrastructure to run network appliances.

7.3 Controller in Software Defined Networks

The “brain” of a SDN is called controller and handels the interactions as shown in Figure 7.1. It must provide the capability to interact to service providers via the northbound API, which can communicate QoS requirements for traffic flows with it. Those instructions can be handeled by the controller instance and set up on the switches through the southbound API using OpenFlow.

Besides, the controller must be able to interact with other networks either via another SDN controller or with traditional networks. Therefore, hybrid SDN demands to integrate and interact with traditional networks within the virtualized SDN domain [104].

Of course, lot of different approaches occured to implement a controller for SDNs, where the basic principles are depicted in Table 7.1. Thereafter, we discuss current implementations and show a comparison in Table 7.2

	Pros	**Cons**
Centralized	reduced costs and complexity of network	high control latency
	easy to deploy	bandwidth issues
	consistency easier to achieve	difficult to achieve scalability and failure resilience
Distributed	solves scalability, reliability, performance and Single Point of Failure (SPoF) problems of centralized paradigm	Permanent state sharing among distributed controllers
	More robust, scalable, flexible and responsive	Challenges in interoperability, consistency, controller placement
Hybrid	Simplifies the shift from legacy distributed control to SDN	Limitations due to the fusion of different devices in a hybrid network
	Two control planes can complement each other, providing optimization and better UX	Higher complexity of network configuration, topology discovery, and overall network control
	Fine-grained control of traffic flows and better performance	

Table 7.1: Pros and Cons of different SDN control plane architecture paradigms [cf. 1].

NOX[22], along with its Python-based counterpart POX[23], emerged as one of the initial SDN controllers, developed at Stanford University. It facilitated the opportunity to conduct experiments including novel network control protocols and algorithms, where researchers were able to create and evaluate new SDN applications in a versatile and customizable setting. Although NOX is no longer actively developed, POX is still in use by the community.

[22]https://github.com/noxrepo/nox
[23]https://github.com/noxrepo/pox

Ryu[24] is a Python-based open-source SDN controller with a modular structure and comprehensive documentation that makes it highly favored among developers. It is extensively utilized in academic and industry environments with a thriving community of developers.

Floodlight[25] is a Java-based open-source SDN controller overseen by the Open Networking Foundation (ONF) that offers a strong and adaptable framework for implementing SDN solutions in real-world settings. Besides the compatibility with OpenFlow, it offers programmability via REST APIs, which enterprises and organizations have widely embraced for constructing SDN-based networks.

OpenDaylight[26] is an open-source SDN controller platform that is community maintained and hosted by the Linux Foundation. Their objective is to expedite the implementation of SDN and NFV by offering a shared platform for cooperation and innovation. Therefore, OpenDaylight provides a modular structure, allowing for a high level of customization and expandability.

Frenetic[27] is a Domain-Specific Language (DSL) that was created at Cornell University, which defines network policies and configurations at a higher degree of abstraction. Therefore, it automates the translation of policies into specific data plane configurations, enabling network administrators to prioritize policy intent above implementation specifics. It has been employed in research initiatives and academic settings for the purpose of conducting SDN experimentation and facilitating development. In the initial approach [55], Frenetic communicates with NOX to setup it's specific settings.

Pyretic is the Python-based enhancement of Frenetic and offers an advanced programming language and runtime created at Princeton University, which allows network managers to articulate intricate network policies. It automates the process of converting policies into optimized data plane configurations, making it easier to design and manage SDN applications. It is very suitable for managing network policies and has been utilized in both research and educational environments.

[24]https://ryu-sdn.org
[25]https://github.com/floodlight/floodlight
[26]https://www.opendaylight.org
[27]http://www.frenetic-lang.org

	NOX	POX	Ryu	Floodlight	ODL	ONOS	Trema
Language	C++	Python	Python	Java	Java	Java	C & Ruby
Performance	Fast	Slow	Slow	Fast	Fast	Fast	Fast
Distributed	No	No	Yes	Yes	Yes	Yes	No
OpenFlow support	1.0	1.0	1.0, 1.1 1.3, 1.4	1.0	1.0, 1.3	1.0-1.4	1.0, 1.1 1.3
Multi-tenant Clouds	No	No	Yes	Yes	Yes	Yes	No
Learning Curve	Moderate	Easy	Moderate	Steep	Steep	Steep	Steep

Table 7.2: Comparison of important SDN controller implementations.

Procera [153] has been developed as an open-source project at the University of Wisconsin-Madison with the primary objective to deliver scalable and effective solutions for network monitoring and management. Procera provides functionalities such as instantaneous traffic analysis, adaptable policy enforcement, and network diagnostic tools.

RouteFlow[28] has been created as an open-source project at CPqD in Brazil. The system combines OpenFlow-based SDN with traditional IP routing protocols, such as Border-Gateway Protocol (BGP), to facilitate the smooth integration of SDN into pre-existing networks. RouteFlow serves as an intermediary between SDN and conventional networking, enabling operators to transition to SDN in a stepwise manner without causing any disruptions to the current network architecture.

The open-source platform Trema[29] was developed by NTT Labs, with a C extension. The platform offers a versatile and expandable framework for constructing SDN applications and exploring novel networking protocols and algorithms. Trema provides a comprehensive range of APIs and libraries for creating personalized SDN controllers and apps. Although Trema has been utilized to some extent in research and educational settings, its implementation has been rather restricted in comparison to other controllers.

Open Network Operating System (ONOS)[30] is open-source and was developed by ON.Lab, which is now a part of the Linux Foundation, along with other collab-

[28]https://routeflow.github.io/RouteFlow/
[29]https://github.com/trema/trema
[30]https://opennetworking.org/onos

orators. It offers a distributed control plane for SDN deployments on a wide scale and provides scalability, high availability, and fault tolerance.

A complete list of SDN controllers can be found by Zhu *et al.* [163]. We give a brief overview of relevant implementations in Table 7.2 by integrating the results of [103], [113], [136], [163].

In this work, we focus on the open-source controller ONOS to create our SDN solutions, and adjust the controller's functionality. ONOS offers an OS for software applications that are customized for certain use cases or goals, including SDN administration, routing, or monitoring. In the case that one server fails, it can function as a distributed system over several servers, increasing resilience and fault tolerance [13]. Because ONOS is broken up into numerous smaller projects as shown in Figure 7.3, each with its own source tree, it can be built and extended independently[31]. To enable autonomous construction, it uses a hierarchical Project Object Model (POM) file and gives each sub-project its own `pom.xml` file. Protocol-aware network-facing modules, a protocol-agnostic system core, and applications that use and act upon the data supplied by the core comprise the ONOS project's components.

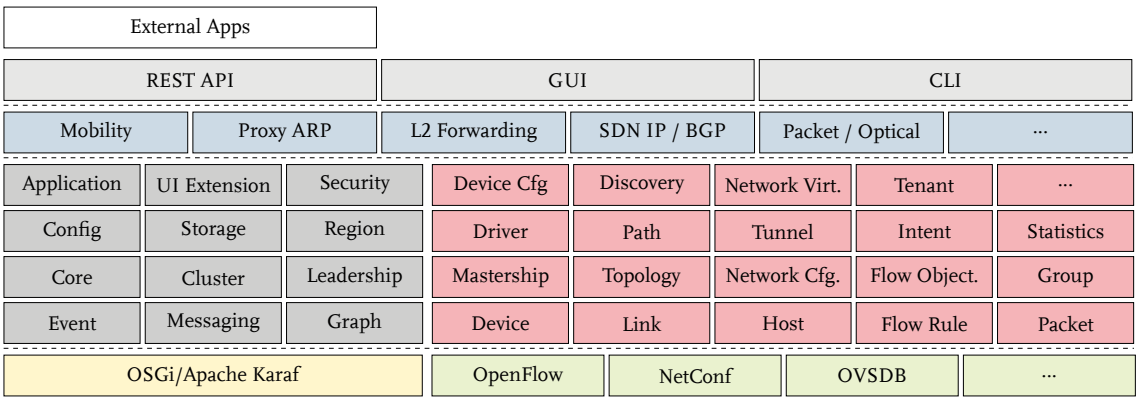


Figure 7.3: Current and future subsystem[31] of the ONOS controller.

Using a Karaf[32] OSGi container, ONOS apps, Kernel, and services are written in Java Programming Language (Java). Java's OSGi module framework is used to build dynamic, decoupled, and reliable applications. It provides segregated class loaders, service-oriented design, dynamic updates, and modularity. Because Karaf employs a polymorphic application container, it may be deployed using Docker images, Java Virtual Machines (JVMs), or CC resources in numerous locations.

[31] https://wiki.onosproject.org/display/ONOS/System+Components
[32] https://karaf.apache.org

The ONOS software is a distributed system that may run on several OSs and is intended to program and control a switch's data plane. It can run on platforms that support the installation of a JVM. An extension of the Karaf CLI, the ONOS CLI provides Secure Shell (SSH) access and details on devices, ports, groups, current flows, and linkages. The ONOS Web GUI is a one-page web application that gives the ONOS controller a visual interface.

An essential component of the system, the Manager component receives data from providers and distributes it to applications or other services through a variety of interfaces. The application and essential components receive information and details about the current network condition from the Northbound Service.

Through the AdminService, applications obtain and process data from managers; each application is given a distinct ApplicationId. While events alert listeners to network changes and promote peer awareness, descriptions help data move through the Southbound API. Only the intended recipients are reached thanks to event dispatch.

For information to be maintained and used for other applications, network state construction is essential. Network setup and network discovery are the two primary tools that ONOS provides for its protocol-agnostic topology. Device, Port, Host, Link, EdgeLink, Path, and Topology are among the elements that make up the graph that represents the ONOS network topology.

"Match + action" pairs are used to set network control at the application level. With various role models the ONOS instance's role controls whether modifications are applicable to the device in question.

To sum up, ONOS is a system that preserves data and renders it accessible for various purposes. To administer the network and guarantee effective communication between apps and devices, it makes use of a variety of parts and interfaces.

7.4 The OpenFlow Protocol

OpenFlow is a standardized protocol that represents the fundamental idea of SDN by defining the tie between the control and data plane [88]. It standardizes the communication occurring at the controller's southbound interface by specifying the content, structure, and actors participating in the data exchange. OpenFlow is overseen by the ONF, and there are several versions of the standard in dissemination. The initial version was released in December 2009.

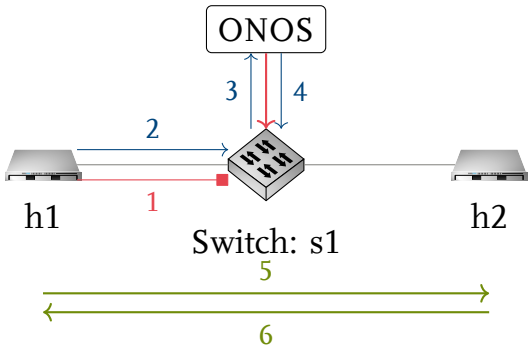


Figure 7.4: Connection establishment in SDN. Normally traffic is blocked (1), until the controller recognizes the flow and sets up forwarding rules (2 – 4). Finally, (bi-)directional communication is possible (5 – 6) [cf. 73].

Normally, distributed applications often provide data in the form of a continuous stream of packets between the server and the client. OpenFlow specifies a "flow" as a defined sequence of packets transmitted between two endpoints that are consistently forwarded in the same manner. The controller leverages the OpenFlow protocol to set up these flows and transmit their definitions to the switches.

OpenFlow switches can be categorized as either specialized OpenFlow switches that exclusively support OpenFlow Protocol or as general-purpose commercial Ethernet switches and routers that have integrated OpenFlow Protocol and interfaces as an additional feature [62]. The switches need to perform three fundamental functions: forwarding packets to a specific port, encapsulating and forwarding packets to a controller, and deleting packets for security purposes or in response to denial of service assaults.

The specific criteria for an OpenFlow Switch are outlined in the OpenFlow Switch Specification[33]. OpenFlow-enabled switches can be upgraded with the OpenFlow functionality with the addition of the Flow Table, Secure Channel, and OpenFlow Protocol. The switches usually utilize current hardware, like a Ternary Content Addressable Memory (TCAM), and adapt the Secure Channel and Protocol to operate on the switch's OS.

[33]https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

- The principle behind consists of wildcarded matches:
  - *Layer 1*: input port
  - *Layer 2*: source MAC, destination MAC, Ethernet frame type, ...
  - *Layer 3*: source IP, destination IP, ...
  - *Layer 4*: source TCP port, destination TCP port, ICMP type, ...
- Combinations may be limited by different hardware Application-Specific Integrated Circuits (ASICs)
- and are sometimes configurable, but sometimes not.
- Software implementations (e.g., OvS) are usually flexible

Packets of a decreased size are often forwarded to the controller when there are no flow entries in the switch's cache or when the current flow entry has expired. Truncating those packets decreases traffic between the controller and switches, minimizing delays in forwarding. Usually, only the initial packet needs to be exchanged between the switch and controller as shown in Figure 7.4.

- First, `h1` wants to communicate with `h2`, but
- the OpenFlow switch stops `h1` talking to `h2` (1)
- until a magic unlock token is seen (2, 3, 4).
- Then `h1` is allowed to communicate with `h2` (5, 6).
- No assistance is required from either `h1` or `h2`

An OpenFlow flow consists of a list of matches, allowing for assignments such as directing traffic to a certain port of a TCP connection. OpenFlow allows for a wider range of filters to be utilized and enables the adjustment of the priority of a flow entry. Fields from the package header of Ethernet, IP, and transport layer protocols are allowed. (ICMP, UDP, TCP, or SCTP).

7.5 The Implementation of an Open vSwitch

OvS is a system that relies on two primary components for packet forwarding [cf. 131]: `ovs-vswitchd`, a userspace daemon, and a datapath Kernel module. The datapath module processes packets from a hardware Network Interface Card (NIC) or a VM's virtual NIC based on instructions. In the second scenario, the packet is delivered to `ovs-vswitchd`, which decides how to process it and returns it to the datapath with the specified treatment. Flow caching has developed from a microflow cache to a more advanced structure consisting of two layers: a microflow cache and a megaflow cache. OvS is frequently utilized as SDN switch, with Open-

Flow being the primary method of controlling forwarding. OpenFlow enables a controller to manage flow tables by adding, removing, updating, monitoring, and obtaining statistics on flows. It also allows the controller to redirect certain packets to itself and inject packets from the controller into the switch. The flow programming model of OvS significantly influences the supported use cases and includes numerous changes to regular OpenFlow to cater to network virtualization. Key components of this design that are crucial for performance are packet classification and the interface between the Kernel and userspace.

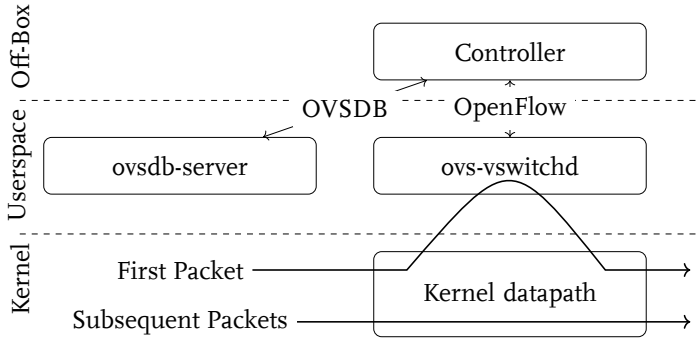


Figure 7.5: Components of an OvS. It is clearly visible that every new "first packet" is forwarded to the userspace and matched against the routing tables. If there is a corresponding forwarding instruction found, the routing of "subsequent" packets is performed in the Kernel space directly. [cf. 131]

DevOps Principles

CHAPTER 8

"DevOps is about fast, flexible development and provisioning business processes. It efficiently integrates development, delivery, and operations, thus facilitating a lean, fluid connection of these traditionally separated silos. In this instalment of Software Technology, Gorka Gallardo, Josune Hernantes, Nicolas Serrano, and I present a brief overview of most recent DevOps technologies such as delivery tools and microservices and discuss what they mean for industry projects. I look forward to hearing from both readers and prospective column authors." – Ebert *et al.* [45]

8.1 The Lifecycle of the DevOps Paradigm

Jabbari *et al.* [95] define DevOps as "[...] a development methodology aimed at bridging the gap between Development and Operations, emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment utilizing a set of development practices."

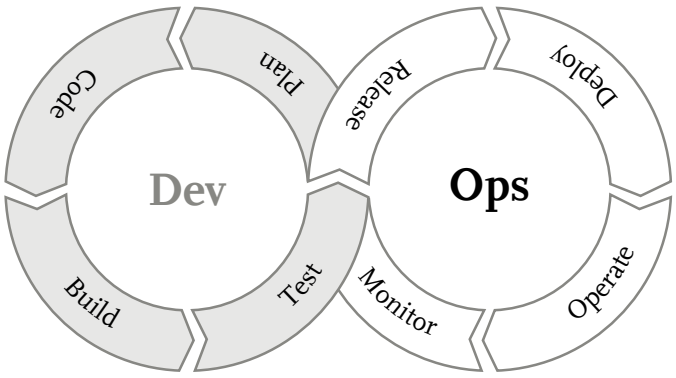


Figure 8.1: DevOps life-cycle [cf. 35]

The DevOps lifecycle has eight stages, where four of them are responsible for the transition between development and operations: *Planning*, *Coding*, *Building*, *Testing*, *Release*, *Deployment*, *Operation*, and *Monitoring* as shown in Figure 8.1. The *Planning* stage delineates the objectives and needs of software creation, together with an initial plan for upgrades and delivery. The *Coding* stage entails the creation and evaluation of software code by developers, sometimes including regular commits and integration tests. *Building* is the process of compiling and packaging code into usable artifacts, often by integrating code from multiple developers. The *Testing* stage encompasses ongoing automation testing to guarantee the quality of the software artifact, before transmitting it to the *Release* stage of the operations part. The *Deployment* stage is centered around the ongoing process of redeploying the software in the production environment. This includes managing the setup of target platforms and resources. In DevOps cycles, the *Operation* phase involves handling tasks related to the configuration and administration of the software application after it has been deployed. This includes activities like resource provisioning and autoscaling. The *Monitoring* stage oversees the performance of deployed applications through the collection and analysis of usage data, identification of exceptions, and provision of feedback for enhancement, which is reiterated in the planning phase again. The DevOps lifecycle bears resemblance to conventional software engineering approaches, however with a perpetual and extensively automated process.

DevOps combines the processes of development and operations through the utilization of automated development, deployment, and infrastructure monitoring. This method prioritizes the continual delivery of operational features, minimizing issues caused by miscommunication and expediting the resolution of difficulties. DevOps is a paradigm change that promotes collaboration across development, quality assurance, and operations teams. Companies such as Amazon and Google have spearheaded this strategy, successfully achieving cycle times measured in minutes. DevOps can be used across many delivery methods, but it needs to be customized to suit the specific environment and product architecture. Some items, like safety-critical systems, do not support continuous delivery. Nevertheless, enhancements can be strategically scheduled and executed promptly and dependably. DevOps for embedded systems presents greater challenges compared to cloud and IT services due to the amalgamation of outdated code and architec-

ture with the implementation of continuous delivery. To summarize, DevOps is a beneficial methodology for achieving faster and more effective delivery of value.

DevOps automation is essential for ensuring high-quality delivery within tight timeframes. It is crucial to select the appropriate tools for your environment or project. There are two categories of tools: build tools and CI tools. Build tools facilitate rapid iteration and minimize the need for laborious manual operations. CI systems integrate code from several developers and perform checks to identify any faulty code, hence enhancing the overall quality of the software.

Apache Ant[34] is the predominant tool used for constructing applications, particularly in open source settings. Nevertheless, the combination of its uncomplicated syntax and excessive use of words renders it challenging to comprehend. Maven[35] seeks to address the issues of Ant by use XML to specify the build process and the POM. Nevertheless, it can occasionally result in rigidity when specific operations need to be customized.

Rake[36] and Gradle[37] are programming-language-specific tools used for building apps. Rake employs Ruby for writing code, but Gradle utilizes a DSL based on Groovy for configuration.

Devices located at the network edge, particularly in an IoT setting, are commonly distinguished by their heterogeneity, which may encompass the architecture of the systems. To meet this criterion, multi-architecture supported images should be delivered without relying on architecture-specific tags. Docker will automatically provide the appropriate image to the requesting host utilizing a manifest. Before creating various system architecture-specific images, we must first develop a foundational image containing an application.

8.2 Continuous Integration for Containerized Services

In the sense of CI, dependence on a trigger to manually initiate the build process is insufficient, as the majority of necessary procedures are already integrated into a singular Makefile or an equivalent build script. The main CI concept is outlined in Figure 8.2 for Github Actions[38], regardless of the specific execution [cf. 67].

[34]https://ant.apache.org/
[35]https://maven.apache.org/
[36]https://ruby.github.io/rake/
[37]https://gradle.org/
[38]https://github.com/features/actions

For example, our work on a plugin for ONOS [73] has progressed to a point where performing a commit is appropriate, a "commit" is started from the "development machine". The developer uploads the modifications to a central repository which hosts the application's source code, and a CI service, in our example "Github Actions", is monitoring for any updates. Upon detecting the event, the workflow is divided into many jobs and sent to an execution environment, commonly referred to as a runner. Jobs are defined and set up simultaneously, allowing them to be executed on many runner instances simultaneously. We created several Docker images that are dependent on other images. Hence, it is necessary to upload the completed artifacts to a registry such as Dockerhub and retrieve the necessary images as needed. CI may manage the organization, verification, and deployment of current artifacts in the build process. Every CI service provides a logging and monitoring system for managing and evaluating previous integration processes. We analyzed three services, Gitlab-CI[39], Circle-CI[40], and Travis-CI[41], for our ONOS build chain [73].

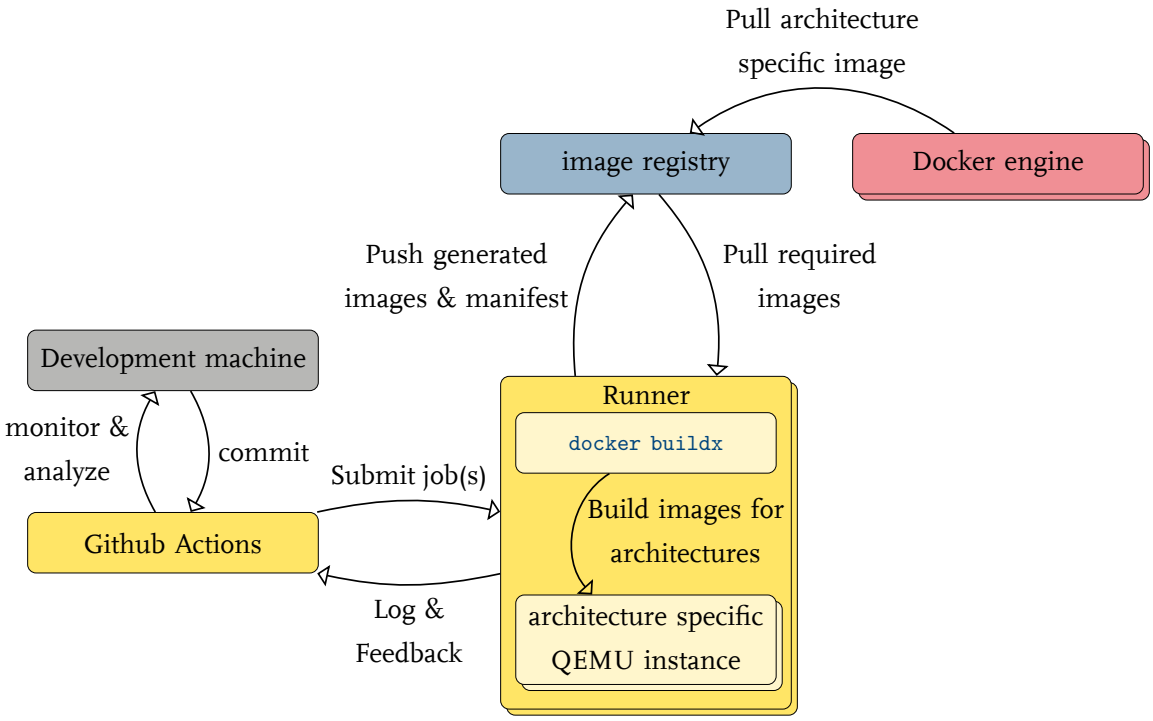
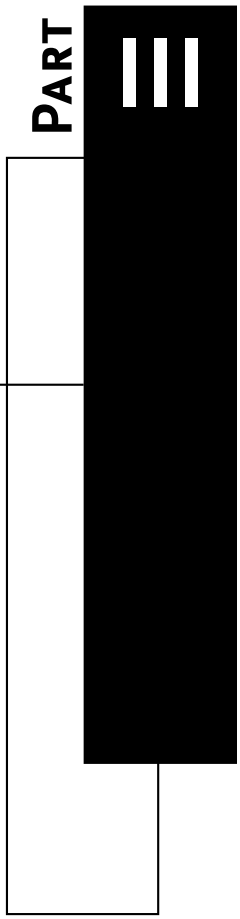


Figure 8.2: CI workflow for Github actions to build multi-architecture Docker images [cf. 67].

[39]https://docs.gitlab.com/ee/ci
[40]https://circleci.com
[41]https://www.travis-ci.com



Middleware and Emulation Concepts

Design is a funny word. Some people think design means how it looks. But of course, if you dig deeper, it's really how it works.

S. Jobs

Parts of this chapter are taken from [17], [43], [46], [70], [76], [137]

An overview of the theoretical concepts of distributed computing models is provided in Chapter 9, without a specific implementation. Commencing with the ODA loop model in Section 9.1, a contemporary methodology, MAPE-K, is presented in Section 9.2. In the area of network virtualization, the concept of Network Function Virtualization Management and Orchestration (NFV MANO), described in Section 9.3, conceptualizes an orchestration middleware for NFV.

Subsequently, for possible realizations in the Cloud-to-Fog-Continuum, we examine contemporary cluster computing implementations in Chapter 10 to illustrate the most pertinent realizations for containerized services within the CC framework. We commence with an elucidation of Docker Swarm in Section 10.1, succeeded by the more commercialized iteration, Kubernetes, in Section 10.2. Frameworks for FaaS can operate atop such clusters, as demonstrated in Section 10.3.

To address the concepts of traffic routing, we examine prominent network emulation tools with capabilities to support NFV placement in Chapter 11. The emulation tool Kathará is subsequently expanded to Kathará as a Service (KaaS) and assessed in comparison to other network emulators in Section 11.2. A practical example demonstrates the capabilities of Kathará by examining multipath transmissions in Section 11.3, facilitated by innovative programmable switches, leading to advanced prototypes discussed in Section 11.5, and concluding with a summary in Section 11.6. Moreover, even attacks, like DDoS, on the network infrastructure are emulated in KaaS as shown in Section 11.7.

Chapter 9
Distributed Computing Models

Autonomic Computing is a characteristic of software systems that satisfy the following four qualities [36].

Self-configuring: The system must possess the capability to autonomously arrange itself in order to adjust to varying conditions.

Self-healing: The system must identify and respond to errors.

Self-optimizing: The system must efficiently allocate resources to enhance overall system performance.

Self-protection: The system must identify and respond to malicious activities.

There are two distinct theoretical designs that realize those autonomous system paradigms. The first design is known as the ODA loop, while the second one is referred to as the MAPE-K architecture. Finally, we investigate the practical concept of NFV MANO, which is used within a NFV enhanced SDN.

9.1 The Observe Decide Act Loop

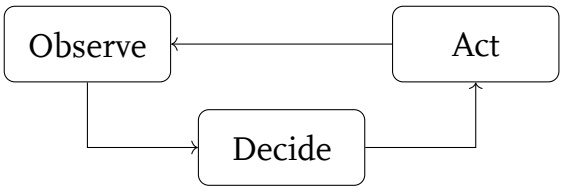


Figure 9.1: The ODA loop [cf. 87].

The ODA architecture is divided into three components [86]:

Observe: The Observe module tracks and analyzes metrics and optimization objectives.

Decide: The Determine feature evaluates the measured metrics and objectives and indicates if the system should initiate measures to enhance the overall performance of the system. The data analysis might encompass many facets of statistics, Machine Learning (ML), or other methodologies.

Act: The Act component's purpose is to carry out the activities determined by the Decide component.

The ODA architecture is a perpetual cycle of observing (Observe), making decisions (Decide), and taking action (Act) as shown in Figure 9.1. It is designed to identify problems and mistakes in a system that is currently operational and develop strategies to minimize the effects of these failures.

9.2 The MAPE-K Architecture

IBM introduced a reference architecture known as MAPE-K for autonomous systems [36]. The design is depicted in Figure 9.2.

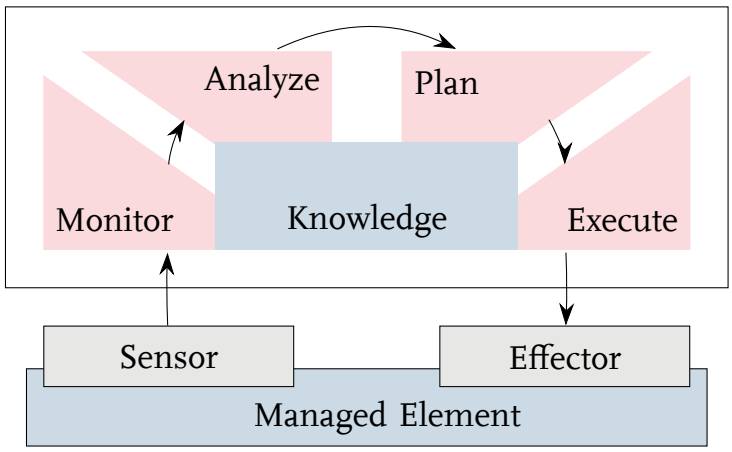


Figure 9.2: The MAPE-K model [cf. 36].

The MAPE-K methodology is separated into five elements [36]:

Monitoring: The monitoring module gathers resources and other metrics that will be evaluated or optimized.

Analyzing: The analysis element evaluates the data of the Monitoring element and decides if the system needs to implement modifications.

Planning: The Planning part is responsible for planning the activities that are conducted in response to change requests if the Analysis element determines that adjustments are necessary.

Execution: In response to these activities, the execution component operates accordingly and adjusts the system's state.

Knowledge: The Knowledge feature is a shared component. It facilitates decision-making by utilizing prior knowledge.

MAPE-K is a continuous cycle that involves monitoring the present state, assessing the desired state, planning the necessary system modifications to evolve from the current state to the desired state, and implementing the optimal plan. Background knowledge underpins all of these actions.

MAPE-K distinguishes itself from ODA by dividing the Decide aspect into two distinct parts: the Analyzing component and the Planning component. This phase simplifies the complexity of the Decide part. In addition, the MAPE-K framework has a Knowledge base that facilitates the sharing of information across all components. This knowledge base enables the customization of the internal configuration of the MAPE-K architecture to facilitate the exchange of information across its components.

9.3 Network Function Virtualization Management and Orchestration

NFV MANO is an essential framework tasked with overseeing the lifetime of Virtual Network Functions (VNFs) and orchestrating NFV Infrastructure (NFVI) as depicted in Figure 9.3. It guarantees the efficient and automatic deployment, management, and scalability of network services in a virtualized environment. The principal objective of NFV MANO is to streamline network operations, augment

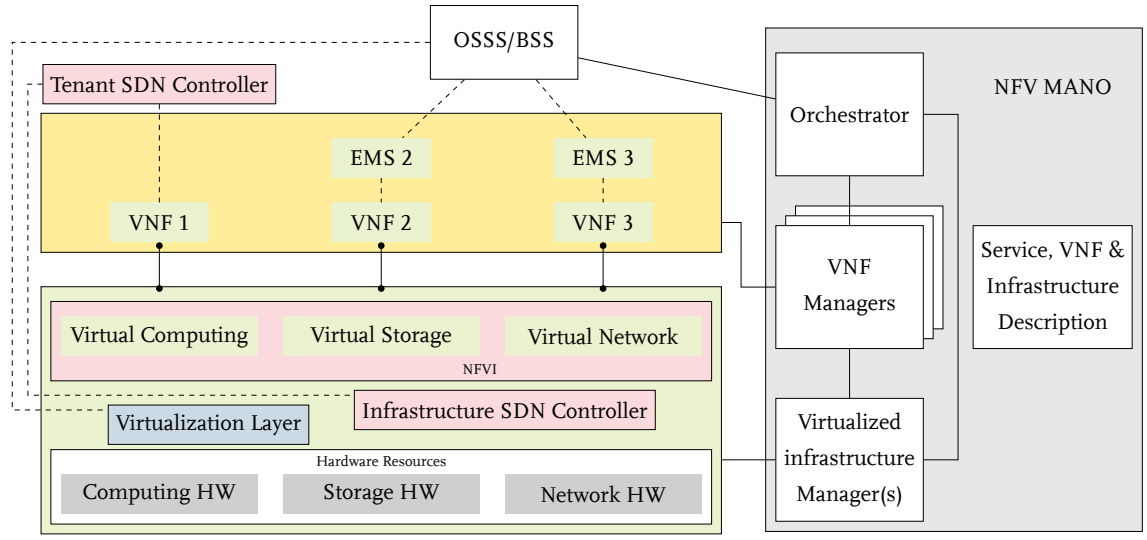


Figure 9.3: Orchestration of virtual network appliances by the concept of NFV MANO [cf. 124]

flexibility, and improve resource allocation within a multi-vendor, heterogeneous framework.

It consists of three core elements, where one, the VNF Manager (VNFM), oversees the whole lifecycle management of VNFs. This encompasses responsibilities including VNF onboarding, instantiation, configuration, monitoring, scaling, and termination. It guarantees that the VNFs are implemented and managed in accordance with the established Service Level Agreements (SLAs). Besides the Virtualized Infrastructure Manager (VIM) watches the fundamental NFVI resources – computation, storage, and networking – necessary for the operation of NFVs. It is accountable for the provisioning, configuration, and monitoring of these resources, ensuring they satisfy the requirements of the VNFs operating atop the NFV infrastructure. The NFV Orchestrator (NFVO) is pivotal in orchestrating and managing the comprehensive NFV ecosystem, encompassing both NFVI and VNF resources. Its duties encompass optimizing resource allocation, ensuring connectivity among components, and sustaining overall network performance. The NFVO collaborates with the VNFM and VIM to guarantee the appropriate instantiation, scaling, and decommissioning of VNFs as required.

The essential functions of NFV MANO are

- on-boarding and lifecycle management of VNFs,
- orchestration and management of the NFVI resources,
- providing a unified view of the NFV infrastructure and services,
- automating the deployment and configuration of VNFs – scaling VNFs up/-down or in/out based on demand,
- upgrading and patching VNFs, and
- providing monitoring and analytics capabilities.

The NFV MANO framework has developed over time to accommodate emerging technologies and architectures. A significant breakthrough is its incorporation with cloud-native architectures and network slicing, allowing operators to optimize resources and provide more precise, tailored network services. Furthermore, NFV MANO is progressing towards enhanced autonomous network administration and greater integration with external systems such as Operations Support Systems (OSSs)/Business Support Systems (BSSs), which facilitate service provisioning and management.

Nonetheless, the implementation of NFV MANO presents certain obstacles. Overseeing the intricacies of distributed, multi-vendor ecosystems presents a significant challenge. Various VNFs, hardware, and software components frequently originate from distinct manufacturers, necessitating NFV MANO to provide seamless compatibility. Ensuring performance and stability within a dynamic, virtualized network presents a challenge, as does attaining end-to-end orchestration across several network domains.

The NFV MANO is crucial for the automated and dynamic administration of virtualized network operations and infrastructure. It facilitates telecom operators and service providers in managing complex and scalable networks with flexible resource management, service orchestration, and lifecycle automation. Notwithstanding obstacles, NFV MANO serves as the foundation of contemporary, virtualized, and agile network management, promising enhanced efficiency and cost-effectiveness in network operations.

Distributed Computing Implementations

CHAPTER 10

In CC different provisioning models for services are available, as depicted in Figure 6.1. Since our goal is service provisioning in the Cloud-to-Fog-Continuum, we focus on the middleware needed to provision a service infrastructure to deploy containerized services and evaluate Docker Swarm in Section 10.1 and Kubernetes in Section 10.2 in this context. Additionally, we have a look at the FaaS model in Figure 10.4.

10.1 Functionality of Docker Swarm

A "swarm" is a functionality in the Docker Engine that handles the administration and orchestration of clusters. It is developed using swarmkit, which is an independent project responsible for implementing Docker's orchestration layer. A swarm is a collection of Docker servers that operate in swarm mode and serve as both managers and workers as shown in Figure 10.1. Each host has the capability to function as a manager, a worker, or both. When developing a service, its optimal state is determined, which includes specifying the number of replicas, the available network and storage resources, and the ports via which the service is accessible to external entities.

Swarm services have benefits compared to individual containers since they enable the adjustment of a service's configuration without requiring a manual service restart. Docker modifies the configuration, terminates service jobs with obsolete configuration, and generates new ones that align with the desired configuration.

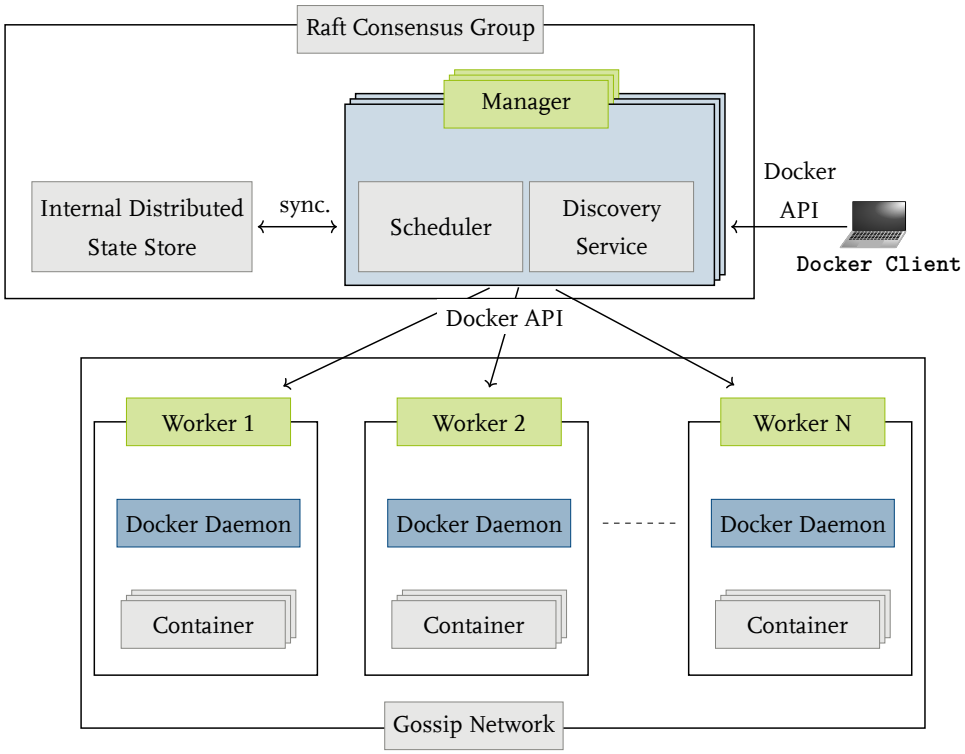


Figure 10.1: Architecture of Docker Swarm [cf. 75]

Nodes are Docker engine instances that actively participate in the swarm, capable of being executed on either a single physical machine or a cloud server. In order to distribute an application to a swarm, a service definition is given to a manager node, which then assigns tasks to worker nodes to carry out the necessary work. Manager nodes are responsible for performing orchestration and cluster management tasks necessary to maintain the intended state of the swarm.

Services and tasks form the core framework of the swarm system and serve as the foundation for user engagement with the swarm. When developing a service, you define the specific container image to utilize and the precise instructions to execute within the active containers. In the replicated services paradigm, the swarm manager allocates a predetermined number of replica jobs to the nodes, according to the desired state's scale set.

The swarm manager utilizes load balancing to provide services externally. The internal Domain Name System (DNS) component automatically allocates a DNS entry to each service inside the swarm. The swarm manager utilizes internal load balancing to evenly distribute requests among services in the cluster, taking into account the DNS name of each service.

10.2 Orchestration by Kubernetes

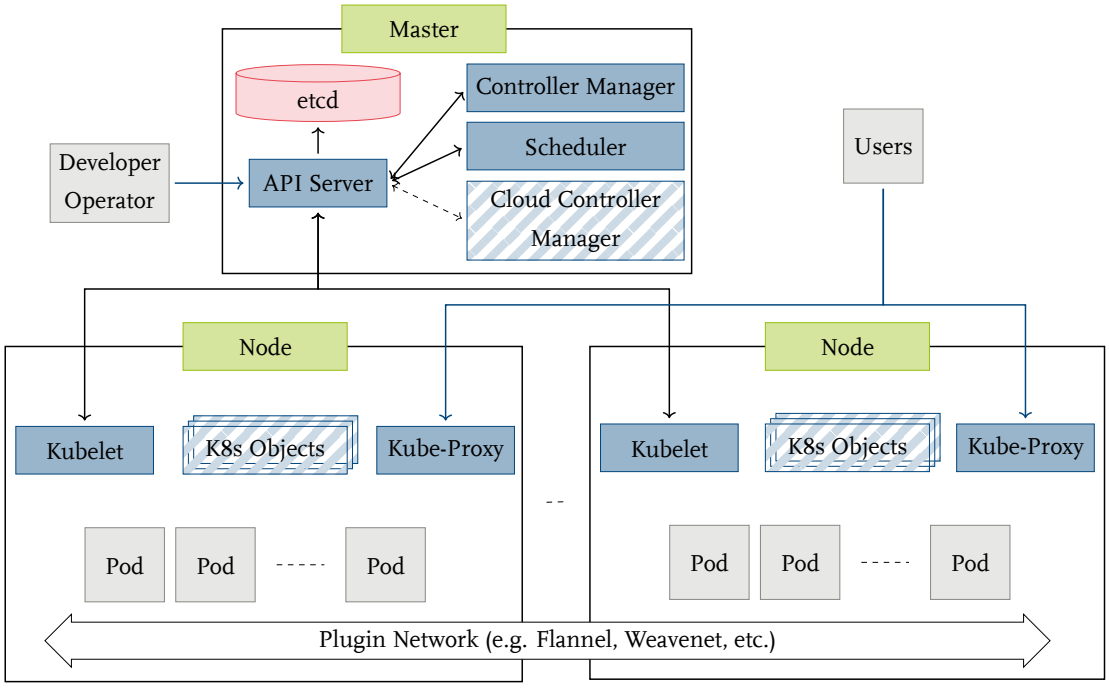


Figure 10.2: Architecture of Kubernetes [cf. 75]

When it comes to Docker container orchestration, Kubernetes is widely recognized as the most prominent solution. The architecture of Kubernetes is demonstrated in Figure 10.2. It is organized into two elements: the *Control Plane* that runs on the *Master* and the workhorses called *Nodes*.

The *Control Plane* provides the central intelligence for Kubernetes' orchestration. It features the *Controller Manager*, *Cloud Controller Manager*, *API Server*, *Scheduler*, and a distributed state store *etcd*. The *Controller Manager* is tasked with detecting and adjusting the infrastructure in the event of node outages. The *Cloud Controller Manager* incorporates cloud-specific control logic and establishes connections using the cloud provider's APIs. The *API Server* offers an API, which all nodes interact with. Additionally, it is accountable for performing the operations specified by the scheduler. The scheduler determines the allocation of services (called pods in Kubernetes) to nodes based on factors such as individual and collective resource demands, hardware and software limitations, policy restrictions, affinity and anti-affinity specifications, data locality, interference between workloads, and deadlines[42]. All of these components integrate the *etcd*[43] component as a common key-value store.

[42]https://kubernetes.io/docs/concepts/overview/components
[43]https://etcd.io

The nodes supply the necessary resources for the Docker container to execute it. There are two services operating on the nodes: the *kubelet* and the *kube-proxy*. The *kubelet* agent operates on every node and guarantees the execution of all requested containers on the respective node. The *kube-proxy* facilitates network connectivity within and outside the cluster by implementing policies at the network layer.

The *Cloud Provider API* provides interaction with cloud providers to validate the deletion of nodes or to establish network paths and set up load balancers. Should infrastructure providers want to implement Kubernetes on their own hardware, the *Cloud Provider API* may be absent.

Recognizing the importance of scheduling in Kubernetes, let us examine how Kubernetes addresses the difficulties associated with container orchestration. Figure 10.3 illustrates the process of determining the node to which Kubernetes will deploy a pod.

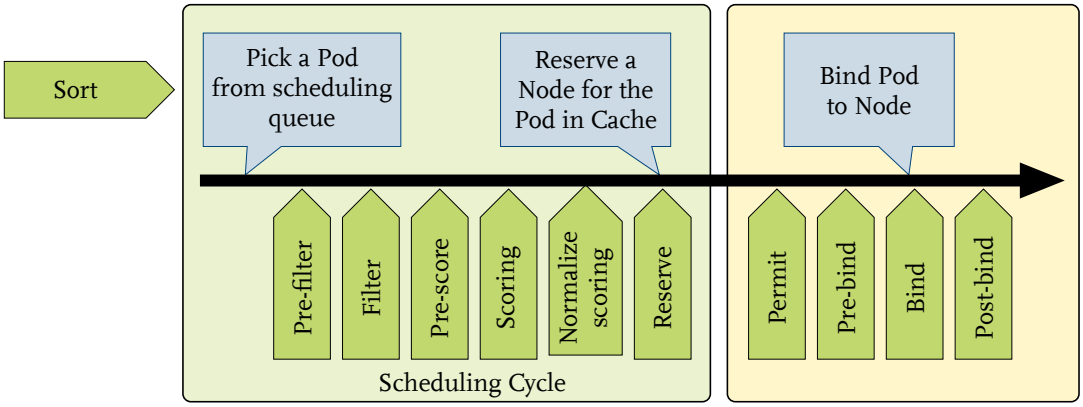


Figure 10.3: Scheduling principles for pod deployment in Kubernetes[44]

The official Kubernetes Documentation[44] provides comprehensive explanations for each step with meticulousness. Firstly, let us provide a concise summary of the initial phase of the pod scheduling pipeline, followed by an overview of the subsequent phase. This will allow us to gain a comprehensive understanding of the techniques employed in Kubernetes' node scoring.

First and foremost, it is necessary to enable only one `queueSort` plugin. It offers an ordering method specifically designed for sorting pending pods[45]. Additionally, the pre-filter plugins execute preparatory procedures prior to the filter step, enabling the selective removal of specific pods from the scheduling pipeline. The

[44]https://kubernetes.io/docs/reference/scheduling/config/#extension-points
[45]https://kubernetes.io/docs/reference/scheduling/config/#scheduling-plugins

filter plugins comprise a collection of plugins utilized to selectively remove pods from the scheduling pipeline through the application of filters. The Kubernetes' network administrator has the ability to modify the sequence of the established filters in order to enhance performance. If a pod fails to pass even one criterion, it will be excluded from the node selection process. After passing through the filtering phase, every pod enters the pre-scoring stage, when a preparation sequence is initiated once again before the scoring stage. The election procedure is determined by a set of plugins that calculate the real pod scores and then normalize them. The pod with the largest total of weighted points is declared the winner. According to the Kubernetes' documentation, the winner node informs plugins when resources have been allocated for a certain pod. Additionally, it deploys an "unserve" operation that Kubernetes uses to invoke routines in the event of a failure at any point, either during or after the reserve stage.

In the second phase, the winning node is conducting four successive phases. During the permit stage, some plugins are activated that have the possibility to impede or postpone the deployment of a pod. Following the completion of the permission step, the deployment of the pod continues with the pre-bind stage. During the pre-bind step, plugins conduct any necessary procedures prior to binding resources on the node. Ultimately, the resources of the node will be allocated to the pod during the binding phase. The post-bind stage occurs following the binding of resources in order to inform extension points.

In the *queueSort* phase the default plugin *PrioritySort* provides the default priority sorting algorithm[45].

The *preFilter* phase features five plugins: *NodePorts*, *podTopologySpread*, *NodeResourcesFit*, *VolumeBinding*, and *InterpodAffinity*. The *NodePort* plugin filter verifies if the ports that have previously been assigned are included in the required ports, rendering them unsuitable for the present deployment request. The *podTopologySpread* is a metric designed to guarantee the dispersion of resources across different topologies. The concept of topology spread enhances redundancy in pods by distributing them over many nodes, hence preventing node failures from affecting whole services in Kubernetes. The *NodeResourcesFit* plugin selectively excludes nodes that fail to satisfy a pod's resource criteria. The system provides three strategies: use the *LeastAllocated* (default) node, the *MostAllocated* node, or the *RequestedToCapacityRatio* node. The *VolumeBinding* plugin verifies if a node

has the capability to associate the volumes of the desired pod. The *InterpodAffinity* plugin effectively enforces inter-pod affinity and anti-affinity. The principle of affinity compels pods to deploy in close proximity (e.g., to minimize latency) or at maximum distance from one other (e.g., to provide geo-redundancy).

Upon examining Kubernetes' scheduling plugins, we can conclude that the orchestration tactics are intricate and sophisticated. In the setting of FC, characterized by restricted node resources and a high number of nodes, the scoring and filtering procedures may be overly complex, necessitating a balance between performance and assessment speed.

10.3 Concept of Function as a Service

With the help Docker Swarm, Kubernetes or similar platforms, FaaS can be adopted on top, offering the option of external management or self-deployment. Serverless computing provides freedom but may result in unpredictable latency. One option is to relocate latency-sensitive applications nearer to the network edge, particularly for IoT applications. However, this may not be practical for projects with limited or no Internet connectivity.

OpenFaaS[46] and similar platforms provide their own adaptations of the FaaS model along with related services. OpenFaaS needs to be self-deployed and therefore offers flexibility and scalability. Docker serves as the essential framework of this system, offering a versatile and scalable technological backbone. The main program is the serve Watchdog, which facilitates contact with a specific process through HTTP, improving Docker images to serve as a FaaS function.

The API gateway is essential for directing external requests to the appropriate function, offering a user-friendly web interface for interaction, and scaling registered functions as required. Functions are implemented as services and can be managed by communicating with either the Swarm or Kubernetes API. Scaling is accomplished by adjusting the replica count of Swarm.

[46]https://www.openfaas.com/

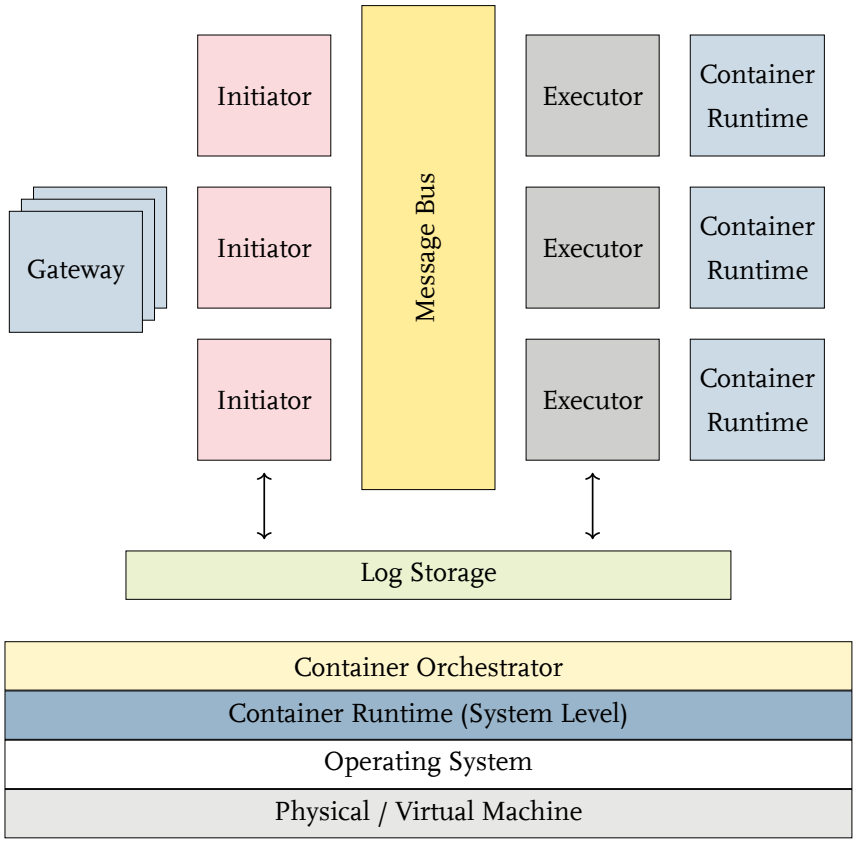


Figure 10.4: The FaaS model [cf. 99]

System Layers:

- OS with modern Kernel that supports container isolation.
- Container runtime (system level) is responsible for provisioning a cluster
- Optionally, a container orchestrator

FaaS Layers:

- Gateway as a frontier component
- Initiator is responsible for the invocation requests
- Message bus is the backbone of FaaS
- Container runtime (application level) is responsible for starting the function container

FaaS is a cloud service paradigm in which a provider offers a managed execution environment and handles almost all operational responsibilities, resulting in a gain of control for the user. It enables the automatic adjustment of cloud functions that have a limited lifespan and lack awareness of their context. Additionally, it allows the scaling down of functions to zero, meaning that no functions are operating when they are not needed, which is a unique feature for FaaS. FaaS facilitates

a payment structure based on usage, allowing for the most detailed billing option available in current cloud technologies. FaaS is a cloud service model in which users create efficient, stateless cloud functions that are capable of running on a single thread. Users simply need to provide a few setup parameters, such as the amount of Random Access Memory (RAM) and the timeout duration [51].

When it comes to interoperability, every cloud vendor has its specific interpretation of cloud functions, highlighting the necessity for vendor-agnostic definitions. The evaluation of function performance should be conducted by comparing it to traditional computing models. This will help determine if a FaaS method is more suitable, especially for highly distributed contexts such as IoT systems. The engineering is designed to maximize cost performance by taking into account various setups and workload patterns. Furthermore, it is crucial to assess integration testing for FaaS systems, particularly when a significant portion of the operational logic is outsourced to the provider. Expressing the function composition is necessary to prevent duplicate charges. Ultimately, it is necessary to discover methods for transferring outdated systems and dividing them into separate functions.

Chapter 11
Network Emulation

In this section we will take a glance at the emulation of networks with an emulation tool called Kathará developed by Bonofiglio *et al.* [17]. With an extension in Section 11.1, we demonstrate an easily usable GUI that supports network emulation in the browser [76]. With this SaaS approach, we want to deliver an easily scalable system, which every newbie can use. With the possibility to use own containerized services, network influences can easily be evaluated. Therefore, we give an example on emulation of DDoS attacks in Section 11.7 and it's recognition methods in a SDN [68]. Besides, we demonstrate multipath transmission with P4, which are easily emulatable in KaaS [71].

11.1 Kathará as a Service

Network research necessitates access to software, scripts, input data, and testbeds, which can be shared via public platforms for collaboration and version control (e.g. *git*). Obtaining a flexible, deployable, and cost-efficient test bed is arduous due to its high expense and scalability difficulties. Network emulators offer economical and effective solutions; yet, implementation faults may result in a deficiency of reproducibility.

There are numerous programmable emulators for prototyping, such as GNS3[47], Mininet[6], Containernet[48], and Kathará[49] and each possess distinct advantages and disadvantages. For instance, GNS3 provides network researchers with diverse net-

[47] https://docs.gns3.com/docs/
[48] https://github.com/containernet/containernet/
[49] https://github.com/KatharaFramework/Kathara

work experiments by employing different VMs to run switches, routers, and hosts, which are interconnected by virtual interfaces. The emulator appears attractive initially, since it allows for the straightforward replication and assessment of various practical topologies; nevertheless, VMs are excessively resource-intensive due to local resource constraints [83]. Mininet and Containernet, conversely, employ lightweight, OS-level virtualization methods to replace VMs with "containerized" network components, thereby offering economical experimenters flexibility and reproducibility. Nonetheless, each requires particular hardware, software, and OS configurations for effective installation and functionality, in addition to a significant effort to learn about those tools. Finally, all the aforementioned emulators are tools that run on a single machine, hence limiting the size for testing topologies due to local resource constraints.

We offer a vision for a multi-tenancy, cloud-based Network Emulator as a Service (NEaaS) platform by incorporating modern technologies into an open-source network emulator, enabling users to quickly create and test a diverse array of network architectures. We provide a first-generation prototype of the platform to showcase its capabilities in comparison to existing network emulators. Our prototype is publicly accessible on GitHub at *uniba-ktr/KaaS*[50].

11.1.1 Implementation by a Sandboxed Realization

The prototype, illustrated in Figure 11.1, has four services, each represented as a tailored Docker container, with the links indicating the inter-dependencies among these services. The GUI service is a containerized web application that enables users to interact with the API endpoints offered by the Kathará REST API service, such as establishing devices and attaching them to collision domains. Additionally, users can access any simulated host remotely to execute supplementary commands via the web sockets provided by the Web-TTY container. Both Kathará REST and Web-TTY services rely on a Docker-in-Docker (DinD)[51] container functioning as the Docker engine. Ultimately, the Kathará REST, Web-TTY, and DinD containers constitute a unified, fully operational network emulator. The prototype prioritizes microservices over monolithic architecture for several reasons: it enables developers to implement and deploy new core functionalities indepen-

[50]https://github.com/uniba-ktr/KaaS
[51]https://hub.docker.com/_/docker

dently; it aligns effectively with the cloud-native paradigm, as a single microservices emulator is allocated on-demand to each user; and it enhances security by providing each user with a dedicated DinD, ensuring complete isolation of their emulated hosts from others.

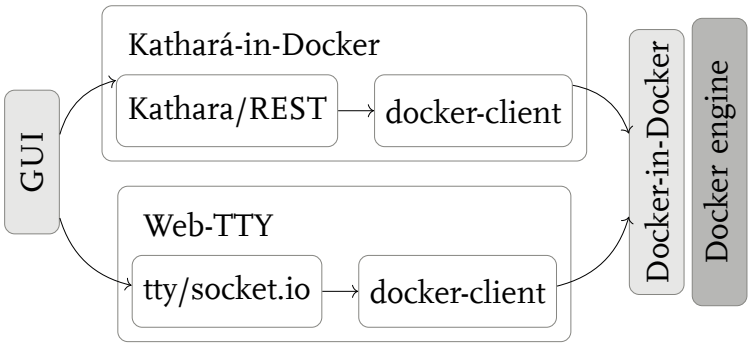


Figure 11.1: Single user prototype of Kathará as a Service [76].

11.1.2 The Workflow of the Kathará Service

Figure 11.2 illustrates the sequence diagram for the establishment and execution of a Kathará lab within the prototype. Initially, users include diverse networking devices into the GUI, subsequently executing an HTTP POST request including the details of these devices to the *create_lab* API endpoint. The emulator core verifies the correctness of the submitted JavaScript Object Notation (JSON), subsequently generating a Kathará lab and its configuration files within the *KinD* component itself. Afterwards, the emulator core issues a *state_msg* that conveys the present status of the supplied laboratory. Following that, upon successful creation of the lab, users may initiate it by using the *start_lab* endpoint. As a result, the emulator core attempts to initiate the required Kathará lab asynchronously. At that point, when users inquire about the current status of the lab via the *lab_info* endpoint, and the lab has been successfully initiated, they will obtain a list of container IDs corresponding to the devices within that lab. Users may occasionally execute commands on a device by clicking its icon in the GUI. The device's container ID is transmitted to the Web-TTY service, initiating a terminal session via a websocket, which is then presented on the Kathará GUI as an Hyper Text Markup Language (HTML) "iframe".

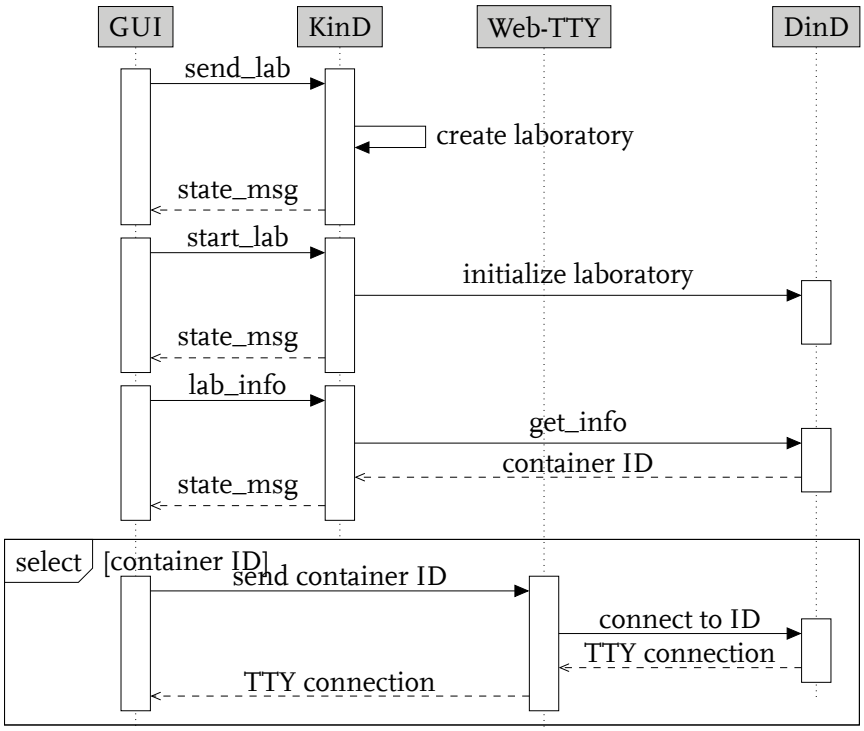


Figure 11.2: Micro-service interactions in the Kathará as a Service framework [76]

11.1.3 Access Interface for Kathará

According to the documentation, Kathará does not support any REST API services; it only provides a Python package containing various APIs for developers to utilize within Python applications[52]. Consequently, this significantly limits the functionalities of the network emulator. A new RESTful API interface is introduced and implemented to facilitate remote control of the Kathará emulator from another service, specifically the Kathará User Interface (UI) in this design. The FastAPI framework[53] is selected for its robustness, quickness, and compatibility with superior libraries such as Pydantic[54] and type hints. Furthermore, supplementary Python wrapper classes (e.g., `LabWrapper`, `LabController`, `KatharaWrapper`) are implemented to obscure the intricacies of interfacing with the core libraries. As a result, the new Kathará REST API offers a API endpoint for every command of Kathará. For instance, Listing 11.1 illustrates the development of the API endpoint for creating a lab:

[52]https://github.com/KatharaFramework/Kathara/wiki/Kathara-Python-API
[53]https://fastapi.tiangolo.com/lo/
[54]https://docs.pydantic.dev/latest/

```python
@app.post("/lcreate", response_model=Message)
async def create_lab(info: Laboratory):
    log.info("Create Lab {}".format(info.json()))
    result = LabController().gen_lab(info)
    return result
```

Listing 11.1: Lab creation API endpoint [76]

Additionally, both the Kathará emulator and the new REST API framework are integrated into a single Docker image, therefore establishing the idea of *Kathará-in-Docker* (i.e., *KinD*). This innovative functionality significantly enhances the NEaaS concept, as an emulator within a sandboxed environment may be readily launched and expanded on-demand utilizing existing container orchestrators such as Docker Swarm or Kubernetes.

11.2 Kathará as a Service in Comparison to Alternative Competitors

The necessity for reproducibility of works within the research community, especially within networking research groups, has been a longstanding concern. Others have delineated the requisite properties that a network emulator must include to meet this demand. The insights acquired from utilizing those emulators are encapsulated in the subsequent list:

Scalability: The platform must enable extensive simulation of network topologies, accommodating from several to thousands of network devices without modifying the system architecture.

Flexibility: Creating network experiments should be seamless and convenient.

Extensibility: New functionalities or categories of network devices must be seamlessly integrated into the platform without necessitating alterations to its design.

Quality of Experience (QoE): The time and effort expended by a user, irrespective of their proficiency, should be low when utilizing the instrument. A robust and efficient GUI is advised.

Realism: The system and its components must exhibit the identical functionalities present in the actual hardware. Likewise, the traffic produced by the platform must closely resemble authentic traffic.

Cost: The total expenses (e.g., capital expenditure, operational expenditures) should be minimized.

Finally, Table 11.1 illustrates the accomplishments of the aforementioned network emulators, along with our prototype about the specified features.

	Mininet	Containernet	GNS3	Kathará	KaaS
Scalability	+	+	-	+	+
Flexibility	-	-	+	+	+
Extensibility	-	-	(limited)	(limited)	+
QoE	-	-	+	(limited)	+
Realism	+	+	+	+	+
Cost	+	+	+	+	+

Table 11.1: Feature comparison of network emulators [76]

11.3 Multipath Transmissions emulated in Kathará

This work presents a methodology for conveying critical data via packet duplication, guaranteeing swift and dependable transmission over many channels. The receiving switch computes a hash of a packet and retains its value in the switching register, much to Bloom filters [149]. This approach guarantees that a single copy of the traffic is transmitted to its destination, ensuring that even if one copy is lost the other one successfully arrives at its destination.

Moreover, multipath transmissions can enhance data delivery speed by partitioning the route. Data is transported across a network by randomly dispersing it through several routes to reach its destination, hence accelerating transmission time.

Motohashi *et al.* [119] implemented a multipath transmission paradigm in P4 for wireless access points that randomly allocates traffic. However, their prototype only transmits UDP packets. In contrast, we developed two distinct multipath realizations for P4 routers by either partitioning or replicating transmitted packets for both transport layer protocols.

Lindner *et al.* [111] attempt to address this issue using an alternative method, wherein they develop a specialized header. Our methodology provides enhanced versatility, as it is applicable to all protocols and is not contingent upon specific protocol parameters (only if the hash is generated over such parameters), eliminating the necessity to build a new header for identifying duplicate packets.

Our objective is to develop a replicable emulation of multipath transmissions in P4 capable networks. Consequently, we developed various prototypes using a P4 network emulated in Kathará capable of transmitting packets via two distinct pathways. A `random_split` configuration is established, compelling packets to transit those paths randomly. Secondly, a `duplicate` prototype is executed, which replicates packets for transmission over both routes. It guarantees the elimination of duplicates on the receiving switch. Our implementation and trial configurations are publicly accessible on GitHub under *uniba-ktr/p4_multipath*[55] to guarantee the repeatability of the experiments.

11.4 The Programming Protocol-Independent Packet Processor Language

P4 [19] is an advanced programming language. P4 is proficient in operating with a SDN control protocol like OpenFlow. The P4 programming language enhances flexibility and enables programmers to modify packet processing at deployed P4 switches. Moreover, P4 is protocol-agnostic, allowing switches to operate independently of any particular network protocol. The programmer can articulate the packet processing capability independently of the hardware.

Rather than augmenting the OpenFlow definition, *"future switches should support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to leverage these capabilities through a common, open interface (i.e., a new "OpenFlow 2.0" API)"* [19]

Upon the arrival of a packet to the P4 parser, as illustrated in Figure 11.3, it extracts the header fields from the packets. The header fields delineate the protocols endorsed by the switch. Subsequently, the header information are utilized in the input and output port "match+action" tables, which ascertain the designated output port and queue for the packet. At the input port, a determination is made regarding whether a packet should be deleted, forwarded, flow control initiated, or

[55] https://github.com/uniba-ktr/p4_multipath

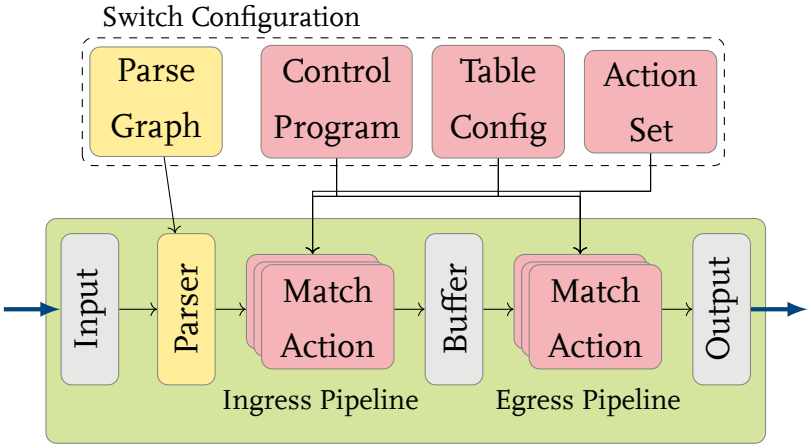


Figure 11.3: P4's abstract forwarding model [cf. 19]

replaced. The output port "match+action" is utilized for pre-instance alterations to the packet header, such as multicast copies [19].

The P4 language articulates the processing of packets by the data plane of a programmable target. It is just intended to delineate data plane operations and does not address the control plane. In traditional switches, the manufacturer delineates the data plane capabilities. The control plane manages the data plane by overseeing routing table entries, configuring specialized objects, and processing control packets.

The most recent iteration of P4 is $P4_{16}$ (version 1.2.2)[56] and has implemented numerous non-backward compatible modifications relative to $P4_{14}$ (version 1.0) to attain a more stable language definition. Numerous functions were delegated to libraries, such as counters and checksum units, to convert P4 from a complex language with over 70 keywords into a streamlined core language including approximately 40 keywords. As a result of these modifications, the new keyword "external" was implemented to characterize library components.

Every manufacturer is required to supply a P4 compiler along with a corresponding architecture definition for their target. The input controls supply data to the P4 program, while the output control can be modified by the P4 program.

The Very Simple Switch (VSS) is an exemplary architecture delineated by the P4 specification[56]. It can receive packets via input ports, where the arbiter conducts a checksum verification and discards unsuccessful packets. The VSS comprises a single parser, a match-action pipeline, and a deparser. Upon deparsing a packet,

[56]https://p4.org/p4-spec/docs/P4-16-v1.2.2.html

it is transmitted over an Ethernet output port. There are three distinct Ethernet output ports:

- drop port, which eliminates a packet,
- recirculate port, which redirects a packet to an Ethernet input port,
- and CPU port, which transmits a packet to the control plane.

11.5 P4 Enabled Multipath Transmission Prototypes

Our prototypes employ the topology illustrated in Figure 11.4, consisting of four switches, designated as *s1* through *s4*, operating on our *unibaktr/ubuntu:p4*[57] image, alongside two computers, *h1* and *h2*, which utilize our *unibaktr/ubuntu* base image. The switches are presently configured manually and solely display the capabilities of a P4 program, with the primary functionality for determining multipath behavior located in *s1* and *s4*, which connect the network to the end users, respectively. The hierarchical execution of the distribution algorithms is not currently addressed.

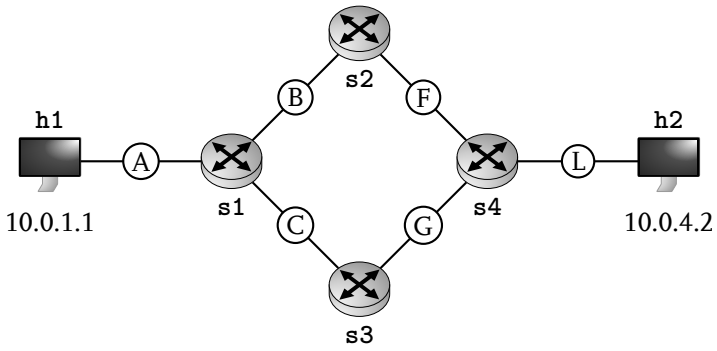


Figure 11.4: Emulated Network topology used for P4 prototypes. The switches *s1* and *s4* are responsible for the path selection [71].

11.5.1 A Random Split Prototype for Faster Delivery

This prototype employs two tables for IPv4 addresses; the first is instantiated on *s2* and *s3*, as only standard forwarding is required in this context. The second table is incorporated on *s1* and *s4* to direct the packet to either *s2* or *s3* using the function `random_split_group_to_nhop` from Listing 11.2.

[57] https://hub.docker.com/r/unibaktr/ubuntu

When a packet is directed to the IP address 10.0.4.2, the method `random_split_group` is invoked, requiring three parameters: the `random_split_group_id`, the `threshold`, and the `maxNum` parameter; `maxNum` is utilized to generate a random number.

If it exceeds the `threshold`, *egress* port 0 is utilized, thereby forwarding the packet to *s2*. If this is not the case, the packet is transmitted to *s3*. This is similarly applicable in the reverse direction, when a packet is transmitted from *h2* to *h1*.

```
table random_split_group_to_nhop {
    key = {
        meta.random_split_group_id: exact;
        meta.random_split_port: exact; }
    actions = {
        drop;
        set_nhop; }
    size = 1024; }
```

Listing 11.2: Ingress of the P4 implementation for the random_split prototoype [71]

11.5.2 A Duplication Prototype for Reliable Transmissions

The duplication prototype is designed to provide a secure, rapid, and dependable pathway for all packets from *h1* to *h2*. To accomplish this, all packets are replicated and transmitted over layer 3. All packets from *h1* traverse the normal path through *s3* to arrive at *h2*. To guarantee a swift and, most importantly, dependable path, the packets are duplicated at *s1*, and the replica is transmitted through *s2* to *h2*. The first packet to arrive at switch *s4* is transmitted to *h2*, whereas the second packet is discarded.

For instance, when *h1* transmits a packet p to *h2*, the packet p is sent from *h1* to *h2* through *s3* and *s4*. At *s1*, p is replicated and $\tilde{p}$ is transmitted to *h2* through *s2* and *s4*. The packet that arrives at *s4* last will be discarded, while the other will be transmitted to its destination.

At *s1*, a `mirroring_add` function is executed, enabling the cloning of each packet at the ingress port and routing it through the secondary egress port of *s1*. Listing 11.3 presents the `action clone_packet`, indicating that all packets are cloned from ingress to egress (I2E) when the `REPORT_MIRROR_SESSION_ID` equals 500.

```
action clone_packet() {
    const bit<32> REPORT_MIRROR_ID = 500;
    clone(CloneType.I2E, REPORT_MIRROR_ID); }
```

Listing 11.3: Ingress of the P4 implementation for the duplicate prototype [71]

Switch *s4* operates with its own P4 file and employs a mechanism akin to Bloom filters [149] for packet deduplication. A hash value is generated for each packet received at *s4*. The hash value is retained in a register and is compared against pre-existing hash values to identify duplicate packets. The hash value is generated using the source IP address, destination IP address, source and destination ports of the transport layer protocol, employing the Cyclic Redundancy Check (CRC) hash method from P4[56]. The generated hash value of a packet is compared to that of preceding packets; if a match is identified, the packet is discarded and the stored hash value is eliminated. Our prototype can accommodate up to five hash values and store them in the register. Registers function analogously to an array in relation to other programming languages. The hash values must be retained in a register rather than in a table, as regular updates to the value are required, which is unfeasible with a P4 table. To obtain several packets amidst duplicates, it is essential to retain multiple values, which can be augmented if required. Upon the arrival of a new packet, the hash value is recorded in register 0, while the existing values are shifted to the subsequent registers (for instance, the value in register 0 is transferred to register 1, the value in register 1 to register 2, and so forth, with only the value in register 4 being discarded).

11.6 Feasibility of P4 Prototypes

The prototypes were successfully developed and are compatible with UDP and TCP. The P4 utilizing the Kathará emulator can replicate and eliminate duplicate incoming packets. Packet deduplication utilizing hash generation over a packet is effective and readily adaptable for various protocols. The drawback, however, is that the switch requires a register, and depending on the hash method employed, there exists a danger of hash collisions.

The `duplicate` prototype represents a minor advancement in guaranteeing efficient and rapid packet transport across a network. It is essential to incorporate many functionalities, including load balancing, congestion control, and au-

tomatic packet re-routing in the event of a switch failure. The dependability of the `random_split` prototype can be improved by incorporating Raptor Codes [47] into the relevant P4 implementations. At now, all configurations are executed manually, and if a switch fails, it will continue to attempt to transmit the packet through the failed route.

The subsequent phase of our resilience concept involves the incorporation of network control through SDN capabilities by incorporating a controller such as ONOS into our prototypes. It would facilitate enhanced regulation of the control layer. P4 is intended to delineate the interface for communication between the control and data planes; however, P4 is not applicable for articulating the control plane functionality of the target [19]. Utilizing SDN, we can devise methods to address these shortcomings.

Beside implementing new transmission paradigms, network resilience is of utmost importance. Therefore, the evaluation of potential network assaults is essential to increase robustness and security of network transmissions. Through proactive threat assessment, effective, resilient systems capable of enduring and responding to cyber assaults can be developed.

11.7 Distributed DoS Attacks in SDN

DDoS attacks pose considerable threats to both conventional and novel networks, encompassing IoT, CC, 5G communication networks, and SDN. There is a significant lack of study about the mitigation of DDoS attacks, with no focus on their prevention. Many research investigate the mitigation of DDoS attacks; however, the prevention of DDoS attacks is hardly addressed [48]. Our aim is to offer a multifunctional playground that enables students to address the specified knowledge deficiencies.

A real-time monitoring system is essential for effective DDoS attack security solutions [9], [12]. Consequently, our methodology incorporates the real-time monitoring tool sFlow-RT[58]. This will build the foundation for anomaly detection and facilitate automatically triggered notifications in future advancements.

Research on SDN generally emphasizes network performance above the security challenges associated with the deployment of programmable networks [48]. Numerous research examine how the use of SDN might enhance security issues

[58]https://sflow-rt.com/

in networking [39], [132]. Ahmad and Mir [1] present a comparative analysis of several SDN controllers, evaluating their scalability, consistency, reliability, and security. The research on SDN security and DDoS assaults mostly focuses on certain forms of DDoS attacks, predominantly volumetric attacks. In accordance with Dayal and Srivastava [41], we intend to compare various forms of DDoS attacks, illustrating the effects of each attack on the SDN infrastructure.

Numerous research have concentrated on utilizing ML methodologies to identify DDoS attacks, e.g., [2], [80], [135]. Various DDoS attack datasets for ML detection methodologies are analyzed in [9], while diverse mitigation strategies are encapsulated in surveys such as [12], [145].

We presently lack readily deployable, highly flexible, lightweight frameworks for assessing and understanding DDoS attacks on SDN networks utilizing commodity hardware. Investigations of DDoS attacks and practical exercises typically overlook SDN networks. A software platform developed by Fuertes *et al.* [57] instructs and evaluates security against DDoS attacks and their life-cycle, utilizing virtual networks. Among others, Das and Sarac [40] establish a laboratory within a VM utilizing OvS[59], ONOS[30], Mininet[6], and Docker[11]. Students can get knowledge about SDN security and configure networks safely. A separate report [20] underscores the security issues of SDN in 5G networks and stresses the necessity of thorough security training for students.

The Mininet emulator is primarily utilized to simulate testbeds for SDN networks; however, it has fewer configurable tools than Kathará. The user-friendly characteristics and extensive use of Mininet facilitate students' initial education on SDN. We propose KaaS [76], an innovative technique, to enhance the utilization of Kathará. We assert that detailed configurations improve students' understanding of the SDN network, node interactions, and attack scenarios.

11.7.1 Attack Categorization for SDN

Attacks within the SDN paradigm can be categorized according to the typical architectural tiers or planes pertinent to SDN: the application plane, the control plane, and the data plane.

[59]https://www.openvswitch.org/

Assaults in the application layer may focus on one or many SDN applications and the north-bound API [48]. A focused assault on a particular SDN application may impact or jeopardize other applications, amplifying the attack's consequences and potentially resulting in authentic security violations within SDN.

Although centralized control in SDN provides comprehensive benefits, it simultaneously poses a considerable risk, especially when the control plane is subjected to targeted attacks [1], [138]. If an attacker successfully inundates the network to the extent that the controller cannot handle the load, it may result in a total disruption of the network for legitimate users [1]. In this context, we would reference a SPoF.

Potential attack strategies to induce network failure encompass inundating the north-bound APIs, the south-bound APIs, or the controller itself [48]. In distributed control plane implementations, supplementary security concerns emerge, necessitating the design of effective authentication procedures for the validation and verification of controller instances [1]. Consequently, the control plane is regarded as the most susceptible component of the SDN architecture concerning DDoS attacks. An instance of this attack entails sending a substantial volume of newly generated data streams from devices with counterfeit IP addresses. Switches generally transmit a header-only packet to the controller upon encountering flows that lack an entry in their local flow table, as they cannot independently manage these flows. A DDoS attack entails transmitting a substantial volume of packet-in messages to the controller, resulting in the consumption of bandwidth, memory, and CPU resources in both the control and data planes. OpenFlow switches retain packet-in messages in a buffer prior to relaying them to the controller [48]. Upon receiving a packet that does not correspond with any current flow rules, it is either forwarded in its entirety to the controller or only a segment of the packet header is transmitted as shown in Figure 7.4. When many packets arrive concurrently and the buffer attains its maximum capacity, the switch will transmit whole packets to the controller rather than merely delivering header information. This scenario would result in the use of control plane bandwidth, leading to a total network failure and inaccessibility to legitimate traffic. In a different scenario, substantial inbound packets from the attack may saturate the forwarding table of the OpenFlow switch, inhibiting the controller from implementing new flow rules, which are then rejected by the switch. The switch will be unable

to route packets until memory space in the forwarding table becomes available, leading to delays and the rejection of legitimate inbound traffic. Unique applications can produce contradictory flow regulations, leading to DDoS assaults on the control layer [1].

Assaults aimed at the data plane and communication channel may lead to packet loss, resulting in network unavailability [48]. In SDN, data plane devices must be prohibited from altering stream rules, as this could result in DDoS attacks.

Possible solutions to aforementioned issues include establishing secure communication channels between data and control planes, implementing traffic scheduling and queuing mechanisms, ensuring isolation and monitoring of shared resources, leveraging throughput confidence intervals, and employing Deep Packet Inspection (DPI), among other approaches. Each of these solutions exhibits its own set advantages and disadvantages [48].

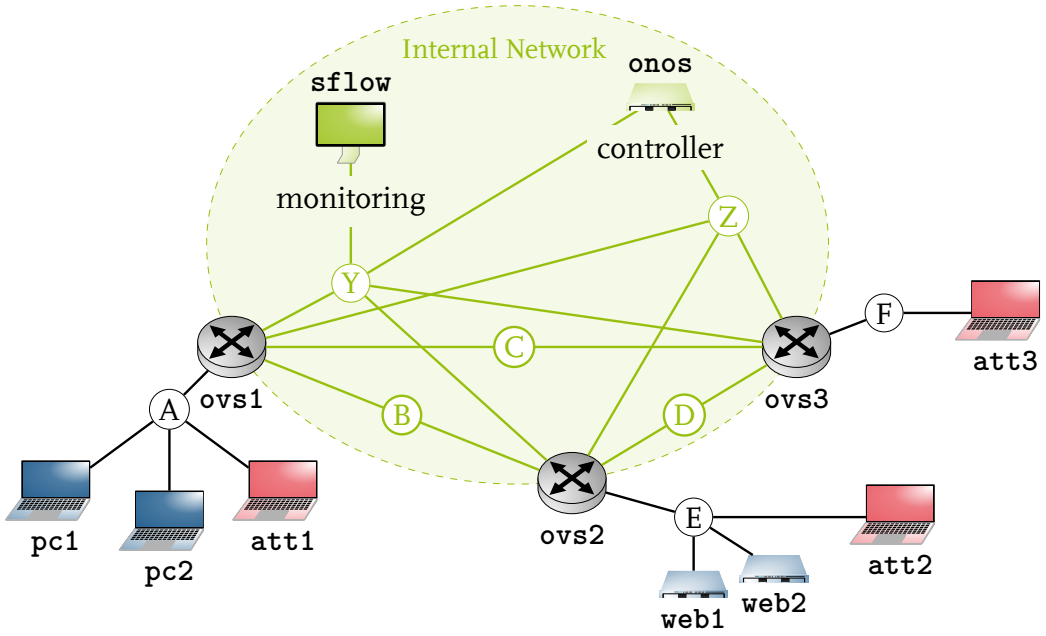


Figure 11.5: SDN network topology deployed for DDoS trials. The "benign" nodes are colored in blue, while the "malicious" nodes are colored in red. The green SDN network is transparent to all external nodes. It is controlled by ONOS and monitored by an `sflow` instance [68].

11.7.2 Emulation of Attacks in Kathará

The initial concept for the design of the first rudimentary implementation aimed to create a network that is straightforward, with a manageable number of components, in order to expedite the process of development and testing. Hence, the resolution was to establish a unified network controller based on ONOS[30] called `controller` alongside three OvSs[59], namely `ovs1`, `ovs2`, and `ovs3`. Additionally, the network consists of four "benign" devices, namely `pc1`, `pc2`, `web1`, and `web2`, as well as three "malicious" devices named `att1`, `att2`, and `att3`. Lastly, a network node is included to house the real-time monitoring tool sFlow-RT[58], referred to as `sflow`. Custom images are utilized for the network nodes and their build instructions are available on Github[60].

11.7.3 Evaluation of Attacks in Kathará

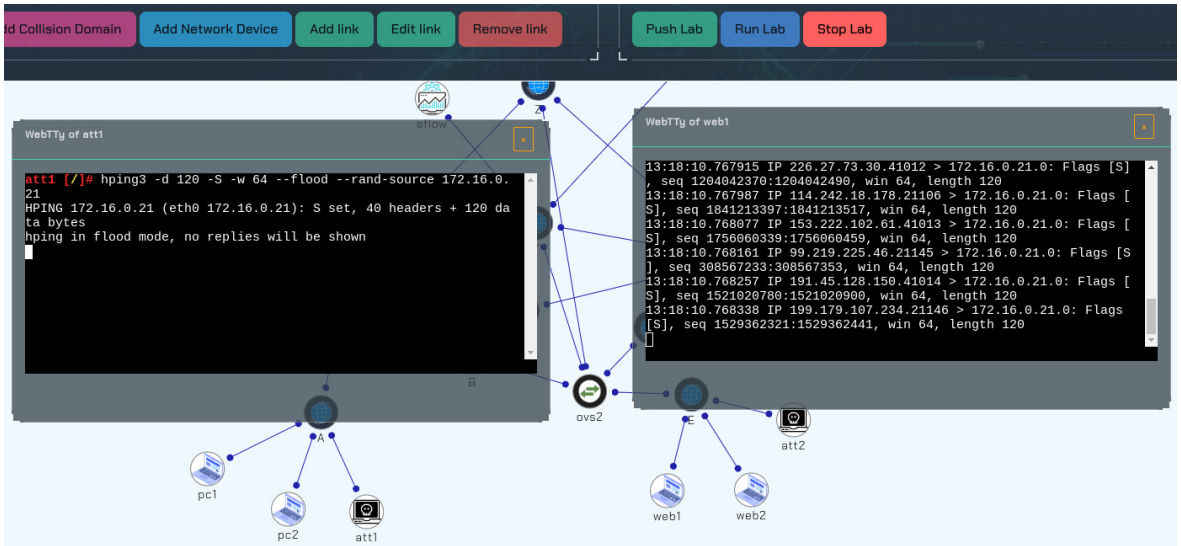


Figure 11.6: Deployment of the DoS demonstration on KaaS. The deployed network of Figure 11.5 was created and started; access is granted to `att1`, which starts an `hping3` to attack `web1` that is capturing the incoming packets with `tcpdump` [68].

Figure 11.6 depicts a Denial-of-Service (DoS) flood attack, specifically a demonstration of a TCP SYN attack, hereafter referred to as *tcp_syn*. The topology illustrated

[60]https://github.com/uniba-ktr/kathara_images

in Figure 11.5 was started in KaaS, and access was acquired to the nodes `att1` and `web1`. We conducted a *tcp_syn* flood attack on `web1` from `att1` by:

```
hping3 -d 120 -S -w 64 --flood --rand-source 172.16.0.21
```

The impending DoS attack on `web1` can be identified using the `tcpdump` program, which discloses multiple random IP addresses sending SYN packets. In the next phase, targets are randomly selected from all prospective attackers within our emulated network to execute a DDoS attack. In this specific situation, capturing traffic with `tcpdump` is unfeasible. Consequently, we employ the `sflow` monitoring capability of our prototype configuration. The `sflow` component identifies and classifies all randomly transmitted flows to a certain destination. At present, it is clear that simulating attacks on network nodes is easily manageable and can be customized to specific tastes through self-developed programs, provided they are compatible with a container environment.

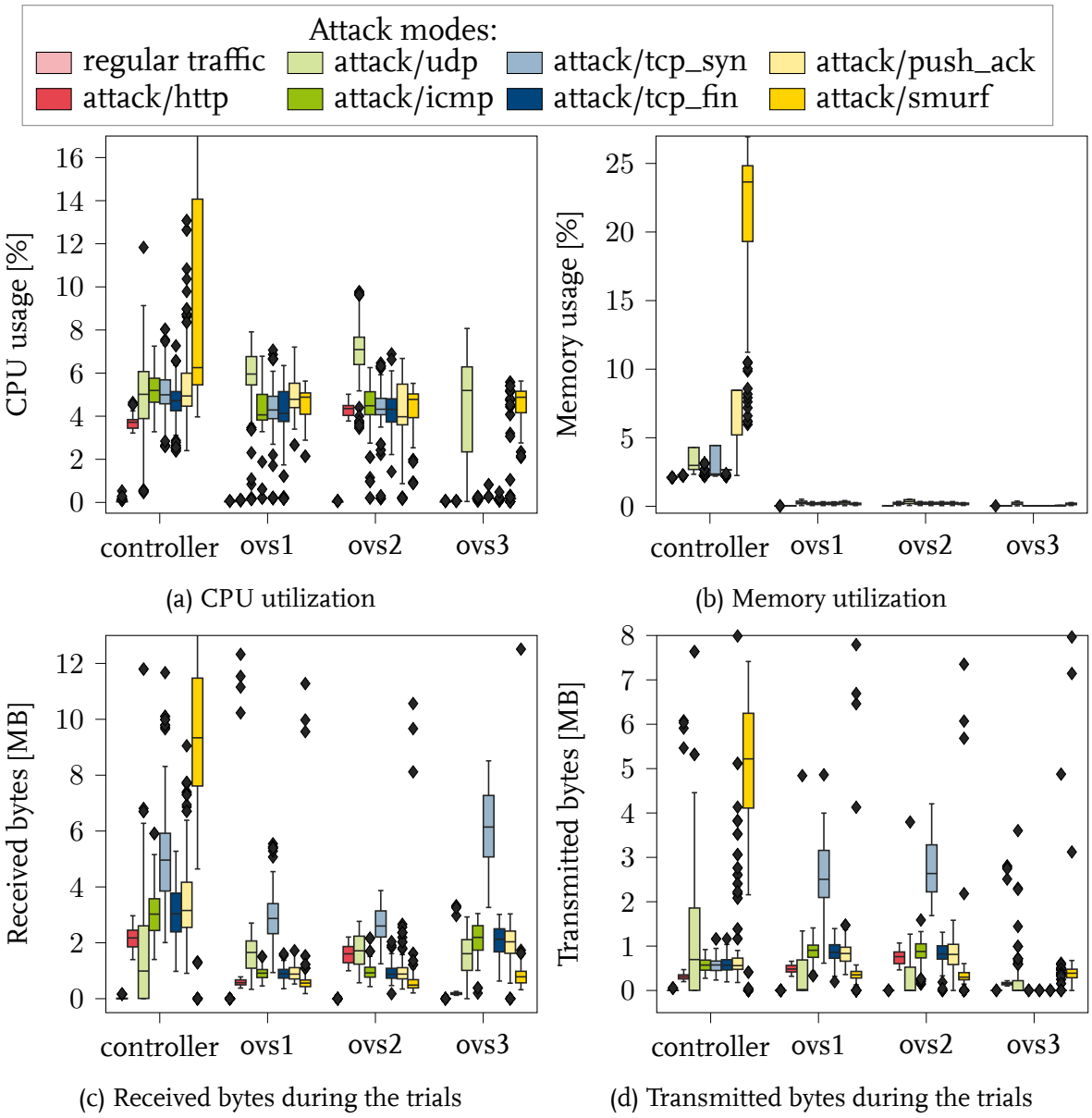
Utilizing cAdvisor[8], we effectively collected statistics from several nodes inside the topology depicted in Figure 11.5. We performed several experiments on a Ryzen 9 processor with 16 physical cores, which effectively quadruple to 32 threads due to Simultaneous Multithreading (SMT), and 128GB of RAM, gathering statistical data throughout a five-minute trial.

We gathered information for a *regular* traffic mode, configured to simulate randomized "benign" web traffic. By augmenting the *regular* traffic mode, we create a distributed *http* attack mode by utilizing `curl` to fetch the web pages. All other attack types entail generating randomized DDoS attacks utilizing diverse protocols from the malicious devices, which employ random source IP addresses to inundate all benign devices through flooding. They employ the tool `hping3` on all "malicious" devices to systematically target other individuals in a distributed fashion.

Our DDoS attacks can be classified into

- *volumetric* attacks, namely *icmp* and *udp*,
- *protocol* attacks, namely *smurf*, *tcp_syn*, *tcp_fin*, and *push_ack*,
- *application* attacks, namely *http*.

In the evaluation presented in Figure 11.7, we omitted some outliers from the box plots to enhance the clarity and readability of the remaining boxes. The graphic of Figure 11.7a displays CPU utilization, with the "benign" setting demonstrating a minimal footprint. This fact is also apparent in the memory utilization depicted



(a) CPU utilization

(b) Memory utilization

(c) Received bytes during the trials

(d) Transmitted bytes during the trials

Figure 11.7: Utilization statistics for the networking nodes of the trial topology of Figure 11.5 [68].

in Figure 11.7b. The analysis in Figure 11.7c focuses on the received bytes on the interfaces linking *ovs1* to *ovs3* with the external networks (A, E, and F) and the *controller*'s interface to Z. In comparing attack modes, it is clear that broadcast *smurf* and UDP attacks need more CPU and memory resources than other attack modes. The controller primarily addresses traffic generated by *smurf* broadcast messages, subsequently focusing on *tcp_syn* attacks, whilst the OvS nodes encounter substantial traffic from *tcp_syn* assaults.

With the extension of the network emulation tool Kathará, we can now assess novel transmission paradigms for services functioning in dispersed contexts. Moreover, even the replication of DDoS attacks to detect vulnerabilities is realizable and vendors may formulate tactics for threat mitigation before deploying services. All in all, the goal of network emulation is to develop services that can endure hostile activities and sustain functionality under difficult conditions. In the subsequent phase, we create a middleware to coordinate these robust services in the Cloud-to-Fog-Continuum, facilitating smooth integration, dynamic load balancing, and adaptive fault tolerance.

PART IV

Towards Service Placement in the Cloud-to-Fog-Continuum

A person who never made a mistake never tried anything new.

A. Einstein

Parts of this chapter are taken from [43], [44], [74], [75], [77], [108]

Initially, we examine the concept of cloudless computing in Chapter 12, demonstrating that services may be disseminated over a network without a fixed location throughout the Cloud-to-Fog-Continuum.

To deploy those microservices on a virtualized network, some groundwork is required: a custom Docker network plugin that supports OvS, as detailed in Chapter 13. The pertinent sections of the Docker API are illustrated in Subsection 5.2.2 for that implementation. Entries in the OvS Database (OVSDB) are modified for interconnection to OvS, as detailed in Section 13.2.

The previously built Docker plugin enables the establishment of a cloudless infrastructure, as illustrated in Chapter 14. In the context of containerized services, as discussed in Section 14.1, a plugin for the SDN controller ONOS is developed in Section 14.2 and compiled using a CI tool. Upon identifying the transmission packets, Section 14.3, intents can be established to appropriately route the traffic, Section 14.4. Ultimately, the plugin facilitates the management of Docker hosts, as demonstrated in Section 14.5.

For delivering services in such an environment, a robust monitoring system is needed to facilitate such a complex service orchestration, applicable in SDN enabled deployments as shown in Chapter 15. Therefore, existing monitoring solutions are explained in Section 15.1 and are compared in Section 15.2. However, for the service monitoring in a Cloud-to-Fog-Continuum environment, we adjust the the former solutions in Section 15.3, subsequently depicting the comprehensive architecture of the Cloudless Resource Monitoring (CRM) system in Section 15.4. The system's capabilities are demonstrated in Section 15.5.

Finally, the orchestration framework is established in Chapter 16 and further refined by FL. In accordance with the MAPE-K paradigm, a proposal for a FL middleware for service distribution within the Cloud-to-Fog-Continuum is presented in Section 16.1. The fundamental notion of FL is delineated in Section 16.2, while the positioning of services is elucidated in Section 16.3.

Chapter 12
Cloudless Computing

With the progress in computational, storage, and network technologies, along with the growth of the Internet, a new computing paradigm called CC has risen as described in Section 6.1. Mainly, resources such as CPU and storage are offered as general utilities that users can rent and relinquish as needed via the Internet. The building of large data centers has been preferred due to the high expenses associated with design, setup, and maintenance, which helps reduce the need for repetitive and expensive tasks. In addition to the clear benefits of adhering to this approach, there are several consequences that, depending on the specific situation in which it is applied, might make such a CC service inappropriate. Constructing a reduced number of larger data centers results in an increased average distance between users and the nearest center. This has two implications: Initially, it is important to consider that potentially confidential information may need to be transmitted over national boundaries, so traversing several legal authorities. Furthermore, the physical separation between locations might result in an undesirable delay for certain applications that need strict real-time performance [151]. Nevertheless, there are other situations in which the given latency is deemed unsatisfactory, particularly in the context of the IoT. While it is evident that offloading several activities from resource-limited, highly portable devices has advantages, the inability to fulfill real-time requirements is sometimes a decisive factor that prevents it from being viable. Autonomous automobiles, for instance, produce around 40GB of data per hour. However, it is not feasible to transmit this data to the cloud and process a decision, which is then sent back to the car within a few moments [3]. With the extension of EC, as described in Section 6.2,

the relocation of services (computation, storage, network) to the network edge is introduced to decrease the distance between service and their requester. Furthermore, with FC, as shown in Section 6.3, hosting of services is expected on the path from CC data centers to the network edge. Fog, in a metaphorical sense, has resemblance to a cloud that is in closer proximity to the ground, symbolizing the hosts inside the network [18]. The enhanced proximity to end users, namely potential buyers of the offered services, significantly reduces transmission times and traffic volume. This is a crucial milestone in facilitating latency-sensitive applications that require near-real-time responses. Furthermore, IoT applications benefit from the compact size of data centers utilized in FC, which can be conveniently spread and efficiently expanded at a low cost. These small-scale centers also allow for the establishment of additional fail-safe and redundant services [3]. The evolving computer paradigms necessitate the corresponding networks and their configuration to adjust correspondingly. In a microservice oriented design, significant traffic is created between several computers before the end-user receives a satisfactory result. In this highly dispersed paradigm, the seamless functioning of the expanding interdependent applications relies on the accessibility and functionality of all their components.

Definition 12.1: Cloudless Computing

Cloudless computing, in analogy to serverless computing, denotes the ability to run apps and services autonomously from centralized cloud infrastructures in the Cloud-to-Fog-Continuum. Its purpose is to use processing power and data storage, which is locally nearer to consumer devices, hence reducing or even eliminating the dependence on faraway cloud services. Nevertheless, depending on the deployed service, cloudless computing primarily enhances CC and particularly improves metrics like latency and throughput.

Due to its reliance on conventional networks, there is a lack of support for the automated adjustment of rules that regulate access to these services as required. The ensuing intricacy makes these networks nearly unchanging, since the effort required for maintenance, updates, and the related danger of service disruption is rapidly increasing. The word “ossification” is widely used to characterize the process of slowing down innovation and the advancement of the Internet and network technologies [126]. Due to the increasing insufficiency of traditional net-

works, the architectural concept of SDN, as shown in Section 7.1, has acquired significant traction and appeal. The separation of the control plane from the data plane is a fundamental idea that allows for more flexibility. Historically, networking hardware has followed a vertically integrated approach, where both the control and data planes were contained within the device itself. This design limited external manipulation and hindered innovation by restricting flexibility. The resultant control layer separates itself from all network devices that were previously closely connected, hence eliminating the requirement for manual and individual configuration. The network hardware is programmed using a defined communication protocol, such as OpenFlow, based on the requirements of the controller and controlled switches. Enhancing efficiency in a conventional network has historically been accomplished by employing middleboxes that provide a highly specialized function. The construction of such a highly specialized gadget provides opportunities for optimizing power consumption, efficiency, and output. The fundamental proposition of NFV, proposed by a specialized interest group inside the ETSI, is to shift from application-specific integrated circuits to software-based solutions that may be executed on general-purpose hardware, as described in Section 7.2. The primary objective is to dismantle the strong relationship between function and physical device. Deploying virtual appliances no longer necessitates laborious, manual, and individual device setup and configuration. Instead, it may be accomplished in a coordinated, automated, and remote manner. The specification of such services is separated from hardware development, promoting the growth of a more extensive software ecosystem. Within this context, the availability of potential network functions increases, allowing for the acquisition of updated or novel features from a remote location. This notion greatly benefits from being utilized in a network environment that implements SDN concepts. Ultimately, combining all the aforementioned technologies results in a middleware stack that offers a cloudless computing environment to future cloudless service programmers. It is important to have clear visibility into the deployment of a (micro-)service application or a series of functions in the Cloud-to-Fog-Continuum, including all the devices that have both high capabilities and limited resources. In our example test-bed, we are putting up an SDN environment where the controller, which is in charge of the network infrastructure, also orchestrates the NFV deployment. Consequently, it necessitates a variety of virtual and real soft- and hardware goods.

We start from the bottom and create an standardized interconnection plugin for Docker to enable newly distributed containers to be directly attachable to OvS in Chapter 13 With this basic communication setup a controller can easily manage services, create them on a corresponding not and setup the traffic flow rules, where we developed an ONOS plugin in Chapter 14.

Chapter 13
Docker Network Plugin

There are multiple options available to achieve the desired functionality. One approach is to create a middleware that establishes connections with all the pertinent components.

Therefore, Docker offers a range of techniques for initiating, terminating, and removing containers, among other functions, all of which can also be accessed via a CLI, which connects to the Docker Engine API. On the other hand, OvS allows for the modification of the configuration through a standardized interface. With a plugin we achieve the capability to establish the necessary connections and interfaces using OvS, as well as retrieve the necessary data or control containers through the Docker Engine API.

13.1 Plugins for Docker

Docker offers supplementary interfaces to enhance functionality by means of plugins. The following interfaces are referenced[61]:

Volume - Volume Plugins enhance containers by adding external, persistent storage functionalities, such as „Amazon S3“ or „Azure File Storage“. There are numerous existing implementations for this interface.

Graph - Enables the expansion or management of container storage. For instance, a storage location can be simultaneously incorporated into multiple containers.

[61] https://github.com/docker/go-plugins-helpers

IPAM - Can be seen as an expansion of network plugins. The IP Address Management (IPAM) interface facilitates the assignment of IP addresses to containers, which can be controlled by external software. For example, it has already been effectively utilized in conjunction with a network plugin that has been published by Weave[62].

Authorization - In order to gain access to the APIs, external programs must authenticate themselves using a certain technique. Docker enables the development of custom backends for this subsystem, which subsequently provide access. By default, a user has unrestricted access to all functions of the API. An example of implementing user permissions using the API is the abstraction of user permissions. The Docker Engine Network Plugins allow Docker to incorporate a range of additional network protocols, expanding its capabilities in networking. Some examples include Virtual Extensible LAN (VXLAN), IPVLAN, MACVLAN, and various others.

The interface for network plugins is evidently intriguing. First, we can establish a network in Docker and specify that your own plugin will assume the responsibility of managing it. If done, we can start a new container as shown in Listing 13.1.

```
1 docker network create --driver mynetworkdriver mynetwork
2 docker run --network=mynetwork busybox
```

Listing 13.1: Network creation with Docker

Relevant data will now be transmitted to our personal plugin. Subsequently, it can react to it and set up the network element, which in this instance is OvS. The Docker Engine API allows for the execution of the same commands. This is a straightforward approach for connecting OvS to the forthcoming Docker Network plugin.

The potential use cases for such a plugin are believed to be varied. The implementation of a notion for utilizing containers on routers or switches should take advantage of the practice of deploying applications on edge or fog nodes. In order to streamline the process of outsourcing, the utilization of Message Queue Telemetry Transport (MQTT) or Constrained Application Protocol (CoAP) could be advantageous. The plugin can be utilized to further investigate these technical

[62] https://www.weave.works/docs/net/latest/overview

capabilities, as demonstrated in a study on SDN by Xu *et al.* [160]. An alternative method would involve assessing if the flow controller is capable of performing the same automatic detection as mentioned.

Switches or routers usually have a significantly restricted computational capacity. Hence, if the interconnected devices exhibit a substantial need for Fog applications, there is a potential for rapid overload, notwithstanding the enhanced operational efficiency facilitated by Docker. Implementing load distribution over multiple nodes would effectively address this issue. Several potential technologies exist that could be utilized for this purpose.

Our trial system utilizes Docker Swarm to centrally oversee the Docker instances of participating devices, which provide computational resources and spread the workload among themselves. Additionally, there is an API available here that enables automated administration. Technologies like VXLAN or Network Virtualization using Generic Routing Encapsulation (NVGRE) facilitate the implementation of overlay networks, allowing remote programs to communicate with one other. The centralization of IP assignment can be achieved by expanding the plugin with the IPAM driver. An overlay network is a network that is created and managed on top of an existing network infrastructure. This refers to the installation of one or more separate layer 2 networks on a layer 3 network.

This example can be applied by analogy to other domains. The implementation of cluster environments would be straightforward using our plugin. Therefore, in [70] we established a controlled setting for evaluating the performance of a group of ARM boards. In order to minimize expenses, one might utilize ARM boards, such as the widely renowned RPi. Additionally, presented here is a suggested application case for the plugin.

Relevant data will be transmitted to our personal plugin. Subsequently, it can react to the request and configure the network element, which in this scenario represents OvS. The identical instructions can also be carried out using the Docker Engine API. This is a straightforward approach for connecting OvS to our custom Docker Network plugin.

13.2 Open vSwitch Database

As previously mentioned, it is feasible to generate many virtual switches and allocate ports using OvS. OpenFlow enables the controller to manage or control the flows in these switches. Using OpenFlow directly, it is not feasible to establish or remove these switches or their ports. The mapping between the controller and the switch must also be recorded as information. OvS maintains this information in a database known as OVSDB. The utilization of the OVSDB protocol is necessary in order to gain access to the database. Essentially, OpenFlow manages dynamic and quickly changing data, while OVSDB oversees data that is infrequently altered.

To obtain a portion of the existing content, you can straightforwardly use the given command.

```
1 ovsdb-client dump
```

The CLI program `ovsdb-client` is an invaluable tool for interfacing with the OVSDB.

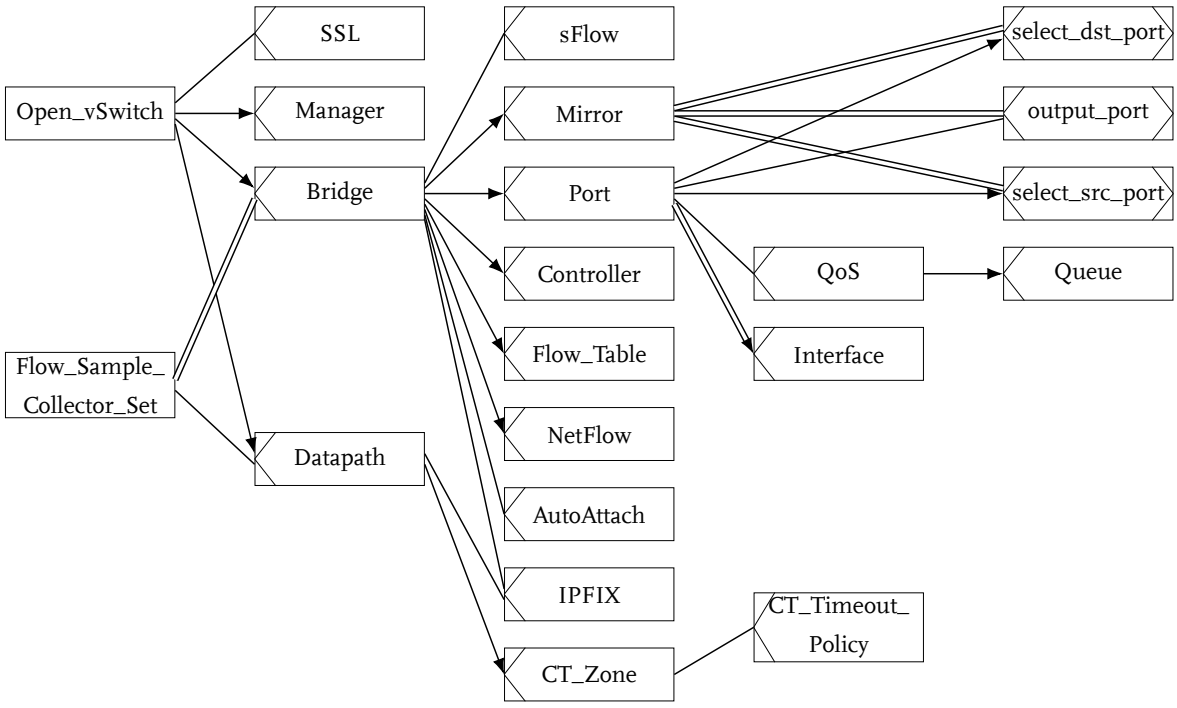


Figure 13.1: Table structure of the OvS database in SERM semantic

The primary table in the database is named “Open_vSwitch”. Entries in the secondary tables are reliant on entries in the primary table, either directly or indirectly, with the exception of a small number of specifically chosen entries. If a new entry in any of the auxiliary tables does not have a reference to the primary

table, it will be automatically removed. The image labeled as Figure 13.1 provides a clear representation of the tables that rely on the primary table "Open_vSwitch" and their interrelationships.

Below, a more detailed description of some of the selected tables is depicted:

controller - Authorized controllers interact with the database via the OVSDB API.

bridge - A table containing information about the bridges that already exist. To add an entry in the "Open_vSwitch" table, it is necessary to modify an existing record to create the reference. To complete this task, you need to enter the identification number of the new bridge in the "Bridge" column.

Port - Configuration of the port. This table relies on the table "Bridge".

Interface - Each port requires one network device, which can be either physical or virtual. This table is reliant on the table "Port".

controller - Designated OpenFlow controller. There could be one or more entries.

Figure 13.1 displays other tables, but only the ones that were assigned numbers are pertinent to this study.

The switches establish a connection with this database using a protocol based on JSON-RPC. This provides them with port configuration as well as tunnel settings. The database server can be deployed on alternative servers as well. Nevertheless, the default configuration makes a connection to a server that is located on the same machine. The protocol is standardized and defined in RFC7047[63]. It contains definitions for all configuration functions. The `ovs-vsctl` utility also use this interface. The JSON-RPC protocol employs JSON for the transmission of instructions, information, and the reception of responses. A JSON-RPC query is composed of three parameters:

Method indicates the specific name of the function to be executed. The answer to this question is contingent upon the specific protocol employed by JSON-RPC.

Params is an array consisting of objects that are used to pass the arguments.

ID is an unique identifier for the request. Utilized for assigning the responses.

[63]https://datatracker.ietf.org/doc/html/rfc7047

The response consistently adheres to the subsequent structure:

Result is a JSON object and the server address.

ID is the request ID of the call.

13.3 Interconnection of OvS and a Docker Container

A virtual Ethernet device (veth) pair closely resembles an actual network connection. Here, a "cable" will be connected between two network ports to facilitate the transport of data. In the simulated scenario, the veth pair assumes the role of the cable. Subsequently, the packages are transmitted within the OS from one virtual network port to another. The virtual ports function in a manner that is indistinguishable from actual, tangible ports. Veth pairs are commonly utilized for connecting to virtual switches like Linux Bridge and OvS, or to namespaces [cf. 114]. Given the circumstances, it is appropriate to discuss Test Access Point (TAP) device files. These are access points in Linux systems that allow programs to send or receive data packets to the Kernel, thus simulating a NIC.

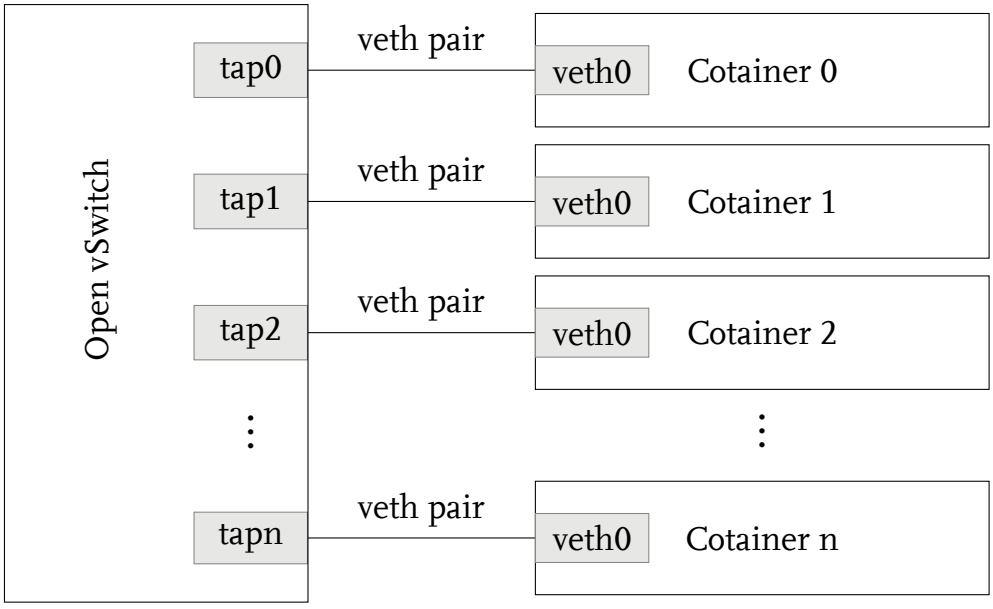


Figure 13.2: Connection of OvS TAP interfaces to Docker's veth ports

The diagram depicted in Figure 13.2 illustrates an architectural design that enables the connection of various containers to the network using OvS. On the left, there is a switch or bridge of OvS. On the right, you may observe many containers. By default, OvS bridges automatically generate an internal port that shares

the same name as the bridge. This enables the assignment of one or several IP addresses to the bridge.

Docker utilizes the former mentioned technologies to achieve network connectivity between containers. The "libcontainer" component requires the veth pair to be configured and provides the newly formed TAP device file to the container or namespace.

The API for network plugins is utilized as an interface to Docker, as previously explained. The fundamental protocol for this is already implemented as a component of "libnetwork". "Libnetwork" is a Go Programming Language (Go) library designed to simplify the utilization of the network API for developers. It offers a versatile framework that enables both built-in and remote drivers to oversee the network connectivity of the containers and offer a tool that facilitates the convenient testing of one's own implementations [cf. 165]. "Libnetwork" explicitly implements the Docker Container Network Model (CNM), which outlines the necessary procedures for configuring and managing network connections of containers using plugins. The plugins that have been created are registered and loaded by Docker in a dynamic manner.

As per documentation, the CNM consists of three primary components:

Sandbox - A sandbox includes a container's network stack configuration. Depending on the OS, it can be defined as a Linux network namespace, a FreeBSD Jail, or a similar notion and has the capacity to accommodate one or several endpoints. It is feasible to deploy sandboxes for many OSs in order to enable the implementation of network functions for each of them. Currently, the developers have just realized the implementation for namespaces.

Endpoint - An endpoint is a connection point that links a sandbox to the network.

Network - A network refers to a collection of endpoints that have the ability to establish direct communication with one another. Examples include a Linux Bridge, an OvS Bridge, or a Virtual LAN (VLAN). Furthermore, a network can encompass a multitude of diverse endpoints.

The library includes the "NetworkController" object, which serves as a straightforward means of accessing "libnetwork" and executing its API operations. For instance, the "NetworkController" enables the connection of a certain driver or plugin to a network.

Additionally, a RPC-based protocol is utilized in this context. RPC queries are dispatched to the registered plugin immediately when a certain event necessitates them. Subsequently, the plugin is capable of executing specific actions and returning responses. Currently, there are a total of fourteen API methods, out of which six are specifically related to labor.

NetworkDriver.CreateNetwork - The execution of this remote procedure call necessitates the plugin to establish a fresh network. For instance, a recently created OvS bridge. In addition, the plugin receives information to precisely configure the network, including the designated IP pool for both IPv4 and IPv6 addresses, as well as the default gateway.

NetworkDriver.DeleteNetwork - When a network that has been associated with a specific plugin is removed, that plugin is notified of this action.

NetworkDriver.CreateEndpoint - is invoked when a new endpoint is created on a specified network. The provided information includes additional parameters that are not required but can be used for help. These parameters can be given during the creation of a container using the Docker CLI, as an illustration.

NetworkDriver.DeleteEndpoint - Just like the function "NetworkDriver.DeleteNetwork", this applies when deleting an endpoint.

NetworkDriver.Join - When a sandbox is provided an endpoint, the plugin receives a call. Typically, this occurs when you initiate a container.

NetworkDriver.Leave - This refers to the action of removing an endpoint from a sandbox or ending a container.

The function "NetworkDriver.Join" possesses certain idiosyncrasies, hence it will be examined in further depth in the following section.

When a sandbox is allocated an endpoint, a "POST" message is transmitted to the plugin in the specified format:

```
{
 "NetworkID": string,
 "EndpointID": string,
 "SandboxKey": string,
 "Options": { ... }
}
```

Listing 13.2: Join request

The terms “NetworkID”, “EndpointID” and “SandboxKey” serve to consolidate the corresponding components. Additional information can be sent to options of Docker’s remote drivers[64].

```
{
 "InterfaceName": {
  "SrcName": string,
  "DstPrefix": string
 },
 "Gateway": string,
 "GatewayIPv6": string,
 "StaticRoutes": [{
  "Destination": string,
  "RouteType": int,
  "NextHop": string
 },...]
}
```

Listing 13.3: Join answer

The use of a gateway is not required, and it is worth noting that IPv6 is already compatible. The entries listed under the “InterfaceName” section specify the name of the interface as it is recognized by the operating system within the container. “SrcName” refers to the name contained within it, whereas “DstPrefix” is the prefix used to name the interface within the sandbox. For example, if the prefix is “eth”, the interfaces would be named “eth0”, “eth1”, “eth2”, and so on.

The items in the “StaticRoutes” section represent the routes that will be added to the sandbox. Once again, this is an array, meaning that multiple routes can be added.

[64]https://github.com/docker/libnetwork/blob/master/docs/remote.md

The “RouteType” item can only have a value of either 0 or 1. If the value entered is 0, then the NextHop field must be completed. The gateway at 1 is identified by the item “Gateway” (refer to Listing 13.3). Additional routes will be included if necessary. This will replace the entry in the “StaticRoutes” section.

If a default gateway or default route is not specified in the “StaticRoutes” section of a join response, “libnetwork” will add an additional virtual NIC in a sandbox. Next, it is linked to the Docker network called “docker_gwbridge”. Furthermore, the sandbox will be assigned a regular pathway across this supplementary network.

13.4 Structure of the Docker Open vSwitch Plugin

The plugin’s source code is mostly separated into three primary components.

driverHandler - establishes the connections to various APIs and constructs an object to hold the information for future utilization.

driverImpl - contains the functions that are invoked when “libnetwork” receives a message from Docker.

ovsdb - handles the connection to the OVSDB and includes functions to handle it (create, remove bridges, etc.).

```
1  func (driverObject *DockerDriver) Join(j *network.JoinRequest) (*
       network.JoinResponse, error) {
2
3   joinresponse := &network.JoinResponse{
4    InterfaceName: network.InterfaceName{
5     DstPrefix: "eth",
6     SrcName:   containerVethName,
7    },
8    Gateway: gatewayIP,
9   }
10
11  return joinresponse
12 }
```

Listing 13.4: Excerpt of the Docker OVS driver implementation

Docker offers a range of frameworks for Go. One of the updates includes a feature that simplifies the integration of network plugins, which was also utilized for the plugin being discussed here.

In order to accomplish this, the class “driverImpl” implements the “Driver” interface, which is specified by the Software Development Kit (SDK). Once a specific event occurs, such as receiving a message from Docker, the SDK immediately invokes the implemented methods of the interface.

The following extract is displayed in Listing 13.4 from the file “driverImpl”. The “Join” method is invoked upon receiving a message with the same name. In Go, there are no “Implements” operators available, unlike to Java. Conversely, the implemented methods are allocated to an object. In this scenario, a `Struct` defines an item called “DockerDriver”. The SDK receives this object and subsequently handles the connection to Docker.

The library “libovsdb”[65] is used for communication with the OVSDB and was crucial in implementing the functionality. SocketPlane is the firm that administers this library.

13.5 Trial Topology for Evaluating the Functionality of the Docker Plugin

The experimental setup consisted of four Docker containers and five OvS switches. These containers are capable of executing a range of programs, including Mininet, ONOS, a basic web server that serves web pages, and a client for fetching those web pages. The network’s structure is shown in Figure 13.3. Through the red pane, we have a comprehensive view of all the components that are being managed by Docker. The green lines depict the network connections, while the blue lines indicate the connections between the OvS switch and the OpenFlow controller ONOS. S1 to S3 denote the OvS bridges realized with Mininet, while the last two indicate the OvS switch of the host system.

Multiple configuration files were generated for the “docker compose” tool to facilitate the quick initialization of the test environment. Subsequently, a script assumed control of the network connection setup. On this occasion, the network areas are inputted into the configuration files prior to the commencement of operations. Upon startup, the plugin automatically takes on the responsibility of assigning network connections.

In the case that Docker containers are to be connected to external networks, a connection between the NIC and the OvS switch of the individual subnets is still

[65] https://github.com/socketplane/libovsdb

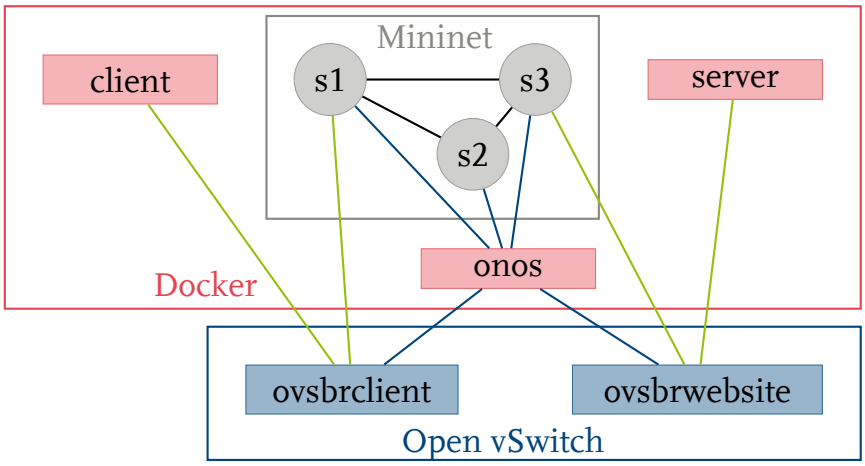


Figure 13.3: Trial network for the Docker OvS plugin

missing. In addition to using veth and Bridge interfaces, the option of connecting the bridges to the outside world using IPtables is often utilized. This strategy has proven to be less performant than directly using the host system's network interface.

Docker also partially implements this through Network Address Translation (NAT) using Netfilter/IPtables rules. This is not easy to implement with OvS, as it does not implement the Netfilter API[66].

Instead, another OvS bridge/switch is used and connected to a physical network port that is connected to the external network. The OvS bridges of the individual Docker networks are also connected to this OvS switch.

13.6 Docker Plugin Capabilities

The development of a plugin for Docker to dynamically connect OvS networks to containers was scrutinized, and the technologies employed to attain the needed functionality were evaluated. The software code is now comprehensible and provides a foundation for subsequent investigation. The plugin could be used and extended in various ways, such as providing significant added value to run outsourced applications in Docker containers.

Several outstanding issues require implementation, including centralized administration through the integration of various interfaces, the addition of flows via the Northbound API, and the implementation of port release for containers. Al-

[66] http://docs.openvswitch.org/en/latest/faq/issues

though still in the nascent phase of development, FC possesses significant promise that requires additional advancement. This plugin establishes a robust basis for the swift and effortless configuration of future apps and testing environments.

The plugin is primarily autonomous from external factors and is adaptable for use. Nonetheless, Docker developers restrict certain decision-making autonomy of the plugin developer. Custom parameters may be provided to the plugin either the start command or the “docker-compose.yml” file; however, this must be done manually or executed via flow entries.

Chapter 14
Cloudless Infrastructure

In the previous configuration of Figure 13.3, a trial network is shown, which is initiated with the custom Docker network plugin of Chapter 13. The various components were incrementally incorporated based on the requirements of each development stage, ultimately resulting in the final form. The Docker container hosting the Mininet[6] topology and the container running the ONOS controller instance are taken from the topology of Figure 13.3 and we concentrate on the CI efforts of developing ONOS apps now. The discoveries from this investigations are used to construct and distribute the ONOS controller image on Dockerhub, along with the custom application. Mininet[6] utilizes OvS to extend a network, enabling the seamless integration of its components into the overall SDN network structure. To register the OpenFlow enabled switches to the controller, simply direct them to the controller's IP, which in our example is 192.168.16.99, and specify the right port, 6653. The Internet Assigned Numbers Authority (IANA) has designated this port number for OpenFlow communication, making it the protocol's standard. The diagram highlights the communication between the switch and the relevant controller with a dotted green line. This connection links all OpenFlow enabled network devices with the Docker container that hosts ONOS.

Several additional containers, with various purposes, are included in several networks, thus expanding this topology. The process of connecting these networks to OvS switches is accomplished by using the plugin of Chapter 13. The containers' names correspond to their primary function: The primary purpose of *docker_client_1* is to serve as a container for transmitting requests across the network. It is equipped with essential utilities such as `dig`, `ping`, and `curl`. The

address to access it is 172.21.21.21. The *docker_website_1* image serves as a basic web server that may be used for testing and debugging by making a `curl` request. The container *docker_dind_2* is located in the same subnet and connected to the same OvS switch. The objective of this is to facilitate the use of Docker within Docker and provide access to the Docker host using a REST API by exposing TCP port 2375. This interface will serve as the foundation for transferring and overseeing services and capabilities to and from Docker hosts in the network. The API enables sophisticated administration of the Docker host operating within the container, encompassing the ability to inquire about its status and manipulate its state. This includes initiating the retrieval of images, creating containers, and managing their life cycle. The container with the IP address 172.21.19.19, named *docker_dind_1*, offers the identical functionality.

The provided configuration enables the fully virtual testing of ONOS and an application specifically built to coordinate services operating on Docker hosts. Given the availability of two distinct Docker hosts, it is possible to implement and test the selection process for a suitable Docker host for a service request. To create and evaluate the topology with a more authentic configuration, the testbed was expanded on the physical aspect.

We added the Zodiac FX[67], which is an OpenFlow enabled switch that facilitates the connection of physical hosts to the topology. Port 4 of this switch is designated for communication with the controller and must be linked to the host computer. The PC port is referred to as `enx00` and establishes communication with the controller using the OpenFlow protocol. To facilitate normal network communication between the linked hosts and switches, an additional port must be attached to the host. This is accomplished using port 3 and `eth1`. A link must be created on the host's side between the network interface `eth1` and an already existing OvS switch.

Two SBCs, including a RPi, serve as representations of actual hosts. This gadget is directly linked to port 1 with a physical connection. The final step of the project setup involves connecting an Orange Pi to the Zodiac FX[67] switch using its built-in LAN connector. To establish contact with the controller and ensure a clear distinction between OpenFlow and controller traffic from other network communication, an extra port is connected to the board. The sole function of this port's connection to the host is to serve as the parent interface for a Macvlan network. The ONOS

[67]https://github.com/NorthboundNetworks/ZodiacFX

controller is located within this network, enabling direct communication between the controller and the Pi. The Orange Pi has two interfaces that are connected to an OvS bridge. This bridge provides an internal port with an IP address that may be used to access the Orange Pi. Additionally, this device has a Docker host that can be accessed and controlled by the controller through the Macvlan network stated above. This allows ONOS to directly communicate with the Docker API.

This configuration leads to a multitude of interconnected devices, including hosts and switches, both virtual and physical, spread across multiple independent subnets. Configuring the endpoints is important to enable communication between them. While this project is centered around the networking paradigm of SDN, which emphasizes flexibility and centralized management, traditional configuration via IP routes offers faster and simpler debugging and development processes. The tool utilized to establish communication between the controller and the diverse endpoints of the network will be `dig`, which stands for Domain Information Groper[68]. This application is commonly utilized for querying DNS servers, transmitting its messages through UDP on port 53. Additionally, it must have the capability to transmit the packets to their intended destination based on the designated IP address. The typical application of the dig command is demonstrated in Listing 14.1.

```
1 dig [@Server] [Domain] [Typ] [-x IP-Address] [queryopt...]
2 dig @10.0.0.3 +time=1 +tries=1 +short stopTraffic
```

Listing 14.1: Syntax of the dig command (l.1) with example (l.2)

Following the execution of the `dig`[68] command, it is possible to direct the request to a particular DNS server using its IP address, which begins with the symbol "@". In the absence of a specified server, the program will search for a server in the `/etc/resolv.conf` file that is responsible for supplying default name servers. In Listing 14.1, the second line sets the destination of the DNS packet to 10.0.0.3 and specifies query options that aim to expedite the transmission of basic instructions. The *time* modifier sets the timeout to the specified value, which is 1 second in the given example. Additionally, the "tries" option decreases the number of UDP attempts to 1, as opposed to the usual 3. The term "short" is used to abbreviate the output. This is because the information collected is no longer of interest to the host, notwithstanding the initial intention. The query is named `stopTraffic`

[68]https://linux.die.net/man/1/dig

and it serves as an instruction at the end of the call. This text will be subsequently parsed by the ONOS controller and the custom-built application.

`Dig`[68] requires that packets be sent to the initial network device using conventional IP routes, which contradicts the SDN model. However, it is predictable due to the default utilization of UDP at destination port 53. The produced packets can be augmented with customized strings, which provide the instructions for the SDN controller. One additional benefit of utilizing DNS packets to coordinate services is the capability to monitor and analyze the resulting traffic by examining the corresponding flows. Any additional computational burden caused by these types of orders can be identified and addressed at an early stage with ease. Using the services relies on traditional IP routes, whereas a genuine SDN network would require an extra name resolution service. The basic configuration of an intent is typically insufficient, as the service provider still needs to be accessed through a known IP address, which ideally should no longer be necessary.

Creating a resolution service of this nature is a comprehensive undertaking that may be explored in a future endeavor.

After ensuring that all the containers have been correctly initiated, the topology will comprise three Mininet[6] switches that are interconnected in the central area, with the remaining switches positioned loosely around them. Due to the dynamic assignment of names to the OvS switches that connect to the Docker containers during startup, it is not possible to predict their names in advance. Therefore, a script is required to establish the connection with these switches. This script assigns the appropriate interfaces as ports to the switches in Mininet[6]. This functionality is achieved by incorporating the Mininet[6] container into each of the other networks specified in the Docker Compose file, so granting it access to the network interfaces of the other containers. The OvS switch can be modified by using the `ovs-vsctl` program, which enables the manipulation of virtual switches. The operation `ovs-vsctl add-port s1 eth1` adds the network interface `eth1`, which is connected to the 172.21.21.0/24 subnet, to the switch called `s1`. Listing 14.2 displays the result of the `ovs-vsctl show` operation for bridge `s1`. Lines 2 and 3 indicate that the switch is linked to an SDN controller at the specified address. Next, the device has various ports accessible, with `s1` being the standard internal port. The most intriguing aspect is the final line highlighted in red, which is appended subsequent to the execution of the script, thereby incorporating `eth1` into the roster of

ports. This is the method by which all autonomous Docker containers are linked to the pre-existing Mininet[6] topology of OvS switches.

```
Bridge "s1"
 Controller "tcp:192.168.16.99:6653"
  is_connected: true
 fail_mode: secure
 Port "s1"
  Interface "s1"
   type: internal
 Port "s1-eth4"
  Interface "s1-eth4"
 Port "s1-eth3"
  Interface "s1-eth3"
 Port "s1-eth2"
  Interface "s1-eth2"
 Port ''eth1''
  Interface ''eth1''
```

Listing 14.2: Adding the interface eth1 as a port to bridge s1 in Mininet

To establish a connection between the Zodiac FX[67] and the testbed, certain supplementary procedures are necessary. Initially, it is necessary to establish the connection on the controller port. After connecting the device to a computer using a Universal Serial Bus (USB) cable, the configuration menu can be accessed by establishing a serial connection with the device's `/dev/ttyACM0` port. Subsequently, it is feasible to configure the controller to the appropriate address by adhering to the standard, which designates 10.0.1.8 as the target, while the device's address is 10.0.1.99 within the identical subnet. The port that ONOS listens to for OpenFlow devices must be configured as 6653. Subsequently, the Ethernet connection must be formed by allocating the 10.0.1.8 address to the network interface that the switch is attached to.

To achieve this, we need to execute the following command on the host using the names shown in Figure 14.1: `ifconfig enx00 10.0.1.8/24`. To connect the OvS to the ones in the Mininet[6] topology, you can add the second interface connected to the Zodiac FX[67] by using the command `ovs-vsctl add-port ovs_switch_3 eth1` to an existing switch.

Ultimately, an additional step is required to guarantee that all the intended features are fully supported on the test network. More precisely, the ONOS controller

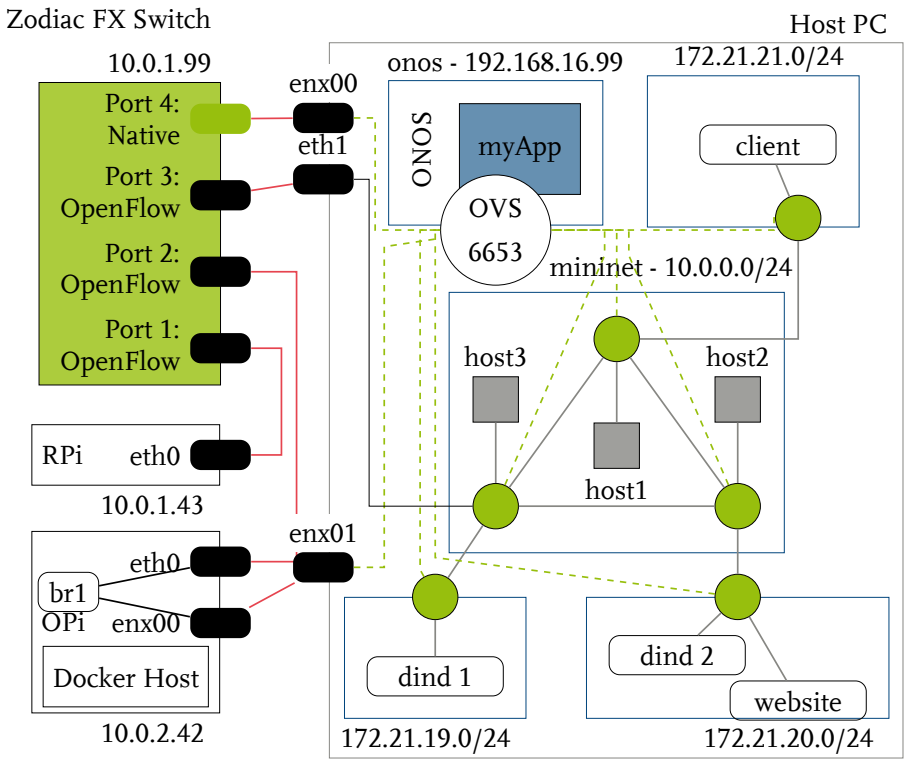


Figure 14.1: Trial test bed for NFV placement via a ONOS plugin to deliver containerized services [74].

must establish a connection with the Docker REST interface available on one of the SBCs, especially the Orange Pi. Enabling the controller to route traffic via the OpenFlow infrastructure would go against the fundamental principles of the SDN paradigm. To ensure the division of responsibilities, the tiny computer was linked to the host using a specific interface. In order to establish a consistent and simple method, an OvS bridge was implemented, utilizing physical NICs as ports. The internal port `br1` is assigned the IP address 10.0.2.42 and can be accessed from both interfaces. The host PC utilizes the `enx01` interface as the parent interface for a Macvlan network that is connected to the ONOS controller.

14.1 Definition of Adjacent Docker Containers

A Docker Compose file can include configuration settings that govern the behavior during the construction process. This has been done to four Docker containers in order to simplify their setup. The Docker client, Docker website, and two DinD[51] containers are described as Dockerfiles and will be utilized for image building if necessary.

The website container is built using the *httpd:alpine* image[69], which runs a basic webserver. The client is built on the Linux Alpine OS, which includes the installation instructions for the package that contains the `dig`[68] command and `curl`[70]. The entrypoint option specifies the location of `/bin/sh`, which enables easy attachment to it through the console. The Linux Alpine image was selected due to its tiny and lightweight characteristics. In its default setup, it has a size ranging from 2 to 3 MB.

The DinD[51] image is more intricate as it must independently transmit DNS packets to the controller in order to register itself as a Docker host in the topology. The image is derived from the official *docker:dind*[51] image, which offers a Docker host during container initialization, making its API accessible over port 2375. The image's entrypoint is situated in the `/usr/local/bin` directory as a shell script. To enable the automatic delivery of these messages, a custom script will be added. This script will send a dig query every 20 seconds. The insertion of this script can be observed in Listing 14.4, specifically in line 2. Additionally, the call to this script is being put in line 5.

```
1 while true; do
2    sleep 20 && dig @172.21.21.21 +time=1 +tries=1 +short docker/
        registerDockerHost;
3 done &
```

Listing 14.3: Registering as Docker host

Listing 14.3 displays the script that pauses for 20 seconds and thereafter dispatches a dig query using the name `docker/registerDockerHost` and 172.21.21.21 as the target IP address. This message will be conveyed to the controller and subsequently scrutinized. The Dockerfile responsible for the DinD image is displayed in Listing 14.4.

```
1 FROM docker:dind
2 ADD digit.sh /usr/digit.sh
3 RUN apk update && apk add bind-tools curl && \
4     chmod +x /usr/digit.sh && \
5     sed -i '/set -e/aexec /usr/digit.sh &' /usr/local/bin/dockerd-
        entrypoint.sh
```

Listing 14.4: Dockerfile of the DinD containers

[69]https://hub.docker.com/_/httpd
[70]https://linux.die.net/man/1/curl

The objective of this trial setup is to develop an SDN controller application capable of establishing connections with Docker hosts and issuing commands to configure containers that offer predetermined virtual network services. Currently, these hosts have a distinct Docker host operating within their individual containers. However, in the future, the deployment of additional services will occur alongside them, operating on the same host system. By connecting them to an existing OvS switch, all the necessary connectivity may be established. This will enable independent addressing of the devices without any conflicts, such as port assignments. This necessitates further configuration exertion, such as the automatic association of the recently generated container to an existing OvS port using the Docker network plugin of Chapter 13. Unfortunately a constraint of this approach exists after the DinD[51] container has been launched, it is presently not feasible to connect it to another virtual bridge without manual intervention. After completing the initial process and establishing connectivity, the new containers can be added to the same switch along with the managing DinD[51] instance. The implementation of this capability involves two primary steps:

- Set up the DinD[51] containers to function solely as a client, rather than a complete host, that connects to the main Docker host.
- Enhance the configuration of new containers in the plugin to accurately define the network setup. This entails utilizing the appropriate network driver of Chapter 13.

14.2 ONOS Plugin to Initiate a Containerized Service

Once the network topology, consisting of various virtual and physical clients, Docker hosts, and webservers, has been established successfully, the network appliances must be programmed by the central controller. ONOS enables the expansion through bespoke applications, utilizing the bundle notion of the underlying Apache Karaf[32] platform. The plugin is implemented using Java, with the class `AppComponent` serving as the entry point.

In Figure 14.2, the class `AppComponent` is positioned at the bottom right, serving as the main class. Subsequently, the connection with all other classes becomes evident. The primary function of ONOS is to register as a component within the Open Services Gateway initiative (OSGi) framework and establish the application's

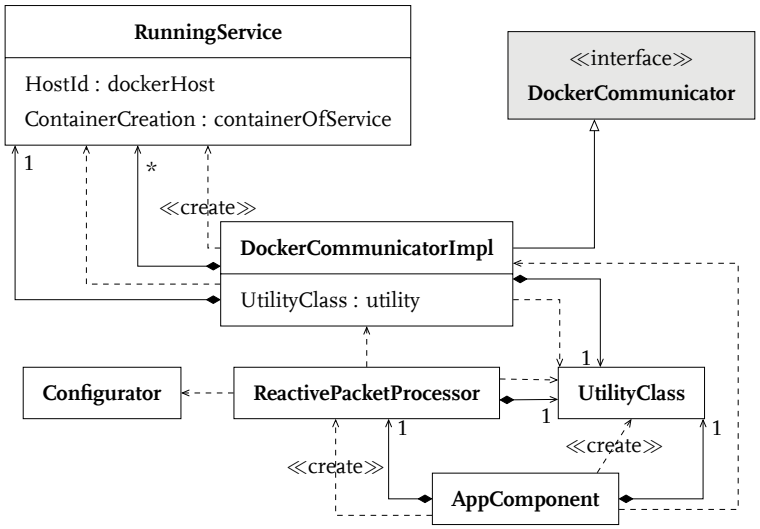


Figure 14.2: Class diagram of the ONOS plugin to deliver containerized services to any registered Docker host [74].

activation and deactivation routines. To activate the app, it is necessary to register it with the core service subsystem. To retrieve an instance of a service, you can declare it as a field and use the annotation `@Reference`, as is the case with all services. The cardinality "MANDATORY_UNARY" signifies that it is necessary to provide one instance in order to fulfill the reference. After all the required services have been mentioned, the instances are transferred to the utility class. This class encompasses many methods that are essential for the project and utilize the references to the subsystems to manipulate the network appliances. The purpose of the configurator class is to facilitate the passing of parameters and information into the application, eliminating the need to hardcode them into the code. The utility class is declared immediately after the `DockerCommunicatorImpl` class, which is responsible for interfacing with the Docker REST interface of Docker hosts in the network. The `RunningService` class contains information about running services on Docker hosts. The main class responsible for the core logic is the `ReactivePacketProcessor`, which implements the `PacketProcessor` interface and is registered as such with the `packetService` subsystem. Subsequently, it captures all the packets that are being transmitted to the controller.

14.3 Recognition of Relevant Packets

The bespoke packet processor must discern network packets that encompass pertinent data and respond accordingly. The implemented interface mandates the utilization of the *process* method, which receives an instance of a *PacketContext*. This method is invoked for each packet that needs to be handled by the controller. The given context allows for the analysis to serve as a foundation for making subsequent decisions. It is essential to determine how to handle these packets, as the switch that transmitted them to the controller did so either based on a specific rule or according to the specified table-miss policy [146]. A significant portion of the packets that the processor must handle are related to locating a particular host in the network and are therefore in the form of ARP. To facilitate the forwarding of those packets and enable the identification of endpoints, they are initially identified and subsequently disseminated, as seen in Listing 14.5.

```
1 if (ethPkt.getEtherType() == Ethernet.TYPE_ARP) {
2  utility.flood(context);
3  return; }
```

Listing 14.5: Identify and flood ARP messages

While it may appear superfluous to enable ARP discovery in an SDN context, it has been demonstrated that network devices utilize the dig tool[68] to transmit packets to the controller, which include customized commands. These messages must be transmitted until they reach the controller, at least until they encounter the first switch. The most straightforward method to ensure the success of this proof-of-concept is to forward messages of this protocol, unless it has been replaced by another tool or technology.

An alternative to the instruction-based intent setup could be the automatic setup that occurs with the first packet of a communication flow. Bianco *et al.* [15] examined the scalability characteristics of the *fwd* and *ifwd* applications of ONOS by evaluating the control traffic they generated. The distinction between these two rests in their respective methods of routing packets across the network. The *fwd* application utilizes traditional and unrestricted ARP flow, whereas *ifwd* takes a shortcut by promptly establishing an intent between the source and destination as the controller becomes aware of the destination. This allows for forwarding at a MAC level. This greatly reduces the amount of controller traffic, but it is not

appropriate for this particular proof-of-concept. The installation of intents is only intended for situations where it is explicitly requested, and intents should not be automatically set up for every communication attempt.

To detect custom issued commands transmitted using the DNS protocol, one can scan the unprocessed packet for a particular sequence of bytes. To accomplish this, one must initially obtain the pertinent messages as arrays of bytes using the standard String function *getBytes*. To ensure that this does not become limited to the default character set of the system, UTF-8 is established as the standard. To establish a fundamental naming convention, the issued instructions will first denote their purpose by commencing with either the term “traffic” or “docker”. The concrete command for the controller follows, separated by a forward slash.

14.4 Intent Setup

The *traffic* commands have the ability to either initiate an intent configuration or terminate it. The command formats are as follows: *traffic/allowTraffic* or *traffic/stopTraffic*. In a `HostToHostIntent`, the first participant is the initiator of the communication intent, while the second participant is the host with the IP address that the DNS query is directed to. This class exemplifies the implementation of the builder design pattern, which allows for the avoidance of an excessive number of parameters during instantiation. Once a message of this nature has been discovered, the condensed process outlined in Listing 14.6 follows. The initial attribute that needs to be configured is the ID, which designates the program accountable for establishing the particular purpose. Subsequently, it is necessary to provide a key for subsequent referencing. The method in the sample application provided by the ONF has assumed control of the key algorithm. The source and destination host IDs are compared, typically consisting of their respective MAC addresses, arranged in ascending order. This guarantees the ability to replicate results, even while the key is being acquired from the second party involved in the communication. The two *HostId*s must be supplied after the key, with the source being specified first and the destination being specified second. Subsequently, it is necessary to provide a selector and a treatment for the intent. For regular forwarding, the *DefaultTrafficSelector* and the *emptyTreatment* are sufficient. Prior to presenting it to the responsible subsystem of the controller, a backchannel will be established.

```
HostToHostIntent hostIntent = HostToHostIntent.builder()
 .appId(appId)
 .key(key)
 .one(srcId)
 .two(dstId)
 .selector(selector.build())
 .treatment(treatment)
 .build();
openBackChannel(context);
intentService.submit(hostIntent);
```

Listing 14.6: Setup of an intent

This is crucial because after the intention is filed and all the required flows are installed in the switches that connect the source and destination hosts, `dig` calls between them will be forwarded and executed without any complications. These packets will be unable to reach the controller, resulting in the cessation of command transmission. The backchannel guarantees that these packets are not seen as regular traffic. To effectively manage network traffic, it is sufficient to install Flow rules in the neighboring network devices. However, it is crucial to limit the selector for the packets to the greatest extent feasible. Otherwise, the controller will encounter an excessive influx of new traffic. Typically, while using the dig command, packets are delivered using the UDP to port 53. While other protocols such as TCP or port 853 can be used, it is adequate to concentrate on UDP and port 53 in our architecture. The selector must traverse the entire IP stack in ascending order of specificity, starting from IPv4 and ending with UDP, as shown in Listing 14.7. Furthermore, the specific port to which the device is attached and the MAC address of the host will be utilized to provide more precise criteria for determining a match.

```
//Match UDP Packets with destination port 53
selector.matchEthType(Ethernet.TYPE_IPV4);
selector.matchIPProtocol(IPv4.PROTOCOL_UDP);
selector.matchUdpDst(TpPort.tpPort(53));
//Match port of incoming packet
selector.matchInPort(context.inPacket().receivedFrom().port());
selector.matchEthSrc(context.inPacket().parsed().getSourceMAC());
```

Listing 14.7: Selector for backchannel

The configuration of the first Flow is rather simple, as all the required information is readily available in the *PacketContext*. The selector and treatment have been specified for the source device. A device, which is a switch that supports OpenFlow, can be acquired via the `DeviceService`. The priority of the device is set to 40001, which is one unit higher than the priority of the intent. In the second flow, it is necessary to initially acquire the information regarding the device and the port to which it is connected, in order to reach the destination host. The `HostService` and the `DeviceService` furnish all essential information to guarantee that both hosts can transmit additional instructions to the controller. This can be accomplished by revoking the intents through the execution of the command *traffic/stopTraffic*. To acquire an intent, one can utilize the *IntentService* which is capable of retrieving a particular intent by utilizing its key. To withdraw it, simply invoke the *.withdraw()* function of the *Intent* object.

14.5 Controlling of Docker Hosts

In addition to the traffic commands that initiate and terminate intents, commands related to Docker have also been incorporated. The naming convention "docker/..." will be employed to initiate VNFs in the network topology. There are presently two supported commands:

dockerInfo This command is intended to assess the capability of selecting, connecting, and coordinating the docker hosts within the network. When a host sends a message with this keyword, the controller will choose a host and ask for specific details about that host. This information will be recorded in the console, together with the IP address, in order to evaluate the efficiency of the underlying procedures.

LaunchWebsite The second command will initiate the configuration of a basic demonstration web server on the nearest host in relation to the requester. The website will be launched within a Docker container, with port 80 exposed and an intention set up to facilitate seamless communication. If this webserver has already been initiated somewhere on the network, it will simply establish an intention to the appropriate Docker host. The function "registerDockerHost" is being referenced. Receiving this particular message from a host within the network indicates that a Docker host is actively execut-

ing and accessible at this address. The data is being kept in a *ConcurrentHashMap<HostId, IpAddress>*, ensuring that several threads can interact with it safely. The REST API will be exposed via the normal Docker port, 2375, as per tradition (over configuration). In order to ensure future adaptability, the configuration class has already been implemented. It is also possible to consider that the `registerDockerHost` message may include the appropriate port, e.g., `docker/registerDockerHost/2375`.

```
try (DockerClient client = DefaultDockerClient.builder().uri("http
    ://" + closestDockerHostAddress.toString() + ":2375").build()){
 if (!"OK".equals(client.ping())) {
  dockerHosts.remove(closestDockerHost);
  closestDockerHost = getClosestDockerHost(srcId);
 }
}
```

Listing 14.8: Ping Docker host

The information retrieval method from a Docker host can be conveniently invoked on the instance of the *DockerClient*, similar to the ping method shown in Listing 14.8. The outcome is being retrieved as a String and displayed on the terminal. This was primarily implemented to initiate the selection process and evaluate the functionality as anticipated.

Lastly, there is the capability to initiate a sample webserver on the nearest Docker host. Initially, the procedure verifies if this service is already existing in the network. A `ConcurrentHashMap<String, RunningService>` is used to store the information of all currently running services. The key is essentially a designation, which may eventually be either the name or ID utilized by the hosts to solicit the service within the network. The value is a bespoke Plain Old Java Object (POJO) that includes details about the Docker host it is operating on, as well as an instance of `ContainerCreation`. The class is returned promptly after the creation of a container and mostly contains essential identifying information. If the service is operating, it is enough to build an agreement between the requester and the Docker host where the service is hosted. If the service is not currently operational, some supplementary measures must be taken. Initially, as is customary, it is imperative to identify an appropriate and accessible host. Subsequently, the host must retrieve the Docker image necessary for the function to operate effectively. The *pullImage*

function initiates a connection with the designated host and retrieves the image with the provided name from a Docker registry. Prior to doing so, it verifies if the client is accessible. Afterwards, a container is being instantiated with supplementary configuration details, enclosed within a *ContainerConfig* object as seen in Listing 14.9. To set up a basic web server, we just need to specify that the container should make port 80 accessible and utilize the image that was downloaded earlier as the starting point.

```
ContainerConfig config = ContainerConfig.builder()
 .hostConfig(HostConfig.builder()
 .portBindings(ImmutableMap.of("80/tcp", Arrays.asList(PortBinding.
    of("", "80")))).build())
 .image("unibaktr/alpine:whoami")
 .exposedPorts("80/tcp")
 .build();
```

Listing 14.9: Configuration for the creation of the webserver container

Once the container is created, it can be submitted to the Docker host for execution. The pertinent data is saved within a *RunningService* entity and ultimately, after all components are activated, an intent is established to guarantee connectivity and communication.

This developed approach enables the development of middleware to orchestrate services in SDN environments, which are dispersed across several heterogeneous nodes, particularly in a Cloud-to-Fog-Continuum context. A monitoring system is essential for every orchestration, which we are evaluating and implementing in the subsequent phases.

Chapter 15
Cloudless Resource Monitoring

We chose ARM-compatible data center orchestration engines in contrast to CC monitoring solutions like OpenStack[71], Pacemaker[72], and Marathon[73], as our monitoring environment. After assessing four potential choices, we determined that both Mesos[74] and Nomad[75] are either excessively intricate or have limited compatibility with ARM platforms. Therefore, we opted for Kubernetes, which has support for ARM, as well as Docker's Swarm mode. Docker Swarm (see Section 10.1) and Kubernetes (see Section 10.2) have emerged as two viable competitors in the realm of small platform high-availability solutions. Assessing the performance of these two technologies and comparing them based on their fundamental resource usage is intriguing.

For our trials, we utilized Docker for conducting measurements of Docker Swarm and Kubernetes. The hardware foundation was supplied by PicoCluster Ltd.[76], consisting of a stack of three RPis. In addition, we included a fourth RPi that serves as the master node in our experiments. We installed Hypriot OS[77] on each RPi and connected all devices using an Asus RT-AC68U[78], which offers a switching fabric with a capacity of $1^{Gbit}/_{s}$. We enable Monit[7] to operate directly on all RPis, allowing for seamless switching between Kubernetes and Docker Swarm

[71]https://wiki.openstack.org/wiki/Operations/Monitoring
[72]https://clusterlabs.org/projects/pacemaker/
[73]https://github.com/d2iq-archive/marathon
[74]https://mesos.apache.org
[75]https://www.nomadproject.io
[76]https://www.picocluster.com
[77]https://blog.hypriot.com/downloads
[78]https://www.asus.com/de/networking-iot-servers/wifi-routers/asus-wifi-routers/rtac68u

without interrupting the data gathering process. The following table describes the resource consumption under the following three scenarios:

Scenario 1: Base utilization, where only Monit and Docker daemon are running.

Scenario 2: After starting Docker Swarm Mode.

Scenario 3: After starting Kubernetes.

Scenario	CPU usage [%]	Memory usage [%]	Process CPU Docker [%]	Process memory Docker [MB]
1	0.44	4.21	0.05	2.47
2	1.43 / 0.83	7.87 / 7.4	1.1 / 0.44	57.45 MB / 2.42 MB
3	14.74 / 2.01	34.21 / 12.68	5.76 / 1.31	479.51 / 160.77

Table 15.1: Results of resource consumption measurements of Docker Swarm vs. Kubernetes. If there are two data points in the cell, the first denotes the manager node, the second the worker node.

As we can see in Table 15.1, Kubernetes necessitates significantly greater resources than Docker in Swarm Mode. According to the table, Docker's CPU utilization is lower than that of Kubernetes. The results for memory utilization are analogous; Kubernetes allocates more RAM than Docker operating in Swarm Mode, etc.. However, this comparison is somewhat inequitable. Kubernetes functionalities are significantly more advanced and customizable than those of Docker Swarm Mode. Kubernetes is significantly more feature-rich than Docker, as it inherently includes monitoring services, centralized logging, and a dashboard that enables cluster management primarily through a graphical user interface. Disabling add-ons that are enabled by default is not feasible. Disabling them by difficult methods is likely hazardous to the cluster's stability; hence, we refrained from disabling any of Kubernetes' default services.

Based on the results of Table 15.1, we evaluate and develop existing monitoring systems on Docker Swarm, due to it's lower footprint on SBCs.

15.1 Available Monitoring Frameworks for Single Board Computer Cluster

First we will show the functionalities of the PyMon software stack in Subsection 15.1.1 and the Prometheus stack in Subsection 15.1.2 to be compared against each other in Section 15.2.

15.1.1 The Client Server Monitoring Approach PyMon

PyMon is predicated on an improved version of Monit[7] for client-side host monitoring and django-monit-collector[79] (hence referred to as *Monitcollector*) for data acquisition and visualization on the server side.

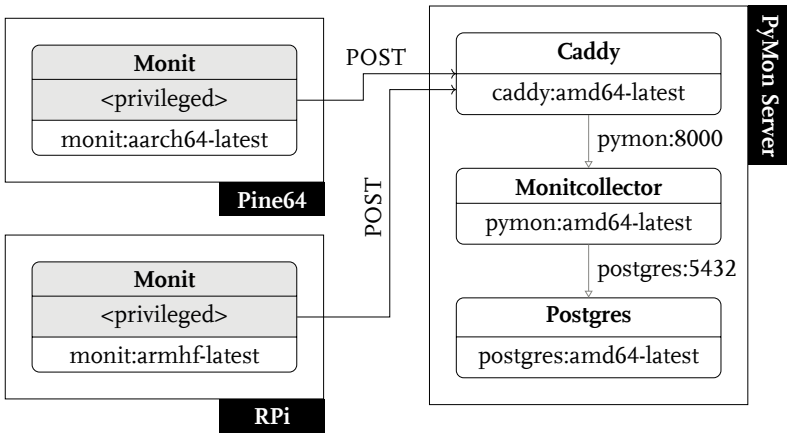


Figure 15.1: Modular architecture of the PyMon framework [cf. 75].

Monit[7] is a compact open-source utility designed for monitoring UNIX-based systems. The primary objective is the monitoring of processes, systems, networks, and files. Monit[7], by default, lacks capabilities for monitoring Docker containers. Consequently, the version incorporated in PyMon has been augmented with an additional script to get container statistics via Docker's HTTP API. The aggregated metrics for CPU and RAM utilization per container, together with additional details such as the container name and status, are made available at a basic HTTP endpoint.

[79] https://github.com/nleng/django-monit-collector

Monitcollector is built on the Python Programming Language (Python) web framework Django[80]. It acquires monitoring metrics from Monit instances and archives them in a PostgreSQL[81] database or, for testing purposes, in a SQLite[82] database. The aggregated data can be presented via a web interface, providing a comprehensive overview for each Monit instance, featuring zoomable timeline graphs for CPU, RAM, and network utilization [75].

15.1.2 The Monitoring Stack including the Prometheus Database

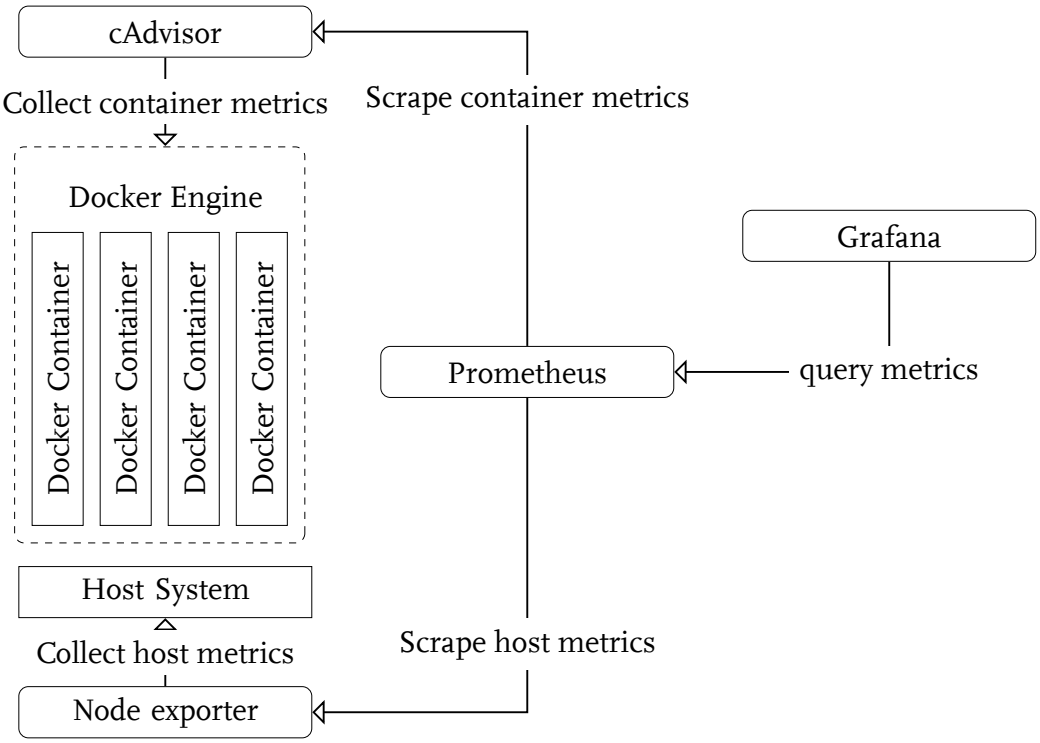


Figure 15.2: Data collection stack around the Prometheus database.

The Prometheus[83] stack is a system monitoring environment centered on the Prometheus[83] server, which integrates with several external open-source components such as metric exporters and dashboards.

The Prometheus developers offer the utility *Node exporter*[84] for system monitoring, which gathers metrics on UNIX-based systems. Supplementary software is necessary for container monitoring. CAdvisor[8], created by Google, is a tool that

[80]https://www.djangoproject.com
[81]https://www.postgresql.org
[82]https://www.sqlite.org
[83]https://prometheus.io
[84]https://github.com/prometheus/node_exporter

offers insights on the resource utilization of active containers on a single machine. Ultimately, as Prometheus offers merely a rudimentary online interface for testing and maintenance, a more comprehensive GUI is required. The creators of Prometheus consequently endorse the dashboard application *Grafana*[85].

15.2 Performance Investigations on PyMon and Prometheus

The performance assessment of the two rival frameworks was executed on a cluster of ARM SBCs. The cluster comprises three RPi 3 devices, coupled to a consumer-grade router with Gigabit Ethernet capability. The RPi 3 is a highly regarded, cost-effective, energy-efficient, and adaptable SBC. It is powered by a 64-bit ARM quad-core CPU, enabling its use as an edge-located micro cloud to provide a substantial array of services, as demonstrated by Elkhatib *et al.* [49].

We utilize Hypriot OS[77] for our evaluation, a Debian-based OS specifically designed for ARM devices, characterized by a compact image size, preinstalled Docker, and a Linux Kernel tuned for Docker. PyMon and the Prometheus stack are deployed on the RPi cluster operating in Docker Swarm mode.

15.2.1 Measurement Approach

To assess the resource overhead and consumption generated by both frameworks, we choose to utilize Docker's built-in statistics tool, which continuously monitors and displays the CPU and RAM utilization, as well as the network and disk I/O for each container.

Upon selecting the measurement tool, two inquiries emerge: First, which metrics are essential for collection, and second, how can we ensure the reliability of the measurements conducted?

The response to the initial inquiry is quite clear-cut. FC aims to address potential latency difficulties associated with certain IoT applications. Merely decreasing the geographical distance between the IoT device and the associated computational back-end system is insufficient to accomplish this. The fog node must do the necessary computations more rapidly than the cloud, accounting for edge-to-cloud network latency. Therefore, it is essential to minimize the CPU overhead caused by the monitoring framework. In addition to CPU utilization, Elkhatib *et al.* [49]

[85] https://grafana.com

also regards RAM as a limited resource crucial for various service. An example of a FC use case that necessitates sufficient available memory is content caching to enhance video streaming efficiency [31]. In summary, we concentrate on assessing the CPU and RAM overhead introduced by the monitoring frameworks in our study.

The Docker statistics output from the Linux terminal is recorded into a csv file for a length of 300 seconds to ensure data reliability.

To assess the CPU utilization of the Prometheus stack and PyMon, both must execute identical code throughout each scrape/poll cycle or UI request, leading to minimal fluctuations in average CPU usage. Conversely, RAM use typically escalates over time when a software allocates extra memory to retain newly computed data. Applying the aforementioned approach to RAM utilization would yield a snapshot of the current memory load with minimal significance. Consequently, the methodology for assessing RAM utilization varies: RAM usage is measured over time for each scenario. Multiple terminal sessions were simultaneously initiated with varying sleep periods to reduce measuring effort and establish a uniform start time.

15.2.2 Performance Comparison

Prior to the presentation of the performance measurement data, it is essential to underscore some pertinent observations concerning them and the conditions under which they were acquired:

Monitoring focus: This comparison examines the resource use of the monitoring programs Monit[7], Node Exporter[84], and cAdvisor[8], which gather host and container metrics on a single node.

Platform limitations: All tests are performed on a RPi cluster of three devices.

CPU usage: The calculation of Docker statistics for CPU utilization does not consider the number of CPU cores. Therefore, the greatest achievable usage is consistently 400% (for the four cores of a RPi 3).

Deployment limitation: To ensure uniform circumstances for both frameworks, the Prometheus[83] server and Grafana[85] were deployed on a single node. This deployment option is improbable, resulting in significant congestion due to the high resource requirements of both apps.

Measurement timeouts: Timeouts of Docker's statistics API are a consequence of this congestion.

The measurements for both CPU and RAM utilization are categorized into three segments depending on the distinct processing stages of a monitoring framework:

Monitoring (Node exporter, cAdvisor and Monit): The aggregation of host and container metrics and the dissemination or transmission of this data.

Aggregation (Prometheus and Monitcollector): The procedure of gathering, consolidating, storing, and disseminating the metrics acquired from the prior monitoring phase.

Visualization (Grafana and Monitcollector): The graphical depiction of metrics via a dashboard.

Evaluation of CPU utilization.

The initial measurements assess the impact of a reduced scrape/poll interval on the monitoring program. This is also designed as a simple test performed on a worker node that exclusively operates Node Exporter, cAdvisor, or Monit. In the case of Monit, an extra cAdvisor container is deployed as a placeholder to guarantee that both frameworks are evaluated under identical conditions (two containers in total on each worker node). The results in Figure 15.3 (I) indicate a substantial disparity in the CPU consumption between the two frameworks. Node exporter averages a consumption of 1.45% with the default interval of 15 seconds, whereas cAdvisor averages 12.51%. However, Monit has already utilized 28.69%. The interval reduction to 5 seconds significantly exacerbates the distance between the Prometheus stack components and Monit.

The second series of observations illustrated in Figure 15.3 (IV) demonstrates the effect of extra containers on worker nodes. This situation is more practical, as a fog node is designed to host numerous containerized apps at the network edge. For testing purposes, such apps are emulated using a variable quantity of

a *Alpine* Linux container, which intermittently executes `ping docker.com` in a terminal. Node exporter is excluded from consideration as it lacks container monitoring capabilities. The scrape/poll interval remains set to the default value of 15 seconds. As the number of operating Alpine containers increases, cAdvisor's CPU utilization correspondingly escalates. With a higher quantity of operating Alpines, the disparity in utilization between cAdvisor and Monit diminishes; still, only cAdvisor can monitor 32 containers. The measurements from Monit reveal approximately 30% timeouts, suggesting a potential bottleneck in either Monit or the script utilized to gather container statistics via Docker's HTTP API.

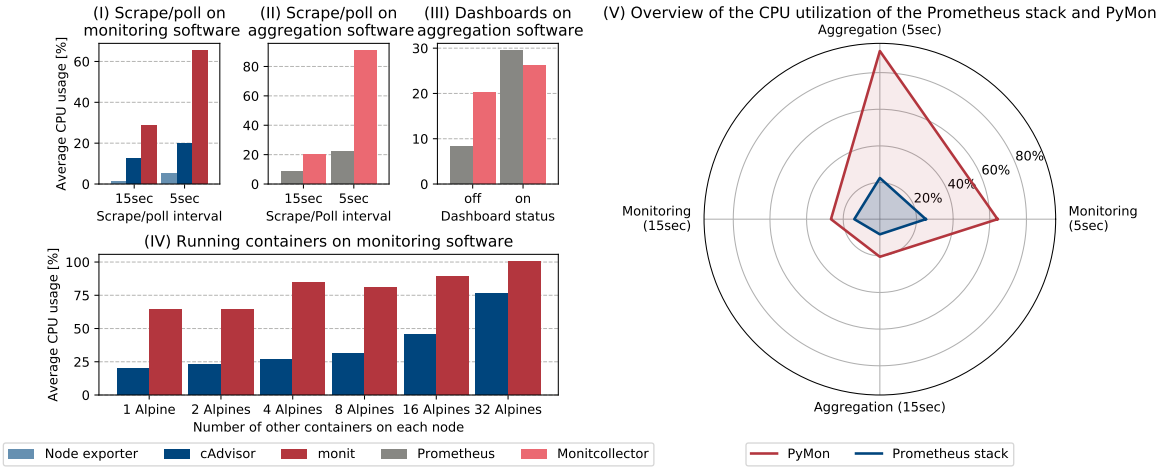


Figure 15.3: Comparison of CPU utilization of the monitoring systems PyMon and Prometheus. In (1) - (3) the CPU utilization is described for the different monitoring layers. In (IV) we compare the CPU usage of the monitoring services for a various number of running containers. (V) shows an overall comparison of both monitoring stacks [75].

In summary, cAdvisor's CPU overhead is minimal and grows effectively with an increasing number of containers. The current version of Monit incurs significant overhead, permitting the monitoring of only 20 to 30 containers on a single RPi 3.

The aggregation of metrics is illustrated in Figure 15.3 (II), which shows the average CPU utilization of Prometheus and Monitcollector, together with the effect of a decreased scrape/poll interval. During the collection of metrics from two nodes at 15-second intervals, and in the absence of any requests or inquiries from a user interface, both frameworks utilize a minimal portion of the available CPU resources. Reducing the scrape/poll period to 5 seconds considerably elevates Monitcollector's utilization to above 90%, indicating an additional bottleneck.

After setting the baseline CPU consumption of Prometheus and Monitcollector, the subsequent step is to investigate the impact of database queries generated by the visualization software. The measurement results in Figure 15.3 (III) indicate a minimal increase of around 6% for Monitcollector and around 20% for Prometheus. In the case of Prometheus, the actual load in this scenario fluctuates based on the quantity of metric queries executed by the currently displayed Grafana dashboard and the frequency of its refresh.

The measurements for the visualization component, specifically Grafana and Monitcollector's user interface, are excluded from this comparison because to the previously stated platform constraints. For the purpose of thoroughness: Grafana utilizes approximately 72% CPU on average with a refresh rate of 10 seconds. Furthermore, obtaining precise utilization values for Monitcollector is entirely unfeasible because to the software's lack of distinction between aggregate and the user interface.

In conclusion, the CPU consumption analysis is encapsulated in Figure 15.3 (V), which highlights the key performance metrics for monitoring and aggregation. This picture illustrates the detrimental effect of a decreased polling interval on PyMon in comparison to the Prometheus stack. A 15-second interval may suffice for long-term performance assessment, but short-term load balancing for services necessitates more current data. Moreover, PyMon encounters difficulties when the quantity of containers it must oversee rises. Overall, the Prometheus stack utilizes fewer CPU resources, particularly with its primary function of container monitoring on SBCs. The Prometheus stack effectively scales with a growing number of containers to monitor and can manage brief scrape intervals.

Evaluation of RAM utilization.

The assessment of RAM utilization consistently employs a scrape/poll interval of 5 seconds, as its total effect is negligible in contrast to CPU consumption. Another distinction is the altered measuring methodology: The RAM utilization is assessed against individual measurement intervals spanning from 1 minute post-container initiation to 12 hours.

The initial set of measurements presented in Figure 15.4 (I) contrasts the RAM utilization of the **monitoring** software on the manager node. The usage of Monit and Node exporter remains minimal overall, with only a tiny increase over time. In contrast, cAdvisor necessitates greater RAM and exhibits a distinct rise during

the initial three hours, subsequently declining between three to six hours. Monit demonstrates a distinct advantage in this context, utilizing only $13MB$ on average, whereas Node Exporter and cAdvisor collectively average around $50MB$ to $60MB$.

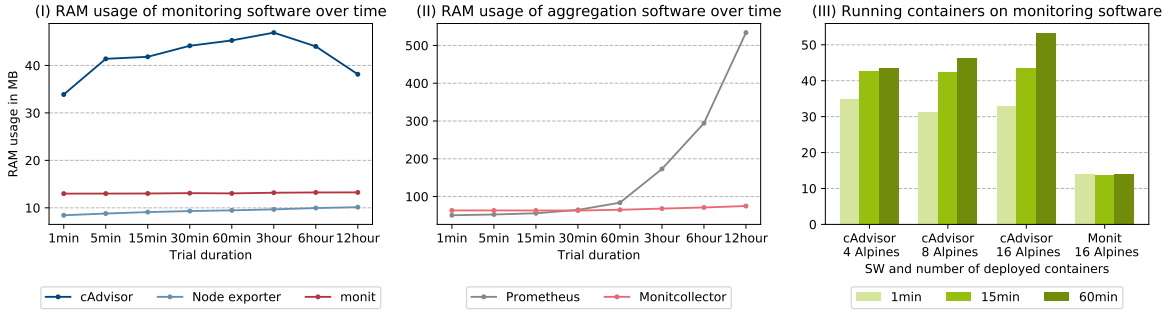


Figure 15.4: Different influence factors on the RAM consumption of our evaluated monitoring frameworks. In (I) the usage of the monitoring services is evaluated over time, while (II) shows the utilization of the aggregation services. In (III) different numbers of containers are running and compared over time on the monitoring level [77].

The prior findings were obtained on a manager node with only the necessary containers operational. A subsequent test was performed to assess the effect of the extra deployed containers, utilizing the established Alpine dummies. Figure Figure 15.4 (III) presents the principal findings. The effect on Monit is rather minimal. Despite the inclusion of 16 Alpine containers, the use only rises by around $1MB$ relative to the baseline measurements depicted in Figure 15.4 (I).

The variance for the one-hour measurement period is likewise insignificant. Conversely, cAdvisor's RAM requirements escalate in accordance with the quantity of active Alpine containers. After 1 hour, the usage escalates from approximately 43 MB with 4 additional containers to roughly 53 MB with 16 containers.

Ultimately, Figure 15.4 (II) demonstrates the application of Prometheus and Monitcollector (**aggregation**). The RAM use of Monitcollector experiences a modest rise during 12 hours, rising from approximately 63 MB to approximately 75 MB. Upon initial container startup, Prometheus exhibits lower memory use than Monitcollector; however, after 30 minutes, the usage begins to rise significantly. After 12 hours of metrics collection, Prometheus requires over 500 MB of RAM, whereas Monitcollector necessitates less than 75 MB.

In terms of RAM utilization, PyMon has a distinct advantage initially. Monit and Monitcollector utilize less RAM than their Prometheus stack equivalents. Nonetheless, these findings are deceptive. Both cAdvisor and Prometheus utilize metric caching. CAdvisor offers a runtime option to decrease the length of short-term metrics caching. The Prometheus manual for version 1.8 advises a minimum of 3 GB of RAM when utilizing the default configurations. The RPi 3 is equipped with only 1 GB of RAM and hence necessitates additional optimization for sustained operation. Nonetheless, the anticipated Prometheus version 2.0 is projected to significantly reduce RAM consumption [133].

Evaluation of the network overhead.

In addition to these two metrics deemed most significant in the case of FC with tiny SBC, which may be associated with a speed and data volume-constrained mobile communication network, the resultant network overhead must be determined. Table 15.2 presents a comparison of incoming and outgoing network traffic between PyMon and the Prometheus stack. The data was obtained from Docker's statistics API after 60 minutes of uptime, with 16 Alpine containers operational on each worker node. The findings indicate that Monit generates approximately $8MB$ of network traffic to oversee the 16 containers and itself, while Node Exporter produces only $4.6MB$, and cAdvisor over $60MB$ under same conditions. The data is transmitted to Monitcollector and Prometheus, with Monitcollector receiving a total of 20.6 MB, and Prometheus manages 154 MB.

Name	**Network I/O (MB)**
Monit	5.5/8.1
Node exporter	0.4/4.6
cAdvisor	1.5/59.9
Monitcollector	20.6/11.5
Prometheus	154.0/4.8

Table 15.2: Comparison of incoming and outgoing network traffic for the different modules of the evaluated monitoring solutions [77]

The significant disparity in the outgoing traffic of Node Exporter and cAdvisor cannot be attributed only to the quantity of metric samples they provide, as Node Exporter has a baseline of 448 samples, but cAdvisor has 332, assuming no additional containers are operational. The inclusion of 16 Alpine containers results in

a substantial rise in the number of samples to 1543. Nonetheless, this quantity of samples fails to elucidate the significant disparity of approximately 13.

15.2.3 Feature and Qualitative Metrics Comparison

A choice should not rely exclusively on quantitative performance indicators but should also include qualitative metrics.

One of the most critical elements of a monitoring framework is the quantity of metrics offered. The standard exporters for Prometheus, cAdvisor, and Node exporter provide a substantial array of diverse metrics. Node exporter may examine practically every conceivable feature of UNIX systems, many of which are likely unnecessary for performance monitoring. Conversely, PyMon adheres to the principle of data reduction, supplying only the essential metrics necessary for its function.

In addition to gathering and preserving those measurements, their display is also essential. Grafana's user interface is engineered to accommodate various dashboard styles and data sources to meet diverse monitoring objectives. Moreover, Grafana accommodates several concurrent users and enables them to customize the dashboard according to their specific requirements by utilizing unique time frames and a straightforward drag-and-drop functionality for repositioning graphs. Conversely, PyMon is developed for a singular objective and has minimal modification possibilities, limited to a fundamental zoom feature for graphs.

The Prometheus stack exhibits a distinct benefit in overall adaptability owing to its loose coupling of separate software components, facilitating the addition, removal, or replacement of framework elements. Additionally, the Prometheus server is compatible with numerous exporters, such as those used to gather metrics from web servers or databases. It is advisable to store the metrics in an external database for long-term retention. PyMon exclusively utilizes PostgreSQL as its default external database and does not allow additional data sources.

Software compatibility and updates are other areas of concern. Concerning compatibility, only PyMon provides multi-architecture Docker images to facilitate seamless deployment in environments with varying CPU architectures. Although the Prometheus stack components undergo consistent updates for bug fixes, enhancements, and new features, only Monit is maintained in the case of PyMon.

Ultimately, the quantitative performance metrics have shown a significant limitation: PyMon exhibits poor scalability with an increasing number of containers to monitor and/or a diminished polling period. Elkhatib *et al.* [49] demonstrated that a single RPi can operate a minimum of 60 containerized *httpd* web servers. Experiencing a limitation of fewer than 30 containers due to excessive CPU overhead from Monit, which results in the loss of its measurements, should be seen as a significant disadvantage.

Last but not least, both frameworks need to be adjusted to fit into an SDN Cloud-to-Fog-Continuum environment, where we want to monitor services. Due to that fact, we need to implement a new solution, which is feasible for a cloudless deployment and capable of monitoring nodes within a SDN network in the next step.

15.3 Cloudless Resource Monitoring Foundations

Given that OpenFlow and SDN have fundamentally altered the networking landscape, these advancements represent the most cutting-edge technology; however, researchers have encountered numerous challenges and difficulties during the development and testing of prototypes for new functionality integrated into an SDN controller. Mininet[6] is a network emulation tool specifically designed for testing SDN functionalities. It provides a solution for developers and testers seeking to circumvent dependence on large and complex testbed infrastructures. Nonetheless, the default host in Mininet[6] offers just fundamental functionalities. Consequently, this constraint renders Mininet[6] inefficient to emulate networks for increasingly complex applications. Addressing this challenge, Peuster *et al.* [130] developed an iteration of Mininet called Containernet[86]. Containernet[86] facilitates the operation of more resilient and expansive Docker containers to function as hosts within a simulated environment.

It offers benefits like the deployment of fully operational OS containers as testbed hosts, permitting testers to effortlessly incorporate supplementary runtime configurations, including constraints on CPU, RAM, storage, and I/O, thereby enabling a more precise emulation of physical machines.

However, the proposed approach requires a more sophisticated emulated host that accurately replicates a physical system with all features, including the capability to execute several applications or daemons. The DinD image has been altered to

[86]https://github.com/containernet/containernet

the `ubuntu:18.04` OS image during the initial construction process to guarantee compatibility with Containernet[86] and mitigate potential difficulties. This modification resolves the problem of installing Python packages within an Alpine base image, because it uses a different C-library called `musl`[87]. In the second part of the initialization process, Containernet[86] installs supplementary packages, including `net-tools`[88], `iproute2`[89], and `iputils-ping`[90].

15.4 Architecture of the Cloudless Resource Monitoring System

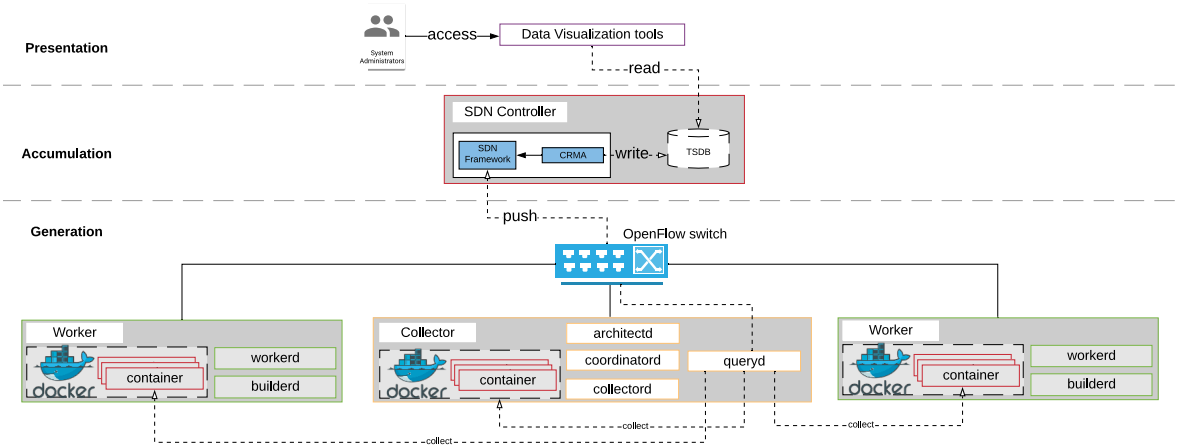


Figure 15.5: Blueprint of the CRM system [108]

Now we will present and discuss the general architecture of the proposed system, which is a CRM that inter-works with SDN. The visual representation of Figure 15.5, albeit simplified, accurately depicts all the components in the system without diminishing or distorting the importance of any of them.

The complete system will be elucidated in a sequential manner, starting at the lowest level and progressing upwards, in accordance with the flow of information inside our proposed system. From a logical perspective, there are three distinct layers: **Generation**, **Accumulation**, and **Presentation**.

[87] https://musl.libc.org/about.html
[88] https://packages.debian.org/de/sid/net-tools
[89] https://packages.debian.org/de/sid/iproute2
[90] https://packages.debian.org/de/sid/iputils-ping

15.4.1 CRM Data Generation Layer

The Collector and Worker nodes, which provide the principal computational capacity of the CRM system, are situated in this bottom layer. Their principal functions, as well as their internal components, are delineated as follows:

- The basic functions of a Collector node entails identifying potential Workers, managing them, and relaying their resource consumption to the higher tier. The execution of these tasks is facilitated by the deployment of four daemons, specifically `coordinatord`, `architectured`, `collectord`, and `queryd`. Each background process has designated responsibilities: The coordinator oversees communication between the Collector and various Workers through customized messages and synchronizes the Collector's workflow by utilizing multiple intra- and inter-process communication channels (i.e., queues). The architecture alters the registered Workers remotely by calling their specific API endpoints. The Collector and `queryd` collaboratively assess several machine resources (e.g., CPU, disk, RAM utilization), produce relevant reports, and transmit them to upstream specialized applications for further utilization. In order to initialize those complex daemons, the Collector employs a *config.yml* file to delineate the required parameter, including as port numbers, API endpoints, and Docker options).

- The Worker is explicitly engineered to do complex and resource-demanding tasks. The system comprises two daemons: `workerd` and `builderd`. Similar to the collector, each daemon has designated responsibilities. `Workerd` communicates with `coordinatord` and registers itself with the monitoring system. Conversely, `builderd` processes the HTTP requests generated by the `architectured` daemon of the Collector. Subsequently, it executes various operations to modify the local Docker environment. The Worker utilizes a *config.yml* file to maintain the essential configurations for its two principal daemons.

15.4.2 CRM Data Accumulation Layer

Two services in this layer, *crma* and *influxdb*, execute the **Monitoring** functionality. The primary engine is "crma", a Java program specifically engineered to integrate itself into the ONOS application subsystem as a plugin. The system's aims are explicit and defined: to intercept certain packets transmitted from the generation layer, modify the payload data if necessary, and archive them in a database for subsequent applications, like data analysis or visualization. The second service is a TSDB, as the proposed system predominantly uses data to track variations in resource utilization across several compute nodes over time. In this context, a TSDB is more pertinent than a relational database. Multiple TSDB options, such as InfluxDB[91], TimescaleDB[92], and QuestDB[93], demonstrate significant potential. Following extensive evaluations, InfluxDB[91], the preeminent leader in TSDB technology, has been chosen as the preferred option due to its capacity to handle high data velocity. Moreover, InfluxDB[91] provides a unique query language known as InfluxQL[94], allowing users to efficiently and accurately query, extract, and aggregate valuable information.

15.4.3 The Presentation Layer

This layer, positioned on top of the CRM system, provides a graphical depiction of resource utilization data. System administrators can swiftly identify and analyze real-time trends, patterns, and anomalies in the underlying system through visual features such as charts or graphs. Numerous open-source data visualization tools are available: Kibana[95], Grafana,[85] and Graphite[96]. Upon thorough evaluation of the advantages and disadvantages of the tools, Grafana[85] ranks best. It enables users to create detailed dashboards using various visual components (e.g., line graphs, tables, singlestats) together with several customization possibilities, demonstrating great versatility. Moreover, Grafana[85] is compatible with other database systems, facilitated by integrated data-source plugins, including InfluxDB[91], PostgreSQL[81], Elasticsearch[97], and Prometheus[83], among others.

[91]https://www.influxdata.com/get-influxdb
[92]https://www.timescale.com
[93]https://questdb.com
[94]https://docs.influxdata.com/influxdb/v1/query_language
[95]https://www.elastic.co/de/kibana
[96]https://graphiteapp.org
[97]https://www.elastic.co/de/elasticsearch

15.5 Evaluation of Cloudless Resource Monitoring

This chapter begins by providing detailed instructions on how to construct a containerized ONOS controller that is seamlessly connected with the specialized CRM application. Furthermore, the ONOS container has the capability to function in many architectures, specifically x86_64 and arm64 [73], and the utilization of the Docker CLI plugin known as *docker/buildx*. The next part provides an overview of the preparation for four experiments that replicate real-world scenarios. Following each testing scenario, the comprehensive outcomes pertaining to significant system resources such as memory or I/O are displayed. These results can be used as the reference values in future test cases.

15.6 Multi-Architecture ONOS Image

One of the key characteristics of IoT and FC is the wide diversity of hardware architectures employed in fog and edge devices. Yet, the official ONOS image does not support any other architecture than amd64. To overcome such a predicament, an innovative approach utilizing several tools and technologies is facilitated [73]: a two-stages Dockerfile relies on a Makefile to provide necessary instruction and other information (e.g., libs, packages, qemu binaries, ONOS source code); then, with the help of Docker manifest command, three different images for `amd64`, `arm64v8`, and `arm32v7` are built and pushed to the public registry of Docker. To lessen the efforts required in that lengthy procedure, we take advantage of the CI tools such as Gitlab-CI[39] or Circle-CI[40], integrating them into their ONOS build chain as described in Section 8.2.

The CRM system introduces a newer strategy to build multi-architecture ONOS Docker images using Docker *buildx*. It is a CLI plugin providing extra features, such as concurrently building multi-platform (i.e., multi-arch) images or builder instances supported by the Moby Buildkit[98] toolkit. Compared to the old way of Docker *manifest*, *buildx* has two significant improvements:

[98]https://github.com/moby/buildkit

1. *buildx* is more manageable: While utilizing a Makefile can greatly simplify the management of multi-architecture building procedures, it remains a complex process that requires a significant amount of time and effort to understand. Using *buildx*, users can provide a single Dockerfile and it will trigger the build process for all accessible builder instances. Moreover, after a builder instance completes its construction process, it promptly uploads the resulting image to the predetermined registry, often the public Dockerhub.

2. *buildx* exhibits significantly higher speed: The lack of parallelism in Docker's *manifest* feature is what causes it to be slow and inefficient when constructing complex multi-platform images. Users systematically construct and upload the images for each chosen architecture to the registry, and subsequently consolidate all of those photos into a manifest list, which is then uploaded to the same register. *buildx*, however, constructs one picture for each chosen architecture simultaneously, effectively harnessing the capabilities of contemporary microprocessors.

Prior to describing the multi-arch ONOS image using buildx, it is important to note a few significant changes related to this specific use-case of ONOS:

- The official Dockerfile of ONOS cannot be used since the use of "buildx" necessitates that the base image specified in the Dockerfile must likewise be compatible with multiple architectures.

- The compilation and assembly process of the installable `onos.tar.gz` in ARM architecture is much slower compared to the `x86_64` architecture. There are a few possible causes for this decline: ONOS possesses a substantial and intricate subsystem due to the inclusion of numerous interdependent and autonomous modules. In addition, there are significant incompatibilities because some NodeJS libraries and Java packages do not have support for a non-x86_64 architecture.

Hence, the more recent technique consists of several sequential steps as outlined below:

1. "Initiate": Referring to the authoritative ONOS documentation, it is essential to install the required tools and packages. The installation and configuration of *buildx* has been completed in the previous round. Next, the specified

version of ONOS will be retrieved and unpacked to a precise location. The next task is to duplicate the source code of "crma" and place it in the "apps" directory within the main ONOS directory. Finally, the two Java packages, specifically *org.knowhowlab.osgi.sigar* and *io.netty.netty-tcnative-boringssl-static*, must be substituted as their present versions are incompatible with the ARM architecture.

2. The Bazel[99] build tool generates the installable `onos.tar.gz` file during the initial round of this phase. Afterwards, the installable ONOS file is transferred into a customized Dockerfile that utilizes a new multi-platforms base image. Ultimately, the *buildx* tools will commence the construction process to generate and upload the ONOS image, which supports several architectures, to the public DockerHub registry.

In order to decrease the time it takes to build and allow for CI pipelines, the "init" and "Build" phases mentioned earlier are reintroduced in a Makefile, within the "init" and "build" functions, respectively. The ONOS multi-arch image, combined with the required "crma", is now prepared to monitor and display the utilization of resources on all hosts.

15.7 CRM System Evaluation

We utilize 15-minute trial runs in Containernet[86] for the evaluation and present the mean data in this section. All trials can be redone by reusing our Jupyter[100] script, which directly connects to the InfluxDB[91] of the CRM system. The first assessment is presented in Table 15.3, wherein the architecture comprises one switch, one collector node, and two worker nodes.

	Collector	Worker 1	Worker 2
CPU [%]	6.4	4.9	5.1
Memory [%]	10.1	6	5.9
Disk I/O [kBps]	2.2	0.5	0.5
Network I/O [kbps]	100.7	19.8	18.7

Table 15.3: Utilization of the CRM system on a single switch topology.

[99] https://bazel.build
[100] https://jupyter.org

The second evaluation is shown in Table 15.4, where the topology is build up on two switches, one collector node, and four worker nodes.

	Collector	Worker 1	Worker 2	Worker 3	Worker 4
CPU [%]	7.9	6.3	7	4.7	6.5
Memory [%]	10.4	6.1	5.9	6	6.1
Disk I/O [kBps]	3.9	0.4	0.4	0.5	0.4
Network I/O [kbps]	171	18.5	18.5	18.4	18.5

Table 15.4: Utilization of the CRM system on a linear topology.

The final evaluation is presented in Table 15.5, where the topology comprises three switches, one collector node, and eight worker nodes.

	Collector	W.1	W.2	W.3	W.4	W.5	Wo.6	W.7	W.8
CPU [%]	11	9.3	9.7	8	9.8	9.4	9.4	9.7	8
Memory [%]	12.3	6.1	6	6.1	6	6	6.2	6.1	6
Disk I/O [kBps]	7.7	0.4	0.4	0.4	0.4	0.4	0.4	0.5	0.5
Net. I/O [kbps]	321.3	18.5	18.5	19.5	19.8	18.5	18.5	19.8	18.5

Table 15.5: Utilization of the CRM system on a mesh topology.

The three test scenarios demonstrate patterns in the fundamental statistics of system resources, including CPU and network I/O. The Controller node consistently exhibits the highest average CPU use throughout all tests, due to its complex subsystem and the operation of the greatest number of containers relative to other Worker nodes. As the quantity of registered Workers rises, the CPU time consumed also escalates. This tendency is equally applicable to other resources such as memory or network I/O.

With the CRM system we provide the foundation to distribute services in a Cloud-to-Fog-Continuum, which we use for a cloudless FL framework.

Chapter 16
Cloudless Federated Learning

For deploying the services in a Cloud-to-Fog-Continuum, we seek to assess the viability of the MAPE-K architecture (see Section 9.2) for container orchestration, focusing on failure scenarios like container failures, network disconnections, and excessive simultaneous container executions. The autonomous system for service orchestration must identify and manage these problems, establishing a framework for dynamic event handling procedures. The research additionally seeks to tackle node scalability in FC and EC contexts, emphasizing distributed node scoring methodologies rather than a centralized solver. We identify a method for estimating container resource utilization prior to deployment, thereby avoiding node overload and assuring sufficient service performance. Moreover, we develop a distributed node scoring mechanism to mitigate decision complexity as the number of nodes increases. All in all, the objective is to develop a more efficient and dependable container orchestration system.

16.1 Architectural Realization of the Cloudless Federated Machine Learning Framework

This chapter presents our proposed architecture for a container orchestrator, as seen in Figure 16.1. This image depicts the distinct levels of CC, FC, and EC, along with their respective capabilities and components, which are visually distinguished via color-coding. In this first section, our aim is to provide a brief overview of each layer and subsequently delve into further elaboration by devoting a separate section to each layer.

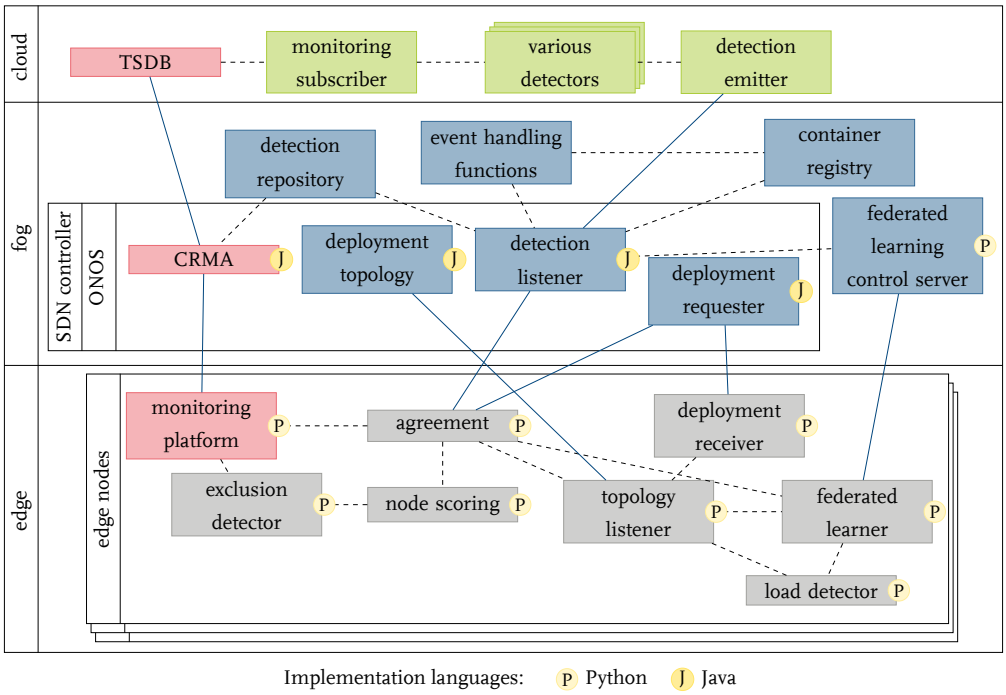


Figure 16.1: High level architecture for FL in cloudless computing architectures. The structure shows the deployment of the modules on the cloud-to-fog-continuum layers and the realization follows the MAPE-K paradigm [43].

Our architecture is primarily influenced by the work of Azimi *et al.* [10], which employs a three-layer architecture. To begin, let us distinguish between the three layers: cloud, fog, and edge. The purpose of the cloud layer is to examine the metrics provided by the edge nodes. The Docker containers are executed on the edge nodes. By utilizing these metrics, the cloud layer may analyze abnormalities in the nodes and identify node failures. Subsequently, these events will be transmitted to the fog layer for processing. The fog layer provides the ability to interface with the SDN controller for executing tasks from the *event handling functions* server or initiating a FL process. The purpose of this server is to enable the application provider to implement customized event handling. This can include actions like as sending email notifications or making deployment requests to the orchestrator in order to scale out the application. Like scaling out, the *event handling functions* can also move an app to a different area, giving application providers more choice over where the program is deployed, for example, to improve latency. The *orchestrator* requests services to be executed on edge devices, which also implement a distributed agreement mechanism to determine the deployment location of the

containers. One characteristic of fog is its presence in the atmosphere. Computing networks are extensive networks consisting of a vast number of devices. Our objective is to develop a filtering algorithm that can assign labels to nodes and organize them into a hierarchical structure as shown in Figure 16.2. Here, it is evident that the nodes can be assigned labels with any level of complexity. Every deployment request necessitates a label to ensure that we can install the services in accordance with the requirements of the application provider. These hierarchies also enable the network operator and application suppliers to have greater authority over the distribution of applications. It is possible to include several nodes inside a single hierarchy, as demonstrated by "Region A". Alternatively, nodes with extensive disk capacity or fast storage can have sub-levels where all data-related applications are stored, as exemplified by "Region B". Another illustration of more intricate hierarchies might have application providers designating particular nodes exclusively for apps, as exemplified in "Region C". Moreover, these ideas can be integrated into the network. Another benefit of this hierarchical network is the reduction in the number of nodes required for each agreement, resulting in speedier deployment as fewer nodes need to be evaluated and compared.

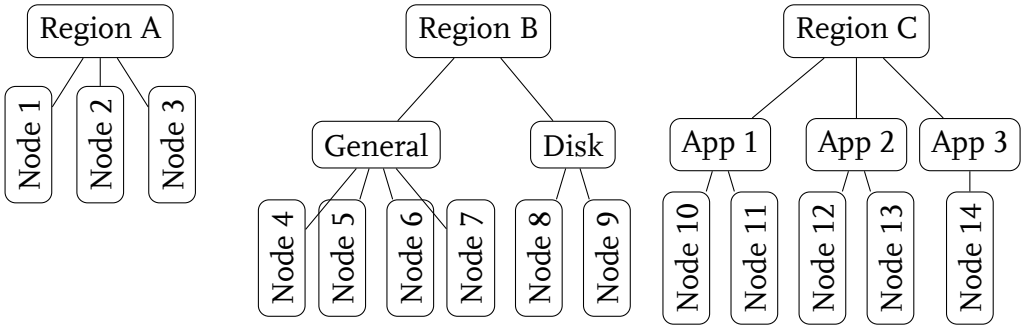


Figure 16.2: Node hierarchies for filtering purposes. Different attributes can be used to define the filter condition of nodes, where applications can run on [43].

Following the MAPE-K paradigm of Section 9.2, the "monitoring" capability is associated with the red-colored components. The *monitoring platform*, described in Chapter 15, gathers measurements from both nodes and containers on edge devices. These metrics are then sent at regular intervals to the SDN controller, which stores them in a TSDB. Since the edge nodes lack direct connectivity to the cloud layer, the *CRM app*, integrated within the SDN controller, transfers the monitoring data to the cloud layer.

The "analysis" feature is visually represented using the color green. The first component is the *monitoring subscriber*, which subscribes to any new entry in the TSDB. Subsequently, other computations are performed by *various detectors* to identify occurrences, including the following:

- *Load detector*: This detection service alerts the fog layer when the system experiences either a low or a hierarchically high load. When there is a generally low workload, we aim to initiate the FL procedure on every edge node. We utilize the concept of FL where each hierarchy is considered as a distinct cluster. Learning is conducted concurrently for each hierarchy, and subsequently, a central server consolidates the data. In the presence of a hierarchical high load, it is desirable to record the resource use of each container as data for the FL process.

- *Taint detector*: We employ the concept of Kubernetes' taints to identify nodes that should no longer be utilized for deployment. These indicators represent the levels of CPU, memory, or disk utilization.

- *Node failure detector*: If the *monitoring subscriber* stops receiving metric reports for a specific node, it indicates that the node is either offline or has lost its Internet connection.

- *New node detector*: Incorporating a novel machine into the existing system necessitates specific activities, such as disseminating this information to the fog layer.

- *Anomaly detector*: Our objective is to identify instances when metrics such as CPU usage, memory usage, network activity, or disk usage exhibit unusually high levels during unexpected periods.

If any of these detectors detect events, we aim to notify the *detection listener* by utilizing the *detection emitter*.

The "planning" component consists of a single entity known as the *detection listener*, which retrieves event handling strategies for the events it gets from the "analysis" layer. The *detection listener* is represented by a blue color. The infrastructure tools facilitate the blue *planning* layer by serving as the *execution* components. Application providers have the ability to put customized functions in the

event handling functions that are activated when specific conditions from the *detection repository* are met. The *detection repository* is a storage system for function Uniform Resource Locators (URLs) and their corresponding conditions. The *container registry* is a storage system for storing functions. Furthermore, when there is a worldwide decrease in demand, the *FL control server* initiates and oversees a FL process. As previously said, network operators allocate each node to a hierarchical structure. The network administrators utilize the *deployment topology* service to modify the hierarchy of the nodes. This service also offers a user interface.

The final feature, mainly emphasized in the color gray, is the orchestrator. The most crucial aspect of a container orchestrator is determining which node should deploy each container. In order to make this scheduling decision, it is necessary to have information about the existing utilization of the node and the resources that are needed for the new container. The current consumption can be measured using the *monitoring platform*. Unfortunately, we presently lack the means to ascertain the container resources of a fresh deployment. Since a centralized approach is not effective in resolving this problem, we must request the container deployment requester for the relevant metrics. An approach to accomplish this is to allow the requester to complete a radar map utilizing CPU, memory, network, and disk as metrics, as depicted in Figure 16.3.

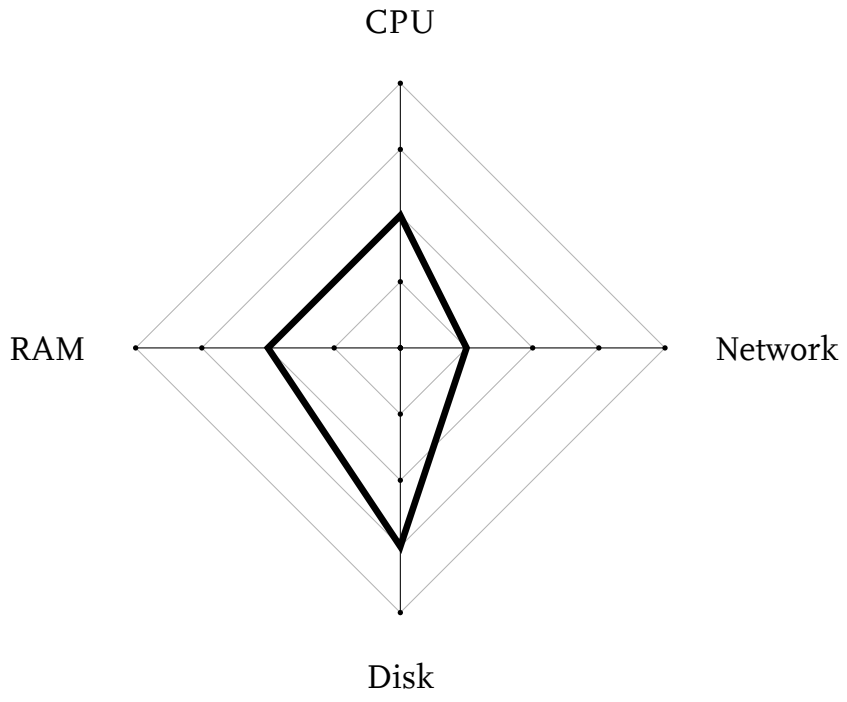


Figure 16.3: Radar chart to specify resource consumption of services [44].

We need application providers to annotate each container deployment request with a corresponding chart. Therefore, we may obtain the necessary resources from the currently active containers by utilizing chart annotations. This can be achieved by employing FL on each node and container. In order to predict the resources required for containers, we require the presence of the *FL control server*. This server communicates with the *federated learners* to learn the desired approximation of the resource function. In order to facilitate communication between nodes in FL, it is necessary for each node to have knowledge of the network topology. This information is obtained from the SDN controller and is provided to the nodes through the *topology listener*. The *node scoring* module utilizes the predicted resource consumption to compute a score for every deployment request. The method of rating the nodes will take into account many parameters, such as the utilization of ports, container names, and resource overload. These factors will be verified by the *exclusion detector*. Given that each node possesses knowledge of its resources, the container's necessary resources (deduced by AI), and the node's condition, all eligible nodes for deployment will engage in a distributed *agreement* procedure to collectively determine and select the node with the greatest score. The node with the highest score subsequently utilizes the *deployment receiver* component to deploy the container. The *load detector* component identifies the appropriate time for the FL process to acquire knowledge from the local model and distribute it throughout the network.

Two additional modules, which are not depicted in Figure 16.1, namely the *container list* and *deployment infrastructure UI*, serve as debugging and data visualization components. While not crucial to the system's functionality, they offer network operators a comprehensive picture of the system's current state.

Now that we have provided a comprehensive overview of the system's architecture and capabilities, we will proceed to provide a more detailed view of the FL algorithm.

16.2 Federated Machine Learning Algorithm

We will now present the algorithm introduced by McMahan *et al.* [115], which established the foundation for FL. Furthermore, we intend to examine the enhancements implemented by Chou *et al.* [32]. First, let us delineate the distinction between traditional ML and FL. In traditional ML, a singular machine aggregates

all training data and develops the ML model. Conversely, in FL, where data is disseminated over multiple servers that may not be independent and identically distributed, the model computes in a federated manner.

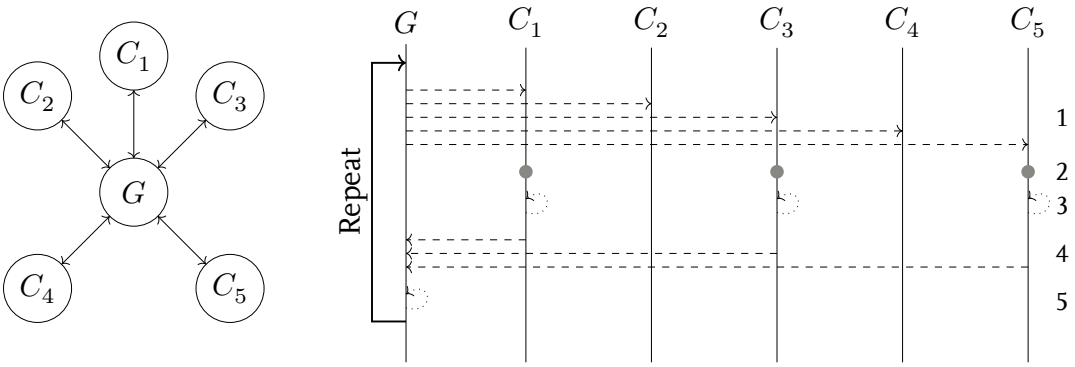


Figure 16.4: The concept of federated averaging, where a central node `G` acts as a coordinator [cf. 32].

Each node trains using its local data, and the models are amalgamated upon the completion of learning by each node as illustrated in Figure 16.4. On the left, a global server G is present, accompanied by several client devices designated as C_i. Initially, we must create a network wherein each device connects to the central server G (step 1). Let C represent the batch size and K denote the number of devices; the server G randomly picks a random set of $max(K * C, 1)$ clients (step 2). In the figure, we select C_1, C_3, and C_5 to train a model provided by the server G utilizing their data (step 3). Each node transmits the results to the server G (step 4), which consolidates the models provided by the clients (step 5). The global server repeats this procedure until a termination condition is met, specifically a predetermined number of rounds. Nonetheless, a major drawback of this ML approach is its reliance on a star network architecture, which results in a communication bottleneck. Chou *et al.* [32] addresses the problem of communication overhead by establishing many Peer-to-Peer (P2P) networks that collaboratively train, hence diminishing the network burden on the central server. We illustrate this method in Figure 16.5. Initially, "FedP2P" must establish local P2P networks (step 1). Subsequently, all devices inside the network train utilizing their local data (step 2) and consolidate their weights by "AllReduce" (step 3), which the nodes subsequently transmit to the central server (step 4). The server subsequently aggregates the received weights to construct a global model (step 5) and transmits the parameters of this global model to the P2P networks (step 6). In our implementation of the

orchestrator utilizing FedP2P, we aim to present the algorithmic description in Algorithm 1, where $F_i(\theta) = l\left(X_i, Y_i, \theta\right)$, with l representing the chosen loss function, X denoting the vector of training data, and Y signifying the vector of training data outcomes. A critical factor to examine is the parameter L, representing the number of P2P networks. We reference the original paper [32] on the calculation of this parameter with minimal communication cost.

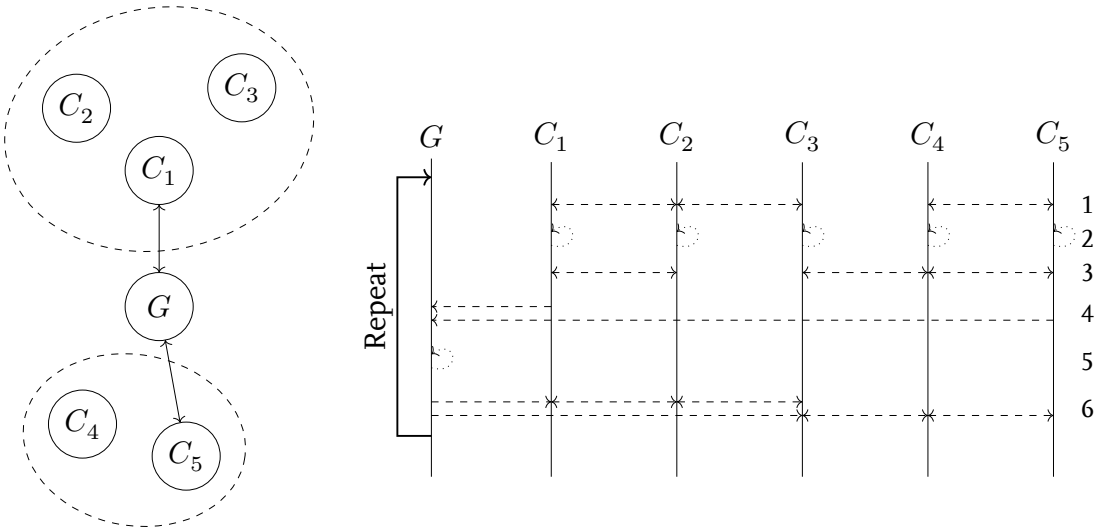


Figure 16.5: The concept of federated P2P learning functionality, where clusters compute local models before synchronizing them via a central node `G` [cf. 32].

16.3 Service Placement in the Cloudless Environment

When the entire system exhibits minimal resource consumption, we aim to initiate the FL process to ensure that the learning procedure does not adversely affect the QoE of ongoing services. When a hierarchy exhibits elevated resource consumption, we aim to record the resource usage data for each container in a node-local database. The federated learning method utilizes this data for training. Therefore, it is necessary to maintain one global model and one individual model for each hierarchy. The machine learning technique seeks to estimate a function that represents resource utilization at a specific period. A two-layer neural network for function approximation takes the timestamp as input and outputs the anticipated $score_{resources}$ value, which averages the CPU and memory utilization (see Equation 16.1).

Algorithm 1 Federated P2P learning algorithm [32]

Input

T	Number of Iterations
L	Number of local P2P networks
Q	Size of the local P2P network
η	step size
O	batch size
E	number of epochs on local client
θ_G^0	global training model at iteration 0

Output

θ_G^T	Global model for T iterations on server G

1: **for** $t = 0$ to $T - 1$ in parallel **do**
2: $Z_1, Z_2, ..., Z_L$ ▷ form P2P networks
3: **for** $l = 1$ to L in parallel **do**
4: **for** $C_i \in Z \subseteq Z_l$ s.t. $|Z| = Q$ in parallel **do**
5: ▷ Device trains on local data with step size η, batch size O and epochs E
6: $\theta_{C_i}^{t+1} \leftarrow \min F_i(\theta_{C_i}^{\theta})$
7: **end for**
8: $\gamma_i \leftarrow |D_i| / \sum_i |D_i|$
9: $\theta_{Z_l^{t+1}} \leftarrow \sum_{C_i \in Z} \gamma_i \theta_{C_i}$ ▷ AllReduce
10: **end for**
11: $\theta_G^{t+1} \leftarrow L^{-1} \sum_{l=1}^{L} \theta_{Z_l}^{t+1}$ ▷ Aggregate
12: **end for**
13: $\theta_G^T \leftarrow \theta_G^{t+1}$
14: **return** θ_G^T

$$score_{resources} = \frac{cpu\_usage + mem\_usage}{2} \tag{16.1}$$

The *detection listener* responds to the analysis conducted by the cloud layer by referencing event handlers in the *detection repository*. When an event handler is designated for the specified scenario, the *detection listener* initiates an HTTP request to the event handler URL.

The *deployment requester* initiates the deployment of new containers and engages with edge nodes for consensus and evaluation. It requires an understanding of the hierarchy for application deployment together with its container configurations. Furthermore, we offer a radar chart for each container, depicting its projected resource utilization on a scale from 0 to 10, enabling ML processes to produce an estimation. This resource estimation enables the deployment of a service on a suitable node.

Based on the hierarchical data, the *deployment requester* transmits the container ratings to the edge devices for the agreement phase, during which each edge node in the hierarchy responds with its score value or indicates a timeout if the node is unavailable. Upon receiving the score values, the *deployment requester* transmits the configuration to the node with the highest score, which subsequently responds with the deployment log to indicate whether the deployment was successful or failed.

The *event handling functions* server is an instance of OpenFaaS[46]. It implements and operates services for event management, the *E* in MAPE-K. We utilize OpenFaaS as an event management platform, offering a standardized and generic method for dynamic event processing.

The *container registry* retains such functions. In OpenFaaS, a serverless function operates as an isolated, self-sufficient container. These containers permit arbitrary code from application providers to manage certain events. Arbitrarily complicated behavior can be depicted by function chaining.

The *FL control server* executes the previously described FL with certain modifications. Regrettably, the original algorithm proposed by Chou *et al.* [32] divides the network uniformly to generate L networks of nearly equivalent size. Our approach differs since we reutilize pre-existing hierarchies, which are not assured to be uniformly sized. Consequently, we modified $\theta_G^{t+1} \leftarrow L^{-1} \sum_{l=1}^{L} \theta_{Z_l}^{t+1}$ to $\theta_G^{t+1} = \sum_{l=1}^{L} \gamma_l \theta_{Z_l}^{t+1}$, where $\gamma_l = \frac{|D_l|}{\sum_{l=1}^{L} |D_l|}$ represents the proportion of datapoints

for learning within the current hierarchy relative to the total datapoints across all hierarchies, as detailed in Algorithm 1. We acquire knowledge daily when the *load detector* forecasts a global low load. Each hierarchy concurrently acquires the ability to expedite the FL process.

The deployment receiver on each edge node implements Docker infrastructure, including containers, networks, and volumes. It accomplishes this by transmitting the deployment request to the *builder daemon* of the *monitoring platform*. The deployment receiver transmits the deployment log from the builder daemon to the deployment requester, enabling the application provider to ascertain whether the deployment was successful.

The *federated learner* acquires the ability to estimate container resource use from a chart diagram. It serves as the equivalent to the *FL control server*, which alone consolidates the models according to each hierarchy. The learner acquires knowledge from the local data model. An elected leader will thereafter consolidate the acquired models from its hierarchy and transmit them to the *FL control server*.

Daily, the *load detector* module on each node receives a notification for elevated load and records the resource consumption of each container in a local database. We intend to record resource usage exclusively during peak load periods, as it is preferable to overestimate resource requirements rather than underestimate them. Exaggerating resource estimations results in increased node exclusions from the agreement, particularly affecting nodes with limited available resources. Therefore, we exclusively acquire knowledge from high load instances to induce overshooting rather than undershooting, as FC and EC possess the attribute of a substantial number of computing devices to mitigate errors arising from our overshooting strategy.

The *node scoring* module employs the FL model and supplementary host information to compute a score value for each deployment request. The necessary metrics are collected by the *monitoring platform*. The node CPU capacity is determined by the provided usage time in milliseconds, while memory and disk capacity are constrained by hardware limitations. The network capacity is derived from the NIC. In the node scoring process, we utilize the fundamental principles of the Kubernetes scheduling framework, as examined in Wei-guo *et al.* [157]. Given that our target devices are edge devices, we aim to incorporate aspects such as disk usage and network usage into the metric set M and compute their utilization. By

integrating the exploitation factor f, which resides in the interval $[0, 1]$, with the resource utilization equations, we ultimately compute the comprehensive node utilization score using Equation 16.2.

$$\begin{aligned} M &= \{cpu, mem, disk, net\} \\ S_x &= \frac{T_x - S_{req\_x} - f * P_{req\_x}}{T_x} \qquad for\ x \in M \\ score_1 &= \frac{\sum_{x \in M} S_x}{|M|} \end{aligned} \tag{16.2}$$

Alongside the overall node utilization score, we also compute the degree of balance among the resources. Upon computing both scores, we inquire the *exclusion detector* component regarding the feasibility of the current deployment request.

The *topology listener* service identifies modifications in the node hierarchy. Each node must be aware of its hierarchy for FL or load detection. The topologies serve as our method for implementing container affinity and anti-affinity. Kubernetes implements the concept of affinity to facilitate the deployment of containers with minimal latency, whereas Kubernetes uses the principle of anti-affinity to ensure that containers are not in proximity to one another. We incorporate the principle of anti-affinity in our architecture by distributing containers across various hierarchies and the principle of affinity in our architecture by deploying containers within the same region, as low latency and real-time connectivity are inherent characteristics of EC.

A chart diagram is the basis for the *federated learner* to estimate the container resource usage. Therefore, it uses a local data model. For each hierarchy, an elected leader will aggregate the learned models of its hierarchy. They are sent to the *FL control server*, which is the counterpart that aggregates those models.

We show the workflow for the whole FL process in Figure 16.6. When an iteration begins, the *FL control server* sends the global model, the hierarchy name, and the list of IP addresses of the current hierarchy to each node. The first IP address is considered the hierarchy leader in this iteration. This way of leader election is simple and efficient. However, it does not give respect to the resources on the node. Once the leaders are elected, they send a “Leader Elected” message to their hierarchies. By receiving the message each node trains the model from its local data. After the *federated learner*s have finished their learning process, the nodes send their models to their hierarchy leader, which aggregates them using the equations

from Algorithm 1 (line 8+9). Once the merging is complete, all hierarchy leaders send the aggregated models to the *FL control server*, which aggregates all hierarchy models. After this aggregation, the *FL control server* sends the finalized model to each node and stores the global model.

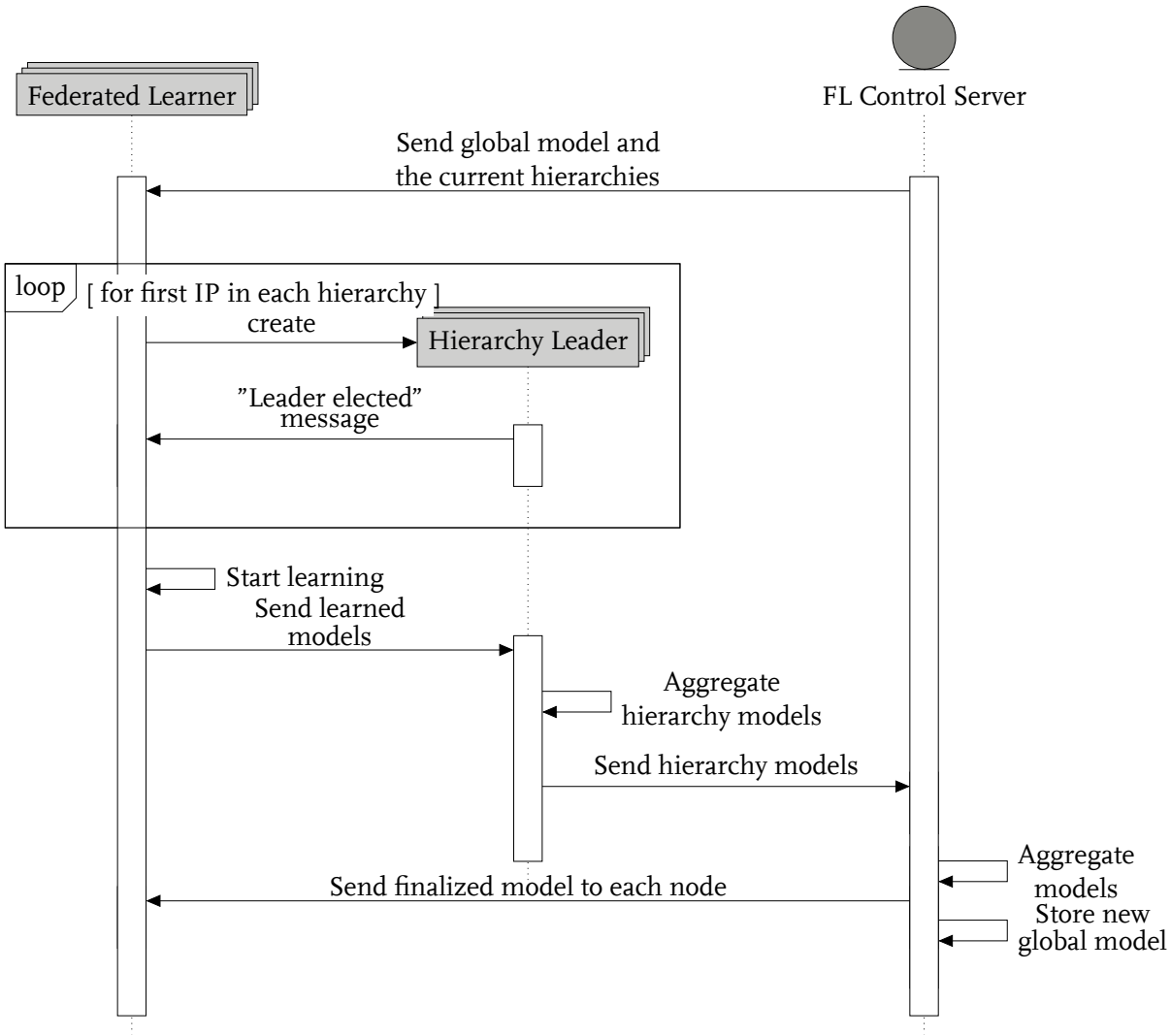


Figure 16.6: Sequence diagram of the FL process [cf. 44].

16.3.1 Evaluation of the FL Process

This part aims to assess our methodology for CPU, RAM, and disk utilization on the edge nodes. Additionally, we aim to assess the agreement duration, the load detection model, and the FL accuracy utilizing a basic target approximation, specifically the identity function.

All benchmarks were executed in a Kernel-based Virtual Machine (KVM) environment utilizing 28 virtual CPU cores (from a host with 16 cores/32 threads of an AMD-5950x, operating at a maximum frequency of 5.0GHz), 48GB of DDR4 RAM (from a total of 64GB DDR4 RAM on the host) functioning at 3200MHz, and a 128GB NVME SSD. The loopback network speed within the VM operates at 45.7 Gbits/sec due to a local simulation. The host system's network operations must not adversely affect the simulation's network speed, given a maximum available host network speed of 89.5 Gbits/sec.

In our simulation, each edge node is allocated two cores and 4GB of RAM from the total of 28 virtual cores and 48GB of RAM available. The disk capacity for each host is restricted to 5GB.

To develop the global model, we aggregate all CPU and memory metrics together with their timestamps, compute the $score$ values, and normalize these scores across all nodes at the specified timestamps. The results are presented in Figure 16.7, where the blue dots represent the $score$ values over time, the orange line denotes the approximated function, and the blue bar indicates the minimum point. We supply the node count for layer 1 and layer 2, along with the computed minimum value of this function approximation. The node count is expressed as a fraction, such as $1/2$ or $1/4$. It signifies the subsequent: We have established a fixed value of 800 for our experiments. A value of 1 indicates that the quantity of concealed nodes in a layer is a constant value. Thus, a value of $1/4$ indicates that, with a total of 800, the specified layer contains 200 hidden nodes. We conducted tests using several fixed values to confirm that our results aligned with differing layer counts.

We employ the eyeballing technique to evaluate this quantity of nodes, as the vast volume of data points in the dataset resulted in an accuracy of zero across all scenarios, with the loss remaining similarly little.

The lowest load time in this scenario ranges from 00:13 to 00:15, and the disparity across the configurations is largely negligible. Furthermore, we evaluated the similar function approximation employing a 3 Layer Neural Network and observed no substantial alterations in the function approximation.

In summary, we determined that the dimensions of the layers do not substantially impact the identification of the minimal value, as illustrated in Figure 16.7, provided there is a sufficient number of nodes in a layer.

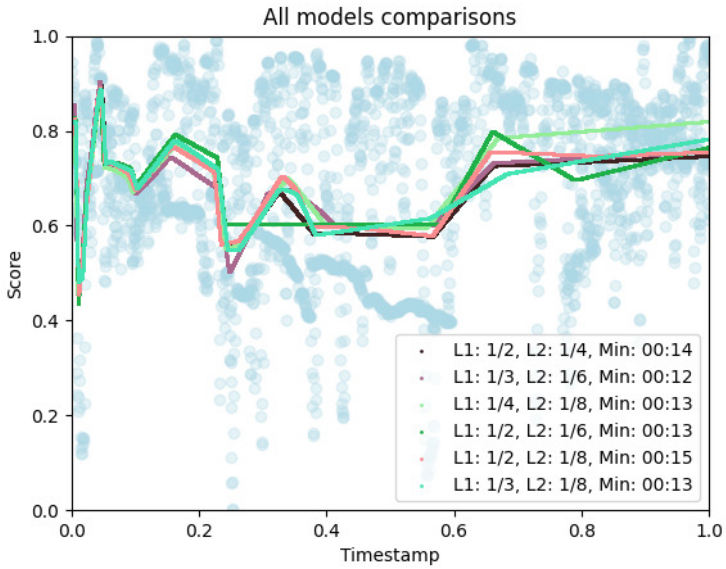


Figure 16.7: Comparison of different approximations through the load detection function [43].

16.3.2 Federated Learning Accuracy

Our learning objective involves the application of an Auto-Encoder concept, wherein the input data is identical to the output data. The input data consists of four random values ranging from 0 to 100. Consequently, the output data consist of identical random values. The four arbitrary values denote the attributes of the radar chart: CPU, RAM, disk, and network. We use a straightforward learning objective due to the uncomplicated nature of resource estimate, which includes an additional scaling component. The RAM use of 2 may correspond to 300Mb of RAM, indicating a scaling factor of 150. In a practical application of our methodology, the data will likely be far noisier, as certain individuals may classify their application with a RAM of 2, resulting in varying output values. Nonetheless, the Auto-Encoder and our resource use scenario should exhibit comparable learning challenges.

We configured our benchmark with the following values:

- Node count ranging between 5, 10, 20, 35, 50, 75, and 100 nodes
- FL iteration count of 1, 7, 15, and 30 iterations
- Hierarchy count of 1 (every node in the same hierarchy), 10, and 25 hierarchies
- Node data entries on each node from 1, 7, and 15

We evaluated each arrangement using the specified parameters. Although our benchmark scenarios may not accurately represent actual data sets, due to the random assignment of nodes to hierarchies and uniform data distribution across nodes, we nevertheless gain insight into the significance of each parameter. We employed three distinct classifiers to assess feature relevance and identify the parameter with the greatest influence.

We have used a random forest classifier [21], which is the left classifier, a decision tree regressor [22] in the middle, and a eXtreme Gradient Boosting (XGB) classifier [29] on the right of Figure 16.8.

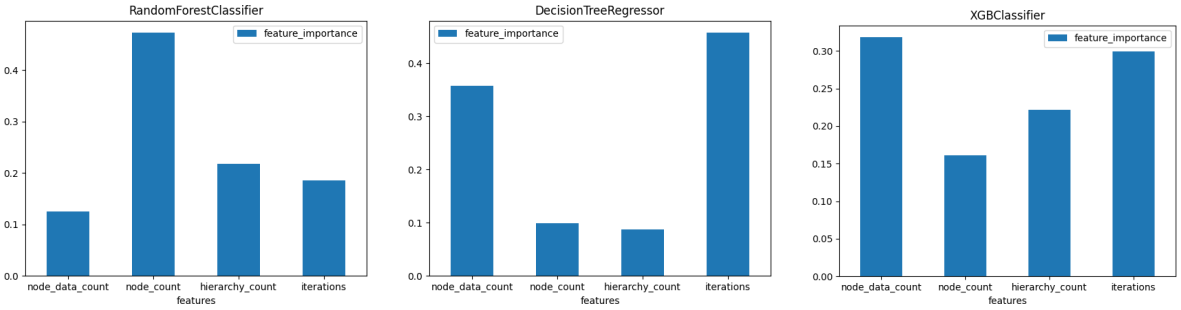


Figure 16.8: Four features for the FL model, the node data count, node count, hierarchy count, and iterations are evaluated by three classifiers, namely a random forest classifier, a decision tree regressor, and a XGB classifier to determine the feature importance [43].

The decision tree regressor and the XGB classifier identify node data entries and iterations as the most significant factors. The random forest classifier does not adhere to this trend. The rationale for the random forest classifier's estimation of node count as a significant factor remains unexplained. Upon examining the benchmark data findings, we arrive to the same conclusion as the decision tree regressor and XGB classifier: the iteration count and dataset size per node are more critical factors than the number of nodes or hierarchies.

With these results in consideration, we can now elucidate the f factor of the resource estimation in further detail. Initially, the volume of available data constrains the precision of estimations. We will extract datasets with a single node data entry, arranged in declining order of accuracy.

The maximum accuracy for one dataset on each node is 0.52, necessitating five nodes, 25 hierarchies (most of which are unoccupied due to the distribution of just five over the 25 hierarchies), and 30 iterations. Furthermore, we confirmed that beyond 30 repetitions does not affect the accuracy with a limited data set. We validated this assertion by executing the precise configuration of 5 nodes and 25 hierarchies with 30, 50, and 100 iterations, which did not yield a substantial increase in accuracy (as evidenced by the benchmark data in the attachment). An increased number of containers is necessary to enhance the model's accuracy over time. Therefore, the learning factor f must gradually escalate over time, particularly when the number of deployed containers is minimal.

PART V

Applications and Services in the Cloud-to-Fog-Continuum

If we did all the things we are capable of, we would literally astound ourselves.

T.A. Edison

Parts of this chapter are taken from [28], [65]–[67], [71], [76], [164]

This part presents applications that can get advantages from the service deployment in the Cloud-to-Fog-Continuum through the FL service orchestration middleware discussed in Chapter 16.

Initially, we examine SensIoT, which facilitated the creation of the service orchestration middleware, in Chapter 17. The motivation for the development outlined in Section 17.1 precedes the description of the initial prototype MonTreAL in Section 17.2. The conclusive solution pertinent to library deployment is illustrated in Section 17.3.

Numerous applications, which are prevalent in the IoT, are utilizing blockchain or elements of its implementation to enhance data encryption. The concepts for implementing these applications in a Cloud-to-Fog-Continuum are presented in Chapter 19. The discussion begins with the storage of sensitive sensor data in Section 19.1, while the functionality of blockchains may provide a solution in Section 19.2. Following the virtualized deployment, the algorithms are executed in Section 19.3 and evaluated in Section 19.4.

Furthermore, surveillance systems can advantageously operate in a cloudless environment, facilitating the automatic distribution of the services outlined in Chapter 20. The development of a surveillance system for the IoT commences with the video transmission on energy-efficient devices in Section Section 20.1 and continues in Section Section 20.2.

SensIoT

CHAPTER 17

SensIoT is an open-source framework for monitoring sensors in the IoT, utilizing established technologies for straightforward implementation, maintenance, and adaptability. It connects highly specialized and rigid sensor monitoring solutions with the creation of alternative solutions from the ground up. SensIoT is founded on MonTreAL, which is explained in Section 17.2, a framework developed for libraries to preserve cultural material and mitigate damage. It employs virtualized microservices provided by Docker, which may be executed on single-board devices such as the RPi. SensIoT augments MonTreAL's capacity to include commodity servers into clusters, improving configuration and maintenance of existing infrastructures, as shown by its architecture in Section 17.3. The technology is first deployed across local computing nodes, circumventing CC paradigms. The system employs a microservice design to attain resilience and fault tolerance, utilizing Docker Swarm for service orchestration. SensIoT has been operational at the University of Bamberg's Library since autumn 2017.

17.1 Preserving Cultural Heritage in Libraries

Libraries have developed into modern service centers for research and education, while also performing an essential role in safeguarding cultural heritage via physical and digital archives. The University Library of Bamberg stores publications in historically protected buildings and considers regular inspections and reporting of environmental factors crucial. Structural issues may result in delayed identification of excessive humidity, leading to the development of mold [60]. Continued monitoring procedures are essential to prevent health concerns for colleagues,

even if the location is no longer in use or the stock is unavailable for borrowing. The library has searched for a cost-effective and adaptable technology solution to monitor publications in real time. This strategy is essential for preserving the safety and well-being of its employees.

Currently, there is no open-source tool available for automatically gathering, combining, and visually presenting environmental data for different magazines in the library landscape. Commercial systems mostly concentrate on monitoring exhibition windows or safe rooms, which are more expensive. Available options for monitoring environmental conditions include Lewis *et al.* [109] calibration laboratory, Jassas *et al.* [98] e-health research, and Hentschel *et al.* [85] distributed software architecture. The solutions utilize a RPi with Python for sensor communication and Cacti[101] for graphical analysis. However, the spatial distribution of magazines renders these notions unsuitable for local use. Hentschel *et al.*'s distributed software design allows individual nodes inside a network to perform local computations, transmit data to a central database, and execute tasks independently. The project intends to adapt current technologies used in huge data centers to energy-efficient devices that are easily updateable and manageable, suited for various locations, and capable of processing local data from numerous sensors.

17.2 MonTreAL - The First Prototype to Monitor Environmental Conditions

The framework MonTreAL is an IoT prototype that utilizes a RPi 3 as its hardware foundation. Users can access the framework through a manager node, which retrieves data from worker nodes connected to sensors. The IoT prototype enables periodic temperature and humidity readings to be collected and stored for visualization. The system provides great mobility for sensor nodes, allowing for simple installation and horizontal scalability. Software delivery and upkeep are accomplished using contemporary virtualization capabilities.

The system has two primary elements: devices functioning as Swarm Workers linked to sensors and dispersed around buildings for monitoring purposes. These devices regularly engage with their surroundings to gather and analyze information. A sensor manager is initiated within the Swarm, which manages a privileged sensor container on the device by using DinD[51] for communication with the sen-

[101] https://www.cacti.net

sors that need to run in privileged mode. The sensor transmits data to the NSQD container via the sensor manager along with its message buffer. Sensor containers may handle different sensor types attached to the General Purpose Input/Output (GPIO) interface or USB port of the RPi by converting multiple manufacturer-specific data formats into a standardized format for additional processing. Every sensor container is programmed to provide a distinct identifier with each package it transmits to the buffer, enabling the data to be matched with its source at a later time.

The second part of MonTreAL consists of management nodes, which are operated by one or more devices responsible for overseeing the swarm, as well as collecting, combining, and storing the gathered data. The nsqlookupd container functions as a directory service for consumers to locate and access buffer messages, enabling users to obtain messages directly from the network.

17.3 Architectural Concept of SensIoT

SensIoT utilizes compact sensor devices attached to an SBC to gather environmental data because of their tiny size and energy efficiency. The RPi 3 for example is equipped with Wireless LAN (WLAN) and Bluetooth Low Energy (BLE) features, making it ideal for distributed sensor networks. The SBC is affordable, easily accessible, and widely manufactured. It offers connections to different sensors via GPIO and USB and is capable of running Linux variants. The device can run all essential software, particularly Docker, which is officially compatible with ARM architectures. The data is transmitted to a server where it is consolidated, stored, and analyzed to generate a valuable environmental profile.

The framework comprises an amd64-based server named *Server* that operates services for receiving, aggregating, storing, and processing sensor data. It also includes multiple connected amd64/arm-based sensor devices known as *Sensor(s)*, which provide services for gathering and temporarily storing environmental data from physical sensors connected via USB or GPIO. SensIoT employs Docker and Docker Swarm for deploying its services, to handle typical software issues such as distribution, installation, removal, upgrading, trust, and software management. The server functions as the “Swarm Manager”, while each sensor functions as a “Swarm Worker”. All services operate in separate Docker containers, which are overseen by Docker Swarm for streamlined deployment and upkeep. The server

operates database and dashboard services for storing data long-term and offers a basic web API for other devices to retrieve sensor data. Docker Swarm does not support running services in privileged mode, which may be required for accessing sensor data. SensIoT utilizes the Docker Engine API to interact with the local Docker daemon on each sensor device in order to execute privileged sensor-reading services.

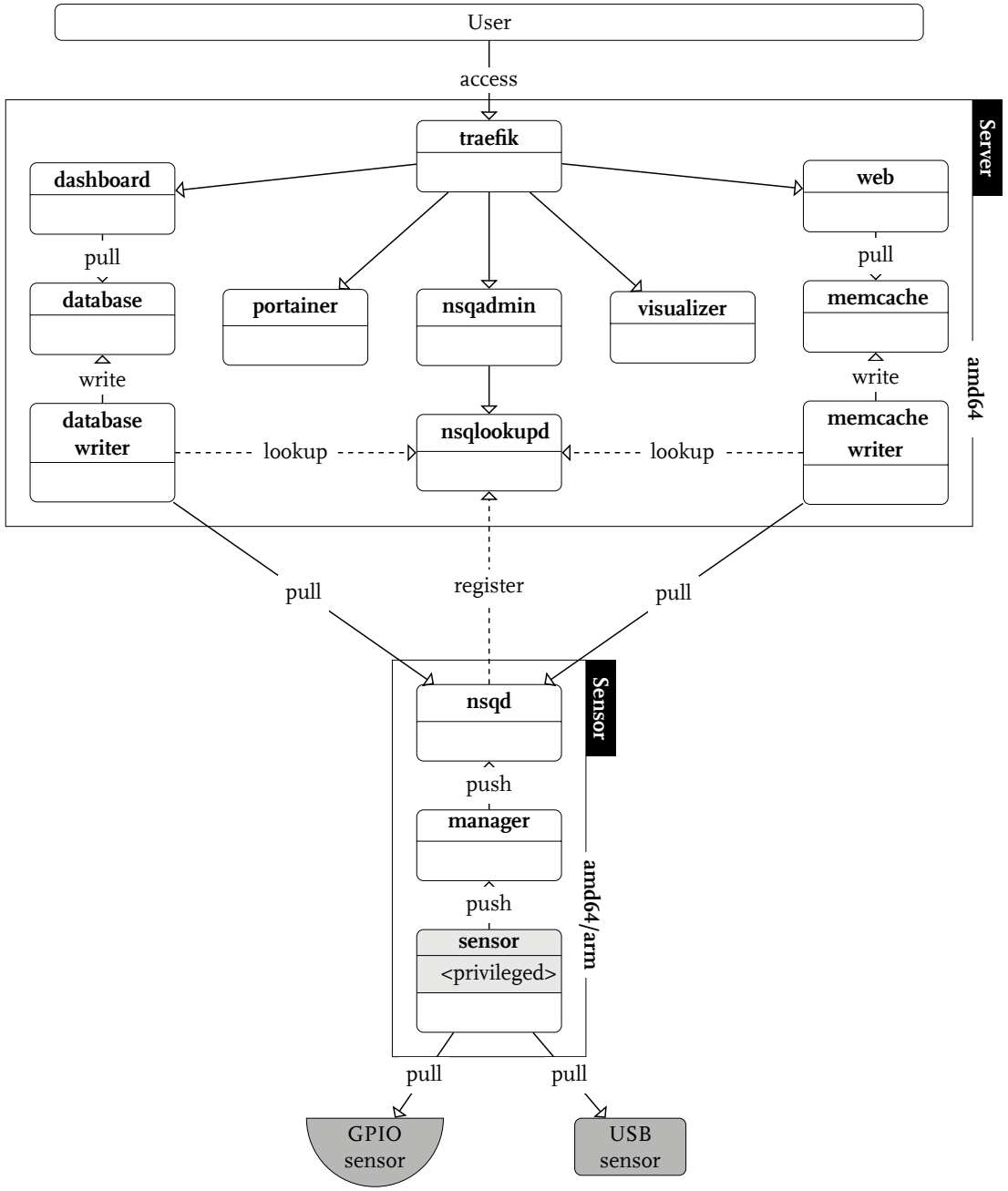


Figure 17.1: Modules to deploy the environmental monitoring framework SensIoT [cf. 65]

17.4 Data Acquisition

SensIoT initially concentrates on temperature and humidity sensors. It may now be outfitted with remote ASH2200 sensors over USB with a USB-WDE-1 receiver[102], as well as economical DHT11, DHT22, and AM2302[103] sensors through GPIO. The RPi can connect with several sensors via its USB and GPIO ports, allowing SensIoT to be readily augmented with other sensor types.

The sensor service employs a designated sensor driver to acquire data from the sensor's interface. The data generated by physical sensors can be intricate, ranging from simple numbers to more elaborate configurations, such as the USB-WDE-1 with connected ASH2200 sensors. To guarantee exchangeability among compatible systems, each sensor reading service transforms its acquired data into a more uniform format utilizing JSON.

SensIoT data is transmitted to a local management service via a TCP socket for further processing. The management service adds more information regarding the sensor's position and time to each measurement, enhancing the description and context of the data collected. This metadata facilitates intricate searches to filter, group, and analyze data, enabling the derivation of conclusions from the collected information.

The data format sent by the sensor device for server-side processing is the fundamental unit upon which all subsequent processing services depend. It has the requisite information to trace each measurement to its source. Ephemeral sensor data is subsequently prepared for transmission to the server via a distributed messaging queue.

SensIoT is a real-time distributed messaging platform that provides significant scalability and user-friendliness for inter-device communication. It comprises a variable quantity of messaging queue daemons (`nsqd`) and a directory service (`nsqlookupd`), which oversee network topology data and provide addresses of `nsqd` instances to potential customers. NSQ[105] facilitates distributed architectures devoid of a SPoF, offers substantial horizontal scalability without the need for brokers, enables communication over TCP, supports client libraries across several programming languages, and incorporates TLS for secure communications.

NSQ[105] employs the publish-subscribe paradigm to enhance network scalabil-

[102]https://de.elv.com/p/elv-bausatz-usb-wetterdaten-empfaenger-usb-wde1-2-P104657
[103]https://www.mouser.com/datasheet/2/737/dht-932870.pdf

ity. Each `nsqd` instance manages several data streams known as topics, with each topic comprising one or more channels. An `nsqd` instance sustains a persistent TCP connection to one or several `nsqlookupd` instances and regularly transmits its status, which is accessible via an HTTP endpoint for consumers. Consumers can query subjects at one or several `nsqlookupd` instances and subscribe to their desired topic by establishing a new channel or joining an existing channel formed by another consumer.

NSQ[105] facilitates the deployment of SensIoT in a dynamically dispersed environment, enhances scalability without introducing complexity, and bolsters overall stability through service replication. It functions as a write buffer to coordinate and synchronize write requests to the database from a potentially extensive array of sensor devices.

SensIoT operates many instances of `nsqd`, `nsqlookupd`, and consumers to ensure high availability despite the potential failure of individual devices. NSQ's throughput is sufficiently high to accommodate messages from several hundred or thousands of sensors concurrently. The dynamic generation of topics and channels facilitates the addition and removal of new nodes and consumers during runtime, without disrupting other nodes or necessitating reconfiguration of active `nsqd` or `nsqlookupd` instances.

SensIoT incorporates NSQ[105] on AMD64 and ARM architectures, enabling customizable deployment of these services by users.

Time series data, a succession of data points recorded at consistent time intervals, is essential in environmental science and is frequently utilized to depict environmental characteristics. Monitoring systems such as SensIoT, equipped with several sensor devices, produce a substantial volume of time series data that requires enrichment with metadata for effective utilization, comprehension, accessibility, and management. Conventional relational databases such as MySQL[104] and PostgreSQL[81] have scalability challenges when aggregating hundreds of thousands of metrics, rendering them impractical for extensive monitoring systems. Time series data possesses distinct properties that facilitate optimizations in the storage of large volumes of data. The increasing trend towards high-performance, lightweight databases tailored for time series data has resulted in the proliferation of TSDB. SensIoT presently accommodates two distinct databases for the storage of sensor data: InfluxDB[91] and Prometheus[83].

[104] https://www.mysql.com

Chapter 18
FaaS enabled SensIoT

By integrating FaaS to the SensIoT system in [66], we replace the NSQ[105] system with an OpenFaaS[46] function invocation process, like described in Section 10.3. The OpenFaaS platform and its services will be deployed on the swarm manager, which will begin and allocate the functions across the entire swarm, which will be shown in Section 18.1 To utilize the OpenFaaS project, it is essential to provide the executable for all architectures present in the environment, where support for numerous architectures is missing.

The OpenFaaS gateway will also replace the NSQ admin and lookup services on the server side. On the sensor side, the NSQ daemons will be eliminated and substituted by a function call through the gateway. The deployment strategy for the function is contingent upon the experimental configuration. One approach involves relying on the OpenFaaS platform to handle scaling and replication, while the other involves duplicating the function on all participants in the swarm, where we show an evaluation in Section 18.2

18.1 Integration of OpenFaaS Functions into SensIoT

To implement our modifications, we initially adapt the SensIoT framework [65] to operate with a FaaS system. Furthermore, we must modify the foundational FaaS system to support our Cloud-to-Fog-Continuum.

[105] https://nsq.io/

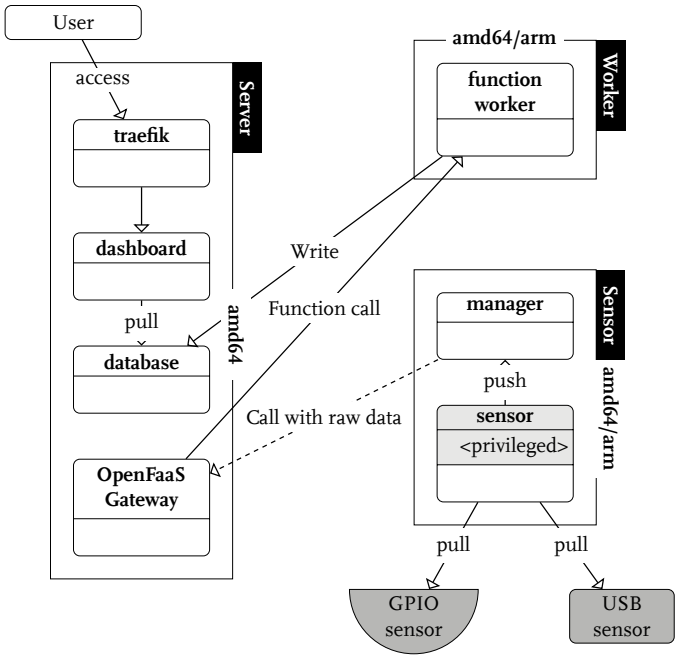


Figure 18.1: Overview of the FaaSified SensIoT platform [cf. 66].

The SensIoT framework consists of two components [65] as shown in Chapter 17:

- Server: A server, which runs different services to receive, aggregate, store, and process sensor data.
- Sensors: RPis, which run services to collect the data from attached sensors.

The part we want to modify is NSQ[105], which serves as the real-time distributed messaging solution to facilitate communication across the server, its diverse services, and the sensor devices. The NSQ administration and lookup containers operate on the server, utilizing InfluxDB[91] as the database service for storing sensor values. The database writer, another component, subscribes to the NSQ for writing to the database. On the worker side, the publication of sensor readings to the queue is accomplished by executing NSQ daemons and rendering them accessible via the NSQ lookup of the server. A local manager service operates on sensor devices and transmits the collected data to various services via the NSQ daemon.

Besides, a supplementary service is necessary to connect the local manager to the corresponding physical or virtual sensor. This container is excluded from the swarm, as services cannot be initiated in *privileged* mode, which is essential for accessing the system's physical interfaces. SensIoT utilizes the Docker Engine

API to interact with the local Docker daemon on each host, enabling the initiation of these privileged sensor-reading containers.

18.1.1 Modifications of SensIoT to integrate FaaS Functionality

We removed non-essential services from the original SensIoT [65] system, such as portainer, visualizer, and memcache, as they were unnecessary for our performance investigation. Subsequently, we modified the information flow by abolishing the NSQ queues and substituting them with HTTP REST calls. The comprehensive and revised architectural overview is available for examination in Figure 18.1.

OpenFaaS, realizing the principles of FaaS of Section 10.3, and its associated services will be deployed on the swarm manager, which will thereafter initiate and render the functions accessible across the swarm. Due to our lack of control over the instantiation of the function, we must supply it for all architectures present in our environment, namely manager (amd64) or worker (arm64). The templates offered by the OpenFaaS project lack support for ARM architectures, necessitating the creation of our own multi-architecture function template, which is accessible on GitHub[106]. To do this, we utilized the official Dockerfile template as a foundation and coordinated its construction using a Makefile. This document outlines the essential procedures for consolidating all sources into a Docker image, executing this for each specified architecture, uploading them to Dockerhub, and aggregating them through a Docker manifest, similar to the CI principle of Section 8.2.

Thus, we substitute the NSQ administration and lookup services on the server side with the OpenFaaS gateway. We eliminate the NSQ daemons on the sensor side and substitute their functionality with a function call through the gateway. The local manager transmits the sensor data to the function by invoking it through the OpenFaaS gateway using a REST call, which specifies the function name and includes the data in the body of the request. The OpenFaaS gateway manages load balancing by directing the entire request to the appropriate function among several replicas of the same function. The function accepts data in JSON format and transforms it into a designated format for transmission to the database service.

In conclusion, the FaaSified version eliminated the publish-subscribe model,

[106]https://github.com/uniba-ktr/python_faas_template

which was done using the NSQ framework. We substituted it with an OpenFaaS function, with its replicas deployed throughout the swarm. The remainder of the system is unchanged in both versions.

18.1.2 Continuous Integration Workflow

The updates encompass alterations to several essential components of the original platform, necessitating constant testing to identify potential errors promptly. Given that this is not a monolithic application that can be executed with the push of a button, but rather a complex array of numerous interacting Docker containers, we opted to design a continuous integration approach. This significantly aids the development process by minimizing human labor and eliminating distracting and repetitive setup procedures, resulting in reproducibility and reduced mistake susceptibility.

CI was initially defined by Fowler and Foemmel [56] and seeks to automate the build process, testing phase, and occasionally the deployment of the freshly constructed, tested, and distributed executable into a production environment.

The completed approach involved constructing Docker images that incorporated all modifications on a per-commit basis for arm64, arm32, and amd64 [73]. Furthermore, we utilized Docker's manifest feature, enabling the dynamic delivery of the appropriate image for the architecture of the asking Docker host, hence facilitating deployment on any Docker-enabled platform. The benefits of this CI strategy should be evident at this point: Regardless of the individual who initiates a commit, all users can download and evaluate the corresponding Docker image upon the completion of the build process.

18.2 Evaluation of FaaS enabled SensIoT

To evaluate and elucidate the efficacy of the serverless computing paradigm within our SensIoT framework specifically, and the IoT domain broadly, we have resolved to develop the following frameworks.

- **SensIoT**: We keep the infrastructure of SensIoT as intact as possible.
- **SensIoT-FaaS-D**: We use the OpenFaaS default settings and our custom function configuration without any additional changes. This means there is only a single replica of the function arbitrarily distributed among available workers.

- **SensIoT-FaaS-M**: We modify our custom function configuration so that it is evenly distributed on all available nodes in the system.

We conducted the following two experiments to evaluate on the performance of the former frameworks:

1. **Idle System**: Running all devices in the infrastructure with least possible services in order to have our baseline metrics.
2. **Workload scenario**: Running all devices with the three frameworks listed above using the workload settings from Table 18.1.

Configuration	Workload	Description
global sensor	1	The global sensor attached to the IoT device, where we only use a single mock sensor to generate data.
sensor reader devices	5 devices per host	The actual sensor reading device which listens and collects data.
emitting interval	60s	The amount of time in seconds between two consecutive emissions of data generated by a sensor reader.

Table 18.1: Workload configurations for SensIoT, SensIoT-FaaS-D and SensIoT-FaaS-M [66].

18.2.1 Trial Setup

In our experimental series, the test apparatus comprises a single local virtual machine, referred to as *manager*, which functions as the management node tasked with collecting performance measurements from all worker nodes, including its own. Three RPi devices – specifically Sheldon, Leonard, and Howard – function as worker nodes that deliver continuous data streams. In all test settings, the data produced by these personnel consists solely of simulated temperature and humidity readings, rather than genuine measurements from the surrounding environment.

We gathered and analyzed the resource use of the three frameworks on both a per-container and aggregated basis. The resource metrics of concern are as follows:

1. **CPU usage**: indicates how much of the processor's capacity is currently in use by a particular SensIoT framework within a predefined period.
2. **Memory usage**: denotes the amount of main memory - measured in MB - which is being consumed by a particular SensIoT framework within a predefined period.
3. **Network usage**: shows the network rate - measured in kbps - of both uplink and downlink channels.

Resource metrics are collected from all nodes utilizing **cAdvisor**[8], which offers insights into resource utilization and performance attributes of active containers, and **NodeExporter**[84], which presents all pertinent metrics of a Linux system. The data is gathered by a Prometheus[83] container operating on the management node as described in Subsection 15.1.2. Subsequently, all metrics are accessible for further extraction through Prometheus' HTTP API queries. Additionally, all trials consist of a 30-minute observation duration with a 20-second query resolution.

18.2.2 Resource Consumption of the Idle System

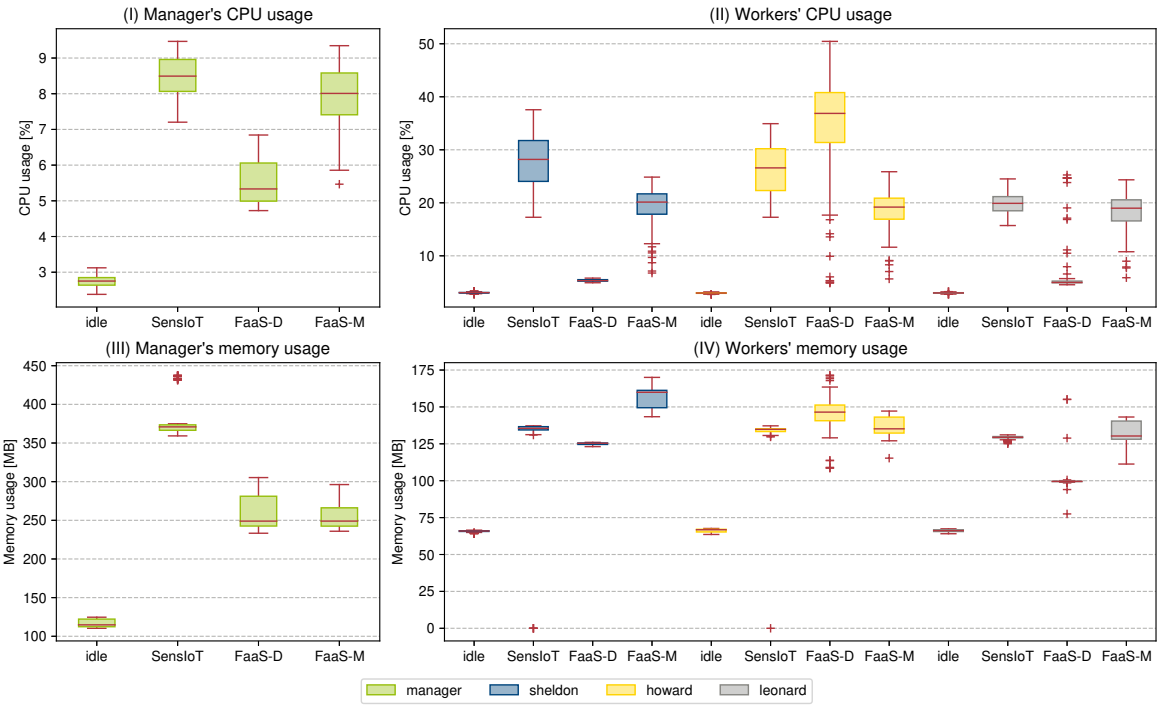


Figure 18.2: CPU and memory resource usage comparison based on container level measurements [66].

In the “idle system”, all nodes operate containers associated with the monitoring service, including cAdvisor[8], NodeExporter[84], and Prometheus[83]. The latter operates on the management node to gather resource statistics from the worker nodes [77]. Utilizing the information derived from Prometheus queries, we retrieved our metrics and compiled the overview in Table 18.2, which serves for subsequent comparison.

Average usage	manager	sheldon	leonard	howard
CPU (%)	2.746	3.044	2.994	2.967
Memory (MB)	116.747	65.712	66.082	66.150
TX (kbps)	5.97	22.877	22.746	22.575
RX (kbps)	61.637	3.630	3.647	3.567

Table 18.2: “Idle system’s” resource consumption [66].

The manager is consistently distinguished from the other three worker nodes due to two factors:

1. It is a completely different system being an instance of a virtual machine running on an amd64 machine and

2. it is the one performing the actual measurements that are being visualized. It gathers data, which results in a higher receive and lower transmission rate.

In the subsequent experiment, we utilize these baseline resource consumption metrics to elucidate the actual performance of our three SensIoT configurations and ultimately, to assess our project objective: whether OpenFaaS can confer any advantages regarding resource efficiency in the IoT domain.

The baseline measures provide a comprehensive account of idle CPU, memory, and bandwidth utilization. The CPU utilization levels are approximately three percent with minimal variation, indicating a steady system. The results on memory are as anticipated: The manager responsible for data collection and accessibility ranges between 110-120 MB, whereas the nodes have stabilized at approximately 65 MB. As the nodes transmit their values to the manager within the swarm, they must maintain heightened transmission rates, while the manager is required to receive this data. This was precisely corroborated by the assessment of the network I/O.

18.2.3 Performance Evaluation

	CPU (%)			Memory (MB)			Network I/O (RX + TX in kbps)		
	S	D	M	S	D	M	S	D	M
manager	+5.8	+2.8	+5.2	+265.3	+142.3	+134	+74.1	+78.4	+65.1
sheldon	+24.7	+2.3	+16.2	+63.5	+59.4	+90.5	+35.4	+25.5	+37.9
howard	+23.3	+30.8	+15.5	+66.5	+78.9	+70.8	+35.9	+35.9	+31.8
leonard	+16.9	+4.3	+15.4	+63	+34.7	+66.9	+35.4	+24.6	+31.5

Table 18.3: Average resource consumption changes compared to the ones of the "idle system" (cf. Table 18.2). The measurements are taken for SensIoT (S), FaaS-D (D), and FaaS-M (M) [66].

In accordance with the "idle system", our second experiment focuses on the default configurations of the SensIoT system. This involves a 60-second gap between sensor readings and five sensor reading devices per host (see to Table 18.1). The three distinct configurations will be evaluated using these parameters and compared to the baseline values in Table 18.3. Overall, SensIoT appears to utilize the most resources across all categories; however, the results require a more comprehensive analysis. The NSQ nodes predominantly utilize CPU resources, with certain exceptions. When the FaaS system utilizes a single replica (SensIoT-FaaS-D), the node hosting the function instance will inherently have a significantly increased burden. In Figure 18.2 (II), the yellow worker *howard* is the node responsible for managing all function calls in SensIoT-FaaS-D. The subsequent exception pertains to the uniform distribution of the function across the swarm, from which the workers collectively appear to derive benefits; yet, the manager is also required to execute the function. This elucidates the reason for its heightened CPU utilization in this instance. The memory on the manager node is largest in the SensIoT configuration (about 380MB), whereas the two FaaS configurations are nearly identical (roughly 255MB). This is logical, as in the SensIoT configuration, the manager must supply the NSQ lookup and an independent database writer, whereas in the FaaS environment, just the database persists. The conversion and function invocation transition to the sensor nodes, which is somewhat evident in the memory usage. Figure 18.2 (IV) illustrates that in the SensIoT environment,

the nodes exhibit very predictable behavior, whereas the FaaS executions demonstrate a modest increase with greater variability. Ultimately, we assessed the network transmission and reception rates, revealing that on the manager node, there is negligible disparity among the FaaSified systems, whereas SensIoT exhibits a marginally reduced traffic generation (refer to Figure 18.3). The nodes in Figure 18.3 (II) operating in SensIoT mode exhibit comparable findings, whereas in FaaS mode, the variation is somewhat greater, and the overall traffic appears to be slightly reduced. The receiving rate presents a comparable scenario.

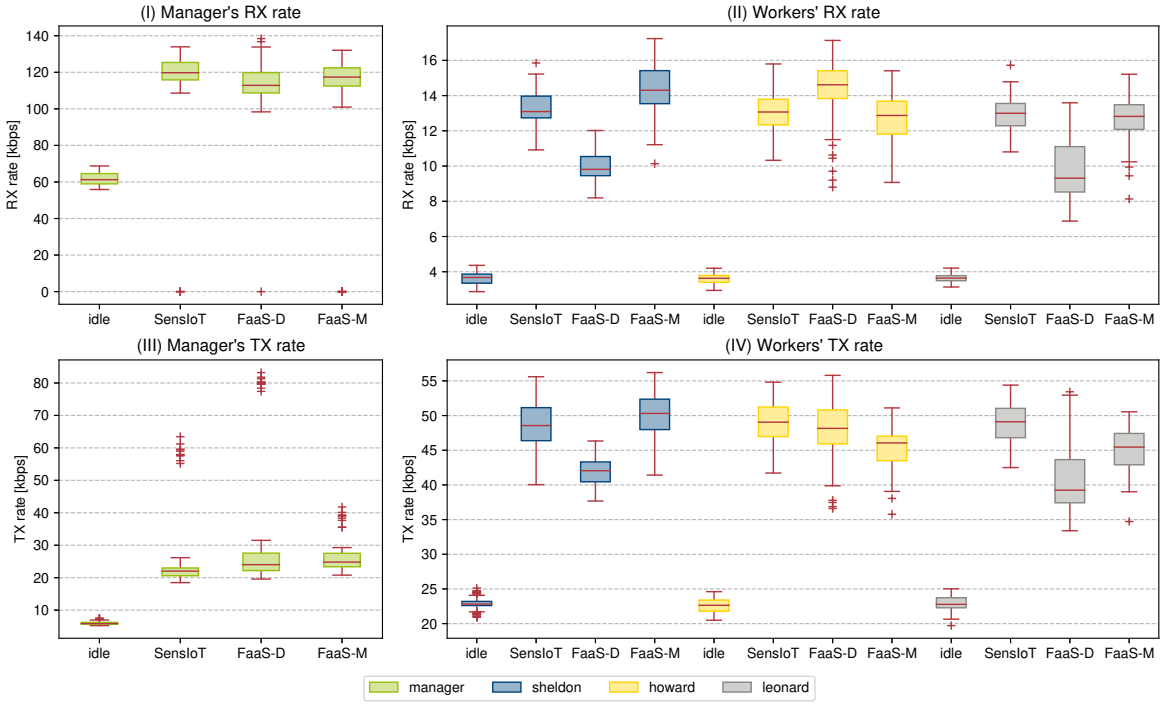


Figure 18.3: Network usage comparison based on container level measurements [66].

18.3 Opinion on FaaS enabled SensIoT

The NSQ system (SensIoT) utilizes marginally more resources; nonetheless, this disparity is insufficient to warrant a preference for a certain system architecture. The FaaSified system necessitates further features, such a caching technique, to address connectivity loss during real-life activities. The decoupling of the software project throughout the modification phase facilitates enhanced maintenance and performance monitoring.

The original SensIoT system comprised many Docker containers, whereas the FaaSified solution features a singular codebase, facilitating identification and management. The revised IoT middleware platform, originally based on a publish-subscribe architecture, was converted to utilize the FaaS approach and evaluated in a controlled environment. A comparison indicated that the method preserved system resources to a certain extent.

The revised system does not include a cached reported value feature, which is advantageous during temporary network loss. The revised system lacks a cached value feature, as the local manager on the RPis communicates sensor readings through a function call that just sends data to the database without disseminating it to other services, unlike the original system.

Implementing caching techniques and assessing performance impact would enhance the overall comparison's significance. Transforming further IoT middleware solutions into FaaS and evaluating performance would facilitate the analysis of result verification in broader contexts. The FaaSification and performance assessment established a foundation for evaluating various service models for IoT middleware, highlighting the benefits of this novel computing paradigm.

Chapter 19
Blockchain Integration for IoT

Novel blockchain technology applications arise for IoT services, such as smart buildings and smart homes. Deploying applications enhanced by blockchain capabilities with our orchestration middleware of Chapter 16 can help to securely gather and disseminate sensor data, especially in private IoT use cases such as ambient assisted living. The proof-of-concept employs a permissioned blockchain and FC nodes, utilizing Docker and the MultiChain framework. Performance measurements are analyzed with RPi SBCs. The FC architecture incorporating blockchain capabilities is examined, and its drawbacks are disclosed. The architecture's prospective applications encompass intelligent buildings, automated residences, and secure electronic health solutions.

The IoT integrated with FC will establish a rapidly expanding foundation for novel, swiftly developing application scenarios of blockchain technology. A blockchain is a decentralized database that is disseminated by replication among its participating peers. Data are not updated directly; instead, a compilation of change records, represented by transactions, is appended as aggregated entities known as *blocks*. All participating peers are capable of validating state changes and must reach a consensus on the status of the data stored in the corresponding data plane [26], [33].

19.1 Foundations of Blockchain Technology

A decade ago, the blockchain technology known as *Bitcoin* was introduced to establish a genuinely decentralized digital currency for its users in a P2P interaction mode, eliminating the necessity for a centralized authority [122].

The fundamental element of Bitcoin is defined by the notion of a *transaction*. A transaction contains a minimum of one input, which represents the digital currency expended by the transaction. The counterpart is represented by a transaction output directed to another peer within the Bitcoin network, transferring a set amount of digital currency. Transactions are deemed "valid" if several domain-specific constraints are satisfied, such as the condition that the total of the currency indicated in the outputs does not surpass the amount provided by the inputs [33].

Concerning Bitcoin, the classification of a *asset* is not an independent entity but is represented through transactions. The asset indicated by the input is, in fact, an output from a prior transaction. At any given moment, a user's digital balance comprises any transaction outputs directed to her that she has not yet transferred to other users. The collection of transactions that meet such criteria is referred to as *unspent transaction outputs (UTXO)*. User balances are determined by aggregating UTXOs categorized by their respective addresses. Transactions are not organized separately but rather in larger aggregated units formed concurrently. Alongside a compilation of transactions, these entities known as *blocks* provide a reference to the most recent block at that moment. This reference is generated, incorporating a hash value of the preceding blocks. Analogous to the transactions, the back-references establish a distinct sequence among these blocks. Consequently, the term *blockchain* for this system derives from the interconnection of blocks.

Bitcoin utilizes a basic stack-based language that permits a restricted level of programmability. Fewer than 100 operations are executed to facilitate fundamental cryptographic computations. A script, typically consisting of a limited number of instructions, can be appended to each transaction output to provide an enhanced specialized capability. Typically, the ownership of transactions is regulated by public-key cryptography. Users are recognized by a public key, and assets are secured in a manner that only the owner's private key can unlock and allow the transfer of those assets. A new transaction is generated to transfer an asset, with its input referencing the asset in question. The transaction output cites the new owner's public key, and the entire transaction is authenticated by the asset owner's private key.

The distributed control model of a blockchain coordinates the interactions of a network of identified, connecting peers within a transaction-oriented system architecture characterized by three fundamental features:

- *Decentralized architecture:* The blockchain operates through a collection of autonomous peers that can dynamically enter and exit the foundational P2P network. These peers can be managed by many entities that need not be aware of their identities or motives.

- *Fundamentally anonymous interaction:* The capability of a public blockchain enables participants to engage in the P2P network without identity. Read access is unrecorded, whereas data writing to the storage segment of a blockchain is facilitated by the concept of pseudonymity. This feature is an essential attribute to protect the privacy of the interacting parties.

- *Stability guarantees:* Participants in a blockchain can securely store data in the storage plane of its P2P network, certain that this data will remain unaltered, even in the absence of trust among peers.

Blockchain peers disseminate new transactions across the whole peer-to-peer overlay, and each peer retains all recently generated transactions within a block before resuming the collection of transactions for the subsequent block. The objective of establishing a shared history of blocks addresses a consensus issue. Nguyen and Kim [125] categorize the consensus mechanisms of a blockchain into *proof-based* and *vote-based* schemes. Bitcoin employs a costly proof-based consensus mechanism, while Byzantine Fault Tolerant (BFT) replication, a resolution to the Byzantine Generals Problem involving malicious nodes, utilizes a vote-based scheme [107], [154]. The algorithms PBFT [27] and BFT-SMART [14] are prominent examples of this latter category. A comparable yet less robust category of protocols is represented by crash-tolerant protocols, which protect solely against failures of honest nodes. A comprehensive summary of these many consensus procedures is available in [125, Table 3].

The selection of an effective consensus mechanism aligns with the peer's access model in a blockchain. Bitcoin exemplifies an open, unpermissioned, or permissionless network. Access to data within the P2P network and participation in the consensus process are unrestricted. An alternative is provided by a *permissioned* blockchain, wherein access to the network necessitates authentication and authorization. The latter can be achieved by established means, such as public-key cryptography, shared secrets, or whitelisted IP addresses.

Inspired by the digital currency application Bitcoin, blockchains are frequently referred to as distributed ledgers, as they facilitate the storage of asset distributions. Additionally, each peer in the underlying network maintains a copy of the ledger. Technically, the ledger is regarded as a state transition mechanism. The state denotes the distribution of units of a digital currency, whereas the transition function processes a state and a transaction, applies the transaction to the state, and produces a new, altered state [23].

In this context, it is evident that a blockchain might serve as an effective method for storing and representing various sorts of data designated for a specific degree of public sharing. For example, the blockchain *Namecoin*[107] facilitates the storage of key/value pairs to provide a decentralized alternative to DNS. Supplementary information is embedded in transactions to impart a special application-related hue to their output. These colored transactions may be utilized to grant their owner a specific access right, such as the ability to unlock a shared lock [16].

19.2 Evaluating Blockchain Functionality to Share Sensor Data

In light of common IoT usage scenarios, the implementation of blockchain technology has already been explored in the literature [37], [112], [140], [159], [166]. For example, Conoscenti *et al.* [37] has categorized approaches pertinent to IoT situations into four distinct categories: data storage management, exchange of products and data, identity management, and rating systems. Wörner and Bomhard [159] has proposed a system in which sensors transmit data to purchasers via the blockchain and receive compensation in digital currency. Their approach pertains to data storage management and the exchange of products and data, leading to significant scaling challenges. Shafagh *et al.* [140] have articulated a comprehensive perspective on data storage management and identity management concerning access control via blockchain technology. Blockchain nodes are responsible for regulating client access by enforcing rules encoded in the blockchain. Zyskind *et al.* [166] have delineated a blockchain system designed to enhance scalability and address privacy concerns. Their technology comprises a blockchain component that hosts public data and a private off-chain component.

To investigate the implementation of blockchain technology in IoT scenarios within a FC environment, it is essential to first select an appropriate blockchain

[107] https://www.namecoin.org

framework, ensuring that its underlying architecture and AIs align with the requirements of the specific IoT application [33], [37]. To sustain a network of engaged IoT devices, it is essential to operate stable fog nodes equipped with blockchain capability to service their peers inside the corresponding network. Operational logical management entities could oversee the functioning of a blockchain and, specifically, adjudicate new membership applications within a specified organizational framework of an IoT scenario. Consequently, a permissioned blockchain with regulated membership is favored. It does not encounter the issues associated with open networks, such as Sybil attacks, and can leverage this advantage to select a more cost-effective and robust consensus mechanism than a proof-based method like Bitcoin's proof-of-work scheme [122], [125]. Nonetheless, the system must continue to offer functionalities that uphold security and privacy. Although blockchain nodes verify their identities to engage in the network, the necessary trust between parties should be minimized as much as feasible. An individual organizational unit should not let new peers to engage in the consensus process without the approval of the majority of other peers. Concerning our FC strategy that incorporates blockchain capabilities, an implementation utilizing an affordable ARM processor environment with open-source frameworks should be feasible as well.

In this regard, we have conducted a comprehensive review of published blockchain frameworks. It has omitted proprietary and obsolete systems, like Hyperledger Iroha[108], Corda[109], Hydrachain[110], and Openchain[111]. Ultimately, three blockchain frameworks listed in Table 19.1, specifically Hyperledger Fabric [5], Quorum [38], and MultiChain [63], were deemed appropriate and subjected to thorough analysis.

[108]https://www.lfdecentralizedtrust.org/projects/iroha
[109]https://corda.net/
[110]https://hydrachain.org/
[111]https://openchainproject.org/

Attributes	Fabric	Quorum	MultiChain
General properties	Apache-2.0 licensed, ~5700 stars on GitHub, written in Go, originating from IBM.	LGPL-3.0 licensed, ~2400 stars on GitHub, written in Go, forked from *go-ethereum*.	GPL-3.0 licensed, ~400 stars on GitHub, written in C++, based on Bitcoin and compatible with its RPC protocol.
Consensus protocol	Pluggable consensus backend, using Kafka as only current production-ready option.	Pluggable consensus backend, applying the Practical BFT (PBFT)-inspired *Istanbul BFT* algorithm as most promising option.	Round-robin vetting, i.e., each node proposes an equal number of blocks by taking turns.
Application features	Potential use of a variety of general-purpose languages to write *chaincode*.	Supporting Ethereum virtual machine (EVM) smart contracts.	Programmability limited to Bitcoin-like scripts, supporting *stream*-abstraction on top of transactions with a permission system on address granularity level.
Confidentiality features	Access can be configured to *channels* which are independent blockchains.	Supporting *private transactions/contracts*, handling encryption and storage by nodes in a P2P-fashion, exposing users' plain-text to nodes.	Proposing an encryption scheme based on its stream feature with additional support for encrypted off-chain storage.

Table 19.1: Comparing relevant contenders' attributes of the selected open source blockchain frameworks [28].

In conclusion, MultiChain [63] has been selected as the foundational blockchain framework to augment our fog computing architecture. The rationale is that its concept closely resembles the original blockchain model of Bitcoin, with pertinent modifications to suit a permissioned environment. Valuable abstractions, such as streams that provide publish, subscribe, and query functionalities, are constructed upon the transaction primitive. Moreover, MultiChain possesses a sophisticated permission mechanism predicated on the capabilities of blockchain addresses. Only specific addresses are permitted to broadcast data to a stream or to initiate new transactions.

19.3 Evaluation of the Fog Computing Node With Integrated Blockchain Functionality

Our objective concerns an investigation of the performance results that are achievable by the implemented arrangement of the blockchain components in a virtualized fog gateway on a low cost SBC platform with its constrained resources.

We have created a test bench to assess several performance indicators of the MultiChain implementation. Percentage in a FC environment. A cluster of three RPis has been employed as a fog gateway network, managed by a singular PC as the control node. Each RPi 3 Model B Rev 1.2 node was outfitted with a Quad-Core 1.2GHz 64-bit CPU, 1GB of RAM, and a 100 Mbit/s Ethernet connector.

The software environment utilized was constructed on an adapted Linux system and facilitates container virtualization and orchestration with Docker as shown in Subsection 5.2.2. The blockchain capability is facilitated via a Docker image of a blockchain created using the MultiChain version 2.0 alpha 3 framework. Since MultiChain officially supports only the x84 CPU architecture, the framework had to be constructed from the ground up for the ARM processor architecture. Docker Swarm, as shown in Section 10.1, has been directed to deploy a solitary container executing the master-multi-chain image on a manager node. To efficiently operate a MultiChain node on each host, it is advisable to deploy node-multichain containers on every non-manager node. Upon startup, the master-multichain image first initializes a new blockchain. The corresponding master-multichain container possesses no intrinsic rights post-setup and can indeed be substituted by any container derived from the node-multichain image.

To monitor critical metrics of the deployed computing system of a fog node, including processor utilization, memory consumption, and file system writing, the monitoring stack of Subsection 15.1.2 was utilized. This template establishes an instance of the monitoring system *Prometheus* and installs system monitoring daemons on all Swarm nodes. The latter are frequently engaged by Prometheus to gather data. Furthermore, the template implements an instance of *Grafana*, which visually displays the aggregated performance data. To minimize burden on the RPi hosts during our experiments, a PC was incorporated into the cluster as a passive management node. The execution of the Prometheus and Grafana instances was scheduled on the latter PC using Docker node “tag” rules. This

supplementary member of the test bed did not execute any functions pertaining to MultiChain.

Transactions are the fundamental component of blockchains, even concerning the sophisticated functionalities of the MultiChain architecture, such as permissions. Consequently, it is essential to ascertain the maximum rate at which a fog gateway can accept and process individual transactions for inclusion in a block.

The MultiChain network, comprising three fog nodes, has been utilized, with each node operating on a distinct RPi. All nodes were assigned the responsibility of mining blocks. In several iterations, a predetermined number of clients consistently contacted each node's API to transmit transactions. The transaction was negligible, delivering a null amount of the integrated currency to an address not possessed by any of the three nodes in the test bed. At the onset of a test round, clients were initiated gradually over a duration of 60 seconds to prevent node overload from simultaneous queries. Although the clients have submitted queries, the metrics were documented only after an additional 60 seconds. This precaution enhanced the coherence of the measurements. Each test was conducted for a duration of 5 minutes. Prior to proceeding with the subsequent test, the test executor awaited the completion of all nodes incorporating the incoming transactions into the blockchain. The MultiChain nodes were deactivated, and a fresh blockchain was launched prior to commencing the subsequent round with an augmented amount of asking clients.

The outcomes of several test executions with varying client quantities are encapsulated in Figure 19.1. Initially, the median transaction rate per second (Tx/s) appears to vary. The client count varies from 30 to 330, representing an elevenfold variance. The disparity between the maximum and minimum recorded median rate is merely 34.5 Tx/s. Likewise, the CPU and RAM utilization on the hosts remains rather stable, whereas the average response time consistently escalates with the growth in the number of concurrent clients, as illustrated in Figures 19.2 (a) and (b). A portion of connections started to consistently fail when reaching 400 clients.

The consistent rate of published transactions per second can be attributed to the installation of MultiChain. A developer elucidated that the API accommodates just a single node concurrently owing to the implementation of locks. The API code is predominantly single-threaded. This corresponds with the CPU utilization

seen during our experiments. Upon observation, one CPU core operates nearly perpetually at full capacity, while the remaining three are predominantly inactive. The MultiChain developers have indicated that an increase in throughput beyond two concurrent connections should not be anticipated. The developers identify the CPU as the principal bottleneck, reporting a throughput of approximately 1000 Tx/s on a contemporary x86 CPU.

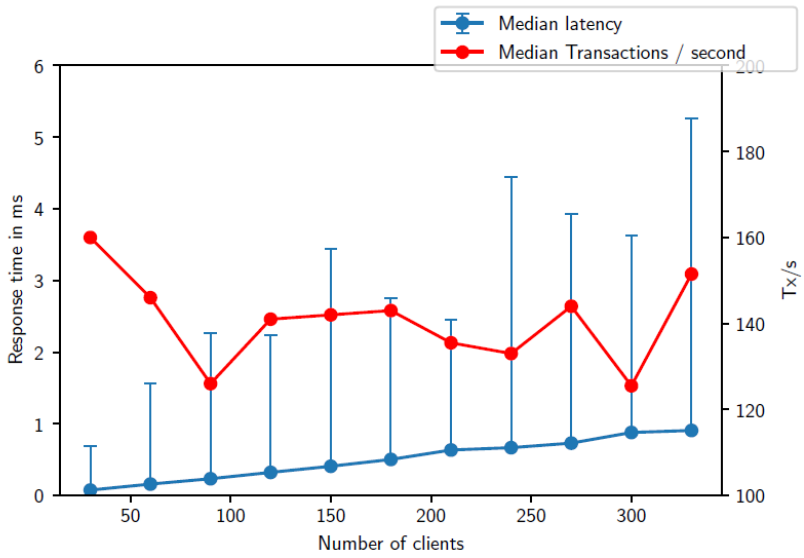
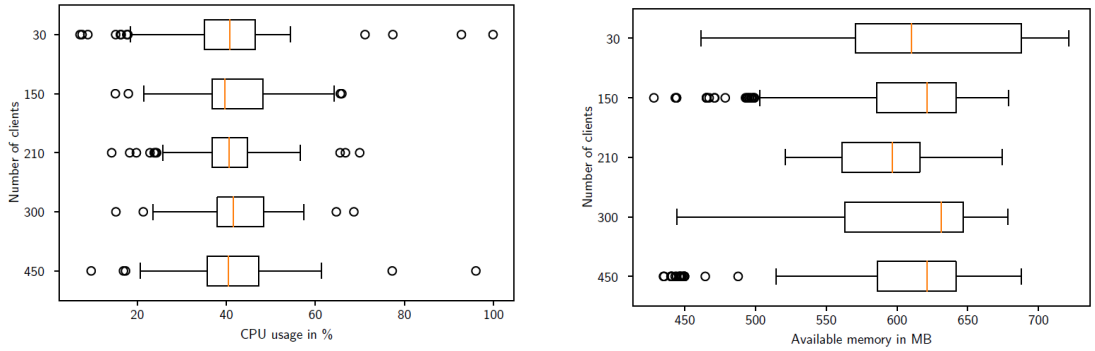


Figure 19.1: The median transactions per second, the median response time in ms, and the standard deviation are shown for a different number of requesting clients during repeated stress tests. As the x-axis depicts the *total* number of clients, each node was only targeted by $1/3$ of the clients [28].

The efficacy of the MultiChain consensus algorithm is contingent upon a mining diversity parameter, denoted as ρ. In our test bed comprising three nodes, the mining diversity has the following effect:

- $0 \leq \rho \leq 1/3$: A node is allowed to mine every block.
- $1/3 < \rho \leq 2/3$: The previous block must have been mined by another node.
- $2/3 < \rho \leq 1$: A true round-robin scheme is enforced.

When the mining diversity ρ is selected to let a node to produce successive blocks, forks occur frequently. During testing with $\rho = 0.3$, the nodes exhib-



(a) CPU utilization on all four CPU cores of the RPi.

(b) RAM usage, where the available memory is defined by the Linux Kernel

Figure 19.2: Utilization of one blockchain node during the stress test with a specific number of clients. It is measured as an average for 189 repetitions [28].

ited a propensity to converge on a consensus about a shared block when a limited number of transactions were reported. However, as further load was produced, the nodes could not reach a consensus on the subsequent block. Nodes either operated on distinct chains or ceased mining entirely for several minutes.

The test findings depicted in Figures 19.1, 19.2 (a), and (b) were obtained for the parameter $\rho = 0.75$, which is advised in the MultiChain white paper [63] because of the mandated round-robin vetting process.

19.4 Blockchain Investigation Results

Blockchain technology has arisen as a remedy for the difficulties encountered by manufacturers and consumers within the realm of IoT architectures, resulting in the aggregation of sensor data and widespread computing activities. Our paper [28] addresses the challenge of securely storing and distributing sensor data via a FC node integrated with blockchain technology. The report investigates the realm of open-source software offering blockchain functionalities and its suitability for diverse IoT applications, assessing and appraising multiple blockchain initiatives that enable developers to build apps on an established FC infrastructure.

We chose the MultiChain framework for first integration into the virtualized, modular FC gateway due to its affinity with Bitcoin and its fundamental blockchain principles. We established two innovative protocols for data storage and se-

curity access control utilizing the foundational blockchain framework MultiChain. The initial protocol enables data sharing with specified entities through a public blockchain channel, whilst the subsequent component allows for the storage of streamed real-time sensor data on the blockchain. A system has been suggested that utilizes these two protocols as core parts to publicly disclose non-sensitive data while restricting access to its sensitive components.

The suggested algorithmic methods for the new FC node depend on the behavior of the underlying blockchain platform, and the impact of essential blockchain attributes on block formation and consensus mechanisms is succinctly discussed. The importance of mining diversity is also highlighted. In conclusion, founded on proven MultiChain primitives, we provide a composable virtualized functionality of a FC node to enable extensive data processing chains for advanced IoT applications.

Chapter 20
Surveillance System

Video surveillance is essential in society, enhancing security and boosting quality of life. Current technologies, however, are costly, proprietary, and lack mobility for small-scale applications. Inexpensive SBCs can be readily deployed and integrated into existing IoT networks, enabling sensor nodes to activate various actuators. The WebRTC architecture facilitates access to P2P communications across all web browsers. By integrating these two paradigms, sensor nodes can identify predetermined events and alert users to an easily available video stream captured by an economical camera node. With the trial framework of Section 20.1 we can perform measurements on any forthcoming hardware platform to evaluate different video settings for the best performance. A prototype of a sensor-activated video surveillance system utilizing container virtualization is presented in Section 20.2, enabling user interaction via a cutting-edge browser and permitting system administrators to deploy the open-source system by executing a micro-service stack.

20.1 Video Transmissions

An evaluation of the video streaming performance of several SBC models is performed through a basic configuration involving a camera connected to the RPi. Common streaming frameworks are enclosed in a container and supervised by a camera. We provide multi-architecture images for FFmpeg[112], GStreamer[113], and VLC[114] on the Github repository *uniba-ktr/streaming*[115]. In every image, a script is included to run experiments, gather resource data, and receive video content.

[112]https://www.ffmpeg.org
[113]https://gstreamer.freedesktop.org
[114]https://www.videolan.org/vlc
[115]https://github.com/uniba-ktr/streaming

We concentrated on the most widely used video codecs for our experimental architecture, providing a brief description of each [30], [53], [162]. The H.264/MPEG-4 AVC codec is commonly utilized in applications including Blu-ray discs, video streaming, and video conferencing. Due to its great compression efficiency and compatibility with various devices, it is suitable for a wide array of devices. It's successor, H.265/HEVC, is a more recent codec that provides superior compression efficiency compared to H.264/MPEG-4 AVC. Compatibility is not as extensive as H.264/MPEG-4 AVC, although its popularity is gradually growing. VP8 was developed by Google as a component of the H.264 video codec. The codec is free, open-source, and provides great compression efficiency. VP9 is a no-cost, open-source codec created by Google as the next version after VP8. The technology provides excellent compression efficiency and works with a variety of devices.

Framework	**Codec**	**CPU [%]**		**RAM [%]**		**Throughput KB/s**	
		RPi 3	RPi 4	RPi 3	RPi 4	RPi 3	RPi 4
VLC	H.264	2.5	47.8	1.3	1.3	2.6	33
	H.265	62.8	75.1	4	4.0	5.4	13.5
	VP8	40.3	72.3	2	3.2	143.8	570.4
	VP9	47.8	48.4	5	6.6	16.3	249
	mjpeg	0.01	0.6	0.6	0.2	39.6	5009.9
FFmpeg	H.264	1.1	16.2	1.2	0.3	5.7	125.3
	H.265	8.5	57.9	0.6	0.6	2.6	33.3
	VP8	27.6	29.4	0.3	0.3	38.1	50.2
	VP9	46.3	46.1	1.3	1.2	3.4	6.8
	mjpeg	31.7	34.2	0.5	0.4	4617.9	4628.4
GStreamer	H.264	9.1	67.8	2.4	0.4	37.7	487
	H.265	31.8	59.6	2.7	0.7	24	91
	VP8	14.2	25	0.5	0.3	18.4	63.2
	VP9	45.8	48.6	1.2	1.2	23.4	43.2
	mjpeg	0.1	4.4	0.4	0.2	—	5019.3

Table 20.1: Average values of the resource consumption on different RPi versions for our measurement trials [67].

The assessment was conducted on two RPi devices: a RPi 3 (v1.2) with 1 GB of RAM and a RPi 4 with 8 GB. Video streams using several codecs and frameworks were run for ten minutes, and cAdvisor statistics were collected for each container. A model railway was utilized to encrypt video streams and record a moving locomotive.

The research illustrates resource utilization for video transmission duration through boxplots. The analysis of Figure 20.1 indicates that the RPi 3 has difficulty achieving consistent encoding at the specified resolution of 640x480 pixels, as evidenced by fluctuations in the boxplots. The H.264 and VP9 codecs ascertain the feasible encoding rate. Conversely, the RPi 4 operates effectively and generates a video stream for nearly all combinations. The VLC[114] framework exhibits the largest footprint, however the disparity between FFmpeg[112] and GStreamer[113] is minimal. The transmission rate of video streams is obscured by the substantial data rate of the MJPEG codec. The RPi 3 inadequately transmits data for nearly all codecs, resulting in an unsatisfactory video stream, whereas the RPi 4 maintains steady streams with adequate data rates, with the exception of the VP9 codec. The mean results for CPU utilization, memory use, and throughput are presented in Table 20.1. The MJPEG encoding exhibits minimal or the lowest CPU use compared to all frameworks. FFmpeg demonstrates optimal performance regarding CPU efficiency and average memory consumption.

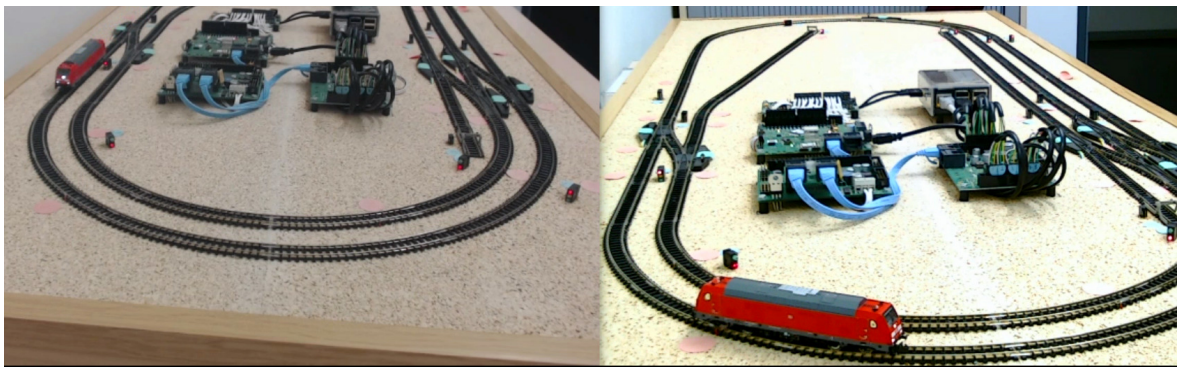
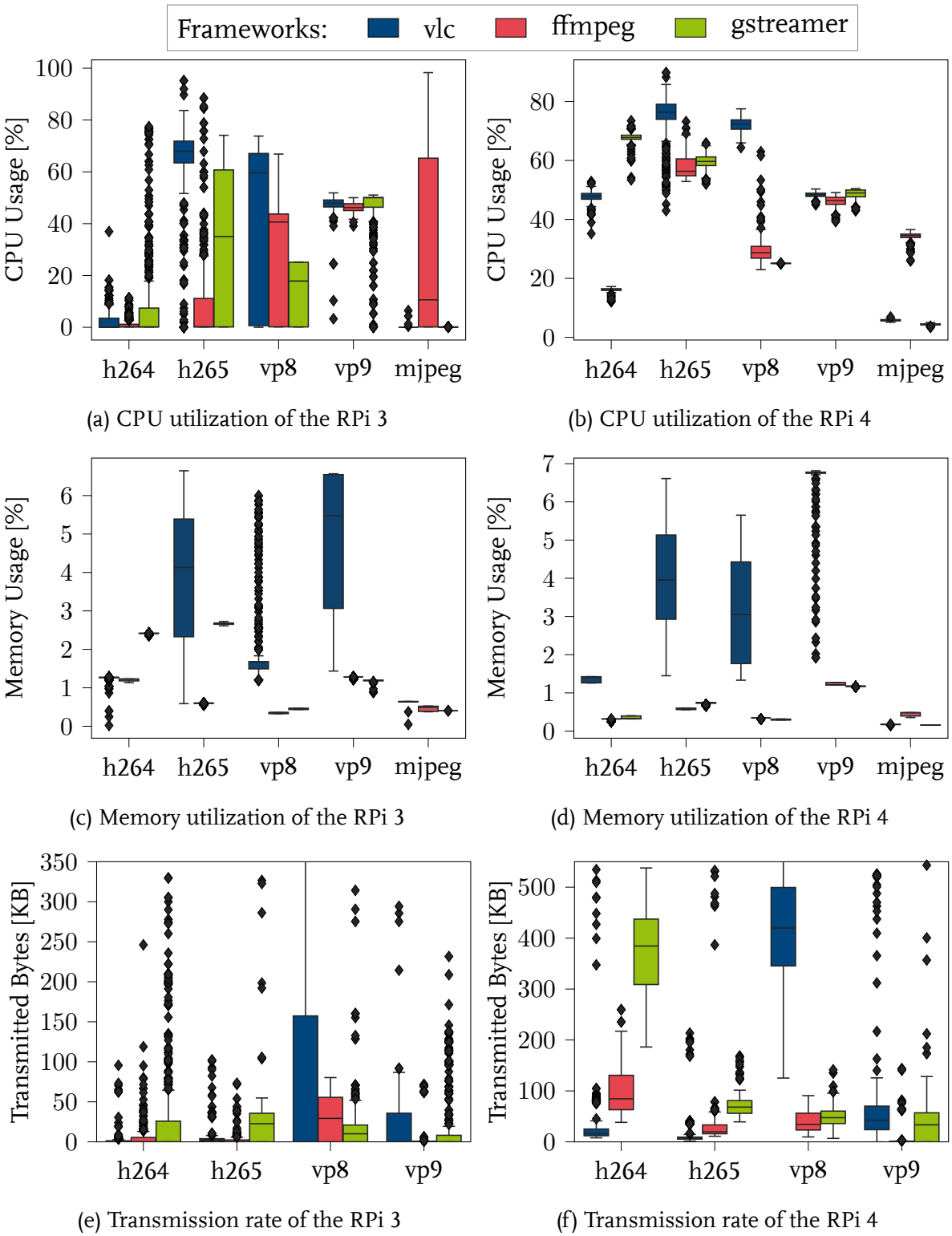


Figure 20.2: A video stream record example for VP9 encoded by GStreamer for evaluating the QoE. On the left side the stream of the RPi 4 is recorded, while the right one shows the live video from a second camera attached to the receiving device [67].



(a) CPU utilization of the RPi 3

(b) CPU utilization of the RPi 4

(c) Memory utilization of the RPi 3

(d) Memory utilization of the RPi 4

(e) Transmission rate of the RPi 3

(f) Transmission rate of the RPi 4

Figure 20.1: Evaluation of the resource consumption for ten minute video streaming using different video codecs on three available encoding frameworks [67].

The QoE was assessed by subjective evaluation and comparative video analysis as shown in Figure 20.2. Latency was quantified by halting and initiating a locomotive, recording the interval in seconds between actual movement and the appearance of the stream. Latency under one second received a rating of ++. Most frameworks successfully streamed satisfactory videos on the RPi 4 for video surveillance. Nevertheless, certain combinations of frameworks and codecs surpassed others for quality and latency. End consumers favored near-real-time transmissions utilizing FFmpeg with H.264, H.265, or VP8, or GStreamer with H.264, owing to anticipated Internet latency.

Framework	**Codec**	**Quality**		**Latency**	
		RPi3	RPi 4	RPi 3	RPi 4
VLC	H.264	−−	++	−−	+
	H.265	−	+	−−	−
	VP8	*o*	*o*	−−	*o*
	VP9	−−	−	−−	−−
	mjpeg	−−	−−	−−	+
FFmpeg	H.264	*o*	++	−−	++
	H.265	−	++	−−	++
	VP8	+	++	*o*	++
	VP9	−−	−	−−	−−
	mjpeg	*o*	++	−−	−
GStreamer	H.264	*o*	++	−	++
	H.265	−	+	−−	+
	VP8	−	+	−	+
	VP9	−−	−	−−	−−
	mjpeg	—	++	—	++

Table 20.2: QoE evaluation of the received video streams [67].
Quality: ++ (very good), + (good), *o* (medium), − (bad), −− (very bad)
Latency: ++ (<1sec), + (1-3sec), *o* (3-6sec), − (6-9sec), −− (>9sec)

20.2 Realization of a Video Surveillance System with WebRTC

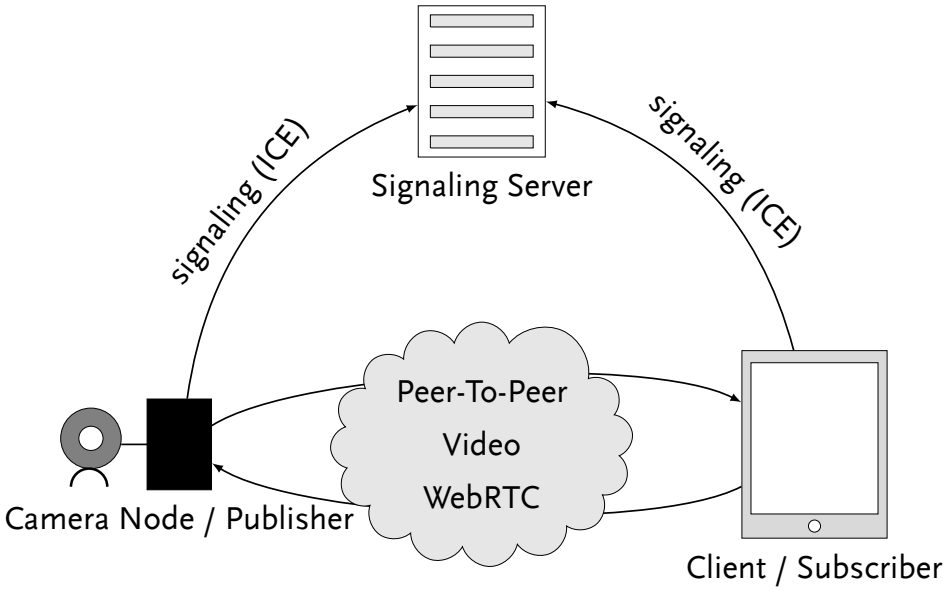


Figure 20.3: High-level architecture [67]

For each component in a standard WebRTC peer connection, the publisher and the subscriber can start a connection and anticipate a response from the other peer. Both peers can join a room in mature implementations, where the room is responsible for maintaining the connections and exchanging Interactive Connection Establishment (ICE) states for both peers. This solution is perfect for situations where both individuals attempt to participate in a video conference and include the media track in the peer connection. In an IoT scenario illustrated in Figure 20.3, the publisher module is an embedded device equipped with a camera module that just records and sends the video. The subscriber module is a lightweight client that solely consumes the video content. Thus, we instructed the subscriber to invoke the `addTransceiver` function on the `RTCPeerConnection` to generate a `RTCRtpTransceiver` object and include it in the peer connection's list of transceivers.

The publish-subscribe pattern facilitates communication between distinct components in a distributed system. The system comprises three primary elements: publishers, subscribers, and mediators. Subscribers indicate their preferences for particular events or event categories to the mediator, who informs them about the events released by the publishers.

The publish-subscribe pattern has attracted interest from both industry and researchers, resulting in the development of numerous innovative prototypes and modifications of the conventional method. We drew inspiration from the classic publish-subscribe paradigm for our prototype, although we did not adopt it entirely.

The publisher is a component that captures and generates a video stream, offering this video track for subscription. The mediator presents the publisher with a proposal from a possible subscriber, to which the publisher replies with its response and the video stream.

The requirements for a publisher node are to have minimal utilization of resources and be compatible with single-board computers that have a camera module. The library `aiortc` [116] implements the WebRTC[117] specification and provides an API that is utilized in our prototype.

Python was selected as the programming language for the publisher service due to its speed, user-friendly nature, and compatibility with resource-limited devices, typically operating on lightweight Linux variants. Python scripts can be executed on Linux shell with minimal setup.

The `aiortc`[116] library utilizes `PyAV`[118], a Python interface for `FFmpeg`[112], a versatile framework for processing audio and video content. `PyAV`[118] provides access to many types of media such as containers, frames, and streams, allowing for convenient manipulations that create an abstraction layer above FFmpeg[112].

The subscriber is a component of a IoT application that consumes the media stream provided by the publisher. It can manifest in different forms, such as web browsers or smartphones. The subscriber must support the WebRTC[117] specification stack for signaling techniques using the Session Description Protocol (SDP).

The subscriber module is a web page hosted by a web application for simplicity. An asynchronous server-side JavaScript application requires a framework such as Node.JS[119], which is an event-driven framework that utilizes non-blocking I/O and considers the event loop as a runtime tool. The program largely depends on callback functions and ends the event loop when there are no more callback functions to be executed.

[116] https://github.com/aiortc/aiortc
[117] https://webrtc.org
[118] https://github.com/PyAV-Org/PyAV
[119] https://nodejs.org

Node.JS[119] is adept and effective for data intensive real-time applications, enabling servers to maintain many connections while handling concurrent queries. A centralized repository, like **npm**, is essential for managing a collection of valuable packages.

The mediator serves as a middleman between the publisher and subscriber components in the classic publish-subscribe messaging design, making it the third fundamental building piece in the implementation. The subscriber creates a session description known as an "offer", which indicates its desire to use the service provided by the publisher and specifies its preferred settings for connectivity and media consumption. The mediator locates the suitable publisher and sends the session description provided by the subscriber, resulting in an acknowledgment known as an "answer" session description.

The mediator facilitates a P2P connection without requiring direct communication between the publisher and subscriber. The implementation maintains loose coupling, scalability, and security characteristics. The mediator must host a web application, provide communication channels for the publisher and subscriber, and handle several subscribers at the same time. Hence, Node.JS[119] is a suitable framework for implementing the mediator.

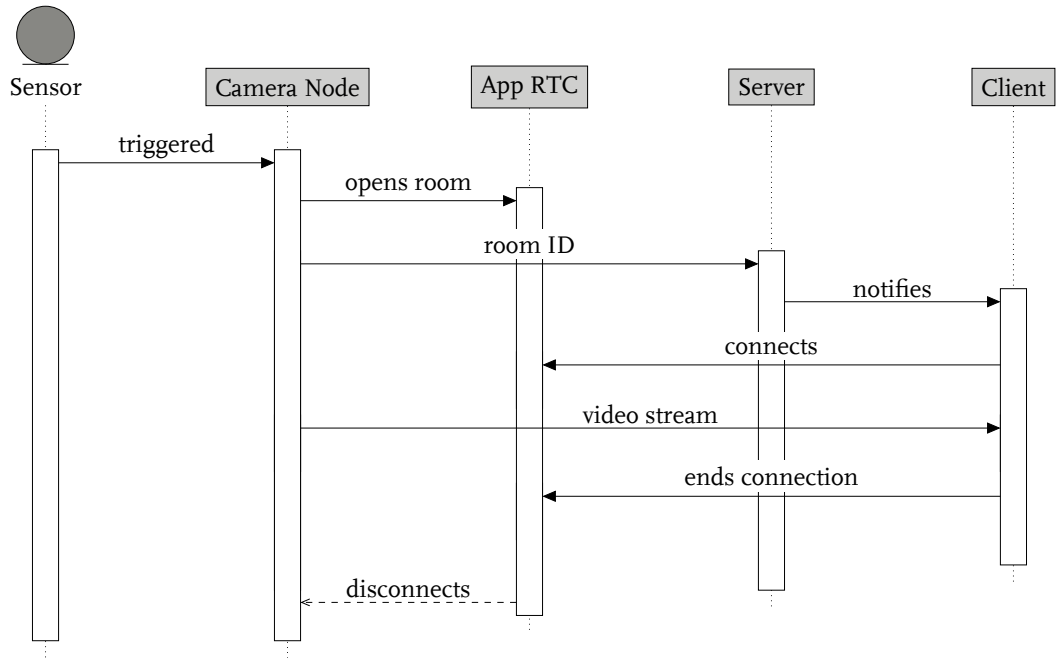


Figure 20.4: Surveillance System [67]

Our system prototype is constructed using Docker and Docker Compose to facilitate the deployment and management of dependencies. The project is accessible on Github at *uniba-ktr/iotrtc*[120] and is compiled using Github Actions. Both the camera and essential server functions are fully containerized and can operate on devices with little setup.

The system operates by deploying one or more camera nodes and connecting a sensor to them. Upon activation, the camera establishes a connection with a new room and transmits the room ID to the signaling server. Subscribed customers receive notifications and have access to watch the stream.

A camera node incorporates a RPi 4, a USB-connected camera, and a sensor attached to the GPIO pins of the RPi. Upon detecting a trigger on a specified pin, the node will automatically start a video connection. This behavior is facilitated by Node-RED[121], simplifying the configuration of the device.

The signaling server component of our prototype delivers the web app to clients and manages notifications. A Node.js[119] server hosts all static HTML files and offers an API for sending notifications to subscribing clients. Currently, only one device can subscribe to the prototype and receive notifications for all cameras. However, this functionality can be expanded to allow for multiple subscriptions and subscriptions to certain cameras.

The Web Push API integrated into all major contemporary browsers has established a standard for managing alerts on the web. A webpage can implement a "Service Worker", which is a small JavaScript file that operates in the background and is triggered whenever a new notification is ready.

The server application provides a single page to clients for subscribing to the service. The server offers a basic API for receiving messages from camera nodes and forwarding them to users. Browsers will only activate a web app's service worker if the connection is encrypted for security reasons.

[120]https://github.com/uniba-ktr/iotrtc
[121]https://nodered.org

PART VI

Conclusion and Future Work

How to play music may be known. At the commencement of the piece, all the parts should sound together. As it proceeds, they should be in harmony while severally distinct and flowing without break, and thus on to the conclusion.

CONFUCIUS

Parts of this chapter are taken from [43], [44], [76], [108]

Finally, we conclude our findings in Chapter 21. Beginning with the obstacles of service distribution in Section 21.1, we illustrate recent advancements and demonstrate how our study offers contributions to this field. Additionally, we examine our first solution for service orchestration in Section 21.2 within the Cloud-to-Fog-Continuum. Consequently, service monitoring is critically important for it, as described in Subsection 21.2.1. Furthermore, we ascertain in Section 21.3 how network emulation facilitates the development of novel paradigms, hence enhancing the value of virtualized services across any continuum.

In Chapter 22, existing shortcomings and prospective enhancements are examined. All network emulations establish a basis for future integrations into real-world systems that have yet to be constructed for testing. Our FL orchestrator now demonstrates a functional prototype on a single machine and requires service distribution and comprehensive testing in the future, potentially through emulation, to be adapted to meet the requirements of the Cloud-to-Fog-Continuum.

Chapter 21
Conclusion

We reexamine the research questions outlined in Chapter 2 and provide a conclusion to them pointing at our work. Modern communication infrastructures, as depicted in Figure 1.1, provide new issues for service distribution.

First we investigate, how modern technologies influence the ability to enable service distribution on the Cloud-to-Fog-Continuum in Section 21.1. With the integration of resource monitoring on resource-constrained devices in Section 21.2 we show the functionality of a orchestration framework in Section 16.1 in our distributed domain. Finally, we show how current emulation paradigms help to develop new protocols for applications and increase the robustness of networks in Section 21.3

21.1 Challenges through Service Distribution

In Chapter 5 we recognized that in the early 2010s, CC emerged as a popular trend, with companies like Google and Amazon introducing centralized CC models. However, the IoT has led to the need for a computing model that supports features such as mobility, geo-distribution, and location awareness demanded by IoT. FC, an extension of the CC model, provides compute, data, and storage services closer to requesters, offering location awareness, mobility support, and low latency.

The IoT devices also demand a lot of bandwidth, making it difficult to facilitate time-critical operations like eHealth and Ambient Assisted Living (AAL). Traditional networking systems are built on traditional infrastructures, which include dedicated hardware devices like ASICs. In contrast, SDN is an innovative archi-

tecture that enables network system management through software applications, decoupling traffic engineering and network configuration from networking hardware pieces.

Moreover, virtualization has emerged as another enabler for the Cloud-to-Fog-Continuum. It allows IT organizations to optimize server utilization and perform vertical scaling depending on user demands, resulting in both operation and economic benefits. However, VMs have disadvantages, especially when applied in the fog environment, such as high system resources, long deployment time, and considerable management overhead. FC requires a more lightweight, versatile virtualization technology called "containerization". Containers have a smaller footprint, are faster to launch, are quicker to scale, and are more suitable in the fog paradigm than VMs.

Conventional container orchestrator tools like Docker Swarm or advanced ones like Google Kubernetes or Apache Mesos have not fully integrated into the fog environment, and there is a necessity for a futuristic container orchestrator that can promptly and functionally manage containers running in multiple geographically distributed fog nodes, collaborating with SDN to control network behaviors based on real-time monitoring data and application demands.

To answer Research Question 1, we see that current developments support the distribution of services along the path from the service provider to the service consumer, paving the way to reduce the bottlenecks of the network and diminishing the transmission delay. Advancements in the virtualization technologies and the support of SBCs, which we developed in [70], as a predecessor of Docker Swarm, is enabled to integrate IoT devices for service placement. We demonstrate a multi-architecture service orchestration for example with SensIoT [65]. However, we showed by using a serverless paradigm in [66] that further benefits are not recognizable, since the underlying network is not taken into the service routing considerations. To be more flexible in this regard, a virtualized network is a necessity. With our concept of cloudless computing [74], we described the foundation, how service placement is achievable in a SDN network enhanced by VNFs.

21.2 Realization of Service Orchestration

To answer Research Question 3, we need to improve the cloudless computing concept of [74]. Firstly, the connection of Docker containers and OvS is explained in Chapter 13, where we implemented the corresponding driver and depicted its functionality. It is the foundation, that new containers can be added to a topology by a central orchestration unit - the SDN controller. In the next step, we chose a SDN controller, that is extensible - called ONOS - and provided a toolchain in a CI manner to easily develop new plugins. Our plugin enables ONOS to place containers in a network and to configure the routing tables accordingly as described in [74]. Especially, for the Cloud-to-Fog-Continuum divergence of devices it is a necessity to setup CI automation to simplify compiling services for different platforms. We already used it for ONOS in [73] to build our plugins for two infrastructures, namely amd64 and arm64v8, and deliver the corresponding Docker images. Based on these two improvements, a service orchestration framework can be developed, where resource monitoring is the first necessity.

21.2.1 Cloudless Resource Monitoring

Therefore, the work on the CRM focused on the development of a context-aware system, consisting of two multi-processes subsystems: “Collector” and “Worker”. The CRM application uses a Docker monitoring framework and a custom ONOS plugin called CRMA to manage, monitor, and report resource consumption of all hosts within an SDN as shown in Chapter 15. However, the CRM system faces several challenges, including the lack of multi-architecture support for required third-party libraries in ONOS. Currently, the CRM system works on two major CPU architectures: amd64 and arm64v8. The CRM tool is operational, allowing system administrators to observe the current utilization level of primary system resources in real-time. However, there are several problems that need to be addressed. The ONOS plugin does not yet support a GUI for users to interact with it, which limits the capabilities of the MAPE-K principle orchestrator tool. The CRM system also has a clear separation of responsibility between “Collector” and “Worker” systems, with “Collector” managing multiple containers and “Worker” managing multiple hosts.

In conclusion, the proposed CRM system [108] provides a solid foundation for creating an intelligent, context-aware container orchestrator tool useful not only in the Cloud-to-Fog-Continuum but also in other paradigms.

By integrating the CRM middleware, we can generate the needed data for a service orchestration layer. It is the foundation to cope with the changes in the Cloud-to-Fog-Continuum.

21.2.2 Service Orchestration with Federated Learning

Chapter 16 introduces a federated autonomous Docker container orchestrator architecture in FC systems using SDN as described in [43], [44]. The architecture consists of three parts: the cloud layer, the fog layer, and the edge layer, which incorporate the MAPE-K paradigm for autonomous computing. The cloud layer analyzes and detects events like high load, container failures, and anomalies, while the fog layer handles these events, groups nodes into hierarchies, and deploys new containers. The edge layer hosts deployed apps and sends monitoring events to the fog layer, which will update the corresponding cloud services. The research aims to evaluate the MAPE-K approach for container orchestration, for handling dynamic events, for supporting a high number of nodes, and for smartly deploying containers. The cloud analysis layer supports node failures, container failures, anomalies, taints, and load detection, while the node scoring functionality is faster than Kubernetes'. FL is used to estimate required container resources before container deployment using a radar chart given by the application provider.

The prototype implementation is still too heavy-weight and needs performance modifications before it can be used effectively on real devices. Node scoring due to parallelization is fast and distributed node scoring is feasible for real scenarios. The system should weaken resource estimation model predictions at the beginning of the orchestrator life cycle due to low accuracy, but over time, the model estimation's weakening must be reduced.

However, to operate programs in a distributed environment, some constraints must be considered and rigorously validated. Revisiting Research Question 2, we examine how emulation can facilitate the development of new paradigms for service orchestration in the Cloud-to-Fog-Continuum.

21.3 Network Emulation Tools

For answering Research Question 2, we have a look at the different concepts we integrated into Kathará. New transmission paradigms can easily be emulated and programmed, by utilizing emulated switches, which are supposed to be present in future network topologies. In our example [71], we improved the network by using P4 and executed own implementations of multi-path paradigms directly on layer 2 of the OSI stack from Chapter 7.

In our "clone filter" emulation [71] we represent a modest advancement in facilitating dependable and rapid packet transmission across a network, both for UDP and TCP packets. However, still in a prototypical state, features like load balancing, congestion control, and automatic packet re-routing in the event of a switch failure are required. At now, all configurations are executed manually, and in the event of a switch failure, it will continue to attempt to transmit the packet through the compromised route. The prototype functions effectively with the Kathará emulator, where packet deduplication utilizing hash generation over a packet is effective and readily adaptable for various protocols. Nonetheless, the switch must include a register, and there exists a possibility of hash collisions contingent upon the hash method employed.

As another example, we presented an easy-to-use, emulated SDN environment with automated scripts for DDoS simulation in [68], [79]. Two different DDoS detection mechanisms were executed on different levels of the SDN architecture and compared against each other. Network monitoring in real-time and an implementation simplifying dashboard creation for forensic network analysis were shown. The simulation testbed can be executed on commodity hardware, except that Snort[122], Prometheus[83], and Grafana[85] might overstrain the system when launched at the same time. The experiment runs smoothly without unexpected hardware behavior, and a kill-switch is implemented in the scripts.

The simulation testbed does not disrupt external network services or escape the testing environment, making it relevant for educational and research purposes where less resources are available. Different types of attacks and their impacts on different layers and devices of the SDN network are illustrated. Half-distributed attacks have a greater impact than fully distributed attacks, and the Smurf attack

[122]https://www.snort.org

has shown interesting behavior that has not yet been analyzed in literature.

Two types of overwhelming high-rate attacks can be detected with easily implemented threshold mechanisms, with the exception of Local Area Network Denial (LAND) attacks. However, threshold mechanisms are not suitable for a large range of DDoS attacks. The differences between eight different types of DDoS attacks, as well as the variants of half-distributed, fully-distributed, spoofed, and unspoofed, are discussed.

In summary, we provided a comprehensive overview of DDoS attacks and possible detection strategies using Kathará, a flexible, easy-to-replicate, and scalable solution made from open source software and frameworks. The solution takes various types of DDoS attacks and variations into account. Real-time monitoring with sFlow-RT[58] and adaptable visualization in Grafana[85] ensure usability.

With the network emulation approach, we can clearly see that alternative transmission paradigms can be developed and tested against containerized infrastructures. Even the recognition of DDoS attacks shows, that the infrastructure is capable to provide resilient services by investigating the currently routed traffic.

Chapter 22
Future Work

In the future, we want to enhance the simplicity of network emulation using our framework [76] and demonstrate prospective enhancements in Section 22.1. Observing the Cloud-to-Fog-Continuum orchestration, the FL orchestrator presently serves as a proof of concept, which can be enhanced as demonstrated in Section 22.2.

22.1 Improvements for Network Emulation

Let us first discuss improvements for the network emulation, which is the foundation to generate new transmission paradigms and to test services with an emulated scenario.

With [76] we developed a framework to easily setup emulated network environments. Based on Kathará every containerized micro-services can be added as a container to an emulation. However, the system is currently running on a single machine, where every user is working on it's own emulated sandbox realized by a DinD daemon as described in Section 11.1. To improve it, we plan to integrate Docker Swarm or even Kubernetes as a backbone for the network emulation service to be able to cope with the requirements of complicated networks and resource intensive services.

Our first test scenarios showed the capability to run DDoS attacks against the ONOS controller and can help to improve it's resilience. It provides a foundation to implement attack recognition frameworks within the network control unit and increase the overall network availability.

In our second scenario, a P4 design was emulated to demonstrate, how easily new virtualized transmission paradigms can be integrated into a network. With a programmable switch it is possible to improve the traffic between services and add additional features, like multi-path transmissions for either faster network speed or duplication principles directly on layer 2. In the future, this prototype may also interact with ONOS directly, to setup the path between services with different QoS levels needed in the Cloud-to-Fog-Continuum to support deployed applications.

22.2 Optimization for the Federated Machine Learning Orchestrator

Besides of the optimizations concerning the single modules of the service orchestrator to run efficiently, we will look at a few extensions for the existing framework. On the one hand, each micro-service is setup by a resource estimation, which the service provider actually is asked to provide. A system for micro-service evaluation can be an improvement as shown in Subsection 22.2.1. On the other hand, the deployment is currently done via a GUI running at the ONOS controller. An implementation of a REST API and a service description language for Cloud-to-Fog-Continuum deployments improves an automated deployment process, which we discuss in Subsection 22.2.2.

22.2.1 Evaluation System for Micro-services

Currently, the FL process in our orchestration framework depends on the knowledge of service's resource estimation, which is actually setup by the service provider with a chart as shown in Figure 16.3.

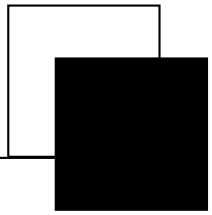
Here, an estimation framework could provide the necessary evaluation of a service to find the initially needed data for the learning process. For the estimation our CRM tool can be utilized together with the network emulation tools of Chapter 11. In such a sandboxed environment a containerized service can be deployed and stress-tested to find it's consumption data.

22.2.2 Service Description for Network Deployments

Another improvement for the service provider would include utilizing the northbound API of the ONOS SDN controller. With it a REST interface can be implemented, where a common service description language can define the requirements of a service deployment.

In a description model, the service requirements need to be defined, such that it fits to a Cloud-to-Fog-Continuum deployment. The service provider should be enabled to easily define the QoS agreements, which should include the distribution requirements.

Bibliography



[1] S. Ahmad and A. H. Mir, "Scalability, Consistency, Reliability and Security in SDN Controllers: A Survey of Diverse SDN Controllers", en, *Journal of Network and Systems Management*, vol. 29, no. 1, p. 9, Jan. 2021, ISSN: 1064-7570, 1573-7705. DOI: 10.1007/s10922-020-09575-4. [Online]. Available: http://link.springer.com/10.1007/s10922-020-09575-4.

[2] N. Ahuja, G. Singal, D. Mukhopadhyay, and N. Kumar, "Automated DDoS Attack Detection in Software Defined Networking", en, *Journal of Network and Computer Applications*, vol. 187, p. 103 108, Aug. 2021, ISSN: 10848045. DOI: 10.1016/j.jnca.2021.103108.

[3] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications", *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015, ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2444095.

[4] N. J. Alpern and R. J. Shimonski, "CHAPTER 5 - The OSI Model and Networking Protocols", in *Eleventh Hour Network+*, N. J. Alpern and R. J. Shimonski, Eds., Boston: Syngress, 2010, pp. 73–88, ISBN: 978-1-59749-428-1. DOI: https://doi.org/10.1016/B978-1-59749-428-1.00006-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781597494281000060.

[5] E. Androulaki, A. Barger, V. Bortnikov, *et al.*, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains", in *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-*

26, 2018, R. Oliveira, P. Felber, and Y. C. Hu, Eds., ACM, 2018, 30:1–30:15. DOI: 10.1145/3190508.3190538. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190538.

[6] Łukasz Apiecionek, M. Großmann, and U. R. Krieger, "Harmonizing IoT-Architectures with Advanced Security Features - A Survey and Case Study", *JUCS - Journal of Universal Computer Science*, vol. 25, no. 6, pp. 571–590, 2019, ISSN: 0948-695X. DOI: 10.3217/jucs-025-06-0571. eprint: https://doi.org/10.3217/jucs-025-06-0571. [Online]. Available: https://doi.org/10.3217/jucs-025-06-0571.

[7] F. Arena, G. Pau, and A. Severino, "An Overview on the Current Status and Future Perspectives of Smart Cars", *Infrastructures*, vol. 5, no. 7, p. 53, 2020.

[8] M. Armbrust, A. Fox, R. Griffith, *et al.*, "A view of Cloud Computing", *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[9] N. Aslam, S. Srivastava, and M. M. Gore, "ONOS Flood Defender: An Intelligent Approach to Mitigate DDoS Attack in SDN", en, *Transactions on Emerging Telecommunications Technologies*, vol. 33, no. 9, e4534, Sep. 2022, ISSN: 2161-3915, 2161-3915. DOI: 10.1002/ett.4534.

[10] I. Azimi, A. Anzanpour, A. M. Rahmani, *et al.*, "HiCH: Hierarchical Fog-assisted Computing Architecture for Healthcare IoT", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 1–20, 2017.

[11] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "On Orchestrating Virtual Network Functions", in *Network and Service Management (CNSM), 2015 11th International Conference on*, IEEE, 2015, pp. 50–56.

[12] N. Z. Bawany, J. A. Shamsi, and K. Salah, "DDoS Attack Detection and Mitigation Using SDN: Methods, Practices, and Solutions", en, *Arabian Journal for Science and Engineering*, vol. 42, no. 2, pp. 425–441, Feb. 2017, ISSN: 2193-567X, 2191-4281. DOI: 10.1007/s13369-017-2414-5.

[13] P. Berde, M. Gerola, J. Hart, *et al.*, "ONOS: Towards an Open, Distributed SDN OS", in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 1–6, ISBN: 9781450329897. DOI: 10.1145/2620728.2620744. [Online]. Available: https://doi.org/10.1145/2620728.2620744.

[14] A. Bessani, J. Sousa, and E. E. Alchieri, "State Machine Replication for the Masses with BFT-SMaRt", in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, IEEE, 2014, pp. 355–362.

[15] A. Bianco, P. Giaccone, R. Mashayekhi, M. Ullio, and V. Vercellone, "Scalability of ONOS reactive forwarding applications in ISP networks", *Computer Communications*, vol. 102, pp. 130–138, 2017.

[16] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies", in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 104–121. DOI: 10.1109/SP.2015.14. [Online]. Available: https://doi.org/10.1109/SP.2015.14.

[17] G. Bonofiglio, V. Iovinella, G. Lospoto, and G. Di Battista, "Kathará: A Container-based Framework for implementing Network Function Virtualization and Software Defined Networks", in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1–9. DOI: 10.1109/NOMS.2018.8406267.

[18] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and its Role in the Internet of Things", in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, ACM, 2012, pp. 13–16.

[19] P. Bosshart, D. Daly, G. Gibb, *et al.*, "P4: Programming Protocol-Independent Packet Processors", *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014, ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. [Online]. Available: https://doi.org/10.1145/2656877.2656890.

[20] C. Bouras, A. Kollia, and A. Papazois, "Teaching Network Security in Mobile 5G using ONOS SDN Controller", in *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, Milan: IEEE, Jul. 2017, pp. 465–470, ISBN: 978-1-5090-4749-9. DOI: 10.1109/ICUFN.2017.7993828.

[21] L. Breiman, "Random Forests", *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001, ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. [Online]. Available: https://doi.org/10.1023/A:1010933404324.

[22] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and regression trees*. Routledge, 2017. DOI: 10.1201/9781315139470. [Online]. Available: https://doi.org/10.1201/9781315139470.

[23] V. Buterin, *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*, [Online; accessed 22. Nov 2017], 2013. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper.

[24] R. Buyya and S. N. Srirama, *Fog and Edge Computing: Principles and Paradigms*. John Wiley & Sons, 2019.

[25] J. P. Buzen and U. O. Gagliardi, "The Evolution of Virtual Machine Architecture", in *Proceedings of the June 4-8, 1973, national computer conference and exposition*, ACM, 1973, pp. 291–299.

[26] C. Cachin and M. Vukolić, "Blockchains Consensus Protocols in the Wild", *arXiv preprint arXiv:1707.01873*, 2017. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/8016/pdf/LIPIcs-DISC-2017-1.pdf.

[27] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", in *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, 1999, pp. 173–186. DOI: 10.1145/296806.296824. [Online]. Available: http://doi.acm.org/10.1145/296806.296824.

[28] H. L. Cech, M. Großmann, and U. R. Krieger, "A Fog Computing Architecture to Share Sensor Data by Means of Blockchain Functionality", in *2019 IEEE International Conference on Fog Computing (ICFC)*, Jun. 2019, pp. 31–40. DOI: 10.1109/ICFC.2019.00013.

[29] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System", in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794, ISBN: 978-14-50342-32-2. DOI: 10.1145/2939672.2939785. [Online]. Available: https://doi.org/10.1145/2939672.2939785.

[30] Y. Chen, D. Murherjee, J. Han, *et al.*, "An Overview of Core Coding Tools in the AV1 Video Codec", in *2018 Picture Coding Symposium (PCS)*, 2018, pp. 41–45. DOI: 10.1109/PCS.2018.8456249.

[31] M. Chiang and T. Zhang, "Fog and IoT: An Overview of Research Opportunities", *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, Dec. 2016, ISSN: 2327-4662. DOI: 10.1109/JIOT.2016.2584538.

[32] L. Chou, Z. Liu, Z. Wang, and A. Shrivastava, "Efficient and Less Centralized Federated Learning", *arXiv preprint arXiv:2106.06627*, 2021.

[33] K. Christidis and M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things", *IEEE Access*, vol. 4, pp. 2292–2303, 2016. DOI: 10.1109/ACCESS.2016.2566339. [Online]. Available: https://doi.org/10.1109/ACCESS.2016.2566339.

[34] "Cloud Service Models", in *Architecting The Cloud*. John Wiley & Sons, Ltd, 2014, ch. 2, pp. 13–22, ISBN: 9781118691779. DOI: https://doi.org/10.1002/9781118691779.ch2. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118691779.ch2. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118691779.ch2.

[35] A. Colantoni, L. Berardinelli, and M. Wimmer, "DevopsML: Towards Modeling DevOps Processes and Platforms", in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2020, pp. 1–10.

[36] I. A. Computing, *White Paper: An Architectural Blueprint for Autonomic Computing*, 2005.

[37] M. Conoscenti, A. Vetrò, and J. C. De Martin, "Blockchain for the Internet of Things: A systematic literature review", 2016.

[38] P. H. B. Correia, M. A. Marques, M. A. Simplicio, L. Ermlivitch, C. C. Miers, and M. A. Pillon, "Comparative Analysis of Permissioned Blockchains: Cosmos, Hyperledger Fabric, Quorum, and XRPL", in *2024 IEEE International Conference on Blockchain (Blockchain)*, IEEE, 2024, pp. 464–469.

[39] F. S. Dantas Silva, E. Silva, E. P. Neto, M. Lemos, A. J. Venancio Neto, and F. Esposito, "A Taxonomy of DDoS Attack Mitigation Approaches Featured by SDN Technologies in IoT Scenarios", en, *Sensors*, vol. 20, no. 11, p. 3078, May 2020, ISSN: 1424-8220. DOI: 10.3390/s20113078.

[40] S. Das and K. Sarac, "Practical Labs for Teaching SDN Security", *Journal of The Colloquium for Information Systems Security Education*, vol. 10, no. 1, p. 7, Mar. 2023, ISSN: 2641-4554, 2641-4546. DOI: 10.53735/cisse.v10i1.166.

[41] N. Dayal and S. Srivastava, "Analyzing Behavior of DDoS Attacks to identify DDoS Detection Features in SDN", in *2017 9th International Conference on Communication Systems and Networks (COMSNETS)*, Bengaluru, India: IEEE, Jan. 2017, pp. 274–281, ISBN: 978-1-5090-4250-0. DOI: 10.1109/COMSNETS.2017.7945387.

[42] J. Dogani and F. Khunjush, "Proactive Auto-Scaling Technique for Web Applications in Container-Based Edge Computing using Federated Learning Model", *Journal of Parallel and Distributed Computing*, vol. 187, p. 104 837, 2024, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2024.104837. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731524000017.

[43] M. Dworzak, M. Großmann, and D. T. Le, "Federated Autonomous Orchestration in Fog Computing Systems", in *Proceedings of Eighth International Congress on Information and Communication Technology*, X.-S. Yang, R. S. Sherratt, N. Dey, and A. Joshi, Eds., Singapore: Springer Nature Singapore, 2024, pp. 639–649, ISBN: 978-981-99-3236-8.

[44] M. Dworzak, M. Großmann, and D. T. Le, "Federated Learning for Service Placement in Fog and Edge Computing", p. 4, 2023. DOI: 10.25972/OPUS-32219.

[45] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "DevOps", *IEEE Software*, vol. 33, no. 3, pp. 94–100, 2016. DOI: 10.1109/MS.2016.68.

[46] A. Eiermann, M. Renner, M. Großmann, and U. R. Krieger, "On a Fog Computing Platform Built on ARM Architectures by Docker Container Technology", in *Innovations for Community Services*, G. Eichler, C. Erfurth, and G. Fahrnberger, Eds., Cham: Springer International Publishing, 2017, pp. 71–86, ISBN: 978-3-319-60447-3.

[47] P. M. Eittenberger and U. R. Krieger, "Performance Evaluation of Forward Error Correction Mechanisms for Android Devices Based on Raptor Codes", in *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, K. Fischbach and U. R. Krieger, Eds., Cham:

Springer International Publishing, 2014, pp. 103–119, ISBN: 978-3-319-05359-2.

[48] L. F. Eliyan and R. Di Pietro, "DoS and DDoS Attacks in Software Defined Networks: A Survey of existing Solutions and Research Challenges", en, *Future Generation Computer Systems*, vol. 122, pp. 149–171, Sep. 2021, ISSN: 0167739X. DOI: 10.1016/j.future.2021.03.011.

[49] Y. Elkhatib, B. Porter, H. B. Ribeiro, M. F. Zhani, J. Qadir, and E. Rivière, "On Using Micro-Clouds to Deliver the Fog", *IEEE Internet Computing*, vol. 21, no. 2, pp. 8–15, 2017.

[50] L. Espe, A. Jindal, V. Podolskiy, and M. Gerndt, "Performance Evaluation of Container Runtimes", in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*, Prague, Czech Republic: SCITEPRESS - Science and Technology Publications, 2020.

[51] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures", in *Proceedings of the 2nd International Workshop on Serverless Computing*, ser. WoSC '17, Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 1–4, ISBN: 9781450354349. DOI: 10.1145/3154847.3154848. [Online]. Available: https://doi.org/10.1145/3154847.3154848.

[52] D. Farinacci, V. Fuller, D. Meyer, D. Lewis, *et al.*, "Locator/ID Separation Protocol (LISP)", *Internet-dra, Dec*, 2013.

[53] C. Feller, J. Wuenschmann, T. Roll, and A. Rothermel, "The VP8 video codec - overview and comparison to H.264/AVC", in *2011 IEEE International Conference on Consumer Electronics -Berlin (ICCE-Berlin)*, 2011, pp. 57–61. DOI: 10.1109/ICCE-Berlin.2011.6031852.

[54] O. K. Ferstl and E. J. Sinz, *Grundlagen der Wirtschaftsinformatik*, 6th ed. München: Oldenbourg, 2008, ISBN: 9783486587555. [Online]. Available: http://www.gbv.de/dms/zbw/582128692.pdf.

[55] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker, "Frenetic: A high-level Language for OpenFlow Networks", in *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, ser. PRESTO '10, Philadelphia, Pennsylvania: Association for

Computing Machinery, 2010, ISBN: 9781450304672. DOI: 10.1145/1921151.1921160. [Online]. Available: https://doi.org/10.1145/1921151.1921160.

[56] M. Fowler and M. Foemmel, *Continuous Integration (Original Version)*, Sep. 2000. [Online]. Available: https://martinfowler.com/articles/originalContinuousIntegration.html.

[57] W. Fuertes, A. Tunala, R. Moncayo, F. Meneses, and T. Toulkeridis, "Software-Based Platform for Education and Training of DDoS Attacks Using Virtual Networks", in *2017 International Conference on Software Security and Assurance (ICSSA)*, Altoona, PA: IEEE, Jul. 2017, pp. 94–99, ISBN: 978-1-5386-4808-7. DOI: 10.1109/ICSSA.2017.19.

[58] J. Gedeon, J. Heuschkel, L. Wang, and M. Mühlhäuser, "Fog Computing: Current Research and Future Challenges", *KuVS-Fachgespräch Fog Computing*, vol. 1, pp. 1–4, 2018.

[59] M. Gerwien, M. Großmann, and U. R. Krieger, "The System Architecture of a Reliable Telesurgery Service and its Performance Analysis", in *Innovations for Community Services*, F. Phillipson, G. Eichler, C. Erfurth, and G. Fahrnberger, Eds., Cham: Springer Nature Switzerland, 2024, pp. 257–274, ISBN: 978-3-031-60433-1.

[60] M. Glauert, "Klimaregulierung in Bibliotheksmagazinen", in P. Hauke and K. Werner, Eds. Humboldt-Universität zu Berlin, Philosophische Fakultät I, Institut für Bibliotheks- und Informationswissenschaft, 2009, pp. 158–173. DOI: http://dx.doi.org/10.18452/2177.

[61] R. P. Goldberg, "Survey of Virtual Machine Research", *Computer*, vol. 7, no. 6, pp. 34–45, 1974, ISSN: 0018-9162. DOI: 10.1109/MC.1974.6323581.

[62] G. Goth, "Software-Defined Networking Could Shake Up More than Packets", *IEEE Internet Computing*, vol. 15, no. 4, pp. 6–9, 2011. DOI: 10.1109/MIC.2011.96.

[63] G. Greenspan, *MultiChain Private Blockchain — White Paper*, 2015. [Online]. Available: https://www.multichain.com/download/MultiChain-White-Paper.pdf.

[64] M. Großmann, S. Illig, and C. Matějka, "Environmental Monitoring of Libraries with MonTreAL", in *Research and Advanced Technology for Digital Libraries*, J. Kamps, G. Tsakonas, Y. Manolopoulos, L. Iliadis, and I. Karydis, Eds., Cham: Springer International Publishing, 2017, pp. 599–602, ISBN: 978-3-319-67008-9.

[65] M. Großmann, S. Illig, and C. L. Matějka, "SensIoT: An Extensible and General Internet of Things Monitoring Framework", *Wireless Communications and Mobile Computing*, vol. 2019, p. 4 260 359, Mar. 2019, ISSN: 1530-8669. DOI: 10.1155/2019/4260359. [Online]. Available: https://doi.org/10.1155/2019/4260359.

[66] M. Großmann, C. Ioannidis, and D. T. Le, "Applicability of Serverless Computing in Fog Computing Environments for IoT Scenarios", in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, ser. UCC '19 Companion, Auckland, New Zealand: Association for Computing Machinery, 2019, pp. 29–34, ISBN: 9781450370448. DOI: 10.1145/3368235.3368834. [Online]. Available: https://doi.org/10.1145/3368235.3368834.

[67] M. Großmann, L. Klinger, V. Krolikowsky, and C. Sarkar, "Efficient Internet of Things Surveillance Systems in Edge Computing Environments", in *Innovations for Community Services*, U. R. Krieger, G. Eichler, C. Erfurth, and G. Fahrnberger, Eds., Cham: Springer Nature Switzerland, 2023, pp. 292–311, ISBN: 978-3-031-40852-6.

[68] M. Großmann and N. Weinmann, "Emulation of Denial-of-Service Attacks for Software Defined Networks", in *Innovations for Community Services*, F. Phillipson, G. Eichler, C. Erfurth, and G. Fahrnberger, Eds., Cham: Springer Nature Switzerland, 2024, pp. 275–285, ISBN: 978-3-031-60433-1.

[69] M. Großmann and A. Eiermann, "Automated Establishment of a Secured Network for Providing a Distributed Container Cluster", in *2016 28th International Teletraffic Congress (ITC 28)*, vol. 01, Sep. 2016, pp. 212–215. DOI: 10.1109/ITC-28.2016.137.

[70] M. Großmann, A. Eiermann, and M. Renner, "Hypriot Cluster Lab : An ARM-Powered Cloud Solution Utilizing Docker", in *Proceedings 23rd International Conference on Telecommunications (ICT 2016), Thessaloniki, Greece.*,

Jahr der Erstpublikation: 2016, 23rd International Conference on Telecommunications (ICT 2016), Thessaloniki, Greece, Bamberg: opus, 2018, p. 2. DOI: 10.20378/irbo-51198.

[71] M. Großmann and T. Homeyer, "Emulation of Multipath Transmissions in P4 Networks with Kathará", p. 4, 2023. DOI: 10.25972/OPUS-32209.

[72] M. Großmann, S. Illig, J. Lappe, and C. Matějka, "Bestandsmonitoring in kulturellen Einrichtungen", *ABI Technik*, vol. 38, no. 4, pp. 344–353, 2018. DOI: doi:10.1515/abitech-2018-4008. [Online]. Available: https://doi.org/10.1515/abitech-2018-4008.

[73] M. Großmann and C. Ioannidis, "Continuous Integration of Applications for ONOS", in *2019 IEEE Conference on Network Softwarization (NetSoft)*, Jun. 2019, pp. 213–217. DOI: 10.1109/NETSOFT.2019.8806696.

[74] M. Großmann and C. Ioannidis, "Cloudless Computing - A Vision to Become Reality", in *2020 International Conference on Information Networking (ICOIN)*, Jan. 2020, pp. 372–377. DOI: 10.1109/ICOIN48656.2020.9016441.

[75] M. Großmann and C. Klug, "Monitoring Container Services at the Network Edge", in *2017 29th International Teletraffic Congress (ITC 29)*, vol. 1, Sep. 2017, pp. 130–133. DOI: 10.23919/ITC.2017.8064348.

[76] M. Großmann and D. T. Le, "Visualization of Network Emulation Enabled by Kathará", p. 4, 2023. DOI: 10.25972/OPUS-32218.

[77] M. Großmann and C. Schenk, "A Comparison of Monitoring Approaches for Virtualized Services at the Network Edge", in *2018 International Conference on Internet of Things, Embedded Systems and Communications (IINTEC)*, Dec. 2018, pp. 85–90. DOI: 10.1109/IINTEC.2018.8695277.

[78] M. Großmann and S. J. Schuberth, "Auto-Mininet: Assessing the Internet Topology Zoo in a Software-defined Network Emulator", in *Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen*, B. E. Wolfinger, Ed., ser. Universität Hamburg. Fachbereich Informatik: Bericht, https://fis.uni-bamberg.de/handle/uniba/47143, 7. GIITG-Workshop MMBnet 2013, 5.6. September 2013. Universität Hamburg, Fachbereich Informatik, Hamburg: Bibliothek des FB Informatik, 2013, pp. 63–72.

[79] M. Großmann and N. Weinmann, "Real-Time Monitoring of Software-Defined Networks in Kathará for Denial-of-Service Attacks", p. 4, 2024. DOI: 10.25972/OPUS-38900.

[80] A. Hamarshe, H. I. Ashqar, and M. Hamarsheh, "Detection of DDoS Attacks in Software Defined Networking Using Machine Learning Models", 2023, Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.2303.06513.

[81] B. Han, S. Wong, C. Mannweiler, M. R. Crippa, and H. D. Schotten, "Context-awareness Enhances 5G Multi-access Edge Computing Reliability", *IEEE Access*, vol. 7, pp. 21 290–21 299, 2019.

[82] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network Function Virtualization: Challenges and Opportunities for Innovations", *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015. DOI: 10.1109/MCOM.2015.7045396.

[83] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible Network Experiments Using Container-Based Emulation", in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12, Association for Computing Machinery, 2012, pp. 253–264, ISBN: 9781450317757. DOI: 10.1145/2413176.2413206.

[84] N. Hassan, K.-L. A. Yau, and C. Wu, "Edge Computing in 5G: A Review", *IEEE Access*, vol. 7, pp. 127 276–127 289, 2019.

[85] K. Hentschel, D. Jacob, J. Singer, and M. Chalmers, "Supersensors: Raspberry Pi Devices for Smart Campus Infrastructure", in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, Aug. 2016, pp. 58–62. DOI: 10.1109/FiCloud.2016.16.

[86] H. Hoffmann, J. Holt, G. Kurian, *et al.*, "Self-aware Computing in the Angstrom Processor", in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 259–264.

[87] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "SEEC: A General and Extensible Framework for Self-aware Computing", 2011.

[88] F. Hu, *Network innovation through openflow and sdn*. Crc Press, 2014.

[89] F. Hu, Q. Hao, and K. Bao, "A Survey on Software-defined Network and OpenFlow: From Concept to Implementation", *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2181–2206, 2014.

[90] P. Hu, S. Dhelim, H. Ning, and T. Qiu, "Survey on Fog Computing: Architecture, Key Technologies, Applications and Oen Issues", *Journal of Network and Computer Applications*, vol. 98, pp. 27–42, 2017, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2017.09.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804517302953.

[91] S. Illig and M. Großmann, "Überwachung von Bibliotheksbeständen mit Hilfe von Raspberry PIs", 2017.

[92] K. Inamdar, O. R. Kottayi Pilapprathodi, J. John, M. Wolff, M. Großmann, and U. R. Krieger, "A Web Architecture for E-Health Applications Supporting the Efficient Multipath Transport of Medical Images", in *Innovations for Community Services*, F. Phillipson, G. Eichler, C. Erfurth, and G. Fahrnberger, Eds., Cham: Springer International Publishing, 2022, pp. 136–152, ISBN: 978-3-031-06668-9.

[93] "Introduction", in *XaaS: Everything-as-a-Service*, ch. Chapter 1, pp. 1–41. DOI: 10.1142/9789811219924_0001. eprint: https://www.worldscientific.com/doi/pdf/10.1142/9789811219924_0001. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/9789811219924_0001.

[94] M. Iorga, L. Feldman, R. Barton, M. Martin, N. Goren, and C. Mahmoudi, "The NIST Definition of Fog Computing", National Institute of Standards and Technology, Tech. Rep., 2017.

[95] R. Jabbari, N. bin Ali, K. Petersen, and B. Tanveer, "What is DevOps? A Systematic Mapping Study on Definitions and Practices", in *Proceedings of the Scientific Workshop Proceedings of XP2016*, ser. XP '16 Workshops, Edinburgh, Scotland, UK: Association for Computing Machinery, 2016, ISBN: 9781450341349. DOI: 10.1145/2962695.2962707. [Online]. Available: https://doi.org/10.1145/2962695.2962707.

[96] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, "Software Defined Networking: State of the Art and Research Challenges", *Computer Networks*, vol. 72, pp. 74–98, 2014.

[97] M. Jarschel, T. Zinner, T. Hoßfeld, P. Tran-Gia, and W. Kellerer, "Interfaces, Attributes, and Use Cases: A compass for SDN", *IEEE Communications Magazine*, vol. 52, no. 6, pp. 210–217, 2014.

[98] M. S. Jassas, A. A. Qasem, and Q. H. Mahmoud, "A Smart System connecting E-health Sensors and the Cloud", in *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, May 2015, pp. 712–716. DOI: 10.1109/CCECE.2015.7129362.

[99] C. Kaewkasi, *Docker for Serverless Applications*. Birmingham, England: Packt Publishing, Apr. 2023.

[100] S. Kaiser, M. S. Haq, A. □. Tosun, and T. Korkmaz, "Container Technologies for ARM Architecture: A Comprehensive Survey of the State-of-the-Art", *IEEE Access*, vol. 10, pp. 84 853–84 881, 2022. DOI: 10.1109/ACCESS.2022.3197151.

[101] S. P. Kane and K. Matthias, *Docker: up and running*. O'Reilly Media, Inc., 2023.

[102] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge Computing: A Survey", *Future Generation Computer Systems*, vol. 97, pp. 219–235, 2019, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.02.050. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18319903.

[103] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, "Feature-based Comparison and Selection of Software Defined Networking (SDN) Controllers", in *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*, 2014, pp. 1–7. DOI: 10.1109/WCCAIS.2014.6916572.

[104] S. Khorsandroo, A. G. Sánchez, A. S. Tosun, J. Arco, and R. Doriguzzi-Corin, "Hybrid SDN Evolution: A Comprehensive Survey of the State-of-the-Art", *Computer Networks*, vol. 192, p. 107 981, 2021, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2021.107981. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128621001109.

[105] T. Kramp, R. van Kranenburg, and S. Lange, "Introduction to the Internet of Things", in *Enabling Things to Talk: Designing IoT solutions with the IoT Architectural Reference Model*, A. Bassi, M. Bauer, M. Fiedler, *et al.*, Eds.

Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–10, ISBN: 978-3-642-40403-0. DOI: 10.1007/978-3-642-40403-0_1. [Online]. Available: https://doi.org/10.1007/978-3-642-40403-0_1.

[106] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined Networking: A comprehensive Survey", *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[107] L. Lamport, R. E. Shostak, and M. C. Pease, "The Byzantine Generals Problem", *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982. DOI: 10.1145/357172.357176. [Online]. Available: http://doi.acm.org/10.1145/357172.357176.

[108] D. T. Le, M. Großmann, and U. R. Krieger, "Cloudless Resource Monitoring in a Fog Computing System Enabled by an SDN/NFV Infrastructure", p. 4, 2022. DOI: 10.25972/OPUS-28072.

[109] A. Lewis, M. Campbell, and P. Stavroulakis, "Performance Evaluation of a Cheap, Open Source, Digital Environmental Monitor based on the Raspberry Pi", *Measurement*, vol. 87, pp. 228 –235, 2016, ISSN: 0263-2241. DOI: http://dx.doi.org/10.1016/j.measurement.2016.03.023. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0263224116001871.

[110] O. Liebel, *Skalierbare Container-Infrastrukturen: Das Handbuch für Administratoren*, 1. Auflage. Bonn: Rheinwerk and Rheinwerk Computing, 2017, ISBN: 978-3-8362-4366-7.

[111] S. Lindner, D. Merling, M. Häberle, and M. Menth, "P4-Protect: 1+1 Path Protection for P4", in *Proceedings of the 3rd P4 Workshop in Europe*, ser. EuroP4'20, Barcelona, Spain: Association for Computing Machinery, 2020, pp. 21–27, ISBN: 9781450381819. DOI: 10.1145/3426744.3431327. [Online]. Available: https://doi.org/10.1145/3426744.3431327.

[112] W. Liu, S. Zhu, T. Mundie, and U. Krieger, "Advanced Block-chain Architecture for E-health Systems", in *2017 IEEE 19th International Conference on e-Health Networking, Applications and Services (Healthcom)*, 2017, pp. 1–6. DOI: 10.1109/HealthCom.2017.8210847.

[113] E. Markakis, G. Mastorakis, C. X. Mavromoustakis, and E. Pallis, *Cloud and Fog Ccomputing in 5G Mobile Networks: Emerging Advances and Applications*. Institution of Engineering and Technology, 2017.

[114] V. Marmol, R. Jnagal, and T. Hockin, "Networking in Containers and Container Clusters", *Proceedings of netdev 0.1, February*, 2015.

[115] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient Learning of Deep Networks from Decentralized Data", in *Artificial Intelligence and Statistics*, PMLR, 2017, pp. 1273–1282.

[116] H. Mekky, F. Hao, S. Mukherjee, T. Lakshman, and Z.-L. Zhang, "Network Function Virtualization Enablement within SDN Data Plane", in *IEEE INFOCOM*, 2017, pp. 1–9.

[117] P. Mell, T. Grance, *et al.*, "The NIST Definition of Cloud Computing", Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, Tech. Rep., 2011.

[118] R. A. Meyer and L. H. Seawright, "A Virtual Machine Time-sharing System", *IBM Systems Journal*, vol. 9, no. 3, pp. 199–218, 1970. DOI: 10.1147/sj.93.0199.

[119] H. Motohashi, K. Nguyen, and H. Sekiya, "Enabling P4-based Multipath Communication in Wireless Networks", in *2020 IEEE Globecom Workshops (GC Wkshps*, 2020, pp. 1–5. DOI: 10.1109/GCWkshps50303.2020.9367468.

[120] A. Mouat, *Using docker*. Beijing: O'Reilly, 2016, ISBN: 978-1-4919-1576-9.

[121] M. R. H. Mullick, M. Großmann, and U. R. Krieger, "A Feasibility Study of a Lightweight Fog Computing Architecture Integrating Blockchain Technology for Smart E-Health Applications", in *Innovations for Community Services*, U. R. Krieger, G. Eichler, C. Erfurth, and G. Fahrnberger, Eds., Cham: Springer Nature Switzerland, 2023, pp. 253–276, ISBN: 978-3-031-40852-6.

[122] S. Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[123] "Network Functions Virtualisation – Introductory White Paper", European Telecommunications Standards Institute, Tech. Rep., 2012.

[124] "Network Functions Virtualisation (NFV); Architectural Framework", European Telecommunications Standards Institute, Tech. Rep., 2014.

[125] G.-T. Nguyen and K. Kim, "A Survey about Consensus Algorithms Used in Blockchain", *Journal of Information Processing Systems*, vol. 14, no. 1, 2018.

[126] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of Software-defined Networking: Past, Present, and Future of Programmable Networks", *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[127] Open Networking Foundation, "Software-defined Networking: The New Norm for Networks", *ONF White Paper*, vol. 2, pp. 2–6, 2012.

[128] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca, and J. Folgueira, "Network Slicing for 5G with SDN/NFV: Concepts, Architectures and Challenges", *arXiv preprint arXiv:1703.04676*, 2017.

[129] C. Pahl, "Containerization and the PaaS Cloud", *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015. DOI: 10.1109/MCC.2015.51.

[130] M. Peuster, H. Karl, and S. van Rossem, "MeDICINE: Rapid Prototyping of Production-ready Network Services in Multi-PoP Environments", in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov. 2016, pp. 148–153. DOI: 10.1109/NFV-SDN.2016.7919490.

[131] B. Pfaff, J. Pettit, T. Koponen, *et al.*, "The Design and Implementation of Open vSwitch", in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA: USENIX Association, May 2015, pp. 117–130, ISBN: 978-1-931971-218. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff.

[132] A. Rahman, M. J. Islam, S. S. Band, G. Muhammad, K. Hasan, and P. Tiwari, "Towards a Blockchain-SDN-based Secure Architecture for Cloud Computing in Smart industrial IoT", en, *Digital Communications and Networks*, vol. 9, no. 2, pp. 411–421, Apr. 2023, ISSN: 23528648. DOI: 10.1016/j.dcan.2022.11.003.

[133] F. Reinartz. "Prometheus 2.0: New Storage Layer". (2017), [Online]. Available: https://coreos.com/blog/prometheus-2.0-storage-layer-optimization.

[134] R. Roman, J. Lopez, and M. Mambo, "Mobile Edge Computing, Fog et al.: A Survey and Analysis of Security Threats and Challenges", *Future Generation Computer Systems*, vol. 78, pp. 680 –698, 2018, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2016.11.009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X16305635.

[135] K. S. Sahoo, B. K. Tripathy, K. Naik, *et al.*, "An Evolutionary SVM Model for DDOS Attack Detection in Software Defined Networks", *IEEE Access*, vol. 8, pp. 132 502–132 513, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3009733.

[136] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab, "SDN Controllers: A Comparative Study", in *2016 18th Mediterranean Electrotechnical Conference (MELECON)*, 2016, pp. 1–6. DOI: 10.1109/MELCON.2016.7495430.

[137] M. Scazzariello, L. Ariemma, and T. Caiazzi, "Kathará: A Lightweight Network Emulation System", in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–2. DOI: 10.1109/NOMS47738.2020.9110351.

[138] S. Scott-Hayward, S. Natarajan, and S. Sezer, "A Survey of Security in Software Defined Networks", *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 623–654, 2016, ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2453114.

[139] S. Sezer, S. Scott-Hayward, P. K. Chouhan, *et al.*, "Are we ready for SDN? Implementation Challenges for Software-defined Networks", *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, 2013.

[140] H. Shafagh, A. Hithnawi, L. Burkhalter, and S. Duquennoy, "Towards Blockchain-based Auditable Storage and Sharing of IoT Data", *arXiv preprint arXiv: 1705.08230*, 2017.

[141] J. Sherry and S. Ratnasamy, "A Survey of Enterprise Middlebox Deployments", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-24, Feb. 2012. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-24.html.

[142] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges", *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[143] M.-K. Shin, K.-H. Nam, and H.-J. Kim, "Software-defined Networking (SDN): A Reference Architecture and Open APIs", in *ICT Convergence (ICTC), 2012 International Conference on*, IEEE, 2012, pp. 360–361.

[144] D. Silva, J. Rafael, and A. Fonte, "Toward Optimal Virtualization: An Updated Comparative Analysis of Docker and LXD Container Technologies", *Computers*, vol. 13, no. 4, 2024, ISSN: 2073-431X. DOI: 10.3390/computers13040094. [Online]. Available: https://www.mdpi.com/2073-431X/13/4/94.

[145] J. Singh and S. Behal, "Detection and Mitigation of DDoS Attacks in SDN: A Comprehensive Review, Research Challenges and Future Directions", en, *Computer Science Review*, vol. 37, p. 100 279, Aug. 2020, ISSN: 15740137. DOI: 10.1016/j.cosrev.2020.100279.

[146] "Software-Defined Networking: The New Norm for Networks", Open Networking Foundation, Tech. Rep., 2012.

[147] H. Stadler, M. Großmann, and U. R. Krieger, "Design of a Secure Mobile Business Communication Platform Utilizing Next Generation Web Technologies", in *2016 IEEE 13th International Conference on e-Business Engineering (ICEBE)*, Nov. 2016, pp. 257–263. DOI: 10.1109/ICEBE.2016.051.

[148] Stefan Strobel, Ed., *Virtualisierungs-Grundlagen* (Ratgeber Virtualisierung & Cloud Computing). TecChannel, 2011.

[149] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and Practice of Bloom Filters for Distributed Systems", *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2012. DOI: 10.1109/SURV.2011.031611.00024.

[150] I. Ullah, I. U. Khan, M. Ouaissa, M. Ouaissa, and S. El Hajjami, *Future Communication Systems Using Artificial Intelligence, Internet of Things and Data Science*. CRC Press, 2024.

[151] B. Varghese and R. Buyya, "Next Generation Cloud Computing: New Trends and Research Directions", *Future Generation Computer Systems*, vol. 79, pp. 849–861, 2018.

[152] A. Vilmos, M. C., and A. Moroni, *Vision and Challenges for Realising the Internet of Things*, Jan. 2010.

[153] A. Voellmy, H. Kim, and N. Feamster, "Procera: A Language for High-level Reactive Network Control", in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12, Helsinki, Finland: Association for Computing Machinery, 2012, pp. 43–48, ISBN: 9781450314770. DOI: 10.1145/2342441.2342451. [Online]. Available: https://doi.org/10.1145/2342441.2342451.

[154] M. Vukolić, "The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication", in *Open Problems in Network Security - IFIP WG 11.4 International Workshop, iNetSec 2015, Zurich, Switzerland, October 29, 2015, Revised Selected Papers*, 2015, pp. 112–125. DOI: 10.1007/978-3-319-39028-4_9. [Online]. Available: https://doi.org/10.1007/978-3-319-39028-4_9.

[155] D. Walden, "50th Anniversary of MIT's Compatible Time-Sharing System", *IEEE Annals of the History of Computing*, vol. 33, no. 4, pp. 84 –85, 2011. [Online]. Available: https://muse.jhu.edu/pub/87/article/464605.

[156] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, "A Survey on Mobile Edge Networks: Convergence of Computing, Caching and Communications", *IEEE Access*, vol. 5, pp. 6757–6779, 2017. DOI: 10.1109/ACCESS.2017.2685434.

[157] Z. Wei-guo, M. Xi-lin, and Z. Jin-zhong, "Research on Kubernetes' Resource Scheduling Scheme", in *Proceedings of the 8th International Conference on Communication and Network Security*, 2018, pp. 144–148.

[158] T. Wood, K. Ramakrishnan, J. Hwang, G. Liu, and W. Zhang, "Toward a Software-based Network: Integrating Software defined Networking and Network Function Virtualization", *IEEE Network*, vol. 29, no. 3, pp. 36–41, 2015.

[159] D. Wörner and T. von Bomhard, "When Your Sensor Earns Money: Exchanging Data for Cash with Bitcoin", in *UbiComp Adjunct*, ACM, 2014, pp. 295–298.

[160] Y. Xu, V Mahendran, and S. Radhakrishnan, "SDN Docker: Enabling Application Auto-docking/undocking in Edge Switch", in *Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference on*, IEEE, 2016, pp. 864–869.

[161] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud Computing: State-of-the-Art and Research Challenges", *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[162] T. Zhang and S. Mao, "An Overview of Emerging Video Coding Standards", *GetMobile: Mobile Comp. and Comm.*, vol. 22, no. 4, pp. 13–20, May 2019, ISSN: 2375-0529. DOI: 10.1145/3325867.3325873. [Online]. Available: https://doi.org/10.1145/3325867.3325873.

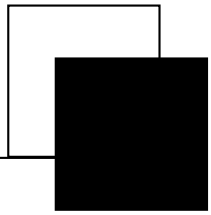
[163] L. Zhu, M. M. Karim, K. Sharif, *et al.*, "SDN Controllers: A Comprehensive Analysis and Performance Evaluation Study", *ACM Comput. Surv.*, vol. 53, no. 6, Dec. 2020, ISSN: 0360-0300. DOI: 10.1145/3421764. [Online]. Available: https://doi.org/10.1145/3421764.

[164] M. H. Ziegler, M. Großmann, and U. R. Krieger, "Integration of Fog Computing and Blockchain Technology Using the Plasma Framework", in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, May 2019, pp. 120–123. DOI: 10.1109/BLOC.2019.8751308.

[165] A. Zismer, "Performance of Docker Overlay Networks", 2016.

[166] G. Zyskind, O. Nathan, and A. Pentland, "Enigma: Decentralized Computation Platform with Guaranteed Privacy", *CoRR*, vol. abs/1506.03471, 2015.

Acronyms



A

AAL Ambient Assisted Living 231

ACL Access Control List 271

AI Artificial Intelligence 5, 43, 172, 209

API Application Programming Interface 37, 44, 55, 56, 58, 60, 61, 63, 83, 84, 86, 90–93, 102, 111, 117–119, 121, 123, 124, 126, 128, 132, 133, 137, 144, 149, 153, 154, 157, 161, 192, 197, 200, 212, 223, 225, 238, 239, *Glossary:* Application Programming Interface

ARM Advanced Reduced Instruction Set Computer (RISC) Machine 15, 119, 147, 151, 164, 165, 191, 197, 209, 211

ARP Address Resolution Protocol 52, 140

ASIC Application-Specific Integrated Circuit 65, 231

B

BFT Byzantine Fault Tolerant 207, 210, 264

BGP Border-Gateway Protocol 61

BLE Bluetooth Low Energy 191

BSS Business Support System 79

C

CaaS Container as a Service 40

CC Cloud Computing 5, 6, 11, 21, 25, 28, 39–42, 44–46, 54, 62, 73, 81, 100, 113, 114, 147, 167, 189, 231, 267, 269, *Glossary:* Cloud Computing

CDN Content Distribution Network 45, 46

CI Continuous Integration 11, 21, 25, 69, 70, 111, 131, 163, 165, 197, 198, 233, 268, *Glossary:* Continuous Integration

CLI Command Line Interface 37, 63, 117, 120, 124, 163

CNM Container Network Model 123

CoAP Constrained Application Protocol 118

CPU Central Processing Unit 30, 39, 97, 102, 105, 106, 113, 148–155, 158, 159, 161, 166, 170, 171, 174, 177, 179–181, 201, 202, 212, 213, 219, 220, 233

CRC Cyclic Redundancy Check 99

CRM Cloudless Resource Monitoring 111, 160–163, 165, 166, 169, 233, 234, 238

D

DCCP Datagram Congestion Control Protocol 52

DDoS Distributed DoS 16, 73, 89, 100–103, 105, 107, 235–237

DinD Docker-in-Docker 90, 91, 136–138, 159, 190, 237, *Glossary:* Docker-in-Docker

DNS Domain Name System 82, 133, 134, 137, 141, 208

DoS Denial-of-Service 16, 100, 104, 105, 262

DPI Deep Packet Inspection 103

DSL Domain-Specific Language 60, 69

E

EC Edge Computing 5, 21, 25, 39, 42–44, 113, 167, 177, 178, *Glossary:* Edge Computing

ERM Entity-Relationship Model 272, *Glossary:* Entity-Relationship Model

ETSI European Telecommunications Standards Institute 57, 115, 270

F

FaaS Function as a Service 22, 40, 73, 81, 86–88, 195, 197, 202–204

FC Fog Computing 5, 11, 21, 25, 39, 44–46, 86, 114, 129, 151, 152, 157, 163, 167, 177, 205, 208, 209, 211, 214, 215, 231, 232, 234, 267, *Glossary:* Fog Computing

FL Federated Machine Learning 14, 17, 111, 166, 168, 170–174, 176–179, 182, 187, 229, 234, 237, 238

FTP File Transfer Protocol 53

G

GPIO General Purpose Input/Output 191, 193, 225

GPS Global Positioning System 43

GUI Graphical UI 14, 63, 89–91, 93, 151, 233, 238

H

HTML Hyper Text Markup Language 91, 225

HTTP Hypertext Transfer Protocol 53, 86, 91, 149, 154, 161, 176, 194, 197, 200

I

I/O input/output 27, 151, 159, 163, 166, 201, 223

IaaS Infrastructure as a Service 40, 41, 46

IANA Internet Assigned Numbers Authority 131

ICE Interactive Connection Establishment 222

ICMP Internet Control Message Protocol 52, 65

IIoT Industrial IoT 47

IoT Internet of Things 5–7, 10, 13, 14, 17, 21, 25, 39, 43–45, 47–50, 69, 86, 88, 100, 113, 114, 151, 163, 187, 189, 190, 198, 199, 201, 204, 205, 208, 209, 214, 215, 217, 222, 223, 231, 232, 263, 268, *Glossary:* Internet of Things

IP Internet Protocol 52, 53, 61, 65, 99, 118, 119, 123, 124, 131–134, 136, 137, 141–143, 178, 207, 263

IPAM IP Address Management 118, 119

IPSec IP Security 52

ISO International Organization for Standardization 51

ISP Internet Service Provider 56

IT Information Technology 28, 232

J

JSON JavaScript Object Notation 91, 121, 122, 193, 197

JVM Java Virtual Machine 62, 63

K

KaaS Kathará as a Service 73, 89, 94, 101, 105

KVM Kernel-based Virtual Machine 180

L

LAC Layered Application Composition 34

LAN Local Area Network 49, 132, 266

LAND Local Area Network Denial 236

LLC Logical Link Control 52

LRPON Long-Reach Passive Optical Network 45, 46

LXC Linux Containers 31–34, 36

LXD Linux Container Hypervisor 33, 34, 36

M

M2M Machine-to-Machine 5, 49

MAC Medium Access Control 51, 52, 65, 140–142

MAPE-K Monitor-Analyze-Plan-Execute-Knowledge 14, 22, 73, 75–77, 111, 167–169, 176, 233, 234, *Glossary:* Monitor-Analyze-Plan-Execute-Knowledge

MIME Multipurpose Internet Mail Extensions 53

MIT Massachusetts Institute of Technology 48

ML Machine Learning 76, 101, 172, 173, 176

MQTT Message Queue Telemetry Transport 118

N

NAT Network Address Translation 128

NDN Named Data Network 46

NEaaS Network Emulator as a Service 90, 93

NFS Network File System 52

NFV Network Function Virtualization 6, 21, 25, 45, 57, 58, 60, 73, 75, 77–79, 115, 264, 271, *Glossary:* Network Function Virtualization

NFV MANO Network Function Virtualization Management and Orchestration 73, 75, 77–79, *Glossary:* Network Function Virtualization Management and Orchestration

NFVI NFV Infrastructure 77–79

NFVO NFV Orchestrator 78

NIC Network Interface Card 65, 122, 126, 127, 136, 177

NIST National Institute of Standards and Technology 39, 44, 45

NVGRE Network Virtualization using Generic Routing Encapsulation 119

O

OCI Open Container Initiative 34, 36

ODA Observe Decide Act 22, 73, 75–77, *Glossary:* Observe Decide Act

ONF Open Networking Foundation 60, 63, 141

ONOS Open Network Operating System 61–64, 70, 100, 101, 104, 111, 116, 127, 131–136, 138, 140, 162–165, 233, 237–239, *Glossary:* Open Network Operating System

OS Operating System 6, 27–31, 35, 41, 62–64, 87, 90, 122, 123, 137, 147, 151, 159, 160, 267, 270, 272

OSGi Open Services Gateway initiative 138

OSI Open Systems Interconnection 49, 51, 53, 235

OSPF Open Shortest Path First 52

OSS Operations Support System 79

OvS Open vSwitch XVI, 21, 22, 25, 65, 66, 101, 104, 106, 111, 116–120, 122–124, 127, 128, 131–136, 138, 233, 264, *Glossary:* Open vSwitch

OVSDB OvS Database 111, 120, 121, 126, 127

P

P2P Peer-to-Peer 173–175, 205, 207, 210, 217, 224

P4 Programming Protocol-Independent Packet Processor 16, 89, 94–97, 99, 100, 235, 238

PaaS Platform as a Service 36, 40, 46

PBFT Practical BFT 210

PC Personal Computer 28, 30, 211

POJO Plain Old Java Object 144

POM Project Object Model 62, 69

PPP Point-to-Point Protocol 52

PPTP Point-to-Point Tunneling Protocol 52

Q

QoE Quality of Experience 93, 174, 221

QoS Quality of Service 6, 49, 58, 238, 239

R

RAM Random Access Memory 88, 105, 148–153, 155, 157, 159, 161, 179–181, 212, 219

REST Representational State Transfer 37, 60, 90, 92, 93, 132, 136, 139, 144, 197, 238, 239

RFID Radio-Frequency Identification 48, 49

RIP Routing Information Protocol 52

RISC Reduced Instruction Set Computer 15, 261

RPC Remote Procedure Call 52, 121, 124, 210

RPi Raspberry Pi 13, 119, 132, 147, 151, 152, 154, 157, 159, 189–191, 193, 196, 199, 204, 205, 211, 212, 217–221, 225

RTP Real-time Transport Protocol 52

S

SaaS Software as a Service 40, 41, 46, 47, 89

SBC Single Board Computer 13–15, 18, 132, 136, 148, 151, 155, 157, 191, 205, 211, 217, 232

SCTP Stream Control Transmission Protocol 52, 65

SDK Software Development Kit 127

SDN Software Defined Network XIX, 6, 14, 16, 17, 21, 25, 45, 54–56, 58–65, 75, 89, 95, 100–103, 111, 115, 119, 131, 133, 134, 136, 138, 140, 145, 159, 160, 168, 169, 172, 231–235, 239, 271, *Glossary:* Software Defined Network

SDP Session Description Protocol 223

SERM Structured Entity-Relationship Model 120, *Glossary:* Structured Entity-Relationship Model

SLA Service Level Agreement 78

SMT Simultaneous Multithreading 105

SMTP Simple Mail Transfer Protocol 53

SOA Service-Oriented Architecture 6

SPoF Single Point of Failure 59, 102, 193

SQL Structured Query Language 52

SSH Secure Shell 63

SSL Secure Sockets Layer 53

T

TAP Test Access Point 122, 123

TCAM Ternary Content Addressable Memory 64

TCP Transmission Control Protocol 52, 53, 65, 99, 104, 132, 142, 193, 194, 235

TLS Transport Layer Security 53, 193

TSDB Time-Series Database 17, 162, 169, 170, 194

U

UDP User Datagram Protocol 52, 65, 94, 99, 106, 133, 134, 142, 235

UI User Interface 92, 152, 172, 262

URL Uniform Resource Locator 171, 176

USB Universal Serial Bus 135, 191, 193, 225

UTXO unspent transaction outputs 206

V

veth virtual Ethernet device 122, 123, 128

VIM Virtualized Infrastructure Manager 78

VLAN Virtual LAN 123

VM Virtual Machine 5, 6, 21, 25, 27–33, 46, 65, 90, 101, 180, 232, 266, 269, 270

VMM VM Monitor 27, 30

VNF Virtual Network Function 77–79, 143, 232, 266, 270

VNFM VNF Manager 78

VPN Virtual Private Network 15

VSS Very Simple Switch 96

VXLAN Virtual Extensible LAN 118, 119

W

WAN Wide Area Network 57

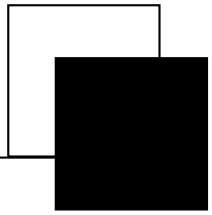
WiFi Trademark of the Wi-Fi alliance 49, 51

WLAN Wireless LAN 191

X

XGB eXtreme Gradient Boosting 182

Glossary



A

Application Programming Interface is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API 37, 261

C

Cloud Computing is the elder of scalabilty and provides users with immediate access to computer system resources, such as data storage and computational power, without requiring user control. Large cloud providers have many data centers, which allow them to share resources and adopt a pay-as-you-go pricing model. 5, 261

Cloud-to-Fog-Continuum is a concept categorizing computing models according to their degree of centralization and proximity to users. It describes the transition from centralized and scalable CC, hosted in extensive remote data centers, to decentralized FC, delivering services to the network's periphery, hence minimizing latency and bandwidth usage [cf. 3] XV, XIX, 3, 5, 7, 9–12, 14, 17–19, 22, 39, 73, 81, 107, 111, 114, 115, 145, 159, 166, 167, 187, 195, 229, 231–234, 237–239

Container is a lightweight, portable, and isolated runtime environment that virtualizes an application together with its dependencies, facilitating consistent execution across diverse computing environments. It virtualizes at the OS level, utilizing the host machine's Kernel while ensuring process isolation.

33, 263

Continuous Integration is a software development approach in the field of DevOps, where developers combine code changes into a common repository and then automatically create and test them. The primary objective is to rapidly identify software defects, enhance the overall quality of the product, and minimize the time required for validation and deployment. 11, 198, 261

D

Docker-in-Docker describes the technique of executing Docker containers inside other Docker containers, hence establishing a Docker environment nested within another. This is frequently utilized in CI pipelines, where Docker images are constructed or evaluated via a containerized setting, in addition to testing Docker itself or setting up isolated development environments. 90, 262

E

Edge Computing is a decentralized computing approach that brings computation and data storage in close proximity to users, hence minimizing latency. Emerging in the 1990s, it subsequently broadened its scope to encompass content delivery networks and early edge computing applications. Although sometimes associated with the IoT, edge computing and IoT are separate concepts. 5, 42, 262

Entity-Relationship Model is used to depict the relationships between entities within a certain field of knowledge. They are commonly employed in software engineering to depict business processes and establish abstract data structures for relational databases. ER models, created by Peter Chen in 1976, are frequently employed for instructing database construction and can also define ontologies relevant to particular domains. Several models illustrate the connection between super and subtype entities through generalization-specialization connections, and can be employed to precisely define domain-specific ontologies. 262, 272

F

Fog Computing was introduced by Cisco and is a cloud concept that involves providing computing capacity in a decentralized manner at the edge of the cloud. This approach reduces the requirement for central data transfer. Alternatively, operations can be executed in small-scale data centers located nearby, which reduces latency and processing times by shortening the distance data needs to travel from end devices to processing facilities. 5, 44, 262

G

Go , also known as Golang, is an open-source, statically typed, compiled language created for simplicity, efficiency, and scalability. Released by Google in 2012, it incorporates garbage collection, comprehensive concurrency support via goroutines and channels, and an uncomplicated syntax, rendering it suitable for constructing resilient and high-performance applications, particularly in CC, networking, and distributed systems. 123, 126, 127

H

Hypervisor is a software or hardware layer that administers and operates VMs by allocating the host machine's actual hardware resources, hence facilitating the concurrent execution of numerous VMs on a single system. 33, 263

I

Internet of Things is a network of tangible entities embedded with sensors and software, which establish connections with other devices and systems over the Internet. This facilitates the flow of data between these entities. 5, 48, 263

J

Java is a high-level, object-oriented programming language engineered to possess minimal implementation requirements. It enables programmers to write once, run anywhere code, permitting compiled Java code to execute on all platforms without the need for recompilation. Java applications are compiled into bytecode executable on any Java Virtual Machine (JVM). Java, created by James Gosling of Sun Microsystems, attained significant popularity, ranking as the third most used programming language in 2022.

Notwithstanding its popularity, Java has experienced a downturn in recent years, as other Java-based languages have risen in prominence. 62, 127, 138, 162

K

Kernel is the central component of the OS, overseeing system resources and hardware interactions, serving as an intermediary between software and hardware, and ensuring secure and efficient system functionality. 29–32, 122, 151, 214, 267

L

Linux is one of the most widely supported UNIX-like OS, which is open source and community-developed and runs on major platforms like x86, ARM, and SPARC. 30–33, 35, 122, 123, 151, 152, 191, 200, 214, 223, 263

M

Monitor-Analyze-Plan-Execute-Knowledge is an ongoing process that entails evaluating the existing condition, determining the target condition, devising the requisite system alterations to transition from the current state to the desired one, and executing the most effective plan. Foundational knowledge supports all of these actions. 14, 263

N

Network Function Virtualization is the virtualization of Layer 4 through 7 services like load balancing and firewalling, converting network appliances into VMs for quick deployment. It addresses inefficiencies caused by virtualization, such as varied traffic flows and issues with fixed appliances. NFV allows for the creation of virtual instances of functions, denoted as VNF, like firewalls, which can be easily deployed and configured 6, 264

Network Function Virtualization Management and Orchestration is an architectural framework developed by the Industry Specification Group of the ETSI. Its purpose is to efficiently manage and coordinate VNF and software components. This framework enables the provision of services and connectivity even when not connected to physical devices. 73, 264

O

Observe Decide Act constitutes an ongoing cycle of observation, decision-making, and action. It is intended to detect issues and errors in an operational system and establish solutions to mitigate the impact of these failures. 22, 75, 264

Open Network Operating System is a prominent open source SDN controller used for advanced SDN/NFV applications. The system provides high-quality solutions for carriers, streamlines programmable interfaces, and enables real-time control of the network. The ONOS cloud controller facilitates innovation by enabling end-users to develop new network applications without the need to modify dataplane infrastructure. 61, 264

Open vSwitch is a software switch that operates within the hypervisor or management domain to establish connections between virtual machines and physical interfaces. 21, 126, 264

OpenFlow is a communication protocol that divides the control and forwarding layers, allowing for network topologies that are built around applications. The system provides support for SDN and Access Control Lists (ACLs), and is overseen by the Open Networking Foundation. 21, 55, 58, 60, 61, 63–66, 95, 102, 115, 120, 121, 131, 132, 136, 143

P

Python is a high-level, general-purpose programming language characterized by a design philosophy that emphasizes code readability and indentation. It is dynamically typed, garbage-collected, and accommodates many programming paradigms. Developed by Guido van Rossum in the late 1980s, Python has evolved significantly since its initial release in 1991, ranking among the most popular programming languages. 150, 160, 190, 223

S

SCRUM is an agile project management framework that focuses on learning from experiences, self-organization, and continuous improvement. Known for its versatility, Scrum is commonly used by software development teams but can be applied to various teamwork types. It comprises various meetings, tools, and roles to assist teams in task structuring and management. 6

Software Defined Network refers to the ability to program the network, which means that network admins can quickly adjust the network based on changing requirements. SDN is made possible by separating the control plane from the data plane 6, 265

Structured Entity-Relationship Model [54], which is an expansion of the Entity-Relationship Model (ERM), relies on existential connections among sets of objects to distinguish between primary and subordinate sets. The SERM utilizes updated modeling constructs and represents information in a quasi-hierarchical graph, emphasizing structural aspects that are already implicitly present in the ERM. The presence of dependency is evident in both the attributes and the symbolic representation of relationships, facilitating the detection of modeling errors. 265

U

UNIX was developed by AT&T and is a portable, multi-tasking, multi-user, time-sharing OS 28, 149, 150, 158, 270

University of Bamberg Press

The emergence of next-generation infrastructures is facilitating the transition from centralized cloud computing to the cloud-to-fog continuum, wherein services are disseminated over many layers, ranging from centralized clouds to decentralized edge devices. This model facilitates more efficient and responsive service provisioning, especially in latency-sensitive and resource-limited contexts. In these situations, middleware is essential for managing the distributed characteristics of services, aiding energy-efficient devices at the network periphery, and enabling seamless service orchestration.

These frameworks facilitate the monitoring of small-scale services and nodes, hence maintaining performance consistency. Subsequently, distributions can be refined by integrating insights from real-time measurements and network topology data, facilitating dynamic resource allocation and improving system efficiency.

Additionally, emulation frameworks are essential for testing diverse networking scenarios and finding nearly perfect service deployments in a sandboxed environment. By determining the most appropriate emulator for application development, we provide an easy-to-use platform to researchers and developers, such that they can evaluate their services‘ needs. Moreover, novel network paradigms and protocols are easily adaptable on given emulations.

This dissertation examines the design of middleware solutions within the cloud-to-fog continuum and analyzes the application of emulation tools to enhance service delivery at the network edge.

ISBN: 978-3-98989-089-3
9 783989 890893
www.uni-bamberg.de/ubp