Leveraging Kubernetes in Edge-Native Cable Access Convergence

Master of Science in Technology Thesis
University of Turku
Department of Computing
Software Engineering
2023
Janne Virtanen

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Computing

JANNE VIRTANEN: Leveraging Kubernetes in Edge-Native Cable Access Convergence

Master of Science in Technology Thesis, 117 p.
Software Engineering
July 2023

Public clouds provide infrastructure services and deployment frameworks for modern cloud-native applications. As the cloud-native paradigm has matured, containerization, orchestration and Kubernetes have become its fundamental building blocks. For the next step of cloud-native, an interest to extend it to the edge computing is emerging. Primary reasons for this are low-latency use cases and the desire to have uniformity in cloud-edge continuum. Cable access networks as specialized type of edge networks are not exception here. As the cable industry transitions to distributed architectures and plans the next steps to virtualize its on-premise network functions, there are opportunities to achieve synergy advantages from convergence of access technologies and services. Distributed cable networks deploy resource-constrained devices like RPDs and RMDs deep in the edge networks. These devices can be redesigned to support more than one access technology and to provide computing services for other edge tenants with MEC-like architectures. Both of these cases benefit from virtualization. It is here where cable access convergence and cloud-native transition to edge-native intersect. However, adapting cloud-native in the edge presents a challenge, since cloud-native container runtimes and native Kubernetes are not optimal solutions in diverse edge environments. Therefore, this thesis takes as its goal to describe current landscape of lightweight cloud-native runtimes and tools targeting the edge. While edge-native as a concept is taking its first steps, tools like KubeEdge, K3s and Virtual Kubelet can be seen as the most mature reference projects for edge-compatible solution types. Furthermore, as the container runtimes are not yet fully edge-ready, WebAssembly seems like a promising alternative runtime for lightweight, portable and secure Kubernetes compatible workloads.

Keywords: cable access, cloud-native, convergence, container orchestration, edge computing, Kubernetes, edge-native, WebAssembly

Contents

1 Introduction **1**

2 Cloud-native orchestration **8**

2.1 Cloud computing 8

2.1.1 Introduction to the cloud 8

2.1.2 Definition of cloud computing 10

2.1.3 History of the cloud 13

2.1.4 Enablers of cloud computing 14

2.1.5 Application deployment models 16

2.1.6 Cloud-native applications 19

2.1.7 Edge computing 22

2.2 Virtualization technologies 26

2.2.1 Hardware virtualization 27

2.2.2 Containerization 32

2.3 Kubernetes 41

2.3.1 Introduction 41

2.3.2 Components 43

2.3.3 Configuration and management 47

2.3.4 Cluster management 48

2.3.5 Communication and security 49

2.3.6 Network and service model 50
2.3.7 Helm package manager 54

3 Distributed cable access 56

3.1 Access networks 56
3.2 Broadband data access networks evolution 58
3.3 Fiber to the X and digital optics 60
3.4 Data Over Cable Service Interface Specification 62
3.4.1 Introduction 63
3.4.2 DOCSIS network elements 63
3.4.3 Physical layer 64
3.4.4 MAC layer 66
3.4.5 Cable modem provisioning 69
3.5 DOCSIS extensions 71
3.5.1 Frequency Division Duplex and Extended Spectrum 71
3.5.2 Full Duplex 72
3.6 Distributed cable access architectures 74

4 Case: Kubernetes in edge-native cable access convergence 78

4.1 Current trends of cable access and cloud-native 78
4.1.1 Full disaggregation and virtualization of cable access 79
4.1.2 Cable access convergence 80
4.1.3 Evolution of cable to multi-access edge-native 82
4.2 Case study introduction 85
4.2.1 Research goals and question 85
4.2.2 Research plan and discussion 87
4.3 Lightweight Kubernetes solutions 89
4.3.1 K3s 92

4.3.2 MicroK8s . . . 93
4.3.3 KubeEdge . . . 93
4.3.4 Virtual Kubelet . . . 95
4.3.5 Kubemark . . . 96
4.3.6 Kind . . . 97
4.3.7 KWOK . . . 97
4.4 Comparison of lightweight Kubernetes distributions . . . 98
4.5 Edge-native application runtimes . . . 99
4.5.1 Alternative Kubernetes runtimes . . . 101
4.5.2 Standalone WebAssembly runtimes . . . 102
4.5.3 Kubernetes as edge-native orchestrator . . . 105
4.6 Prospects of edge-native in cable access . . . 106

5 Discussion **109**
5.1 Cloud-native model evolution to edge-native . . . 109
5.2 Cable access future prospects . . . 111
5.3 Case study considerations . . . 112
5.4 Study evalution and future work . . . 113

6 Conclusion **115**
6.1 Cloud-native, cable access, and future trends . . . 115
6.2 Case study results . . . 116

References **118**

List of Figures

2.1 Cloud-native application architecture with microservices 20
2.2 Edge computing paradigms 23
2.3 ETSI MEC reference architecture model 24
2.4 Hypervisor types 28
2.5 Comparison of container stack and virtual machine stack 33
2.6 Open Container Initiative standards on container engine 34
2.7 Container ecosystem runtime standards and component instances 36
2.8 Container image stacked layer structure 38
2.9 Kubernetes high-level architecture 42
2.10 Kubernetes networking model communication paths 51
2.11 Kubernetes service model 53

3.1 Simplified view of access network 57
3.2 Elements of Passive Optical Network 61
3.3 Elements of DOCSIS network 64
3.4 EuroDOCSIS 3.0 spectrum use 65
3.5 MAC layer functions in CMTS 67
3.6 Traditional cable access network elements 74
3.7 Remote PHY architecture 76

4.1 High-level view of current state of access networks 80
4.2 High-level view of access networks convergence 82

4.3 Cable access Multi-access Edge Computing host architecture 83
4.4 KubeEdge architecture 94
4.5 High-level view of Virtual Kubelet architecture 95
4.6 WebAssembly WASI runtime architecture 102

List of Tables

4.1 Lightweight Kubernetes distributions and Kubelet agents 90

Acronyms

AWS	Amazon Web Services
CATV	Community Access Cable Television
CMTS	Cable Modem Termination System
CNCF	Cloud Native Computing Foundation
CPE	Customer Premises Equipment
CRD	Custom Resource Definition
CSP	Cloud Service Provider
DAA	Distributed Access Architectures
DOCSIS	Data Over Cable Service Interface Specification
DS	Downstream
DSL	Digital Subscriber Line
ETSI	European Telecommunications Standards Institute
FTTH	Fiber to the Home
HFC	Hybrid Fiber-Coaxial
IoT	Internet of Things
K8s	Kubernetes

MAC	Medium Access Control
MEC	Multi-access Edge Computing
MSO	Multiple System Operator
NIST	The National Institute of Standards and Technology
OCI	Open Container Initiative
OLT	Optical Line Terminal
OOB	Out-of-band
OSS	Operations and Support Systems
PON	Passive Optical Network
QAM	Quadrature Amplitude Modulation
QoS	Quality of Service
RAN	Radio Access Network
RMD	Remote MAC-PHY Device
RPD	Remote PHY Device
SCTE	Society of Cable Telecommunications Engineers
SIG	Special Interest Group
SLA	Service Level Agreement
US	Upstream
VM	Virtual Machine

1 Introduction

Distributed computing systems are composed of nodes that are interconnected by a network of some type. Seen as a whole, the system has underlying physical infrastructure layer that executes software for the system. The connected nodes can be seen as logical entities with varying quantity of compute-enabling resources, such as processing, memory or storage. A node can be dedicated physical server machine, as in the traditional system architecture model [1]. However, in recent years, this traditional physical node representation has evolved to virtual representation. Virtualization technologies have decoupled node execution environment from underlying physical resources. Analogously, the diverse set of traditional network functions are evolving to virtual form, decoupled from physical infrastructure. The same virtualization trend can be seen to have occurred also for software. Distributed systems now use abstract service interfaces to communicate with other services. It has become canonical to refer to this aggregation of virtual nodes and services as *the cloud* [2]. Adoption of cloud has been strong in recent years, especially in enterprise, and seems no signs of stopping [3].

While computing systems must have underlying physical representation, increasingly that detail is relevant only for intermediary system agents. For example, a control software that *orchestrates* allocation of virtual nodes over physical infrastructure is an intermediary agent for node end-users. In case of computing nodes, the orchestrator might allocate virtual machines to be run on physical server machines. For network functions, another orchestrator might configure virtualized switching and routing functions to physical net-

work devices. For end-users who operate virtual nodes or network functions, the physical context has lost much of its meaning. While cloud service providers such as Amazon [4], Google [5] and Microsoft [6] are well-known major vendors who offer virtualized base infrastructures as a service, there are others who offer higher-layer platforms that utilize the base infrastructures for more targeted purposes.

The motivation for using virtualized computing resources emerges from its beneficial attributes that increase flexibility and improve physical resources utilization [7]. In essence, virtualization allows for more applications on fewer number of physical servers, which reduces total power consumption, bringing costs down. Traditionally hypervisor-based virtual machines have been used as virtualization technology for infrastructures and applications. However, in recent years *containerization* has emerged as more lightweight virtualization method for applications [2]. Containerization uses so-called *containers*, which are prepackaged applications that run isolated from each other. The main benefit with containers compared to virtual machines is reduced execution overhead and greater portability. However, the challenge until recently has been the orchestration of containerized applications in distributed systems [8].

Kubernetes [9] is currently the most popular platform used to orchestrate containerized applications in the cloud and distributed systems [10]. It has become the canonical interface, or "operating system", between users and cloud infrastructures. Few years back, many trends and studies indicated this would happen [11–13]. Largest cloud providers have already integrated Kubernetes into their platforms. Along with Kubernetes, the concept of *cloud-native* application, as an application that is intentionally developed for the cloud, has also matured. Kubernetes and cloud-native model are inherently coupled. Consequently, Kubernetes has major focus in this thesis, as will be further explained below. While there are other container orchestration platforms, they are out of scope for this thesis.

Yet another development in distributed computing has been the emergence of low-

resource edge devices, which include so-called Internet of Things (IoT) devices. Here the term *edge* is used to refer to distributed systems located close to end-users. While centralized cloud infrastuctures have been virtualized and have mature cloud-native frameworks, the edge devices present a challenge [14]. They generally do not have enough computing capacity to employ virtualization platforms, such as hypervisors or container orchestrators, in their standard form. Furthermore, the edge environment is highly heterogenous which goes against core assumptions behind many of the virtualization platforms, designed for homogenous data center environments. A suggested solution has been *edge computing*, where edge devices offload computing to intermediate servers residing close to cloud edge [2]. To this end, the current desire of the industry and cloud community is to move now-mature cloud-native model from centralized clouds to the network edge [14]. To realize this *edge-native* model, first it requires more lightweight virtualization platforms that can work in constrained edge environments. To this end, there are already some alternative Kubernetes distributions that are targeted for the edge. Secondly, while containers have much smaller resource footprint compared to virtual machines, that may still not be enough for many edge devices. Thus, the edge-native requires even more lightweight solutions to run workloads in resource-constrained devices. Here, WebAssembly [15] seems like a promising alternative to traditional container runtime solutions [14].

Changing the focus a bit, broadband *access networks* are type of networks located in the edge. They enable broadband data access to the Internet for end-users [16]. Cable access networks use coaxial cable as their physical medium in last kilometers. In recent years, cable access has seen transformation of their architectures to more distributed form. The same has happened with other types of access networks. In distributed architectures, some of the network functions that are located at operator's on-premise sites, are moved closer to end-users. The functions, which usually relate to particular physical access medium, are now implemented in *remote access nodes* located in the field. At the

same time, the optical feeder network between operator's centralized site and the nodes becomes pure IP network. This brings opportunity to share the same feeder network for multiple access services. For example, if a cable system operator in parallel manages passive optical network (PON) or radio access network (RAN), they can all use the same IP feeder network. In addition to unified IP, another opportunity is to co-locate more than one access network functions in the same remote node. For example, a remote node could have output ports for both coaxial and optical fiber mediums. While not all network functions can be combined due to fundamental differences between access mediums, it is still possible to use one access medium as overhaul network for protocol packets of other access types [17]. For example, a small-cell of RAN could be located behind cable system's modem, while the RAN protocol packets were sent over a cable system to the RAN management system. While this general *convergence* of access network technologies is an important trend that underlies this thesis, the convergence is not the main focus.

Access network operators have a desire to virtualize their backend site services, while moving physical medium access functions to remote nodes in the field. For this goal, the now mature cloud platforms provide partial solutions [18]. While many network services can be moved to run in centralized public clouds either as VM-based services, or as modern cloud-native services, that is not possible for all the services. Some services may be latency sensitive, or there are other restrictions, which do not permit the services to run far away from access network end-users. Some of these services can still be virtualized on standard server machines running in operator's own premises. Here, the cloud-native approach may still be applicable, depending on capabilities of operator's site. However, convergence and virtualization of remote nodes is also in the sights [18]. Edge devices in end-user premises might also benefit from having deep edge computing support. However, when moving deeper into the access network, virtualization and cloud-native model gets increasingly more difficult to implement using currently existing tools, as already discussed. Therefore, to fully embrace cloud-native model in access networks domain

requires careful analysis of alternative orchestration solutions. Works like by Vanõ et al. [14] indicate there are still many unsolved problems, and that the research for edge-native is still in early stages.

Understanding the requirements and available options for cloud-native model adaptation to the edge-native model is the main focus of this thesis. Here, cable access networks work as the driving background context. However, this is not how this work was originally envisioned to be. The study began with intention to study interfaces of different cloud cluster platforms that existed back then, with goal to implement the interface of the platform that was found most suitable. The implementation would have been used to integrate a specific cable access network product with the chosen platform. While Kubernetes was one of the early canditate platforms right from the start, it was not the only one that was considered at the time. Furthermore, the scope of platforms was not limited to only container orchestrators. Over time, however, the original study transformed into what it is here. At least two events are responsible for this change. First, the target edge product was canceled, which obsoleted part of the thesis. Second, during first half of the study, understanding of the topics of cable access networks and cloud-native model increased to the extent, that there became a realization that the original study goals did not make sense anymore. The learned fact that Kubernetes has practically become the de facto container orchestrator of the cloud, could not be ignored.

Therefore, Kubernetes is another central topic of this thesis. Any edge-native solution most likely will have Kubernetes in some form as one piece of the puzzle that will orchestrate containers. However, in edge-native context, containers may not always be the optimal deployment method for applications and workloads. Even if containers are lightweight, they still require certain capabilities from underlying runtime environment. The edge devices may also lack other resources to run containers. Consequently, alternative Kubernetes-compatible application runtimes suitable for the edge are also studied. Here, the already mentioned WebAssembly is one interesting possibility. To gather better

understanding how to exploit Kubernetes in cable access cloud-native transformation, this thesis conducts a case study which is described in Chapter 4.

This thesis has its research questions come in two parts. The questions in the first part are about understanding the current status and future trends of the topics discussed above, namely cloud-native model and cable access networks. These questions are explored mostly through informal literature study. The research questions for the first part are:

RQ1. How the cloud concept has evolved to its current state with respect to cloud-native model and container orchestration?

RQ2. What is the current state of cable access networks in context of general access networks evolution?

RQ3. What are the future trends of cloud-native model and cable access networks, and how these trends intersect and relate to each other?

The RQ1 and RQ2 are explored extensively in Chapter 2 and Chapter 3. Historical context is also included to better understand the background. The results of RQ1 and RQ2 are summarized in Chapter 6, but the role of these questions is primarily to widen the understanding of their respective topics, in order to help guide the exploration of any following questions. For RQ3, the future trends of the two seemingly separate topics explored for RQ1 and RQ2 are studied, and their relation to each other is analysed. The RQ3 is explored at the beginning of case study Chapter 4. The results of all the questions RQ1, RQ2 and RQ3 in part one are used as a background context and motivation for further exploring RQ4 in the second part of this thesis through a case study.

The second part has only one question. It derives directly from the first part as described above, as the answers to first part's questions took form, and as understanding of the topics increased. A follow-up research question was formulated, which is:

RQ4. What are the current options to leverage Kubernetes in context of edge-native cable access convergence?

The RQ4 is studied and analysed as a case study in Chapter 4. The case study is conducted through the use of literature sources. Among extensive use of literature sources, online sources, such as GitHub for different software resources are also used. Chapter 4 contains analysis and discussion of the case results, while Chapter 5 presents possible steps for future work. The final Chapter 6 gives concluding remarks for this work.

2 Cloud-native orchestration

This chapter provides an overview of concepts related to the cloud-native model. Two main branches of virtualization technologies are covered in dedicated sections, since virtualization is the most important enabler of the cloud. The chapter concludes with description of Kubernetes fundamentals, as the orchestrator platform is at the core of this thesis.

2.1 Cloud computing

This section provides an overview of cloud computing. Relevant concepts, history, technologies and paradigms related to cloud and cloud-native are covered.

2.1.1 Introduction to the cloud

Some ten years ago cloud computing was seen as still evolving paradigm with much potential [19, 20]. Now, a decade later, there is little doubt that the cloud has brought the prospect of computing as a metered utility close to a reality [21, Ch. 1], similar to how electric grids evolved to standardized, always available utility service [22]. Large cloud service providers (CSPs) of today, such as Google, Amazon, or Microsoft have mature and extensive public cloud service offerings, while open source community has produced many mature platforms, such as Kubernetes [9], which can be used to deploy cloud infrastuctures and applications for various needs.

According to McHaney [23, Ch. 1], the cloud emerged as a solution to the problem of IT systems having become too complex to manage by specialists within different organizations. The challenges of IT relate to cost reduction and capacity planning efforts that arise from owning resources, such as on-premise equipment or software. Ownership management of resources leads to unpredictable costs and lack of organizational agility. McHaney and others [24] assert cloud computing is a solution to simplify management of infrastructures that complex software systems need. Furthermore, cloud is a solution to simplify management of applications themselves. The simplication is made possible by a software-based abstraction layer, namely virtualization layer, that allows sharing of computing resources efficiently. Using cloud system, an organization can purchase computing capacity or software services from one of the CSPs, without need to own dedicated equipment. Still, the organization has always an option to purchase equipment as owned on-premise infrastructure and install private cloud to get many of the same benefits as public cloud offerings.

Cloud computing can be simplified to mean a system that delivers various computing applications and services[1] over a network such as the Internet [23, Ch. 1]. But cloud computing has also broader meaning, encompassing not only the interconnected applications and services, but also the whole technology stack underlying all the software [2] [23, Ch. 1]. In cloud computing, the services can be thought as resources, which make use of other cloud resources in layered and shared fashion. Often the exact location of the resources and their technical details are not known to the users of cloud system. Cloud computing can also be seen as a mindset in which various computing resources are shared and can be used almost from anywhere [23, Ch. 1].

[1] Applications and services such as servers, databases, storage, networking, software, data analytics, security solutions, organizational systems, virtual computers and more. [23, Ch. 1]

2.1.2 Definition of cloud computing

While there are many definitions for cloud computing, literature often refers to one proposed by The National Institute of Standards and Technology (NIST):

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [25]

The NIST definition concludes with:

> "This cloud model is composed of five essential characteristics, three service models, and four deployment models."

The referred characteristics are:

On-demand self-service. Cloud system users can decide to independently provision more computing resources as needed. This can be done automatically without direct human interaction.

Broad network access. The cloud services are available over a network and accessed through standard techniques that promote use by different client platforms.

Resource pooling. Same physical computing resources are opaquely shared between users using multi-tenant model. A user has no direct low-level control or knowledge over the location of the reserved resources. The resources can include storage, computing, memory and network bandwidth.

Rapid elasticity. Resources can be manually or automatically scaled outwards or inwards in proportion to demand. To the user, the available cloud system resources can appear to be unlimited.

Measured service. System state and usage is measured at infrastructure level or at some higher service level. Metrics are collected for purposes of control, optimization and billing. Examples of infrastructure measurements are computing and storage usage metrics. At service software level, number of user connections and consumed bandwith are examples of used metrics.

The three service models, or cloud layers, in context of public cloud are:

Infrastructure as a Service (IaaS). Service provider offers fundamental computing resources, such as CPUs, storage or networks to users. Typically the resources are virtualized and exposed as virtual machines or containers. Users deploy software of their own choosing on the virtual resources, while the physical resources are in control of service provider. A physical unit can be opaquely shared by more than one user, as in multi-tenant model. By definition, users can request more capacity on-demand.

Platform as a Service (PaaS). Service provider offers an environment or platform on which users can deploy their application. The platform supports a set of languages, services and tools related to one or more application domain. Users are in control of application and configuration of its environment, but typically have no control over the underlying hardware or operating system.

Software as a Service (SaaS). Applications or services whose functionality is available for users using either a thin client interface or a programming interface. Users usually have control of only limited number of application configuration options.

The four cloud deployment models in NIST definition are:

Private Cloud. Cloud infrastructure is deployed for exclusive use by single organization. Cloud ownership and management may be held by the organization, by a third party, or any combination of them. The physical infrastructure can be located on organization premises or rented off-premise.

Community Cloud. A private cloud that is intended for exclusive use by a specific community of users.

Public Cloud. The cloud provides resources and services that are open for use by the general public through the Internet. The cloud infrastructure is located on provider premises.

Hybrid Cloud. Composition of at least two distinct cloud infrastructures. Commonly it comprises of a private cloud for critical business resources and public cloud chosen for its cost-efficiency.

Research literature and other sources seem to agree with all or parts of the NIST definition, especially with the service and deployment models: for example, taxonomical research in [19], more recent publications [21, Ch. 1] [2, 26], Wikipedia [27] or Amazon web article [28]. For sake of simplicity, this thesis will ignore community cloud model and merges it with private cloud model. Other than that, the NIST definition as presented here is assumed.

Finally, it should be noted that cloud trends include other service models [27]:

Serverless Computing. A model or architecture where service provider fully manages the underlying cloud resources and pricing is based on actual amount of resources used.

Function as a Service or FaaS. A model that leverages serverless computing to deploy user defined stateless functions in the cloud, which are executed in response to events. Each function call is billed on actual execution time. FaaS runtime platform manages lifetime of function instances.

Many of the traditional cloud service models use techniques where at least some of the application software components must be always resident, i.e. always-on, implying that associated resources may be billed continuosly. FaaS is premised on serverless computing

model where resources are meant to be active only for as long as really needed. It is thus possible to scale the resource usage close to zero, which is one major benefit of the serverless model [26].

2.1.3 History of the cloud

The evolutionary path to cloud computing has had many incremental steps according to McHaney [23, Ch. 1] as gradual changes of technology and use of computers has taken place. While the commercial mainframes of 1960s can be taken as a starting point, they had little similarities to today's cloud systems. It was the multi-user mainframes from 1970s that first had some resemblance to cloud systems, only in much smaller scale. In that era, users accessed mainframes through simple terminal device to get access to shared resources. Since the mainframe used virtual machine computing model, it seemed like the users had their own private computer system.

Invention of microprocessor eventually led to the intermediate era of personal computers in 1980s. This event saw some of the computing power move from mainframes closer to users, while mainframes were delegated more to the role of backend servers for request processing. At first custom networkings of the time restricted communication to work only within organizations. This was later solved in the 1980s by the TCP/IP networking standard, which eventually became the protocol behind the Internet. In the early 1990s the Internet got its easy-to-use web interface in form of the World Wide Web (WWW). The protocol behind WWW was called Hypertext Transfer Protocol (HTTP), which made it much easier to access standardized web resources and services.

This all led to explosion of WWW and more advanced Internet services started to appear, such as social media. Many companies developed web applications, which required technology solutions in problem areas such as distributed databases and processing for millions of users. These solutions and systems became known as Web 2.0, and now some of the computing power moved back to shared resources in data centers owned by com-

panies. The web applications were accessed through thin-clients running in a browser on user's computer or mobile device. McHaney concludes that the cloud computing era truly began around 2005, both for individuals and organizations. From that point on, cloud computing can be seen to have been driven predominantly by Amazon, Google and Microsoft [20]. [23, Ch. 1]

2.1.4 Enablers of cloud computing

McHaney [23, Ch. 1] lists two hardware and two software developments as the main drivers of change in cloud computing evolutionary path:

Significant changes in hardware capabilities. Increases in computing power and storage density allow using more complex algorithms and techniques as required by advanced cloud environments.

Advances in network technology. High capacity and low-latency broadband networks in gigabit ranges make distances shorter. It becomes less relevant to have locality of data as performance optimization.

Service oriented architecture. SOA is both an enterprise-level software architecture style and vision of how to develop, build and deploy systems. Its main concept is reusable services or modules that are integrated as large scale system. The difference to traditional design systems is that SOA is specifically network-oriented. Software module communication is defined as well-defined network APIs, not as procedures and their parameters.

Advances in virtualization technologies. Virtualization allows to share hardware resources in isolation between multiple users. This increases resource utilization rate, which reduces power consumption.

The traditional method to deploy only limited set of applications on single physical machine underutilizes hardware resources [1]. This is because the resources reserved

for unused applications could be potentially used for other purpose. The static nature of the underlying system makes it difficult to plan deployments for optimal resource usage. From McHaney's list of major technology developments, virtualization can be seen as the solution to the hardware underutilization problem, because it can be used to efficiently share resources of a single system to achieve economies of scale [23, Ch. 1].

Virtualization means to have an abstraction unit that represents subset of shared computing resources [1]. Other software deploys against the unit as if the software was running directly on real resources. In truth, the abstraction maps its apparent resources, or *virtual resources*, to subset of available real resources. The mapping can typically vary dynamically adjusting to changing workloads [29]. Virtualization enables multi-tenancy, where multiple unrelated services can be deployed simultaneously on the same underlying system. For example, if the abstraction unit presents complete virtualized server machine, as in case of hardware virtualization, multiple virtual servers could be deployed on single physical computer hardware. If the abstraction presents OS kernel and its resources, as in operating-system-level virtualization, multiple separate OS environments along with applications could be deployed on single OS installation. All this leads to more effective utilization of computing resources and enables resource sharing between users [23, Ch. 3].

Virtualization is often implemented as a virtualization layer where the abstraction units are managed by a specialized control software [7]. The control software may run directly on hardware or it might run as user space application inside operating system. The control software can be used to create new abstraction units on-demand, if there are enough free resources in the underlying system. A user with access to an abstraction unit can deploy software of own choosing inside the unit, limited only by constraints of the virtualization technology. For example, in hardware virtualization, a full operating system along with other software can be deployed, while in OS-level virtualization the kernel is shared between all the units [2].

Virtualization can be seen as the most important cloud enabling technology according to McHaney [23, Ch. 3] and others [1, 20], especially for IaaS. This can be seen also from that virtualization directly contributes to three of the five essential NIST cloud characteristics: on-demand service, resource pooling and elasticity. Elasticity by itself is an important characteristic, so much that it could be also listed as one of the cloud enablers [2]. Elasticity is the degree to which a cloud system adapts to changing workloads [26]. An elastic system automatically provisions or de-provisions virtual resources according to current demand. Highly elastic system, however, requires efficient management and monitoring of virtual resources. Efficient resource management is thus of great importance as it results in high scalability of the cloud and in reduced operational costs [2]. To a large degree, virtualization makes this all possible.

2.1.5 Application deployment models

Evolution of cloud computing and its application architecture has been a continuous process to improve resource utilization, according to Kratzke [26]. Over time this has changed application development and deployment models. Early approach was to use virtual machines (VMs) to consolidate large numbers of bare metal machines to utilize physical resources more efficiently. Each server machine had running many isolated VMs along with their applications, which in essence increased the application density of physical servers. At the same time, the VM model affected how applications were deployed to the servers. An application and its dependencies was now deployed as a VM image, which became the standard deployment unit of the cloud. While the images were more lightweight compared to traditional application deployments to bare metal hardware, they were still relatively heavy by their size. Virtual machines now form the backbone of IaaS services [26].

Service oriented architecture (SOA) style was another early deployment model, listed by McHaney as one of the important cloud enabling technologies [23, Ch. 1]. SOA is

monolithic architecture from deployment point of view [26]. This means that the application modules cannot be deployed indepedently, because the modules cannot execute in isolation from the application[2] [30]. The complete application must be deployed as a whole, or not at all. For ease of deployment, this often leads to packaging an application as single VM image [26]. Monolithic architectures are not applicable for many modern distributed cloud applications, as they often have strict requirement of no downtime. Therefore, service oriented architecture eventually evolved to more indepedently deployable microservice architecture model [26]. In a sense, microservices can be seen as more pragmatic version of SOA.

A microservice is an independent software module with a single responsibility that it implements well [30]. Other distributed modules and applications communicate with the microservice using a well-defined messaging interface. Microservice architecture is a composition of many microservice modules, which work as a cohesive unit. Microservice architecture makes a cloud application more scalable and maintainable, because each microservice can be deployed independently [30]. For example, an updated version of a microservice can be deployed in parallel to old one, making gradual transitions to new application versions possible. Consequently, it is not necessary to reboot the full application when a single module gets updated. According to Kratzke, a major reason why microservice architectures have seen success may have been that the service instances could be standardized as self-contained deployment units known as containers [26]. Containers make use of operating-system-level virtualization, which is inherently much more lightweight compared to hardware virtualization. Containers have also faster startup time, further enhancing the inherent elasticity that microservice architectures already have. It can be said that microservices naturally lend themselves to containerization [30]. Containerization will be discussed in section 2.2.2.

[2] The modularisation abstraction of SOA requires that the modules share resources of underlying host machine, such as memory or file system database. [30]

Microservice architecture and containers provide efficient model for cloud deployments, but they are still conceptually always-on [26]. This means that even when there are no service requests, at least some services must be always instantiated, leading to persistent resource consumption. Furthermore, microservice architecture faces the challenge of efficiently managing each service container for elasticity [26]. For the management concern, orchestration platforms such as Kubernetes have emerged as an important solution. However, orchestration platforms do not solve the always-on issue. For that, Kratzke predicted in 2018 that serverless architectures and especially function as the service (FaaS) could be the next trend in cloud deployment model evolution [26]. In FaaS, the deployment units are fine-grained stateless functions, which are fully managed by a platform of a cloud service provider. The function execution is triggered by events from client application, and is billed by function runtime.

Today, all major cloud service providers include serverless and FaaS among their offerings, AWS Lambda being possibly the most prominent [26]. Among organizations that already lean to the cloud platforms and services, FaaS has become one of the mainstream models [31]. FaaS has the beneficial attributes of fine-grained deployment, bounded lifetime and stateless service concept, which can lead to near zero resource usage when there are no service requests. According to case study by Villamizar et al., FaaS can lead to up to 75% cost reduction compared to microservices, at least in web application contexts [32]. However, serverless computing is not without open challenges and drawbacks [26]. Runtime constraints present themselves as startup latency from zero requests state, while function state constraints require some form of external cache to be used. Furthermore, function compositions have "double billing" issue when functions blindly call other functions synchronously. Vendor lock-in, client software complexity, and development time complexities are some other problems. Furthermore, serverless architectures have increased security concerns from their larger attack surface.

2.1.6 Cloud-native applications

As the cloud as a platform has evolved over the years, so have application development, deployment and management models. The industry has transitioned from monolithic application models to service-oriented models [26], with microservices being the current preferred architectural style. As already discussed, loose coupling and isolated state of microservices naturally lend to containerization. Container orchestration platforms, such as Kubernetes, have proven to be efficient tools to automatize much of the complex container orchestration in scalable and elastic manner. Combining cloud models and tools with principles that increase collaboration between software developers and IT operations has enabled faster application development and deployment processes.

An application that is intentionally developed for the cloud from the start using the principles and tools mentioned above is known as *cloud-native application* (CNA) [14]. Along with maturation of the cloud by strong pragmatic push from the industry, the definition of cloud-native application has reached more concrete form, entailing much of the topics mentioned above. Few years back the situation was different, as the definition of CNA was only taking its form, with research like by Kratzke [33] seeking to understand CNA concept on theoretical level. As for one definition of CNA, following is from Cloud Native Computing Foundation:

> "Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.
>
> These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil." [34]

As CNA paradigm emphasizes microservice model, it is worthwhile to examine how that model has evolved. Here, according to Kratzke [26], based on Jamshidi et al. [35], there has been three generations. In the first generation, microservices were simply packaged as containers managed by some orchestration framework. Each service was responsible for discovering other services and implementing communication protocols. However, as the number of services per application increased over time, management of all the complexity and variety among service implementations became problematic. Therefore, in the second generation, microservices started to use service discovery and fault-tolerant communication libraries to reduce complexity and increase service reliability [26]. Service discovery technologies let services communicate with each other without knowing their exact network location, while fault-tolerant communication libraries improve communication efficiency and reliability [35].

The third microservice generation introduced standard service proxies, or *sidecars*, as intermediaries to improve reusability [26], as seen in Figure 2.1. Sidecar encapsulates

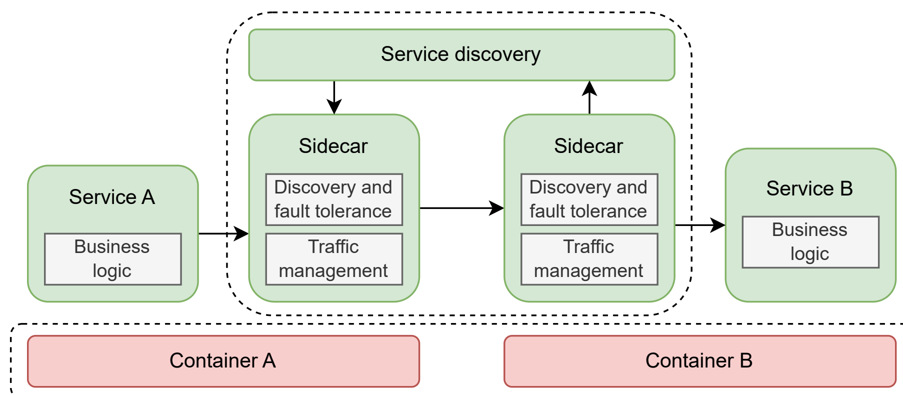

Figure 2.1: Cloud-native application implemented as microservices that run inside containers. A co-located sidecar provides discovery, communication and monitoring services for the application microservice. [35]

reusable communication functions, presenting them as another co-located service to application service developers [35]. Finally, building on sidecar technology, *service mesh* technologies present a fully integrated communication, monitoring and management platform for service networks [35]. While in previous generations a service communicated

directly with other services, now with service meshes, the application communicates explicitly with co-located sidecar interface, which relays the operations to other services through the service mesh.

Jamshidi et al. envisioned in 2018 that the next generation of microservices would be serverless architectures, such as FaaS, to essentially turn an application to collection of functions [35]. Since then, while the serverless model has been successful especially in public clouds [14], it has not fully replaced the traditional microservice model. Serverless computing is still one viable architectural design choice how to implement CNAs for modern cloud.

From the above it is clear that microservice-based applications, and thus CNAs, have many interacting software components. Intermediary service meshes are important design elements which allow building reliable and scalable applications, while also abstracting the runtime environment for individual service component. Still, deployment of CNAs in elastic and scalable manner would hardly be possibly without automatized deployment of application components, sidecars and other support services. The automation is enabled by container orchestration platforms, which are considered as essential pieces in the full CNA paradigm [26]. In practice, Kubernetes has taken the place as canonical container orchestrator that is used for modern CNAs [14].

Finally, as CNAs have their characteristic architecture designs and deployment tools, they also have their own characteristic development methods. The intent behind these methods is to build quality software quickly, to be run at any cloud layer. The underlying methods are to separate software component development to dedicated teams (i.e. by a microservice), to increase co-operation between development and IT operations inside organization, to update software in higher frequency, to test and commit code changes often to shared repositories (Continuous Integration), to deploy applications into production through well-defined testing and validation pipelines (Continuous Deployment), and to automatize all these procedures as much as possible. Together, these methods and

principles are known as DevOps practice [26].

2.1.7 Edge computing

Public clouds are usually located in small number of centralized places far away from the end users at the network edge [36]. The consequence is that these clouds cannot properly serve real-time applications on devices located at the edge [24]. These applications may have strict operating requirements, such as low latency, low jitter, high bandwidth consumption, and mobility functions. On one hand such edge applications would significantly benefit from cloud computing capacity, because they can be hard to fully localize on battery-limited or resource-constrained devices [36]. But the distances to centralized cloud servers bring implementation challenges, and the application requirements cannot be met without degrading the user experience.

The trending of mobile computing[3] in the early 2000s started the evolution of proposals on how to extract the full potential of mobile but low-resource edge devices [36]. After some initial suggestions in the context of pervasive computing, which included Cyber Foraging by Satyanarayanan [38], the soon to be emerging cloud computing eventually gave the answer at least to the question who would own the computing servers. The integration of centralized cloud and mobile computing became known as *Mobile Cloud Computing*. However, because the computing is still done in far away centralized cloud, MCC alone does not meet the requirements of real-time applications [24]. Other practical solutions to move computing nodes closer to the edge devices were needed. Figure 2.2 illustrates differences between some solutions that are discussed next.

One of the next early proposals was *Cloudlet*, introduced in 2009 [40], which is defined as a cluster of Internet-enabled computing infrastructure available for use by nearby mobile devices [36]. Cloudlet was meant to implement a full service, so that it did not access the centralized cloud. Mobile devices offloaded computing to these clusters, which

[3]Computing performed via mobile, portable devices, such as laptops, tablets, or mobile phones [37].

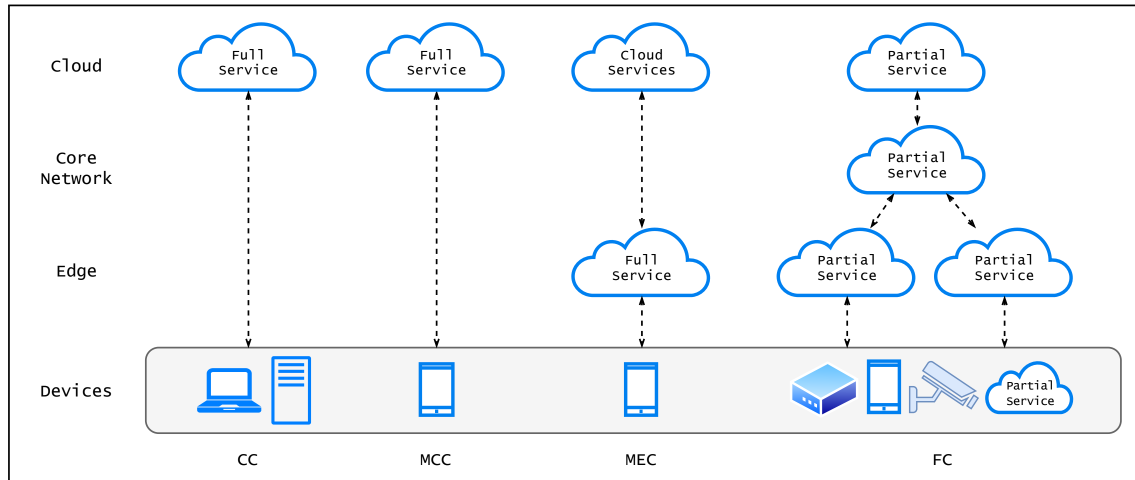

Figure 2.2: A comparison of Cloud Computing (CC), Mobile Cloud Computing (MCC), Mobile Edge Computing (MEC) and Fog Computing (FG) paradigms. Adapted from [36].

made low-latency cloud-like platform available to them. Cloudlet usage for mobile devices was also the de facto birth of a paradigm known as *Mobile Edge Computing* (MEC). Since cloudlet model emerged as a viable solution, the European Telecommunications Standards Institute (ETSI) formed in 2014 an industry specification group to define and create a standard reference implementation of MEC for cellular networks [36], which became known as ETSI Mobile Edge Computing (ETSI MEC). The standard has been recently renamed as Multi-access Edge Computing [39] to emphasize that it can be used for other than mobile networks, even if the driving factor are wireless networks [40]. The ETSI MEC architecture is shown in Figure 2.3.

Fog computing (FC) introduced by Cisco in 2012 was another attempt to overcome cloud integration challenges [36]. Fog computing can be defined as a system architecture that distributes computing, storage, control and networking functions closer to the end users near the cloud edge. Fog computing is based on concept of co-operating Fog Nodes (FNs), which form a multi-level hierarchy that distributes cloud services for edge devices. A fog node can be any device that has enough computing, storage and networking capacity to implement complex services [36]. Fog computing encourages cloud systems where a service is decomposed and provided by many FNs located at any level from cloud edge to

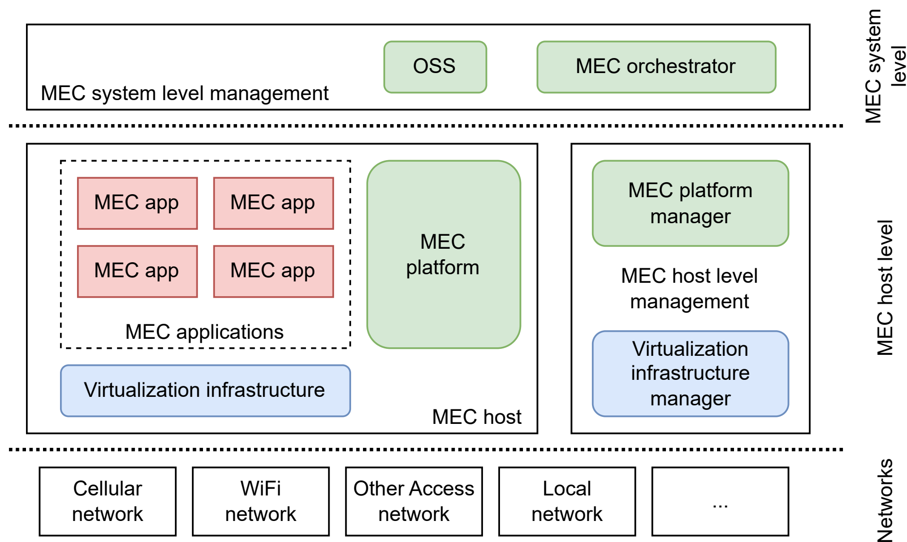

Figure 2.3: High-level view of ETSI Multi-access Edge Computing (MEC) framework. MEC host contains application instances and MEC platform, which is a collection of functions needed to run MEC applications on particular virtualization infrastructure. System level MEC orchestrator has the complete overview of the MEC system, implements constraints based selection of suitable hosts to instantiate MEC applications, and manages application life cycles. MEC host level management handles host specific MEC functionality. [39]

the cloud backbone, as seen in Figure 2.2. Fog computing does not replace the centralized cloud, but complements it. In contrast to early MEC, where the cloudlet implemented the full service, fog computing premise is to include the centralized cloud as part of the overall service when needed.

In general, the extension of cloud from centralized model to a distributed model where data processing and storage functions are deployed close to network edge is known as *Edge Computing* [24]. It encompasses or overlaps many paradigms, including the ones already outlined above:

- Use of cloudlets for any device at the edge, whether mobile or fixed [36].
- Fog computing by its definition.
- Mobile Edge Computing and its successor Multi-access Edge Computing.

Edge computing improves overall application efficiency through reduced latency and jit-

ter, reduced bandwidth usage of core cloud, enhanced mobility at network edge, and improved service availability [24]. Edge computing also improves privacy since data can be stored closer to end-users, while at the same time it brings new security challenges from its wide distribution of heterogenous computing nodes. If definition of edge computing is extended to mean that the cloud is complementary to the paradigm, then that leaves the FC and MEC as the two prominent popular paradigms for edge computing [24]. Both paradigms have seen standardization efforts [36]. While they have differences that inherit from their separate origins, there are enough similarities to say that both have contributed to the convergence of edge computing as a concept. Nowadays, fog computing can be understood either as a near synonym to edge computing [2], or as an umbrella term that encompasses computing at any point in cloud-to-edge continuum, i.e. it is a superset of edge computing [37]. On the other hand, ETSI MEC is more a practical architecture model within edge computing context for wireless edge devices. While many different implementations exist for MEC, it is still immature technology with many unsolved problems [40]. All this said, edge computing and related concepts discussed here are an evolving research topic with no fully accepted definitions.

The fundamental enabling technology for edge computing is virtualization, as it is for the centralized cloud [24]. Virtualization effectively decouples applications from underlying hardware. This is especially important for edge computing because its node computing performances vary a lot. However, there are other distinctive differences with cloud computing, which include location-awareness, mobility-based services, resource-constrained devices and wide-spread distribution of devices. The consequence is that heavy-weight virtualization techniques like virtual machines (section 2.2.1) are not as applicable for edge computing. Alternative, more lightweight technologies such as containers (section 2.2.2) are needed. But even containers are not always enough, as some edge devices such as sensors or actuators are so resource-constrained that they do not support any virtualization techniques [24].

In addition to virtualization of traditional computing resources, edge computing has strong emphasis on Network Function Virtualization (NFV) and Software Defined Networks (SDN), because they further decouple edge applications from underlying hardware [24]. NFV virtualizes different network functions, such as firewalls or load balancers, and implements them as software solutions that run in common servers and use basic network elements [41]. SDN decouples networking to data and control planes, allowing configuration and management by software means [41]. The two networking concepts are complementary and they can be used either in isolation or in parallel.

2.2 Virtualization technologies

The two dominant virtualization technologies used in cloud are *hardware virtualization* and *operating-system-level virtualization* [29]. Hardware virtualization uses *virtual machine* as its deployment unit [7]. It is used especially in IaaS space, and when deployments need security or flexibility in choice of operating system [1]. OS-level virtualization is the more recent technology in cloud context. It is the underlying technology behind *containerization* trend. Containerization has so-called *containers* as deployment units, which share the resources of underlying single kernel. Containerization, however, is more a technique to standardize deployment units in lighter weight form compared to virtual machines [26]. The following sections introduce these two technologies. Unikernels are a third technique comparable to containers as suggested by Kratzke [26]. Unikernels, however, have seen little research and have not reached widespread use. They are therefore mostly out of scope for this thesis.

2.2.1 Hardware virtualization

The traditional technique to virtualize physical hardware resources is to use specialized control software called *hypervisor*[4] to manage instances of *virtual machines* [42]. A virtual machine (VM) is logically isolated abstraction unit that presents full computing environment with CPUs, memory, storage and networking, which map to a subset of underlying physical resources [29]. The virtual machine seems like a real machine to software running in it. The physical machine where the hypervisor runs is known as the *host machine*, while a VM is known as *guest machine*. Each guest machine is loaded with user chosen operating system, libraries and application software, i.e. the full software stack [2]. This type of virtualization technique is known as *hardware virtualization*, since it decouples the guest machine environment from physical hardware resources. [23, Ch. 3]

At high-level view, hypervisor manages life-cycle of guest machines with operations such as create, run, monitor and delete [23, Ch. 3]. The primary function of hypervisor, however, is to provide a virtual environment for each guest OS, arbitrating their access to host hardware functions [29]. The hardware functions include CPU instruction execution, memory operations, and I/O operations. Each shared access function requires different set of software and hardware mechanisms from the hypervisor and host hardware to work. In some cases, the hardware supports direct access, which requires no hypervisor intervention, while in others the hypervisor must detect and intervene to emulate the access [43]. While the distinction is not always clear [7], hypervisor implementations can be classified into Type-1 and Type-2 (Figure 2.4) [23, Ch. 3]:

Type-1. The hypervisor is installed as a standalone operating system. This is known as native or bare-metal hypervisor, since the hypervisor runs directly on host hardware. The hypervisor runs with the highest CPU privilege level, while the guest OSs have lower privileges than hypervisor. VMware ESX/ESXi, Microsoft Hyper-V, open source Xen, and open source KVM are examples of Type-1 hypervisor.

[4] Hypervisor is also known as virtual machine manager (VMM).

Type-2. The hypervisor is installed as ordinary software in conventional host operating system. A virtual machine instance runs as a process of the OS. Type-2 hypervisor is also known as hosted hypervisor. VMware Workstation and Oracle VirtualBox are examples of Type-2 hypervisor. Type-2 hypervisors are convenient for desktop usage, because they offer nearly seamless integration between host and guest OS graphical environments.

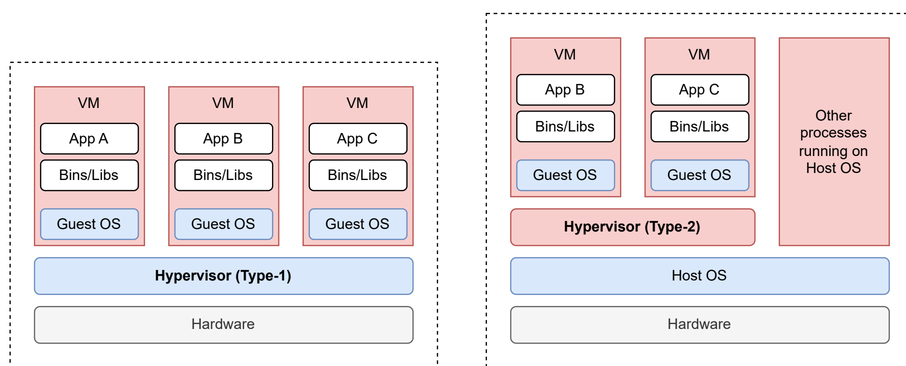

Figure 2.4: Hypervisors are classified to two types depending on their location in software stack. Type-1 hypervisor executes directly on underlying hardware, while Type-2 hypervisor executes as ordinary user process in host operating system.

Generally Type-1 hypervisors are more efficient than Type-2 [2], because Type-1 hypervisor can manage resources directly with no intermediate host OS intervention. Most cloud providers use Type-1 hypervisors to manage their infrastructure.

The underlying hardware architecture defines how much a guest software can use native hardware resources directly, and how much the hypervisor must intervene by other means, such as emulating trapped instructions or using binary translation. If the architecture support for virtualization was known to be heavily limited, in theory a hypervisor could still use intervention to implement most of the hardware features. But such a hypervisor would not be efficient. As Popek and Goldberg formally presented in 1974, for hypervisor to be efficient, it should fulfill three conditions [44]:

- A guest software behaves essentially identical to running directly on host.

- Hypervisor is in complete control of hardware resources.
- A dominant number of guest instructions are executed by the real processor with no hypervisor intervention.

Popek and Goldberg further presented the sufficient conditions for an instruction set architecture (ISA) that fulfills the above properties. First, they classify instructions roughly in two categories: 1) privileged instructions that trap in unprivileged user mode, and 2) sensitive instructions that affect the shared state of the system. In essence, an efficient hypervisor can be implemented when sensitive instructions are subset of privileged instructions, i.e. when all sensitive instructions can be always trapped by the hypervisor. This kind of trap-and-emulate hypervisor was so prevalent in 1974, that the corresponding architecture became later known as classically virtualizable architecture [45].

In addition to CPU functions, virtual machines share the memory functions of the host hardware [46]. In modern architectures, even without any VM present, the host OS virtualizes the physical memory, so that a host process sees only virtual address space [46]. The virtual address space is further partitioned to pages. A hardware memory management unit (MMU) translates virtual addresses to physical addresses using page tables managed by the OS. Since a VM and its guest OS naturally assume the same virtual memory model, the virtualization layer must support translation from guest physical address space to host physical address space, i.e. there must be a second level address translation. Hypervisors can do this using software techniques that use shadow structures for page tables [46]. Software methods are, however, inefficient compared to hardware-assisted memory virtualization. In hardware-assisted memory virtualization MMU supports two levels of address mappings. The first level translates guest virtual addresses to guest physical addresses, while the second level translates guest physical addresses to host physical addresses. The level page tables are managed by the guest OS and hypervisor respectively. Hardware-assisted virtualization improves memory utilization and the performance is higher compared to software methods [46].

Virtualization of I/O functions is another task that hypervisor must be able to do to share resources between VMs. Software and hardware-assisted solutions that exist are full virtualization with device emulation, paravirtualization and direct I/O [46]. The methods will not be explored here further, but the important factor in all of them is to isolate and restrict I/O device access. Similar to CPU and memory functions, hardware-assisted I/O functions are generally more efficient.

Hardware virtualization implementations overall can be classified into three approaches: full virtualization, paravirtualization and hardware-assisted virtualization [23, Ch. 3]. All present hypervisors support full virtualization, and in most cases hardware-assisted full virtualization [47].

Full virtualization. The hypervisor presents complete set of hardware as VM instance. An unmodified guest OS and unmodified guest software running in the VM cannot distinguish the virtual hardware from real. To the guest software it seems as if it was running directly on hardware. Full virtualization approach requires that all hardware functions needed by guest software are supported in the virtualization layer by some method. Classical virtualization is an example of full virtualization.

Paravirtualization. The hypervisor provides an API for a guest OS to communicate its privileged resource needs. The API invocations from the guest OS are known as hypercalls. The guest OS is thus aware of hypervisor presence. Paravirtualization requires that the guest OS is modified by replacing privileged instructions with hypercalls for direct hypervisor access. This approach improves virtualization performance and efficiency [43], but does not permit unmodified guest operating system to be used.

Hardware-assisted virtualization. Host computer hardware has a role in creation and management of virtual machines. The host CPU has built-in command extensions that support virtual machines. In practice, most full virtualization solutions of today require hardware-assistance to improve virtualization performance.

An efficient hypervisor can be implemented for architecture that does not support classical trap-and-emulate style. Originally x86 architecture was thought to be impossible to virtualize classically, because it lacked mechanism to run hypervisor with higher privilege level than guest OSs, and because certain sensitive instructions could not be trapped [48]. However, WMware managed to introduce full virtualization for x86 in 1998, which eventually led for x86 to become the dominant architecture used in cloud [46]. WMware used binary translation technique in their hypervisor, meaning that non-virtualizable instructions are translated at runtime to new instruction sequences that have the intended effect. WMware succeeded also in emulating higher CPU privilege levels for their hypervisor, even if x86 did not have hardware support for such mechanism. Later, major x86 hardware vendors added baseline hardware support for classical virtualization as instruction set extensions: Intel introduced VT-x in 2005, and AMD introduced AMD-V in 2006. However, the first generation hardware had only minor performance advantages over software based techniques such as by VMware [45]. Over the years, and continuing with x86-64 architecture, the vendors have introduced more hardware extensions that improve virtualization performance. For example, Intel EPT and AMD RVI provide support for second level address translation, which improves memory function virtualization performance significantly, while Intel VT-d and AMD-Vi add I/O resource virtualization features [47].

Virtual machines emerged as a solution to migrate services and applications from fragmented deploy form to more compact and manageable image form [47]. Each VM image contains full operating system together with application and its libraries. From application deployment point of view, a hypervisor managed system is the deployment target, while the image is the deployment unit. However, this approach duplicates much of the usage of computing resources making virtual machine inefficient as application deployment method. For example, a typical VM image size can be multiple gigabytes [2]. Containerization technology, as will be discussed in next section, is substitute technology

for application deployments, because it uses resources more efficiently and deploys faster [47]. However, hypervisor is still a necessary virtualization solution when applications require different operating systems or different versions of them [43]. This holds true especially for IaaS model. In addition, virtual machines provide better isolation and security compared to containers [47].

2.2.2 Containerization

Virtual machines are the traditional solution to increase hardware utilization rate in cloud platforms [46]. The solution involves a hypervisor that multiplexes set of operating systems running on a shared hardware. Virtual machines bring significant performance overhead, because the OS instances running inside VMs all have similar system resource needs, which duplicates resource usage [47]. To combat this performance bottleneck, cloud service providers have started to use *containerization* as an alternative method for application deployments [1].

Container as an abstraction unit

Containerization employs *operating-system-level virtualization*. In OS-level virtualization the kernel running on physical or virtual host is shared by each abstraction unit [2]. The abstraction unit is known as a *container*. The Figure 2.5 illustrates differences to hardware virtualization. From kernel perspective each container is an ordinary user-space process, while from inside the container the environment seems like a standard OS distribution [49]. The container is isolated, so that it cannot directly access resources assigned to other containers. Container deployment images include only the binaries, libraries and data files that the container needs at runtime. Compared to VM images containers are significantly smaller in size [1], so that the application density on host can be orders of magnitude higher [47]. Containers also need less resources, have negligible performance overhead, have up to order of magnitude smaller start and stop times, and their life-cycle

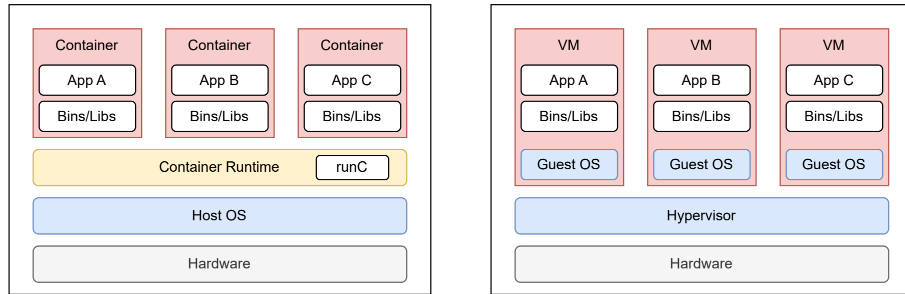

Figure 2.5: Containers (on the left) are abstraction units that package application code and dependencies. They are executed in virtual environments that are managed by a container runtime. The host OS kernel is shared between isolated container instances. Virtual machines (on the right) in contrast are an abstraction of physical hardware. Each VM includes a full copy of an operating system, which makes VM images take much more space compared to containers.

management is easier with help of orchestration platforms [50]. Furthermore, thanks to open standards, containers have better portability across different cloud platforms.

Containers are based on set of Unix and Linux kernel capabilities that have now existed for ten years [49]. The two most important capabilities are 1) the capability to logically separate and isolate process execution, and 2) the capability to set resource constraints for a process. For Linux kernel the capabilities are realized by implementations known as *namespaces* and *control groups*, respectively. Acting as a kind of control software for containers, *container runtimes* (Figure 2.5) use the kernel capabilities to realize containerization [7]. Runtimes include LinuxContainers (LXC) [51], Docker's [52] containerd, and others. LXC was one of the early popular container runtimes, while Docker was released in 2013 as successor to LXC [2]. Docker has quickly become one of the most popular container engines [50], and has established itself as the de facto standard container engine used for applications [33].

Container ecosystem components

While Docker has a large footprint among container ecosystems, the whole landscape is larger than Docker. Four ecosystem component types can be seen [49, 53]: container im-

ages, container engines, container runtimes and container orchestrators. There has been several competing industry formats and technologies for these components. However, container industry has moved towards standards governed under the Open Container Initiative (OCI) [54], which was created to provide common base for containerization. The OCI scope includes three standards: 1) Container Runtime Specification (runtime-spec) that defines how a container file system bundle should be executed, 2) Container Image Format Specification (image-spec) that defines the container image format to be used by OCI implementations, and 3) container Distribution Specification (distribution-spec) that provides standard API to distribute container images. Figure 2.6 shows how these specifications map to different container ecosystem components. Following describes the

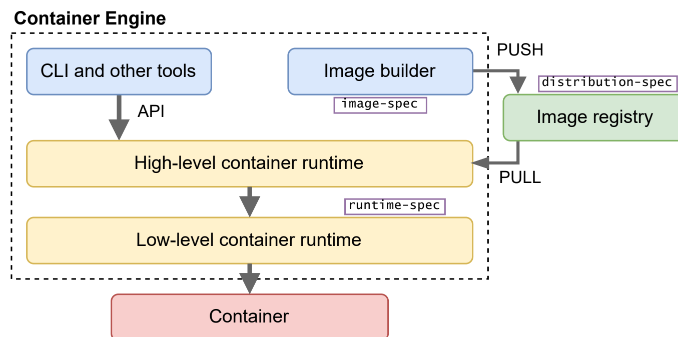

Figure 2.6: Open Container Initiative defines standards for container images, distribution and runtime. The figure depicts how the standards are applied on container engine, such as Docker. [14]

component types in more detail [53]:

Container image. A prepackaged deployment unit that contains the application metadata and files. The complete image consists of one or more *layers*, which are distributed across container ecosystem repositories. While container ecosystems earlier had their own dedicated container image formats, now most of the major ecosystems support standard format as defined in Container Image Format Specification.

Container engine. A front-end component that accepts user requests, pulls container images from repositories, and from user's perspective seems to run the container. To

run the container, the engine compiles various configuration objects and setups the file system layout from container image layers. The layers are merged with union mount to present single file system view with copy-on-write semantics to the container instance [50]. The configurations include meta information that gets passed to the instance, affecting its runtime behavior. As part of the process, the engine will call a *container runtime*, which in turn uses lower level kernel capabilities to run the container instance as isolated execution environment. Docker is a prime example of container engine.

Container runtime. A component that manages container images and interacts with kernel capabilities to launch containerized processes [55]. Container runtimes can be decoupled to high-level and low-level runtime components, as shown in Figure 2.6. The high-level runtime handles the general container image management functions [55]. It delegates runtime control to low-level runtime, which uses the kernel features to create and run container processes. The low-level runtime can also be seen as low-level driver that manages kernel-specific functions [49]. Container runtimes used to be tightly coupled so that no component separation was apparent, even if one might have existed on implementation level. Several runtime subcomponents have since been decoupled from their original projects to open source container libraries and tools. For example, Docker split part of its runtime to high-level *containerd* runtime, which is now used as high-level runtime in other container ecosystem platforms, such as Kubernetes. Furthermore, containerd itself uses low-level *runC* runtime [56], which used to be component of Docker[5]. Today, runC is the standard OCI reference container runtime implementation. All major container engines use runC as their low-level runtime.

[5] LXC was the original container runtime for Docker. Later, Docker team developed libcontainer as a replacement for LXC. When the Open Container Initiative (OCI) was created, libcontainer was donated as a standalone utility known as runC to OCI. [53]

Container orchestrator. Software controller that schedules and manages containerized workloads in cluster of container-enabled hosts. User-provided declarative configuration defines the intended state of the container cluster, while builtin monitoring functions provide realtime information for the orchestrator about the cluster hosts and their container instances. Orchestrator uses the intended cluster state, the current state and current cluster load as basis for scheduling decisions. While not required if intent is to just run containers at single host, orchestrator platforms such as Kubernetes have a crucial role in scalable and highly elastic containerized clouds.

Figure 2.7 illustrates the described container components as layered stack, when Kubernetes orchestrator platform is also included as one possible container tool or engine. Along with Kubernetes comes another important container ecosystem standard: Container Runtime Interface (CRI). It allows Kubernetes to use any CRI compatible container runtime. Kubernetes is described in section 2.3.

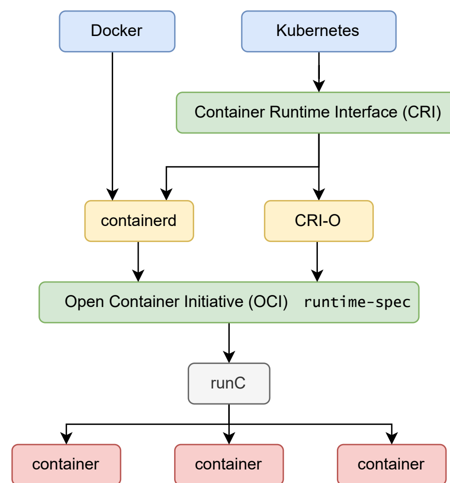

Figure 2.7: Stack of container ecosystem component instances at different layers. There are two major container standards that define various interfaces. The first, Open Container Initiative (OCI) defines interfaces for high-level container runtimes (containerd, CRI-O) and low-level runtimes (runC here). The second, Container Runtime Interface (CRI) is a standard that allows to use many different container runtimes in Kubernetes. [57]

Application and system containers

Containers at kernel level are based on Linux namespaces, cgroups and other kernel capabilities. It is still possible to distinguish between two types of containers based on their usage [50]. *Application containers* are intended to run single application and its dependencies. Preference is for the application to be easily portable and scalable microservice [50]. *System containers* on the other hand are intended to run full operating system distributions along with their applications. System containers have resemblance to virtual machines, with negligible overhead compared to them, but without the flexibility of choosing the OS kernel. While the intents are different, both container types share many similarities, because they utilize the same underlying kernel technology. Docker [52] is a prime example of an application container engine [1], while OpenVZ [58] is an example of system container platform. LXC [51] is container runtime that supports both application and system containers [1].

Image layers

The complete image for application container is formed by a stack of dependent layers, which are union mounted to final file system view [50]. Figure 2.8 depicts the layer structure. The root layer contains the files that form a OS distribution [29]. Above the root layer there can be more layers with specific purposes [59]. For example, a mid-layer might contain the files necessary for a web server, while another layer might contain an environment for a database system. The application layer itself is positioned above the other layers. Container engine sets these layers as read-only. Any file system change by the container is made to a write-only layer that the engine places at the top of the layer stack. The benefit of layered image structure is that the layers can be shared between different application instances [29]. For example, if multiple containers use the same base image as their dependency, the base image needs to be downloaded only once from a container ecosystem repository. Similarly to application containers, system containers

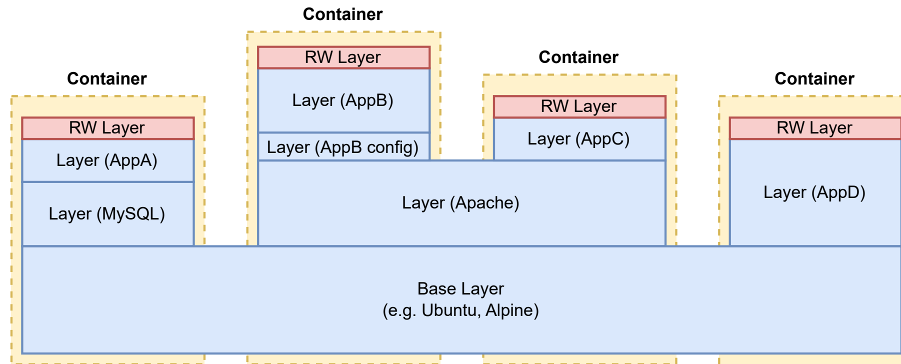

Figure 2.8: The complete container image is a stack of readonly image layers. The container engine mounts the layers as union file system, and adds writable top layer. The base image contains tools, libraries and data needed by a OS distribution. Mid-layers above base layer have additional software packages. The layered structure reduces overall needed network transmits from image repositories, and encourages reuse of layers.

can also use layered image structure. The root layer contains the base operating system, while mid-layers customize it with additional libraries, tools and data. The top layer is writable, like in application containers.

As container instance gets shutdown and deleted, all the data in writable top layer will be lost [50]. If container needs persistent data storage, the container engine may have support for a virtual disk that the application can mount and use. For example, Docker has *volumes* that can be used to write persistent data. Virtual disks may be mounted simultaneously by many containers, which allows containers to share data. Engines may also provide other ways to store data persistently. For example, Docker has efficient but non-portable bind-mount that allows to store data in host system.

Deployment model and orchestration

Both virtual machine images and containers can be used to package application software. However, compared to VMs, containers have much smaller image footprint. They have also other properties that make them suitable for modern collaborative development and IT operations models, i.e. DevOps [26]. These and the success of Docker ecosystem has

made containers popular method to package and deploy applications in cloud environments [29].

At the early stage of containerization evolution, containers were used to simply package and deploy applications to individual hosts. There was no unified solution to automate deployment and management of multi-tier container applications, i.e problem of container orchestration was still largely unsolved [8]. As the number of containers and their dependencies got larger, the container deployment and management became more difficult. Organizations at first used ad-hoc solutions or complex PaaS solutions to orchestrate containers across hosts. Nowadays, the orchestration problem has been mostly solved and Kubernetes [9] has taken the place as industry standard, and as the most popular orchestration platform [60]. While there are other container orchestration frameworks, such as Apache Mesos or Docker swarm, they will not be discussed in this thesis.

Performance analysis

When it comes to performance overhead, containers overall are comparable to native systems [7]. As containers were still rising in popularity, research community had much interest in studying container performance in various settings, especially compared to virtual machines and bare-metal machines [50]. Today, the study results converge to three understandings [1, 42, 50]:

1. Containers have negligible CPU and memory performance overhead. Containers are native processes and most of the overhead comes from kernel that realizes container abstraction [29]. The performance overhead of containers is much smaller than with virtual machines. This is because guest OS is never idle, and the hypervisor must multiplex hardware functions, which requires VM state management [43].

2. Containers have better local I/O performance compared to VMs, because containers have more direct access to system hardware. Containers I/O performance still depends heavily on used filesystem type and configuration.

3. Containers network I/O is worse than with virtual machines.

This noted, the cloud environment is often complex, where diverse set of co-located containers run in VMs on shared hardware resources. Performance characteristics show large variance for containers, as they can be influenced by many different interference effects from co-located applications competing for resources. While VMs have greater overhead compared to containers, VMs offer better noise isolation from neighbors so that their performance characteristics are more robust [7].

Benefits and challenges

Containers are now mature technology with proven benefits. Their small overall performance overhead has been discussed, but in addition their startup latency and power efficiency is better than virtual machines [7]. While a VM instance can take up to a minute to boot, container can boot in mere seconds. However, containers are not without issues. For one, migration of live containers to transfer the runtime state to other host is harder to implement compared to virtual machines [29]. Robust migration requires that large amount of kernel state is transfered along with memory pages. Container security is also a concern. They are properly isolated and secured by design [50], but the shared kernel has consequence that any compromise of the kernel will expose all running container instances. Solutions to improve isolation include hardening of existing security mechanisms, while propositions exist to use various hardware based mechanisms, such has Intel SGX enclaves. Another security issue is how to guarantee confidentiality of container image layer data [50]. Proposed solutions mainly have techniques to encrypt the data at rest and at runtime.

2.3 Kubernetes

Virtualization technologies that were discussed in section 2.2 operate at the granularity of a single server, i.e. a physical host or container-enabled virtual host [29]. Cloud data centers, however, employ many of these servers partitioned as large clusters. The scale of operations makes management of all the infrastructure hosts, virtual hosts, and containers a complex task. Consequently, the task has been for the most part automatized and assigned to different management frameworks that operate over clusters. Kubernetes [9] is the most popular management framework used to orchestrate containerized applications [60]. This section describes Kubernetes fundamentals, and how it is used to manage containers. While other management frameworks have a role in management of cloud infrastructures and virtual machines, they are out of scope for this thesis. Unless cited otherwise, official Kubernetes website [9] and its concepts documentation [61] is used as main source of information.

2.3.1 Introduction

Kubernetes is an open source platform used to orchestrate containerized applications and workloads across a distributed cluster of hosts [9]. While the origins are in Google's Borg [62], nowadays Kubernetes along with many other cloud-native projects are hosted and developed by Cloud Native Computing Foundation (CNCF) [63]. The platform and much of its ecosystem is implemented with *go* programming language.

The platform architecture follows master-slave model [64], as depicted in Figure 2.9. The cluster hosts can be either physical or virtual machines. The slave nodes are known as worker nodes. The master node has the overall responsibility to manage containers running in worker nodes across the cluster. It exposes Kubernetes REST API implemented in the *API server* component, acting as an entry point to control the entire Kubernetes cluster. The API is used by both internal cluster components and clients outside the cluster.

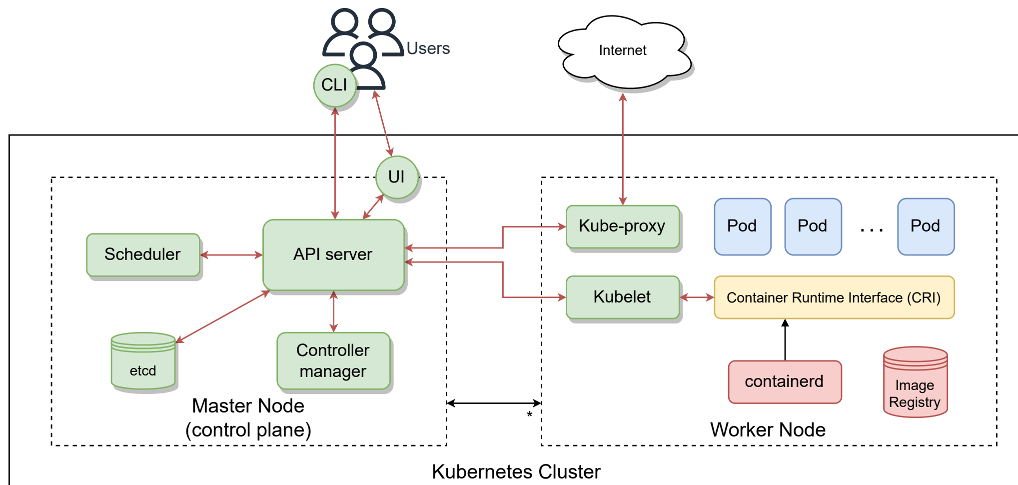

Figure 2.9: High-level view of Kubernetes architecture showing a cluster with master node and representative worker node.

The API is accessible either through built-in `kubectl` CLI tool or through one of the many Kubernetes API client libraries. There is also web-based UI *Dashboard* that can be used to monitor and manage containerized applications, as well as the cluster itself [61].

The distributed key-value pair data storage component *etcd* is responsible for persistently storing configurations and other state information of the cluster [64]. The storage content is shared with rest of the Kubernetes components. The other components use the shared data to synchronize themselves with the cluster state. *Controller Manager* is responsible for monitoring overall cluster state using the state information stored in etcd. When the system state changes, Controller Manager informs of new state to other components through API server.

Each worker node has an agent known as *Kubelet* that is responsible for monitoring deviations of the actual node state from the desired cluster state [64]. When the two states do not match, Kubelet attempts to launch or terminate containers as necessary to reach the desired state for the node. Kubelet is also responsible for reporting node events and resource usage to master node.

A smallest working unit of Kubernetes cluster is known as a *Pod* [64]. It represents a collection of often tightly coupled containers and their associated computing resources.

Containers in different Pods are isolated from each other. The master node schedules a Pod on a specific worker node, based on current cluster configuration and availability of free resources across the whole cluster. The assigned node first downloads the necessary container images from repositories, and then launches the container. The scheduler component in master node implements the actual functions that decide to which node a Pod should be deployed. The default scheduler of Kubernetes is known as *kube-scheduler*.

2.3.2 Components

As already discussed, Kubernetes master-slave architecture consists of multiple components. The master node has the API server, the scheduler, the etcd database, and controllers, while the Kubelet agent on each node manages the deployed Pods and their container instances that represent applications and services. While the master node in Figure 2.9 is presented as a single host, in reality the master node components can be located at different host machines. Furthermore, the master node components can have duplicate instances to implement high-availability clusters. The components of master node, its communication paths, and their endpoints are referred as the *control plane* [61]. The following two sections describe components shown in Figure 2.9 in more detail, starting from worker node [61, 64].

Worker node components

Pod. Smallest working unit of Kubernetes that encapsulates one or more containers with shared storage and shared network resources, along with configuration needed to run the containers. Containers inside a Pod share the runtime context, i.e. the Linux namespaces and cgroups. For example, each Pod gets assigned a single IP address, which along with port space is shared. Pod containers are intended to be tightly coupled, representing a single application or microservice. The most common use case for Kubernetes is to run a single container in Pod. Another, more complex

use case is to have two or more containers co-located in the Pod forming a cohesive unit of service. A common co-located model is to have one container serving the Internet, while one or more *sidecar* containers implement other supporting functions, such as data persistency or logging.

Kubelet. An agent of worker node that manages lifecycle of Pods and their containers. It reads Pod specifications scheduled to run at the node from API server, ensuring that Pod containers are running as desired and healthy. On agent startup, it registers itself to API server, and thereafter acts as a node endpoint for Kubernetes control plane. Kubelet also reports to the control plane the health of the host where it is running.

Kube-proxy. An agent in worker node that operates as a network proxy [65]. It maintains network rules, which enable communication between nodes and Pods. The agent can route traffic directly or it can use node's operating system packet filter functions.

Container Runtime Interface (CRI). A standardized API that allows Kubernetes to use different container runtimes at worker node. Kubelet uses the CRI for its container related management operations. In context of Kubernetes networking, container runtime gets the configuration for its enabled Container Network Interface (CNI) plugins through CRI. The plugins implement the Kubernetes networking model discussed in detail in 2.3.6. Some examples of CRI-compliant container runtimes are *containerd* and *CRI-O*. The first, containerd, originally comes from Docker team, but is now published as an open source container runtime component. It has become industry standard due to its widespread adoption [55]. It is made CRI-compliant with special cri plugin. CRI-O is Kubernetes-optimized and CRI-compliant runtime from Red Hat, IBM, etc. [66]. It has many of the same functions as containerd [65]. Both components use runC as their default low-level container runtime, while also supporting other implementations.

Master node components

API server. A component that implements REST API and acts as the frontend to Kubernetes control plane. It implements the actual API request functions, but also authenticates and authorizes each request using one or more of the configured methods. Kubernetes internal components, application services and external users can access the cluster shared state through the exposed API. The default API server implementation is *kube-apiserver*, which is also designed to be scalable by deploying more instances. Built-in `kubectl` tool can be used to access the API by normal users. Web-based UI addition known as *Dashboard* can be also deployed into system to monitor and manage the Kubernetes cluster. There are many client libraries for different programming languages to help implement custom tools and applications for API server communication. The client library used by Kubernetes internal components is also among these helper libraries.

Controller manager. A component that runs several distinct controller processes. Logically each Kubernetes controller is separate entity with specific function, but to reduce complexity they are compiled as a single controller binary, knowns as *kube-controller-manager*. The purpose of a Kubernetes controller is to implement *control loop* that observes the cluster state through API server, and makes state change requests in attempt to transit the current cluster state closer to the desired state. There are many controllers in Kubernetes, all of which can be read from Kubernetes documentation [61]. For some examples: *Node controller* is responsible for managing various worker node aspects, which include monitoring nodes' health, and keeping contoller internal node list up to date with the actual cluster state. *Job controller* watches for configured one-off batch jobs to create and run as Pods. *Replica controller* manages number of Pod replicas (duplicate instances) for specific configured Pod object. *Deployment controller* is responsible for managing set of Pods as an deployment that supports automatic Pod updates with no down time, and rollbacks when deployment

is unstable. The deployment uses similar replica set model as replica controller.

Scheduler. A scheduler purpose is to ensure that Pods are optimally matched to suitable worker nodes where Kubelet can run them [61]. The default Kubernetes scheduler, known as *kube-scheduler* (KS), uses two-stage selection process for every Pod that needs to be scheduled. The unscheduled Pods are added to a waiting queue, which is being constantly monitored by the KS. As the first step of the selection process, the KS takes a Pod from the queue and triggers node *filtering* process for it. The filtering searches for all the nodes that are *feasible*, i.e. capable of running the Pod. The search process applies a set of filter functions, known as *predicates*, each matching specific set of Pod attributes to corresponding node attributes. For example, the Pod could have minimum CPU and memory resource requirements, which would be matched to free resources of the node by the PodFitsResources predicate. If there are no feasible nodes for the Pod, the Pod is set in unscheduled state and KS triggers an failure event. The second step of selection is *scoring* process that is applied over the feasible nodes. Each feasible node is scored and ranked based on one or more weighted rules called *priorities*. Priorities base their scoring on conditions such as available resources on the node, number of Pods already running on the node, and the node overall status. KS selects the node with the highest score as the Pod deploy target. Then, in a process called binding, the KS informs the concerned controllers about the made decision through API server. The filtering predicates and scoring priorities used by the scheduler algorithm can also be seen as hard and soft constraints, respectively. They can be configured through API server with kube-scheduler *Profiles*.

etcd. A distributed, reliable and highly-available key-value data storage that holds persistent configuration, service discovery, and cluster state information on Kubernetes cluster [67]. API server is the only component that has direct access to etcd for security reasons. With help of watchers, etcd enables notifications about database changes for interested nodes through API server. The notification produces API re-

quest for a node to update its state information.

2.3.3 Configuration and management

Kubernetes has configuration *objects* that map to persistent entities of the system, which together represent the cluster state [61]. The objects describe entities like actual containerized applications, the available cluster resources, and the policies that guide how the applications should behave. The objects describe the intent, the what and how the system should look and behave. Kubernetes works constantly to achieve this *desired state*.

Configuration objects are defined as YAML or JSON files that are applied to Kubernetes database through Kubernetes API [61]. The same API is used to create, update and delete configuration objects. The API is used through `kubectl` or one of the available API client libraries. Configuration files that contain one or more object specifications are known as Kubernetes manifests, while the Kubernetes API URL endpoints corresponding to specific objects are known as API resources.

There are certain configuration fields that must be always set, while others are optional. All objects have field `kind` that defines the object's general type, such as Pod, Deployment, Service and many others. Field `apiVersion` defines Kubernetes API compatibility as a version string. Field `metadata` contains nested fields, such as mandatory `metadata.name` that provides object name, and optional `metadata.namespace` and `metadata.labels`. Together, metadata fields provide identifying information about the object. *Namespaces* is Kubernetes feature that allows for isolating cluster resources from each other, while *Labels* are key-value pairs that provide a method to tag objects with context specific information.

Almost all configuration objects have `spec` field that describes the desired entity state [61]. The `stat` field on the other hand describes the current state of the object. The Kubernetes system constantly compares these two states and strives to correct the situation by actions such as launching new object instances. For example, a Deployment

spec could indicate that there should be ten application instances. If there was less than ten instances, for example because one of them had a failure, Kubernetes would try to launch new instance in its place.

Custom resource definitions (CRDs) are objects that allow extending Kubernetes with third party object types. They are usually accompanied with custom controllers that are implemented and added into the system. The custom resources are manipulated using the same methods as built-in objects. Many common Kubernetes addons use CRDs as part of their implementation.

Specialized ConfigMap objects allow to decouple environment-specific configuration data from container images and application code [61]. Pods and containers can be made to have access to ConfigMap values via environment variables, CLI arguments, or as configurations files in Pod volumes. The configuration for ConfigMap object has fields `data` and `binaryData` instead of `spec`. The fields are used to specify key-value pairs as data entries. The ConfigMap data fields are configured to be accessible for containers in Pod manifests.

2.3.4 Cluster management

Kubernetes control plane operates over a cluster, which is simply a interconnected set of physical or virtual hosts. Each host machine is required to have base Linux operating system with kernel support for container runtimes (i.e. control groups, namespaces and capabilities). The recommended hardware requirements for cluster nodes vary by use case and system provider, but generally they are quite high. The minimum requirements for worker nodes can be estimated as 1 CPU core and 1 GB RAM, while the master node minimum requirements should be doubled from this. Actual recommendations for nodes are from two to four times these limits.

Setting up functional Kubernetes cluster is a complex task that requires many components to be configured correctly. It is easy to make an error that leads to non-functional

cluster. To this end, Kubernetes has designed `kubeadm` tool that can be used to bootstrap minimum viable cluster that conforms to best Kubernetes practices. Many alternative Kubernetes distributions and installers today use kubeadm to bootstrap their clusters. However, kubeadm does not solve all the long term issues of cluster management, which include: provisioning machines and network components across multiple providers, automation of cluster lifecycle management, and scaling these processes on any number of clusters. Cluster API is a project started by Kubernetes SIG Cluster Lifecycle that addresses these issues [68].

Cluster API project provides tools to simplify provisioning, upgrading, and operating multiple Kubernetes clusters [68]. It uses Kubernetes-like declarative API to configure cluster infrastructures and components, which include items like VMs, networks and load balancers. Cluster API uses various *providers* that implement support for any particular infrastructure providers (AWS, Azure, Google, etc.), cluster bootstrap tools, and control-plane implementations. Cluster API separates cluster management concerns to a separate *management cluster*, where the various providers for actual target *workload clusters* are running. The providers are implemented as ordinary Kubernetes controllers that consume custom CRDs, all running inside management cluster. By default, Cluster API uses kubeadm provider to bootstrap clusters.

2.3.5 Communication and security

Kubernetes uses centralized communication model, where all API usage from normal users, worker nodes, Pods, or control plane components, terminates at the API server [61]. The server listens for remote connections on secure HTTPS port. Each API request must be authenticated by some supported method. For this the client must provide valid credentials. Kubernetes has different modules for password, certificate and various token based authentication methods. If the request is authenticated, there are still authorization and admission stages which the request must pass, before the request action is allowed to

execute.

API server holds its own certificate with cluster root CA as the issuer. Nodes can securely connect API server using public cluster root certificate that has been provisioned for them. Nodes can then pass their own credentials, such as a client certificate, to API server. Pods can securely access API server using Kubernetes *service account*. When a Pod is configured to use service account, Kubernetes injects the public cluster root certificate and a bearer token for each Pod instance. The instance can then use the token to authenticate itself to API server.

After the API server request user has been authenticated, Kubernetes authorization modules try to authorize the user requested action on target object. Policy objects get consulted, and if an existing policy indicates that the user is allowed to perform the action, the request gets authorized. Multiple authorization modules can be enabled at the same time, and if any of them authorizes request, the request passes to the last security check stage, which is admission control.

As last stage, admission control modules can reject the request or they can modify the request's target object. They act only on requests that modify objects, i.e. create, modify or delete actions. Readonly requests are not acted on by admission control modules. All admission control modules must accept the request or it will be rejected.

2.3.6 Network and service model

Kubernetes has straightforward networking model that assigns unique cluster-wide IP address to each Pod [61]. One requirement of the model is that a Pod has direct access to all other Pods in any other node in the same cluster. Second requirement is that agents, such as system daemons and Kubelet, have access to all Pods running on the same node. Containers that run inside a Pod share the network context, i.e. they use the same kernel network namespace. Container intances running in Pod have the same IP address, but they can listen for different ports. Communication between containers of same Pod is

possible through `localhost` (as seen in Figure 2.10), shared Volumes, or any standard IPC methods. Kubernetes network model is similar to how workloads running in a VM might work, which helps porting them to container based workloads. One way to look at Pod is as if it represented processes running on a single host. Pod in many respects acts like a single server host.

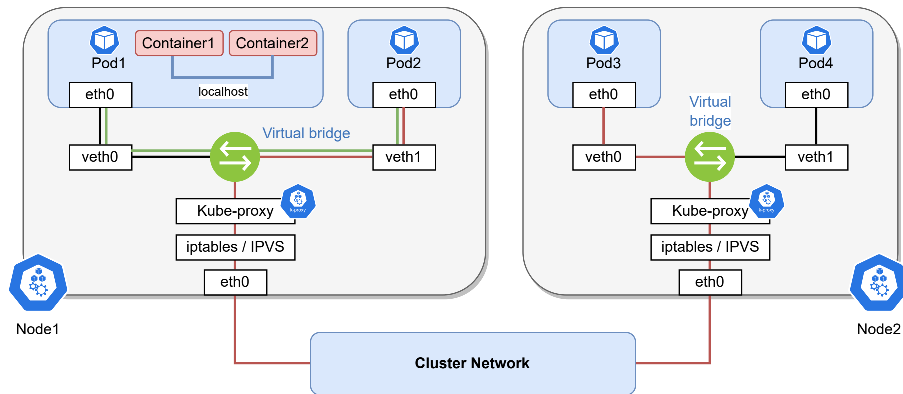

Figure 2.10: Communication paths between different Kubernetes components. Container-to-Container communication (blue) can use loopback interface since Pod containers share network namespace. Node-local Pod-to-Pod communication (green) happens through virtual bridge in the host network namespace. Traffic destined to a Pod located at other node (red), possibly discovered via Kubernetes Service mechanism or DNS, leaves the node according to kernel routing rules configured by Kube-proxy agent. Adapted from [69].

While the network model allows Pods to freely communicate with other Pods in the same cluster, the same is not true for clients outside the cluster. A Pod can be made reachable to cluster outsiders using Kubernetes Service API. Here, Kubernetes uses *Service* concept as a logical unit that maps to a service running as one or more Pods listening at specific ports [61]. The Service abstraction allows decoupling an application from transient network topology details of other services the application is using. Individual application container, which could have multiple running replicas, should not care where another service that it uses is located in the cluster, or how many replicas that service may have. Since containers can be started and stopped frequently, with no guarantee which node will be running them, Service is a very necessary abstraction of Kubernetes. In ad-

dition to service discovery function, Service provides automatic load balancing when it has more than one backing service Pod available.

Services are defined as Kubernetes configuration objects with their `kind` field set to `Service`. Service's `spec` has `selector` field that is used to filter backend Pods that implement the Service, and `ports` field that maps incoming ports to target ports listened by the container instances inside Pods. Service field `spec.type` defines the service type. By default it has value `ClusterIP`, which makes Kubernetes assign unique virtual IP address (clusterIP) for the Service that is accessible only inside the cluster. Figure 2.11 shows an example Service that uses `ClusterIP` type. Here it should be emphasized that Kubernetes Service is a logical concept. The actual implementation of its routing and load balancing mechanism does not map to single physical cluster element. Instead, all cluster nodes participate in the implementation. Specifically, the Service virtual IP is handled by Kube-proxy agent running on every node, as illustrated in Figure 2.10 and Figure 2.11. The agent watches API server for Service and EndPointSlice configuration object changes, and modifies host kernel forwarding rules accordingly. EndPointSlices are objects that hold state information of Pods that are selected as Service backend destinations. They are updated by slice controller that watches for changes in Service objects.

The virtual IP of Service can be exposed to clients outside the cluster by using NodePort or LoadBalancer types. Other expose methods are Ingress and Gateway API. `NodePort` is one of the possible Service `spec.type` values. It has each cluster node listen for a configured port, and forwarding traffic to the Service, as shown in Figure 2.11. If any cluster node is configured to be accessible outside the cluster, the Service is then also accessible. `LoadBalancer` is another `spec.type` value that allows using external cloud provider load balancer to access the service. Here, Kubernetes configures nodes to listen traffic to Service port, and forward it to one backend destination port, similar to `NodePort`. Cloud-provider controllers see the new Service object and configure their external load balancer to route traffic to the node port. *Ingress* manages cluster Service

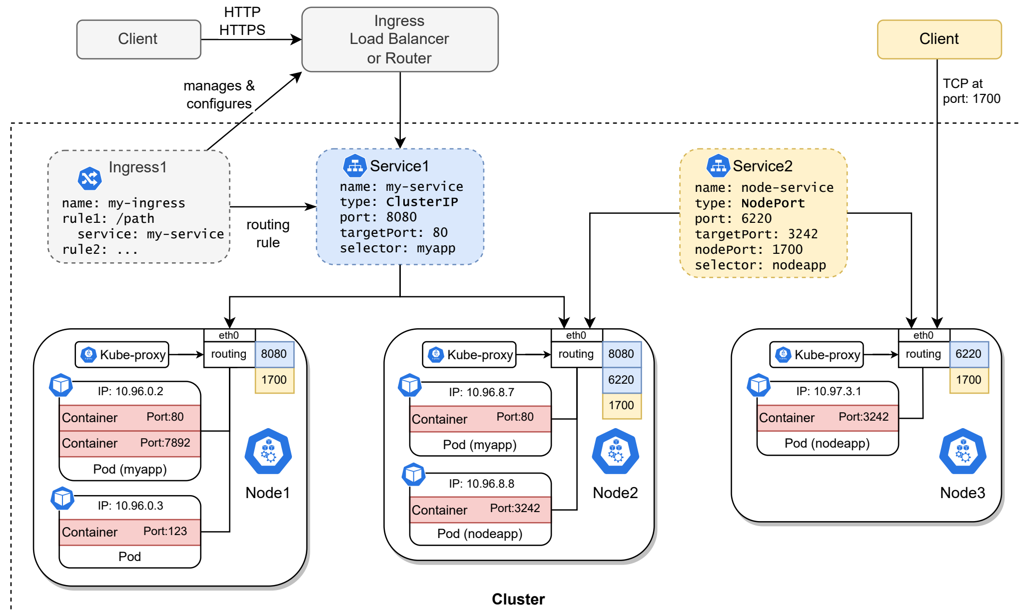

Figure 2.11: Kubernetes logical Service maps virtual service IP address and port to target container port. Service backend Pods are filtered by selector definition spec. Kubernetes automatically updates EndPointSlice objects, which are used by Kube-proxy agents to configure kernel level network routing rules in the node.

external access, mainly for HTTP(S) based applications, as illustrated in Figure 2.11. It is a configurable Kubernetes resource that aims to route and load balance requests that originate from outside the cluster to cluster-internal Service IPs. A Ingress controller is responsible for watching Ingress configuration object changes and implementing necessary changes in physical network elements, such as load balancer or edge router. Ingress resource is configured to have a list of match rules that it listens for, and forwarding rules that tell which Service the request should be sent. *Gateway API* is set of custom resource objects intended to extend Kubernetes networking with role-oriented service interfaces [70].

Containers have two primary methods how to find cluster Services [61]. The first method is built-in to Kubernetes and uses environment variables. As Kubelet starts a Pod, it will also add environment variables of all active cluster Services for the Pod consumption. An application simply uses the variables to connect to services that it needs. The

second method uses Kubernetes add-on, such as CoreDNS, to enable DNS service for the cluster. Unsurprisingly, the add-on by itself is a Kubernetes Service instance. When enabled, the DNS service constantly watches the API server for new Services, and automatically adds DNS records for them. Application containers can then access cluster services using standard domain name resolvers.

2.3.7 Helm package manager

Each application deployed on Kubernetes cluster has a set of YAML configurations which correspond to different target environments. For example, the configurations for a SaaS application are different between staging and production deployments. While custom deployment scripts may be a workable solution for a simple application, the custom solution does not scale. As the number of containers for application increases along with increase in complexity, few issues and limitations are known to emerge [71]:

1. Discrepancies between environments get more significant leading to complex scripts.
2. Templating setups using tools like sed or envsubs become insufficient to manage complex YAMLs.
3. Application version management for different environments becomes challenging.

The solution to these and other issues that relate to application management automation is to use the Kubernetes package manager *Helm* [72]. Helm is based on packaging format called *charts*. A chart is a package of various meta files which describe a set of related Kubernetes resources [72]. A chart contains at least a package description in YAML format, and one or more templates used to generate Kubernetes resource manifests. The templates use Go template language [73] with some add-on functions. Default values to the templates can be provided in YAML values meta file. A chart can also contain dependencies to other existing charts. A single chart can describe Kubernetes resources at any complexity level. For example, a chart can be used to deploy a simple Pod, but as well

it can deploy complex web application that builds from many different resources such as backends, load balancers, or databases.

Helm has a client CLI tool `helm` with various subcommands that can be used to manage chart packages [72]. The tool can reference charts at local host, but the preference is to load them over HTTP from *chart repositories*. The latest Helm versions can also load charts from OCI compliant container registries. The tool has subcommands to handle both chart repositories and OCI registries. When the tool is used to install a chart with install subcommand, a new *release* gets instantiated. The install process first generates the Kubernetes resource manifests from chart templates based on effective configuration values, and then applies the manifests on Kubernetes cluster. A single chart can be installed many times, each release having unique name and its own configuration values. An upgrade subcommand gracefully updates a release to new version by doing the least amount of changes in Kubernetes. The subcommand does this by analysing the differences between the old and new generated release resource manifests. Finally, an upgrade can always be rolled back to old version, while any release can be always uninstalled. Uninstall subcommand removes all the associated chart resources from Kubernetes, which prompts Kubernetes to eventually remove the corresponding containers from cluster.

While the use of Helm is not mandatory to use Kubernetes, it can greatly help in managing complex clusters. It is especially helpful when applications have many dependencies, which should all be deployed to cluster or removed from cluster at the same time as a single unit along with the application. Helm is semantically similar to package managers used in other contexts, such as Debian's `apt` or Node.js `npm`. One clear difference is that Helm commands modify only the desired cluster state, while the addition and removal of actual containers and other resources is executed later by the Kubernetes system.

3 Distributed cable access

This chapter explores the second background for the case study in Chapter 4. The technological context of the study relates to cable access business, which is currently transitioning to become more distributed and virtualized. To this end, and for completeness, the chapter presents broad overview of broadband data access networks evolution up to present day[1]. To better understand the reasons behind historical development of cable access networks, it is necessary to consider all the major technology types which have emerged over the years to serve a specific need, such as voice transmissions, TV broadcasting and wireless voice transmissions. While initially each provider has had unique technology requirements, over the years there has been convergence of service offerings and technologies. This includes broadband data services earlier, and IP convergence recently. Focus here will be on cable technology, but telephony and cellular are considered too. Most of the historical backround is from Gorshe et al. [74], Tornatore et al. [75] and Jia et al. [16, Ch. 1]. Cable data access technology details are based on specifications from CableLabs [76–79] and Gorshe et al. [74, Ch. 11].

3.1 Access networks

Network operators work with distributed systems that cover vastly different geographical areas. On the one hand are long-haul and metro operator networks that cover the full span

[1]Some of the material may be more detailed than the thesis scope would mandate, such as DOCSIS specification details. These sections may be freely skipped when reading.

between the cloud and the edge networks. At the edge, on the other hand, are located smaller-scale operator networks which connect subscribers to the operator core network through a distribution network. Such edge networks are called *access networks* since they provide connectivity for a large number of subscribers. Figure 3.1 illustrates an access network that is made up of a signal aggregation station and the distribution network between the station and subscribers. Cable operators refer to the station as a hub or headend, while telephone and cellular operators use the term central office. Distribution network is further divided to a feeder segment and edge-distribution segments. A feeder segment is used to distribute signals over long distances to a remote node. For the last kilometers or so, the node transmits the signal to subscribers over edge distribution segments which can have drop segments. While the feeder segment is almost always fiber-based, the edge-distribution employs many different physical layer (PHY) technologies. The industry terminology used for access distribution networks varies a lot.

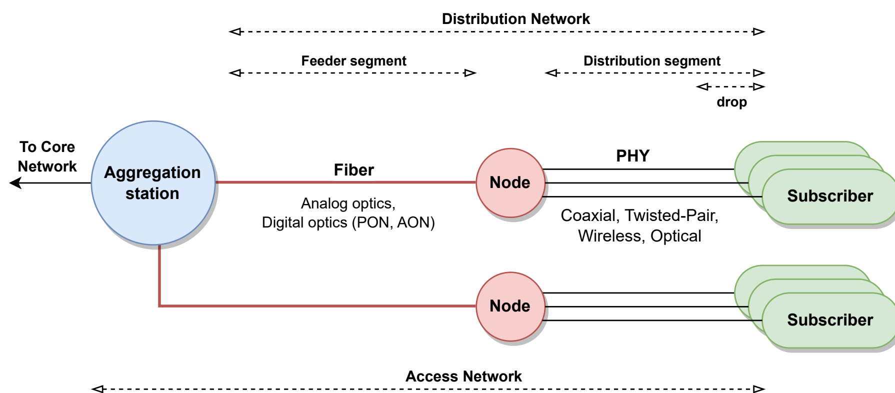

Figure 3.1: Simplified view of an access network that connects subscribers to service provider core network through distribution network. The distribution network employs fiber in its feeder segment up to the node closer to the subscribers. The edge connectivity physical interface (PHY) between the node and subscribers varies by provider.

The variable-scale distributed network systems employ vast number of technologies to transmit video, voice and data between various end-points of the system. Analog payload signals have long been replaced by digital signals because of their many benefits. Optical

fibers dominate in the cloud and core network domains, but in access networks one can still find hybrids of fibre, coaxial, twisted pair and radio based transmission mediums. While coaxial medium is still widely deployed due to cable TV industry legacy, high capacity optical fibers have reached deep into the domain of access networks, often as far as subscriber premises. Traditionally plain telephone, cable and cellular access networks have been designed for the delivery of specific services. Consequently, operators have had distinct technical requirements for their access networks. However, broadband access, digitalization, universal IP delivery and the attractive attributes of optical fiber has led to convergence of access network technologies, resulting in network operator solutions resembling each other at the distribution network. Single technology operators have in many cases become Multiple System Operators (MSOs) who serve customers with more than one technology.

3.2 Broadband data access networks evolution

From their inception, cable television (CATV[2]) access networks were based on unidirectional downstream analog broadcast video transmissions over coaxial medium. In parallel, telephone operators had their own bidirectional point-to-point wireline networks which initially carried only baseband voice signals. Broadband data access on telephone networks began with the introduction of modems that could receive and transmit modulated data signal within voiceband. While in the mid-1990s there was still uncertainty about the future prospects of broadband data access, nowadays it has of course become the norm.

Over the years cable and telephone operators have developed wide range of technologies as a response to progressively increased broadband demand. Traditionally the cable and telephone providers have directly competed for market share in broadband data access

[2]Originally known as Community Access Television or Community Antenna Television.

domain, which has pushed the development forward and provided motivation for infrastructure upgrades. For example, CATV operators were initially forced to upgrade coaxial network repeater equipment to support upstream data transmissions. Telephone operators on the other hand had greater pressure to replace low bandwidth twisted pair distribution lines with optical fibers, which in part has forced a response from CATV operators. Later appearance of cellular based radio access networks (RANs) has given rise to mobile broadband access which has further stirred the situation. The increasing demand for wireless mobile access has rapidly pushed development of cellular technology. For example, in later RAN generations the mobile feeder or backhaul distribution segment increasingly utilizes fast Ethernet/IP-based optical fibers to serve the capacity requirement of more numerous but smaller cells. This in turn has also affected the more general evolution of fiber optics use in wired access networks.

In CATV networks a hub receives video, voice and data content from various sources, such as satellites, terrestial antennaes, the Internet or other hubs. The received content is then filtered, processed and augmented with regional content. Digital data is modulated using techniques such as quadrature amplitude modulation (QAM). The downstream access is provided by combining the final video, voice and data signal elements. The combining is done using frequency division multiplexing (FDM) where signal elements are divided to non-overlapping frequency channels. Multiplex signal consisting of video, voice and broadband data, each modulated at different channel is known as radio frequency (RF) signal.

The data channels of CATV RF signal are received by a cable modem (CM) located at subscriber premises. To enable bidirectional access, part of the shared RF bandwidth is dedicated for the upstream data. Originally the bandwidth reserved for upstream was much smaller compared to downstream, but later cable access specifications have permitted more symmetrical splits. In order to better utilize the upstream spectrum, CMs at first used time division and frequency division multiple access (TDMA/FDMA) meth-

ods to share the medium. Later specifications allow code division multiple access (CDMA/FDMA) method. Most recent specifications include orthogonal frequency division multiple access (OFDMA) to increase the upstream capacity in situations where channel noise characteristics permit it. The current data over cable service interface specification (DOCSIS) is explained in more detail in section 3.4.

In the early days of CATV the downstream RF signals were transmitted from the hub to the CMs over pure coaxial network with no optical fiber feeder segments. Amplifier nodes were used to cover longer distances. However, as the optical fiber signal attenuation issues over long distances were eventually solved, it became cost-efficient to replace all or part of the coaxial network with fiber. Cable networks that started to use analog fiber in their initial 5 to 40 km feeder segment, and coaxial cable in the last few kilometers before reaching subscribers became known as hybrid fiber-coaxial (HFC) networks [80]. In HFC topology one or more fibers extend outwards from the hub, each ending at optical fiber node (FN) located closer to subscribers. In the node the received optical RF signal gets converted to electrical format and is then distributed to subscribers over coaxial medium. The shared coaxial segment is known as a service group, and can be shared by up to 500 subscriber modems.

3.3 Fiber to the X and digital optics

Along with cable operators, other network operators have seen interest in transforming data access networks to fiber-based to meet increasing demand for broadband data capacity. Initially telephone operators utilized modems for data transmissions, first at voiceband and later using various out-of-band (OOB) technologies known as digital subscriber lines (xDSL). Later it became apparent that in order to compete in data services against CATV operators and their high bandwidth coaxial networks, at least partial fiber distribution lines would be needed. Because twisted pair copper has limited bandwidth compared to

coaxial cable, it has further increased the incentive to offer fiber as complete end-to-end solution where fiber extends all the way from the central office to the subscribers. These passive or active optical networks (PON, AON) are known as fiber to the home (FTTH), and they compete directly with the HFC networks. The latest trends point FTTH becoming the most common fixed broadband access method according to OECD statistics [81]. However, deploying fiber is still considered expensive [82], and often fiber is deployed only up to a remote optical network terminal or unit (ONT/ONU) located close to subscribers. Final connectivity to subscribers is then served using other techniques, such as DSL or mobile-cellular. The partial fiber deployments are known as fiber to the node (FTTN). Depending on use case, variations such as fiber to the curb, cabinet, building or premise do exist (FTTC/FTTCab/FTTB/FTTP).

In typical PON system (Figure 3.2) downstream and upstream signals are transmitted over the same fiber segment at two or more distinct wavelengths. The shared fiber segment is power split multiple times so that each fiber ends at a node close to subscriber premises. The fiber ends are called either optical network units (ONU) or optical network

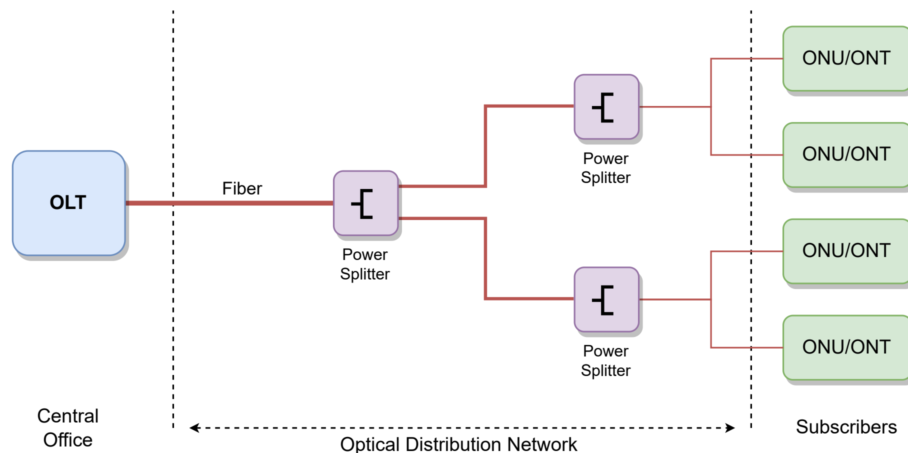

Figure 3.2: Elements of Passive Optical Network.

terminals (ONT), depending on which PON standard is used. The role of ONU or ONT is to do optical-to-electrical conversion for subscriber premise network. Typical number of connected ONUs is 32 or 64, but higher node counts are not uncommon. Optical

line terminal (OLT) located at central office broadcasts downstream traffic to all ONUs. Individual ONU extracts the data intended for it based on time slots, encoded packet address or wavelength. Upstream medium is shared between ONUs, therefore downstream data also includes MAC protocol information on how and when the ONU should transmit upstream. Time division multiple access (TDMA) is the most common protocol used to share the upstream. Earlier PONs used fixed time slots, but newer systems use dynamic allocation model where ONT notifies its bandwidth requirements to OLT. PON systems use wavelength division multiplexing (WDM) to better utilize fiber capacity. Typical method is to use separate wavelengths for downstream and upstream direction, and possibly use a third wavelength for video services[3]. Very high speed PON systems support dense wavelength division multiplexing (DWDM) or use coherent optics. However, full WDM PON is still not as cost-effective compared to TDM. Recent PON systems are based on IP and carry Ethernet frames, contributing to harmonization of access network packetization.

3.4 Data Over Cable Service Interface Specification

HFC access networks today still have partial coaxial medium backing in the last kilometers before they reach customer premises. This is due to the still untapped bandwidth potential in coaxial cables, and the tendency of CATV operators to choose cost effective solutions. In the 1990s there were several competing regional standards aiming to address bidirectional transmissions over HFC networks. Data Over Cable Service Interface Specification (DOCSIS) eventually became the standard which in its original form specifies how IP-based broadband data is transmitted as RF signal over coaxial medium.

[3]Downstream broadband data is transmitted over 1490 nm wavelength, upstream over 1310 nm and video over 1550 nm.

3.4.1 Introduction

The first DOCSIS version 1.0 was released in 1997 and included only a minimal set of features to support broadband data access. Currently cable operators are transitioning their networks to DOCSIS version 3.1 released in 2013 [76, 77]. The latest version 4.0 [78, 79] supports features which enable higher capacities and more symmetric downstream and upstream splits. Important feature of DOCSIS is that each new release is backwards compatible with earlier releases. The specification has regional variability from its historical origin on competing regional standards. For example, the DOCSIS specification for North America has some important differences to European system (EuroDOCSIS).

DOCSIS specifies the required elements, protocols and parameters of functional cable system at different layers of the OSI network reference model. All layers have regional variability and options. Of special interest here are the data link PHY and MAC layers. At the PHY layer, DOCSIS specifies how data is transmitted in RF channel slots over the shared medium, including modulation and line coding formats. The MAC specifies how the medium is shared between subscriber modems in downstream and upstream direction. Seen from service level, DOCSIS specifies how data streams of different services can be classified and divided into QoS service flows to guarantee service level agreement (SLA) measures. Since the transmission medium is shared, DOCSIS also specifies various security features.

3.4.2 DOCSIS network elements

The core elements of DOCSIS network are shown in Figure 3.3. The cable modem termination system (CMTS) is located at the hub and terminates the DOCSIS protocol at operator side. The cable modem (CM) terminates the protocol at the subscriber side. Between the CMTS and CM is the HFC network which includes all the fiber lines, fiber nodes (FNs) and RF amplifiers used to transmit RF signals. A fiber node is relatively simple device whose function is to do optical-to-electrical (O/E) and electrical-to-optical

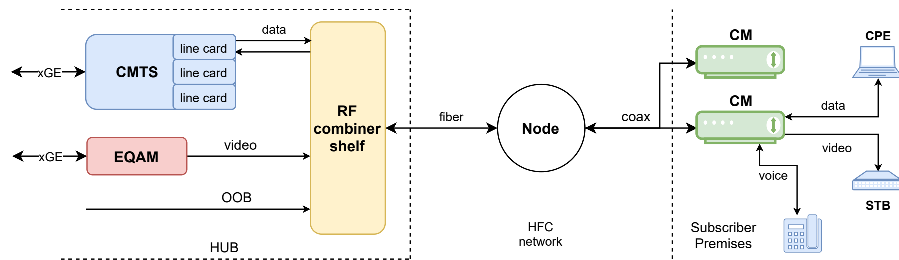

Figure 3.3: Elements of DOCSIS network.

(E/O) conversions. A RF combiner network equipment located at the hub mixes, filters and converts various signals between the hub and HFC network.

A line card in the CMTS initiates DOCSIS downstream protocol RF signal and transmits it over a coax line to the combiner network. The combiner mixes the data signals from all the line cards with video and OOB signals, E/O converts the multiplex and feeds the result to HFC network. A CM extracts packets destined for it from the downstream RF signal and converts them to suitable customer side network interface format, which for data traffic is usually Ethernet based. The customer premise equipment (CPEs) behind the CM send and receive the service data.

The upstream RF signals received from HFC network are O/E converted by the combiner. The combiner filters DOCSIS specific upstream RF components originating from cable modems and transmits them to CMTS line cards, which will demodulate and further process the data signals.

3.4.3 Physical layer

Signaling between cable system elements occurs at the physical layer (PHY). At PHY layer EuroDOCSIS 3.0 specifies spectrum band of roughly 5-85 MHz for upstream (US) channels (Figure 3.4). Spectrum from 108 to 1002 MHz is available for downstream (DS) channels. DOCSIS 3.1 extends the DS band upper edge up to 1218 MHz while future plans are up to 1794 MHz. DOCSIS 3.1 also introduces more flexibility in upstream and DS split plans, so that corresponding upper and lower edges can extend as high as 204

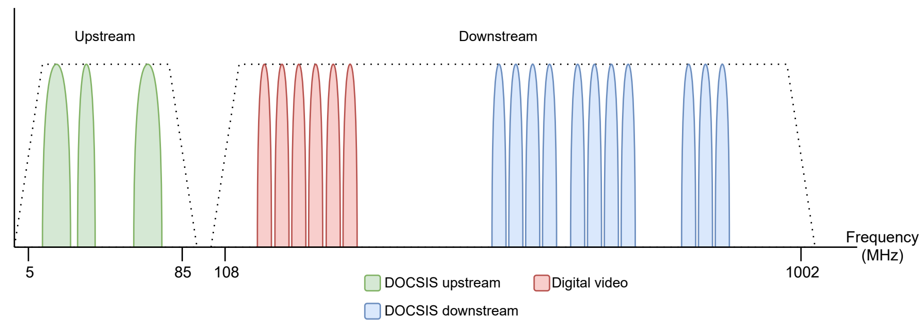

Figure 3.4: EuroDOCSIS 3.0 spectrum use. The DOCSIS 3.1 extends the maximum US and DS frequencies up to 204 MHz and 1218 MHz, respectively.

MHz ("high-split") and 258 MHz.

The only supported DS modulation orders in DOCSIS 3.0 and before are 64-QAM and 256-QAM. EuroDOCSIS uses QAM channel width of 8 MHz while the North American DOCSIS uses 6 MHz. Differences inherit from regional video transmission standards used on cable systems before DOCSIS, such as European PAL system. Consequently, the DS channel band is still shared with digital video signals and other OOB signals.

DOCSIS 3.1 introduces more spectrum efficient orthogonal frequency division multiplexing (OFDM) modulation scheme. Single OFDM downstream channel can occupy frequency band up to 192 MHz divided to large number of smaller subcarriers. Each subcarrier can have different QAM modulation order from 16-QAM to 4096-QAM. Modulation orders 8192-QAM and 16384-QAM are optional. An enabling feature for use of such high-order modulations is the low-density parity-check (LDPC) forward error correction (FEC) encoding introduced in DOCSIS 3.1. The earlier channel schemes are referred as single carrier QAM (SC-QAM) systems in contrast to OFDM.

In upstream the medium must be shared between all the transmitting CMs. To this end DOCSIS upstream access methods are based on separate frequency bands (FDMA) and time slots (TDMA). Synchronous code division multiple access (S-CDMA) method has been also available since DOCSIS 2.0, allowing CMs to transmit simultanously over same time slots with more tolerance to noise. Upstream transmissions have burst nature

since the CMs transmit at dynamic time slots assigned by the CMTS MAC scheduler. The CMTS configures CM upstream channel base burst parameters, which include modulation order. Many of the parameters can be set on each individual burst.

DOCSIS 3.0 supports US modulation orders from QPSK to 64-QAM for TDMA and S-CDMA channels. For S-CDMA channels QPSK to 128-QAM with trellis coding is supported. Supported US channel symbol modulation rates are 1280, 2560 and 5120 kHz.

DOCSIS 3.1 introduces OFDMA upstream channels for more efficient use of spectrum. A single OFDMA channel can occupy up to 95 MHz band. Similar to downstream OFDM, each subcarrier can have different modulation order ranging from QPSK to 4096-QAM. Earlier SC-QAM channel schemes are also supported, and they can be mixed together width OFDMA channels.

3.4.4 MAC layer

A cable system end points transmit service data over shared medium. To allow collision-free and efficient data transmissions, the system needs controller, management and scheduler functions together with associated protocols in the MAC layer. DOCSIS medium access control (MAC) layer specification [77] describes these functions and protocols.

DOCSIS MAC is a complex system and has requirements for upper layers and PHY layer. This section provides only summary introduction of the MAC. Here the CMTS is assumed to contain all the MAC functions. However, the DOCSIS evolution is on a path to have some or all the MAC functions to be relocated to the node for more flexibility. This evolution process is partly described in section on distributed architectures 3.6 and in more detail in specification for flexible MAC architecture (FMA) [83].

Figure 3.5 depicts a high level view of MAC elements of CMTS. A MAC domain is CMTS managed logical unit that provides Layer 2 data forwarding for set of CMs registered to the domain. There can be multiple MAC domains in the CMTS, but a CM

is ever registered to a single domain. Each MAC domain has one or more associated downstream channels and one or more logical upstream channels to enable bidirectional communication. A logical upstream channel maps to a physical channel as defined in PHY layer. If logical channels overlap, CMTS multiplexes the physical channel access in time domain. For management and control purposes MAC domain sends and receives special MAC messages to and from registered modems.

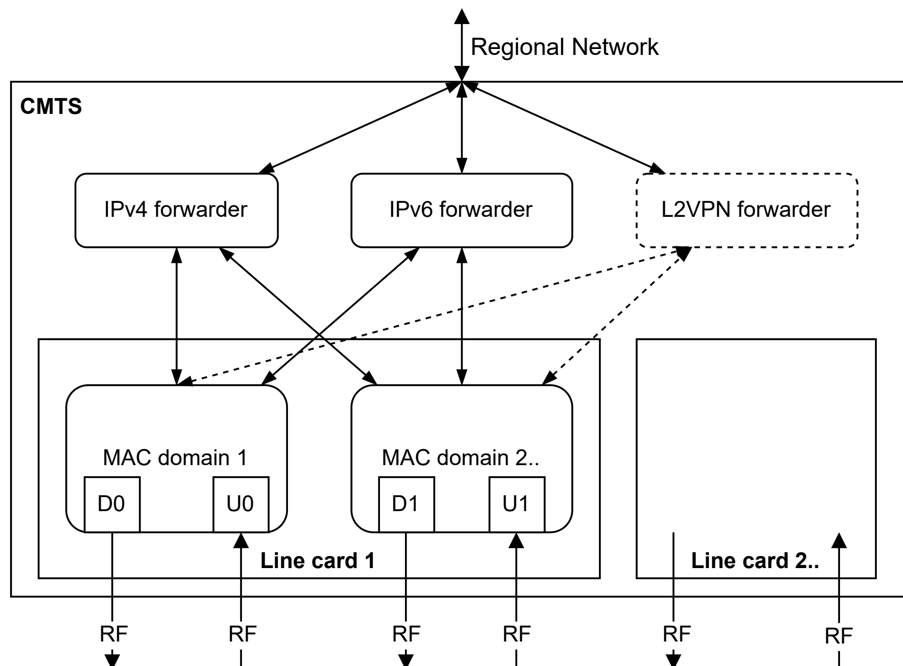

Figure 3.5: MAC layer functions in CMTS. MAC domain provides layer 2 data forwarding for cable modems using set of downstream and upstream channels. The forwarder components provide layer 2 bridging and upper layer 3 routing functions.

The MAC domain receives service level data packets from forwarder components of CMTS. The packets are classified to different service flows based on information in layer 2,3 and 4 packet headers. The service flows in turn are mapped to a set of downstream channels, and the packets are scheduled for forwarding to the channels as determined by a MAC domain downstream scheduler entity in CMTS. The scheduler provides each service flow a Quality of Service (QoS) as determined by the flow classification QoS parameters, which include traffic priority, data rate and latency.

In upstream direction, MAC domain recognizes the source CM of a received packet based on MAC address, and notifies the forwarders of received data. The forwarders bridge or route the packet to the regional network, or send it back to a MAC domain. Similar to CMTS downstream classification, a CM also classifies upstream packets to different service flows and schedules the transmissions according to flow QoS parameters.

Communication between CMTS and CMs involves MAC protocol frames. The generic frame structure includes a header and an optional variable size payload PDU. The PDU includes source and destination MAC addresses, the user data and a CRC. Downstream SC-QAM frames are encapsulated as MPEG-TS packets while OFDM frames are transmitted without any encapsulation. In upstream direction MAC packets have overhead which size depends on used PHY transmission mode. DOCSIS uses special MAC-specific frames for system management and control purposes. Various MAC frames exist for purposes such as initial CM registration, bandwidth management, state management and dynamic modification of protocol parameters.

DOCSIS 3.0 MAC introduced channel bonding as an important feature that increases peak data rates in both downstream and upstream direction. In downstream channel bonding, a CM can receive sequenced frames intended for it from more than one channel simultaneously. The CM resequences the data with help of sequence numbers in MAC frame header before forwarding the packets to subscriber network. Similarly, in upstream channel bonding a CM transmits data to CMTS using multiple channels at the same time. Channel bonding can increase peak data rates dramatically to and from CM, because in a sense multiple smaller bandwidth channels are bonded together as conceptually single superchannel.

Upstream MAC has additional complexities due to shared nature of the medium. To this end, special MAP frames contain information on allocated time-slots when a CM is expected to transmit its queued upstream data. A CM notifies CMTS of its transmission need by sending a MAC request message to CMTS. The CMTS collects all the

requests from different CMs and sends them to scheduler component for processing. The scheduler uses multiple parameters and complex algorithm to allocate time-slots so that available upstream bandwidth is optimally utilized. The CMTS transmits the time-slot allocations to the CMs in MAP messages as grants. For the request-grant scheduling to work efficiently, the scheduler and CM notion of time must be accurately synchronized. For this purpose CMTS periodically sends SYNC messages which contain global timing reference information. CMs determine the exact time to send upstream frames based on the reference, MAPs and timing offsets received in the ranging process. This kind of centralized scheduling intrinsically brings latency to the DOCSIS system. To combat the latency, future trend seems to be to move the scheduler element closer to the subscribers as described in section 3.6.

3.4.5 Cable modem provisioning

The DOCSIS protocol as related to a cable modem initialization has four stages: 1) channel topology resolution, 2) authentication, 3) IP provisioning, and 4) registration.

First, on power-up a CM scans the downstream spectrum and locks on one of the channels flagged as primary. A primary channel carries periodically sent special MAC frames. These include frames that provide time synchronization information (SYNC), scheduled upstream transmit time-slot allocations (MAPs) and descriptors of available upstream channels (UCDs). In addition, DOCSIS 3.0 CMTS transmits periodic MAC domain descriptor (MDD) frame on all downstream channels. MDD contains reference to a primary channel and other meta information on available spectrum channels. After CM has received enough information on PHY layer, it attempts to range on one selected upstream channel. The ranging process involves number of MAC frames being sent and received between CMTS and CM. The aim is to find reliable operating parameters for the upstream channel. The parameters relate to CM transmission power, timing, and frequency offsets. The ranging process is repeated periodically for each CM channel in at

least every 20 seconds to tune the transmission parameters against changing conditions.

Next, after successful ranging DOCSIS 3.0 CMs may try to initialize authentication and encryption functions if they have been enabled. If not, the CM may attempt the same initialization after the CM has been registered.

In third stage CM will first request an IP address from operator network. The address can be either IP version 4 or 6. Depending on provisioning mode, one or both of the address versions may be requested. After the address is known, the CM requests time-of-day information and downloads binary configuration file from the operator network. The configuration file contains important operating parameters and limits for the CM, such as information on access control and parameters of available service flows. The returned CM configuration may be generated dynamically, depending on parameters of the subscriber agreement associated with the CM MAC address.

In fourth stage CM initiates three-way handshake with the CMTS, in which some of the content of the configuration file gets sent to the CMTS. The CMTS validates the content, allocates necessary MAC layer resources for the modem, and notifies modem of registration result. The CM will acknowledge the registration and at that point CM initialization is complete. The cable modem will continue to operate in the network as per the initial operating parameters permit, and bridges ethernet data packets to and from the subscriber network. Changing operating conditions and protocol messages from CMTS may cause the modem behavior to change the parameters, but otherwise the modem tries to keep the connection stable and low latency.

After registration, the CM functions as ordinary IP addressable network element in the operator network. In network layer CM and CMTS support various services, such as SNMP for management or DHCP relaying for CPEs.

3.5 DOCSIS extensions

Over the years, cable industry has been forced to respond to competition from other access network providers. As a result, DOCSIS standard has received many improvements and extensions. Channel bonding was first introduced to enable multiple channels to transmit data simultaneously to and from a single CM. Constellation orders for QAM have been progressively increased. Current specification enables higher spectral efficiencies thanks to OFDM. But the demand is ever for more capacity. To this end the next DOCSIS 4.0 has two more important PHY and MAC layer additions, whose focus is on increasing the downstream and upstream to symmetric 10 Gbps speeds.

3.5.1 Frequency Division Duplex and Extended Spectrum

Frequency Division Duplex (FDD) refers to concept of splitting the spectrum to dedicated frequency bands for simultaneous downstream and upstream transmissions. DOCSIS has always used FDD splits as discussed in section 3.4.3. Split ratios of DOCSIS 3.1 are asymmetric, around 1:10 to the benefit of downstream [84]. DOCSIS 4.0 FDD, or extended spectrum DOCSIS (ESD) as it is commonly termed, significantly increases the maximum upstream and downstream spectrum limits. The upstream upper limit is extended up to 684 MHz and downstream upper limit is extended to 1794 MHz [78]. Research and development efforts are ongoing to eventually reach 3 GHz spectrum limit at the downstream [85].

To achieve the greatest benefits from such high splits requires that HFC network devices such as nodes, amplifiers and passives must be upgraded to support higher bandwidths. This can be expensive for operators, but in many cases it might be less so than moving to full FTTH solutions, such as PON technology.

3.5.2 Full Duplex

DOCSIS Full Duplex (FDX) is an extension that aims to alleviate the asymmetry of US and DS spectrum regions to achieve higher upstream capacity. FDX was first introduced as an addition of DOCSIS 3.1, but is now included as part of DOCSIS 4.0 [78]. FDX extends the available upstream spectrum by allowing frequency overlap with downstream region, so that in the overlap region downstream and upstream signals can be active at the same time. DOCSIS FDX specification defines the spectrum from 108 to 684 MHz as the full duplex band (FDB), where DS and US channels can overlap. The spectrum below FDB is reserved for non-FDX upstream channels, and the spectrum above FDB for non-FDX downstream channels.

Full duplex presents multiple technical challenges, one of which is the interference at a network device receiver port. The interference originates from various sources [86]:

- leakage and reflections from simultaneous transmit in the same channel as the receive channel
- leakage and reflections from out-of-band components of simultaneous transmit in the adjacent channel
- simultaneous upstream transmits from other devices in the cable network.

The first two interference sources are commonly called echo, since they originate from the transmit signal of the device itself. The echo in turn has two components: 1) self-interference, which is the internal signal leakage power from transmitter to the receiver at the same port, and 2) microreflections, which are caused by impedance mismatches at network crossover points, such as device port, taps or coaxial cable itself [87].

Interference problems can be mitigated with echo cancellation techniques. The full details of such technical solutions are out of scope for this thesis, but examples and descriptions can be read in papers such as [86, 88, 89]. At the CMTS end solutions often

involve modeling and estimating the transmitted signal echo components, and then canceling the interference effects to reach acceptable SNR levels. The echo canceling at the node is in fact the enabling technology for FDX DOCSIS [87].

In theory echo cancellation could also be implemented at CM side, but this would lead to high CM complexity and cost. For this reason, CMs in DOCSIS FDX operate in half-duplex FDD mode, with separate transmit and receive channels. The interference from other modems is solved at the CMTS MAC layer. This involves first grouping the CMs in interference groups, so that upstream transmits of CMs in the same group disturb each other, but so that they do not disturb CMs in the other groups [84]. The CMs in the same group are provided the same FDX upstream channels, and the upstream scheduler manages the transmit opportunities so that there is no interference. The CMs present in other interference group use different set of US channels, which can overlap DS channels of other groups, resulting in aggregate FDX at the CMTS level.

Another relevant technical challenge is that FDX requires more power from the node, as the complex circuitry increases the consumption. However, such increase is not exclusive to FDX, because any addition of new functionality for increased data rates increases power consumption. Around fifty percent of consumption can be traced to power amplifiers used to boost transmitted signal [87].

Amplifiers bring another challenge to FDX scheme in that they would also need to have echo canceling support [86]. Replacing the amplifiers would be costly. For this reason DOCSIS FDX initially assumes so-called Node+0 setup, in which there are zero amplifiers in the coaxial part of the network. However, for many operators such a large network change may be too expensive. For this reason, some research and developments efforts towards a FDX amplifier with echo cancellation support are continuing, to enable FDX in N+1 or N+2 architectures [90]. DOCSIS FDX also assumes new distributed HFC architecture, in which the physical RF generation occurs at the node. Distributed architecture is described in section 3.6.

3.6 Distributed cable access architectures

Baseband digital optics can be used in feeder segments of HFC networks to provide bandwidths of 10 Gbps or more over longer distances [80]. However, until recently it has been more cost-efficient to transmit RF signal over fiber using analog intensity modulation. At the same time, analog optics is showing limitations in achieving carrier-to-noise requirements of higher DOCSIS modulations [16]. This and increasing demand for more bandwidth, along with competitive pressure from other network providers, is forcing cable operators to find solutions on how to increase capacity. One technique that operators use is node splitting in which more nodes and fiber are deployed to serve more but smaller size service groups. Traditionally, the most complex cable system equipment has been located at the hub (Figure 3.6), while the distribution network has had relatively simple devices, such as analog nodes or amplifiers. However, as the number of fibers extending from the hub to nodes increases, so does the number of needed network equipment at the hub. This increases spacing, power and cooling requirements at the hub.

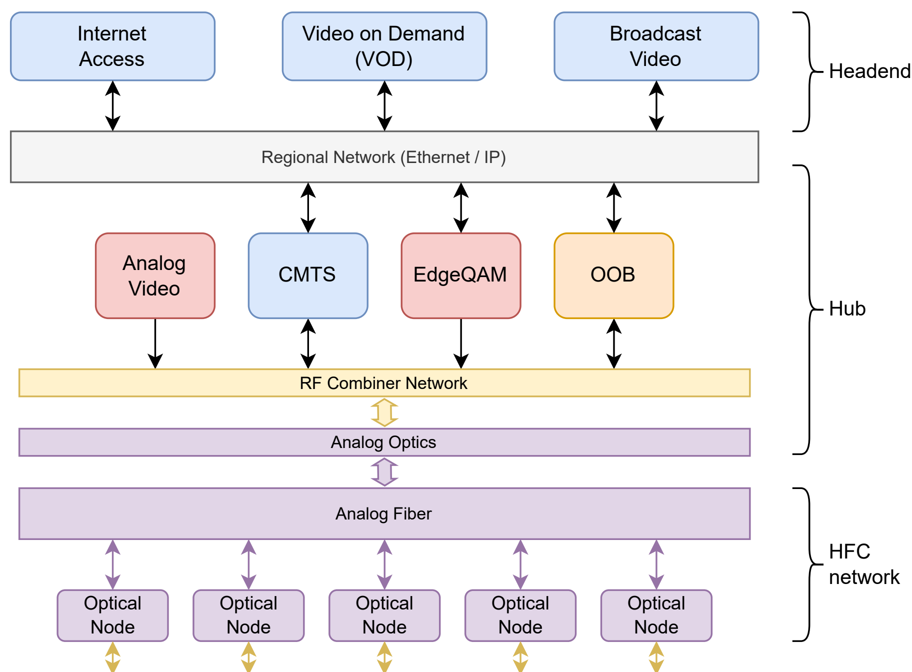

Figure 3.6: Traditional cable access network elements. Adapted from [91].

The traditional cable access network hub has many different functions, such as CMTS, video EdgeQAM, OOB, routing and switching, as shown in Figure 3.6 [91]. The functions are implemented as distinct physical devices whose outputs go through complex RF splitter-combiner network before electrical-to-optical conversion. Consequently the hub has became a complex platform which is challenging to manage, in addition to the spacing and power challenges. A solution to these problems eventually took a form where the traditional cable functions became aggregated to a single platform known as Converged Cable Access Platform (CCAP) [91]. This evolution of functional convergence started already in the late 2000s. The main driver for convergence was the desire to integrate DOCSIS and video data services [91], along with PON functions for multiple system operators [82].

However, as the number of node splittings continue to increase, more than one CCAP is needed at the hub. Power and cooling requirements start to again be the limiting factor. After years of efforts to integrate network functions at the hub, the cable access evolutionary process took a sharp turn and the industry began to consider decentralizing the same functions [82]. The industry chosen solution has been various forms of distributed access architecture (DAA) types that use digital optics in HFC. For one introduction of DAA by one industry actor, see [92]. These systems have started to replace existing cable access network deployments in recent years.

In the first type of DAA that has got industry support, the analog node is replaced with digital optics node. The node has a device that receives digital data over IP network and modulates it to RF format as required by the DOCSIS. In other words, the physical layer (PHY) is moved from CCAP to the node [91] as seen in Figure 3.7. At the hub old CCAP gets transformed into one or more CCAP Cores, one of which must be so-called Principal Core that oversees a set of nodes. Operators are free to divide DOCSIS and video functions between CCAP Cores as fits the purpose. This specific style of distributed architecture is known as remote PHY (R-PHY), while the devices are called remote PHY

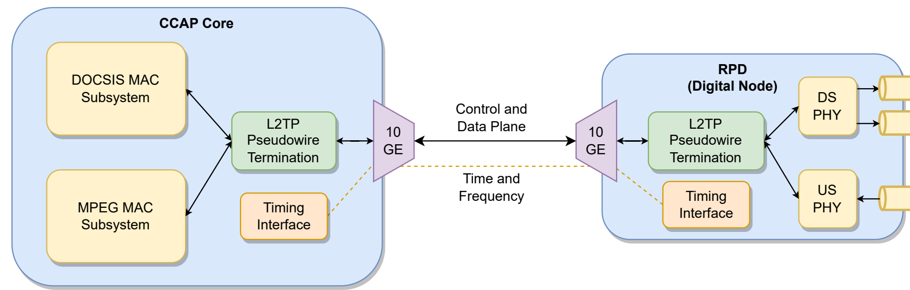

Figure 3.7: Remote PHY architecture. Physical layer (PHY) has been moved from CCAP to remote node. Control and data packets are transmitted over IP-based digital optics feeder segment. [91]

devices (RPD). Remote PHY specification [93] from CableLabs defines the standard[4].

Another DAA variant that is being considered is remote MAC-PHY. Here the cable system MAC functions are also moved out of CCAP to remote location. A node in the field which includes MAC function is called remote MAC-PHY device (RMD). In remote MAC-PHY architecture, the hub might only contain routing, switching and control functions. For even more flexibility a recent CableLabs specification allows locating MAC and PHY functions as separate network entitities. Depending on distance between MAC and PHY elements, one advantage of MAC-PHY architecture is that upstream latency can be improved by a large margin [95]. This is because DOCSIS request-grant round-trip times are reduced the more closer MAC and PHY are to each other. The downside is that system complexity outside the hub increases, thus the cost increases.

Most cable operators are currently in process of converting access networks to use RPDs [92]. However, there are operators who are also using RMDs instead of RPDs. Nevertheless, whichever DAA variant is used, there are common system requirements. First, DAA requires high capacity and low latency IP connectivity between the hub, RPDs

[4]The precursor to R-PHY architecture is Modular Headend Architecture (MHA) [94] as defined in the mid 2000s. In MHA video edge-QAM network element is extended to support DOCSIS downstream PHY. This allows the hub to contain MAC and upstream PHY, while the downstream PHY could be located outside the hub. In addition to other features, R-PHY allows moving DOCSIS upstream PHY also outside the hub. [82]

and RMDs. This network is called Converged Interconnect Network (CIN), as it brings additional synergy advantages from integration of other operator IP services to the cable system. Currently 10G xPON systems are envisioned and being deployed. Second, DOCSIS time synchronization issues get more complex, because MAC and R-PHY are not necessarily co-located. The CableLabs R-PHY specification includes new synchronization model which uses Precision Time Protocol (PTP) to keep MAC and PHY elements synchronized.

4 Case: Kubernetes in edge-native cable access convergence

This chapter describes the case study conducted in this thesis. The technological context of the study relates to cloud-native model and cable access industry, which were explored in Chapter 2 and Chapter 3. The first section discusses cable industry virtualization and convergence trends, providing background and motivation for the case study. Next, the research goals and a plan are introduced based on the discussed background. The following sections include comparison for a set of lightweight Kubernetes distributions and Kubelet agents. Alternative Kubernetes runtimes are also explored, including WebAssembly. The chapter concludes with an analysis of prospects and possible directions for implementing edge-native model in cable access.

4.1 Current trends of cable access and cloud-native

This section discusses the current trends of cable access domain in relation to cloud-native transformation to edge-native model. The discussion here is in context of background information explored in Chapter 2 and Chapter 3.

4.1.1 Full disaggregation and virtualization of cable access

As hinted in Chapter 3, cable access industry is currently on evolutionary path to disaggregate its core functions [96]. While the operator hubs in many cases still have purpose built CCAPs, the relocation of PHY layer outside the hub in form of RPDs or RMDs is already taking place. However, spacing, power and latency issues are driving the industry to find solutions for more disaggregation and flexibility. To this end, Cablelabs has released Flexible MAC architecture (FMA) [83] specification that supports virtualization of DAA. FMA abstracts most of the DOCSIS system physical elements as virtualized software elements, while introducing new management elements and interfaces. To connect the network elements, FMA embraces network function virtualization (NFV) and software defined networks (SDN) models. The specification assumes digital fiber optics to be deployed in access network feeder segments. Another target of FMA is to allow cable system MAC layer to be located almost anywhere in the access network. This makes it possible to serve latency sensitive edge applications. FMA also supports the use of DOCSIS protocol as a backhaul for RAN small cells, which is one part of general convergence trend of access networks, as discussed below.

Virtualization provides many benefits, which include increased cost-efficiency, reduced application development time, and simplified deployment processes. Therefore, it is no surprise that cable access industry is no exception in its desire to virtualize the access network system elements [97]. Flexible MAC architecture is one piece of the puzzle that contributes to this goal. While many of the cable system elements can be virtualized, the PHY layer is one that cannot. As long as the system uses coaxial medium to transmit RF signal, there must be dedicated equipment that do the conversion from digital baseband optics to electrical form. Virtualization efforts will first entail implementing the cable system components as containerized microservices that can run on any commercial-off-the-shelf server, either in operator's premises or in centralized public clouds. To some extent, this virtualization process has already occurred for operations and supports sys-

tems (OSS). For CCAP and other components closer to the edge, virtualization process is still mostly developing, as vendors have only in recent years started to implement containerized CCAPs that run on standard servers located at operator premises [18]. One of the next steps will be to iteratively move the virtualized components out of the on-premise servers to the public cloud, to complete the transformation of on-premise applications and equipment to full cloud-native paradigm [97].

4.1.2 Cable access convergence

Another evolutionary step of cable industry is to be part of a more general convergence of broadband access networks [98, 99]. On high level, Figure 4.1 illustrates the current state of access networks for the three most dominant access mediums and technologies. While coaxial, fiber and radio access technologies all use different technologies to provide

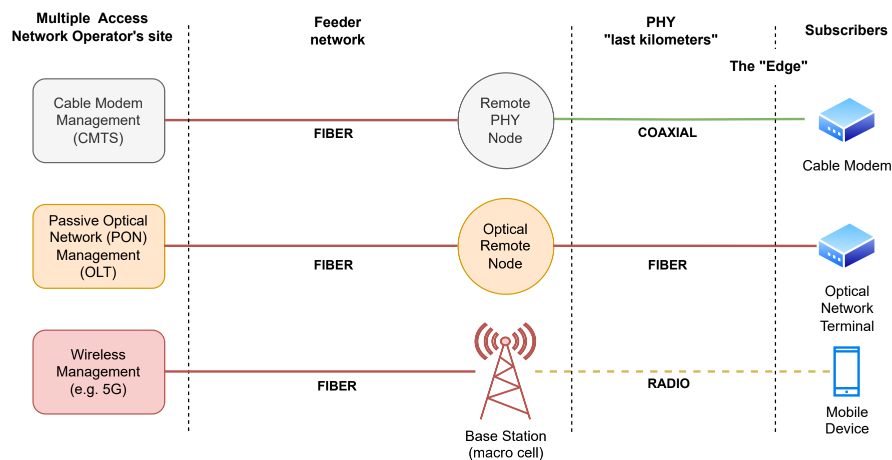

Figure 4.1: High-level view of current state of access networks.

broadband data to subscribers [100], the high-level architectural view reveals similarities. First, as cable access moves to replace analog fiber optics with digital optics of DAA, all the wireline and wireless access systems and their services will be using digital baseband transmission technologies in their feeder networks. This permits converging on IP for all communication between different network elements [96, 100]. Second, all access

technologies will employ some type of remote access node deeper in the edge, which role is to receive IP data packets and convert them to signal format that corresponds with node's access medium. The management protocols are still different, as are many of the employed OSS platforms, while RAN systems in particular have many additional complexities to consider, such as subscriber mobility. On the other hand, when comparing upstream protocols of MAC layer for DOCSIS and PON technologies, as described in Chapter 3, similarities can be found also at protocol level. Third case where convergence opportunities can be seen is in that one access method can leverage existing infrastructure of another access technology. Of particular interest are wireless RAN networks, which can leverage wireline cable DOCSIS or PON networks as overhaul network in the last kilometers [17]. For example, RAN small cell unit located behind cable modem could transmit its PDUs to RAN management systems inside DOCSIS PDUs. Finally, there are convergence opportunities to be found also in management systems and service layers located at the operator premises, and even beyond that when moving towards the core clouds [101]. Clearly, all the various unification prospects of the three different access "silos" at different network layers is enticing, especially for MSOs. Figure 4.2 illustrates some of these on an architectural level.

Convergence can be defined as any system transformation that leads to operational simplification and improved economies of scale [96]. Dictionary meaning of convergence is a process of harmonization and unification. In cable access context, convergence is often used to refer to more specific fixed-mobile convergence, which is an industry term for desire to unify services and components of wireless and wired broadband access networks [98]. More generally, there are many opportunities to harmonize access networks, as touched upon above.

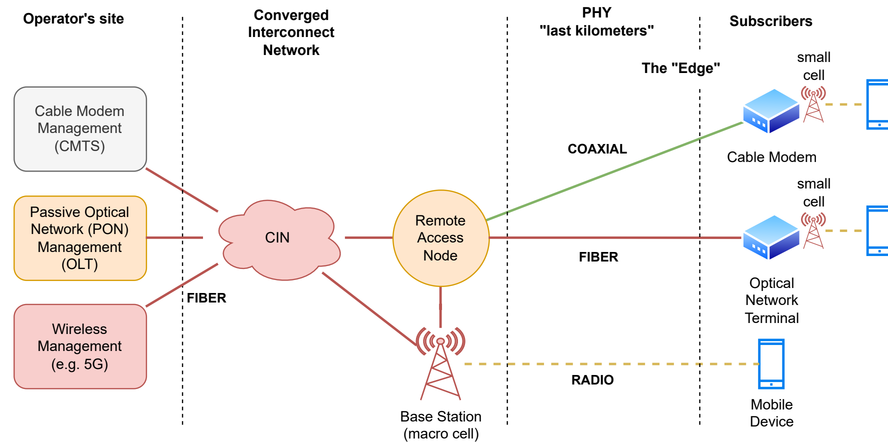

Figure 4.2: Access networks architecture has many convergence opportunities: The feeder network becomes shared IP interconnect network. Remote access nodes at the edge can support more than one access medium technology. Mobile access protocol packets can be overhauled using DOCSIS or PON access protocols.

4.1.3 Evolution of cable to multi-access edge-native

The viability of using virtualized platforms and network functions instead of purpose-built hardware solutions has been acknowledged by the cable, fiber and mobile industries. While the virtualization has at first occurred in OSS level, and now in operators' on-premise aggregation station level, next the process is moving deeper into the access networks edge [100]. Multi-access Edge Computing (MEC) model, despite its origin and major drive coming from mobile industry, is one technology model that cable MSOs are trying to implement or emulate at the network edge. MEC in this context is extended to mean a cloud-native enabled remote device in the network edge with support for one or more access technologies, such as cable or PON. Together with converged IP-based distribution networks, i.e. converged interconnect networks (CIN), MEC can make convergence of all access network technologies to single shared platform or physical edge node a reality.

While the ETSI MEC architecture reference model [39] has its roots in using virtual machines as virtualization infrastructure in MEC hosts, the model is general enough that

other virtualization infrastructures can be used. In fact, recent ETSI work supports use of containers as one possible virtualization infrastructure. Figure 4.3 illustrates a virtualized cable access host platform that has many similarities to MEC architecture, as outlined by Vladyka et al. in their SCTE paper [100]. Another paper suggests that an architecture that

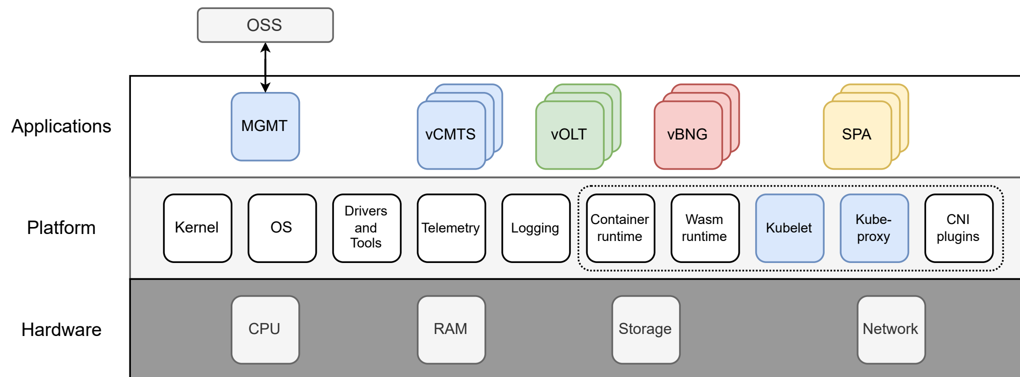

Figure 4.3: Cable access MEC-like host architecture located at operator premises. Application layer corresponds with ETSI MEC applications. In cable context it runs virtualized access management services for Optical Line Terminals (vOLT), Broadband Network Gateways (vBNG), and Cable Modem Termination Systems (vCMTS). Management and Service Provisioning Applications (SPA) provide other converged platform and service functions. Platform layer corresponds with ETSI MEC platform and virtualization entities. The MEC system level is realized by OSS and other services external to the virtualized host platform. Adapted from [100].

fits MEC definition can be deployed using Kubernetes orchestrator [102]. However, the paper elaborates that for complete MEC deployment, Kubernetes must be sided with additional extensions to enable VMs, to chain VMs to containers, and to implement constraint based connectivity scheduling.

As discussed in section 4.1.1, moving applications from operator premises to public clouds as containerized microservices is one of the next steps in path to adopt cloud-native paradigm in access networks. At the same time, as discussed above, and as mentioned in this paper [18], the same process that has already began in operator premises should continue, while taking convergence aspects into account in form of MEC or something else. However, to further continue with this process, the future step is to take the cloud-

native model evolution deeper into access networks. This new deep edge-native paradigm involves finding opportunities to virtualize low-resource edge nodes and devices located outside operator premises. The general convergence of interconnect networks and the consequent convergence of remote access nodes to support different types of physical access technologies is a natural focus point. Therefore, in context of this thesis, convergence of feeder network is assumed, while convergence of remote nodes are given the main focus.

Serving more than one type of access technology from shared access node requires hardware solutions that provide multigigabit speeds and low-latency packet routing to node PHY components. Above the hardware layer in the node is the software stack that implements control plane functions for configuring the hardware and the node. The software components can be designed as dedicated software solutions, without much consideration for cloud-native models. However, access networks and particularly mobile access industry has a strong desire to apply edge computing principles in their edge networks. The driving factor here are emerging edge computing use cases, such as augmented reality, driverless cars and big data analytics [24].

The edge computing facilities are not limited to providing services to the node itself. Their main purpose is to have other edge devices offload computing workloads to them and to make use of dynamic services deployed in the nodes. This kind of capability for dynamic workloads and services in the edge nodes requires portable service deployment units and advanced methods to orchestrate them. Kubernetes and cloud-native model already supports these features. Therefore, an important question is whether cloud-native is applicable in the edge as such. The outlined cable access evolution to converged edge platforms motivates to explore options how this new edge-native can be realized.

4.2 Case study introduction

This section introduces the case study conducted for this thesis. It describes the research goals and states the primary research question. It also elaborates how the research is conducted.

4.2.1 Research goals and question

The research questions in this thesis come in two parts. The first three questions, RQ1, RQ2 and RQ3, were listed in Chapter 1. The first two questions RQ1 and RQ2 were explored extensively in Chapter 2 and Chapter 3. Their results are summarized in Chapter 6. The results were used to understand the background context for this thesis. They were also used to elaborate the single follow-up RQ4 of part two.

The RQ3 was analysed above in section 4.1. The general observed trend in the industry and academy is the desire to bring computing closer to the edge [14]. To achieve this, it is beneficial if the succesful and now mature cloud-native application (CNA) paradigm, along with its tools and methods, was brought to the edge as *edge-native* model. Kubernetes as the de facto container orchestrator is one obvious choice among the tools to move to the edge. However, as can be inferred from overview in section 2.3, the standard Kubernetes release from CNCF is a rather heavyweight platform intended to run on homogeneous set of hosts that have access to significant computing resources, predictable and uniform configurations, strong security, and near unlimited connectivity with high reliability [14]. Kubernetes, like most other container orchestrators, is meant to run in the centralized cloud platforms and data centers [103].

Consequently, the assumption taken for this case study without quantitative analysis is that the standard Kubernetes release [9], or *native* Kubernetes in following, is too heavyweight and inflexible to use for edge computing and edge devices as such. The reason is that the edge environment is highly heterogenous and diverse. Compared to

cloud environments, computing performances, configurations, security features, network bandwidths, and network reliability at the edge vary by a large margin. While native Kubernetes has incorporated large number of components that increase its adaptability to diverse infrastructures, these addons also increase its size, resource needs, and deployment complexity [104]. Furthermore, the approach taken in these addons is more to leverage features that exist in cloud provider's platform, instead of optimizing the actual resource usage. Works by Goethals et al., Marco et al., Kjorveziroski et al., Vanõ et al. [14, 103–105] give validity to the stated assumption.

In parallel to general edge-native trend, broadband access networks as one concrete category of edge networks are evolving to become converged systems in several of their aspects. While the transition to cloud-native in this domain is still occurring, the future trends as discussed are towards edge computing and edge-native. Specifically in cable access context, the edge devices are RPDs, RMDs, and other distributed cable access network devices with spare computing resources. Customer premises equipment (CPEs) must also be considered, as they can leverage edge node resources. Alternatively, the applications residing in edge nodes or in operator premises can be used to manage CPEs. Whether cable systems will adopt MEC-like architectures or something else, the cloud-native model no doubt will have a role. It is precisely here where the looming edge-native trend and cable access trends intersect.

From the stated assumption regarding native Kubernetes, and the related trends of cable access networks, it becomes apparent that to realize benefits of CNA paradigm in cable access, alternative lightweight Kubernetes platforms and solutions should be researched and analysed. If no suitable alternative solution yet exist, analysis of how to implement one should be done. Performances for existing solutions should be measured and compared, or at minimum literature should be searched for applicable performance studies. Therefore, the follow-up primary research question of this thesis, formulated based on results of the three other questions, is as follows:

RQ4. What are the current options to leverage Kubernetes in context of edge-native cable access convergence?

Rest of this chapter explores this question through a case study and presents the findings. Before that, research scope and selection criteria for alternative lightweight Kubernetes solutions are stated. Other relevant aspects and topics that relate to this research are also discussed.

4.2.2 Research plan and discussion

As a major part of this case study, alternative lightweight Kubernetes solutions for edge-native use in cable access context are researched and analysed. A valid solution may be an alternative full Kubernetes distribution. As well a valid solution can be to replace one Kubernetes component, such as Kubelet, with more lightweight implementation. Here, the selection criteria for edge solutions was intentionally chosen as not too restricting. A solution that does not meet all the criteria may still be a valid canditate for analysis. More so, because the research in domain of interest is still fast evolving topic. A solution that might be invalid in respect to the criteria may still contain innovative features that can be useful when adopted in another implementation. The following list has the used criteria:

- The solution is targeted for edge computing, or it can be deployed in the edge context without reducing its inherent performance.
- The solution has lower resource footprint than native Kubernetes for worker node, especially for memory, but also for CPU and storage capacity.
- The size of binaries for worker node is smaller than in native Kubernetes.
- Container runtime is OCI compatible, i.e. it supports standard container images.
- Cluster node component is compatible with native Kubernetes control plane, either directly or via proxy mechanism.

- The solution is open-source.

The case study is conducted primarily using literature review method. The criteria above are used to narrow set of canditate implementations to a core set, which is then used as a baseline for further analysis and research design. However, in addition to the criteria above, limitations derive also from the edge nodes themselves. Hardware limitations, such as available memory, computing power and networking capabilities are important, but software brings other limitations [103]. For example, some operating systems do not by default support kernel capabilities, such as Linux cgroups, which are required to enable containerization. Some of the software requirements could be ignored if intention was to only leverage monitoring capabilities of Kubernetes cluster for the edge nodes, and support for running containerized workloads was considered secondary. However, for true future proof edge-native solution, support for running workloads is close to a requirement.

As already discussed, the edge environment can be harsh with unreliable network connections and low bandwidths, but still with expectations of low latency. Native Kubernetes architecture relies heavily on its distributed etcd database for strong consistency. On one hand, the consistency is necessary also in the edge. However, as noted by Jeffery et al. in [106], increasing the etcd cluster availability by scaling out brings performance issues for applications that require low-latency operations, while also reducing availability and scalability of the overall Kubernetes system. In their work, Jeffery et al. explain why etcd is a bottleneck in Kubernetes clusters, while also giving solutions how these issues might be solved in the edge. As a consequence of these observations, any design for further research should take into account the data consistency model and the cluster database performances. Relying purely on native etcd might not be an option.

Making optimal decisions on resource allocation and provisioning requires more consideration in the edge, because in contrast to centralized cloud, the edge environment is heterogenous. It is therefore necessary to evaluate the extent of how much the lightweight

Kubernetes solutions allow edge nodes to provide state information for the Kubernetes platform. The platform will need this state information, which include amount of available resources, and congestation level of network paths, when making resource provisioning decisions. Works like from Abouaomar et al. [107] have studied and experimented this problem in detail. In many cases the problem solutions require application of advanced optimization schemes, and efficient models for them. For more concrete examples, Kayal [13] has studied how Kubernetes core platform should be modified and extended for edge computing, while Bainbridge et al. in [102] have studied how MEC-like platform can be implemented using Kubernetes. For this thesis, the topic is given at least some consideration when evaluating different edge solutions.

In addition to alternative Kubernetes solutions, this case study considers other possibilities to improve Kubernetes in the edge. While standard Linux containers are mature CNA deployment units in centralized clouds, there are certain issues when adopted in the edge. Therefore, alternative container runtimes that are also Kubernetes compatible are researched. This aspect of edge-native is explored in section 4.5.

4.3 Lightweight Kubernetes solutions

This section describes the alternative lightweight Kubernetes solutions that were chosen for analysis in this case study. Table 4.1 lists all the solutions with descriptions and attributes.

Table 4.1: Lightweight Kubernetes distributions and Kubelet agents. Values inside parenthesis in requirements column indicate recommended values. Native Kubernetes is included as a reference.

Name (Author)	**Description**	**Group**	**Size**	**Min worker node requirements**	**Architectures**
Kubernetes (CNCF)	Official Kubernetes distribution.	-	+1 GB	CPU: 2 cores RAM: 2 GB	amd64 armhf/arm64 ppc64le s390x
K3s (Rancher Labs)	Distribution for IoT and Edge Computing. Single-package binary that encapsulates all control plane components.	A	70 MB	CPU: 1 core (2) RAM: 256 MB (1GB)	amd64 armhf/arm64 s390x
MicroK8s (Canonical)	Lightweight, minimal and low operations production-grade Kubernetes.	A	170 MB	RAM: 540 MB (4 GB)	amd64 arm64 s390x power9
KubeEdge (CNCF)	Kubernetes workloads and device management at the edge nodes.	B	70 MB	RAM: 10 MB	amd64 armhf/arm64
Virtual Kubelet (CNCF)	Kubelet implementation that masquerades as a kubelet for the purposes of connecting Kubernetes to other APIs.	C	+40 MB	Low	amd64 armhf/arm64 ppc64le s390x
Kubemark (K8s SIG)	Kubernetes cluster that runs mock nodes called hollow nodes as Pods.	C	-	Low	-
Kind (K8s SIG)	Tool for running local Kubernetes clusters using Docker container nodes.	C	-	-	-
KWOK (K8s SIG)	Simulation tool for Kubernetes nodes and clusters.	C	-	Low	amd64

Overall, based on literature search and Internet repository readings, the alternative lightweight Kubernetes solutions for the edge can be categorized in three groups. Two of the categories described below are noted in work by Vanõ et al. [14]. Consequently, these two groups A and B are listed also here.

Group A. The first group includes Kubernetes distributions whose approach is to take

the native Kubernetes release and to remove unnecessary components from it, reducing its size and increasing performance by various degrees. These platforms may also employ other optimization methods. One is to converge many components into one, with intention to reduce inter-component communication overhead. The implication of convergence optimization is that Kubernetes internal interfaces are made less generalized, and consequently less scalable. The downside of this approach is that features that might be useful at the edge in certain scenarios may have been selected for removal. While it is possible that such features can be plugged back in, there are no guarantees. That said, platforms that choose reductionist approach generally include features that are most useful and suitable in edge environment, assuming the platform is targeted for the edge. K3s [108] and MicroK8s [109] belong in this category.

Group B. The second group includes edge platforms which aim to keep native Kubernetes cluster and its control plane in the cloud, while implementing Kubernetes compatible edge platform consisting of autonomous edge node agents. An edge node runs container workloads using standard container runtimes, manages edge device communication, and has internal state cache so that the edge node can work even when there is no connection to the control plane at the cloud. One method used to integrate the edge platform and Kubernetes control plane is by using Kubernetes custom controllers and custom resource definitions. In addition to container workload management, edge device management may require their own custom controllers and Kubernetes resources. KubeEdge is probably the most promising canditate in this group [110].

Group C. Third group of solutions reimplement Kubelet agent for specific purpose. This is the most limited alternative solution, since the edge platform side can at most implement what the Kubernetes standard Kubelet API allows. While this approach provides the most potential for optimizing worker nodes corresponding to resources, it is also the most laborous method from implementation point of view. However,

some of the labour can be reduced if an extendable base implementation for Kubelet could be found that already exists. To some extent Virtual Kubelet [111] provides this kind of base implementation.

4.3.1 K3s

K3s is a lightweight Kubernetes distribution from Rancher Labs [112] that packs everything in a single binary less than 70MB in size [108]. One of the goals of K3s is to allow use of edge devices and other low-resource devices as Kubernetes cluster nodes. In K3s cluster, the master and worker nodes are launched from the same binary. The node role as either master or agent is given via command line argument. The binary runs all the Kubernetes control plane components and agent components in single process. In addition to binary size, the runtime memory footprint is smaller compared to native Kubernetes.

The chosen implementation approach of K3s is to take a native Kubernetes source release and remove unnecessary code and functionality from it. K3s is therefore a fully compliant and production-ready Kubernetes distribution. However, since K3s worker nodes have slightly modified cluster join mechanisms, the nodes cannot be used in native Kubernetes clusters. While K3s has limited default feature set, it comes prepackaged with all the necessary components to run a Kubernetes cluster. The features include containerd, Flannel for CNI, CoreDSN, Traefik for Ingress, Helm-controller and more. For CRI, K3s uses containerd runtime that cannot be changed. By default K3s uses database backend that is based on SQLite3. For larger cluster setups that require high availability, K3s has etcd, MySQL and PostgreSQL available as data storage.

In addition to K3s, Rancher provides dedicated operating system for it, known as k3OS [113]. Just like K3s, the k3OS is better tuned for low-resource host nodes. To get best results from K3s, it should be combined with k3OS.

4.3.2 MicroK8s

MicroK8s is a production-grade minimal Kubernetes distribution that aims to have low-operational burden [109]. It runs on Windows, Linux and macOS. Intended use cases are development environments, DevOps, edge computing, and IoT applications. MicroK8s has extensive support for high-availability Kubernetes clusters. Additional features can be enabled via add-on mechanism. Canonical has Core repository that hosts addons that are officially supported by MicroK8s, while Community repository hosts additional non-official third party addons. Some of the addons, such as *dns*, are recommended to be always enabled by MicroK8s team. MicroK8s by default has only core features of native Kubernetes, including api-server, controller-manager, scheduler, kubelet, cni and kube-proxy, and more. It supports containerd and kata container runtimes. Kata containers are lightweight virtual machines that run through container runtime compatible APIs. MicroK8s by default uses snapd to install itself, but other install methods are also provided.

4.3.3 KubeEdge

KubeEdge is an official CNCF hosted project that extends native Kubernetes to the edge for purposes of edge computing and management of edge devices [110]. It consists of cloud part and edge part as seen in Figure 4.4. The cloud part (CloudCore) implements pair of Kubernetes controllers that consume Kubernetes CRDs. The edge part (EdgeCore) implements event-based messaging functions, Pod and container management functions, data persistency functions, device management functions, MQTT functions, and HTTP client proxy functions. KubeEdge employs MQTT as Pub/Sub broker to connect edge devices to the edge node and the cloud. According to [114] the EdgeCore agent memory footprint can be as low as 10 MB at runtime.

The CloudCore and EdgeCore use their respective CloudHub and EdgeHub components to communicate through HTTP over web-socket connection. CloudHub acts as a mediator between CloudCore controllers and edge side components. It watches for

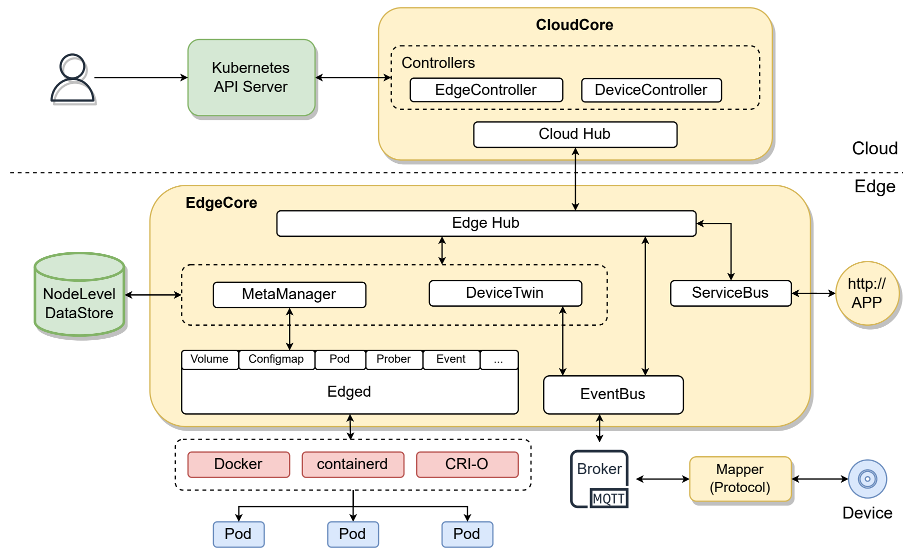

Figure 4.4: KubeEdge architecture. Kubernetes is extended with custom resource definitions and controllers in the CloudCore. EdgeCore has Edged agent that manages containerized applications, DeviceTwin that handles edge device management, and MetaManager that handles data persistency. [110]

changes at the cloud side and reports them to the EdgeHub, while listening for incoming messages from EdgeHub and transmitting them to controllers. EdgeHub has a similar role in EdgeCore to forward messages from the cloud side to the correct components at the edge, and to send messages and responses back to CloudCore. EdgeHub in addition acts as broker to edge site's inter-component messaging.

The role of Edged component is to manage Kubernetes Pod lifecycles at the edge node. Users can launch various workloads using `kubectl` via Kubernetes API. Edged supports several OCI-compliant runtimes through CRI, as seen in Figure 4.4. Edged implements functions as submodules that have different roles. For some examples, Pod Management submodule handles Pod additions, deletions and modifications, Volume Management submodule attaches and mounts volumes for Pods scheduled on the edge node, Status Manager submodule sends Pod status information to the cloud side, and Probe Management submodule provides Pod monitoring functions.

MetaManager component is the message processor between Edged and EdgeHub. In addition, it is responsible for management of metadata in lightweight SQLite database.

MetaManager processes messages that pass through it, and depending on message type, it reads or modifies the local database content. Local data persistency allows EdgeCore to function even when it has no access to the CloudCore. As the data storage is designed to work in offline mode, the etcd performance issues should not affect KubeEdge.

DeviceTwin component stores edge device statuses, handles device digital twin operations, creates memberships between edge devices and the edge node, and synchronizes device status and twin information between the edge and cloud side. In brief, DeviceTwin implements most of the edge device management aspects of the KubeEdge.

4.3.4 Virtual Kubelet

Virtual Kubelet is an open-source implementation of kubelet agent that masquerades as real Kubelet [111]. It can be used in parallel with nodes which are running native Kubelet, as in Figure 4.5. Its purpose is to connect Kubernetes cluster to APIs of other cloud

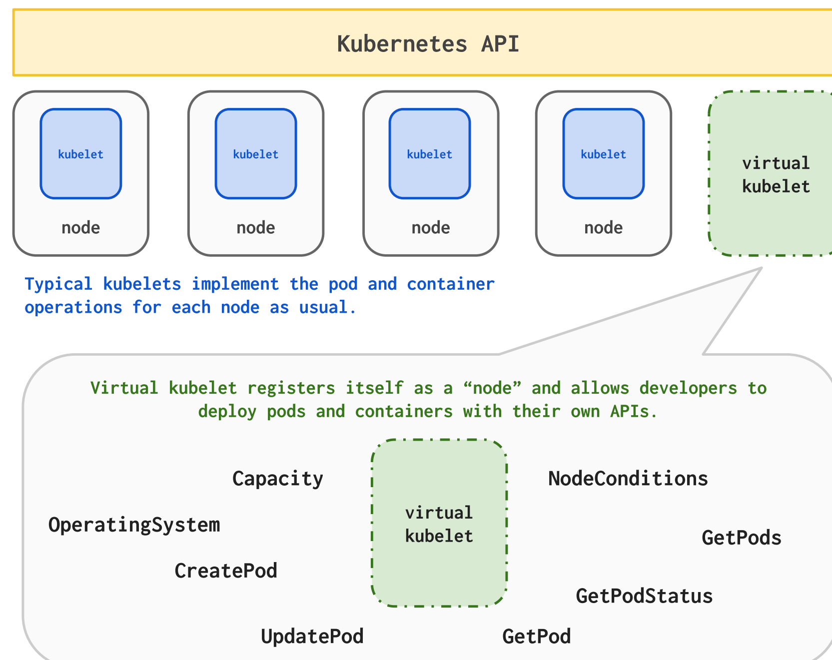

Figure 4.5: High-level architecture of Virtual Kubelet. Virtual Kubelet masquerades as node kubelet agent that provides the cluster access to third party cloud container services. Source: [115]

container services and platforms. The primary use case is to connect with serverless container platforms, such as Azure Container Instances (ACI) or AWS Fargate. When started on a cluster node, Virtual Kubelet registers with Kubernetes API in the same way as native Kubelet would do. Kubernetes sees the Virtual Kubelet as an ordinary Kubelet instance running on a node, and will try to schedule Pod workloads to it. Virtual Kubelet can be installed to run either external to any clusters, or it can be deployed as a normal Pod that is run inside a cluster. There is also a Helm chart that can be used to install Virtual Kubelet as a Pod.

Virtual Kubelet has *providers* as extension points. Virtual Kubelet project [115] implements only some of the core features of Kubelet, and delegates the actual Pod management functions to provider implementations. Providers are expected to conform to Virtual Kubelet API, and they should implement any necessary backend operations that enable proxying workloads to provider-specific external services. Several builtin providers are available. Among these are Kubernetes CRI Provider, which in essence provides similar container runtime support as native Kubelet. However, that provider is not fully featured Kubelet in its feature set, and it does not aim to replace Kubelet. However, it could still be useful starting point to implement custom lightweight alternative runtimes.

4.3.5 Kubemark

Kubemark is a performance testing tool that simulates Kubernetes clusters [116]. The simulated clusters can be larger than real ones, which is useful for scalability testing. Kubemark cluster has a real master node, and number of virtual worker nodes, called *hollow nodes*. The hollow nodes include number of "hollow" components which are mock implementations of corresponding real Kubernetes components. The hollow components, such as HollowKubelet, do not really do anything, but they report to Kubernetes control plane as if they did. The master node of Kubemark runs on dedicated host. Hollow nodes, on the other hand, are run as Pods inside external real Kubernetes cluster. In other words,

the hollow nodes are not real host nodes, but containers running inside Pods.

Kubemark is not suitable as an alternative Kubernetes edge solution as such, since its hollow components do not really do anything. In addition, it may deploy several hollow nodes on same underlying real cluster node. However, it might be possible to use "empty" implementation of the HollowKubelet as a base to extend when implementing own custom Kubelet agent for low-resource edge use.

4.3.6 Kind

Kind (kubernetes-in-docker) is set of go tools to bootstrap local Kubernetes clusters, which nodes are simulated by running them in Docker containers [117]. Kind is not lightweight Kubernetes alternative, since it by default uses fully-fledged native Kubernetes to manage the Docker container based nodes. Kind has been developed for the purpose of testing native Kubernetes platform. To create a cluster, single command line tool needs to be run. It will download prebuilt Docker images when bootstrapping cluster nodes. The prebuilt images contain system features to run nested containers, systemd for Kubelet, and required Kubernetes components. To use different Kubernetes version, Kind command line tool provides option to change the Docker image that is used for nodes. Kind supports cluster configurations with variable number of nodes, including high-availability setups.

Clearly, Kind is not applicable for production use in edge environments, as it is targeted for local testing. It is still included here since it can be useful tool to experiment with alternative Kubernetes distributions that actually are more lightweight.

4.3.7 KWOK

Kubernetes WithOut Kubelet (KWOK) is another testing toolkit that can be used to generate clusters with thousands of simulated nodes [118]. It has low resource footprint which allows it to be used on single development machine. Since the tool only simulates nodes

and Pods running on them, it cannot be used to run any real container workloads. Under the hood, KWOK implements two Kubernetes resource controllers which are responsible for simulating the lifecycle of nodes, Pods and other related API resources. KWOK is relatively new project, with its origins in now-deprecated fake-kubelet and fake-k8s, which were also tools for testing Kubernetes.

As it is purely a testing tool, obviously KWOK is not applicable for true lightweight edge solution. However, as the simulated nodes and Pods are configurable, it can be used to test any lightweight Kubernetes solutions. Furthermore, it is also possible to envision using KWOK as a baseline project to modify and extend for it to have at least some level of support for edge devices. For example, existing KWOK controllers could be replaced with modified versions that were able to communicate with custom lightweight edge node agents to pull state information about the nodes. This approach would not enable any Kubernetes workload orchestration, but it would enable Kubernetes management of edge device states for monitoring purposes.

4.4 Comparison of lightweight Kubernetes distributions

The alternative Kubernetes solutions have different architectures and they have different intended purposes. This makes direct comparison of all of them somewhat artificial. However, comparing solutions that share architectural style is beneficial to find out best solutions. Literature was searched for comparisons between these. Following has performance comparisons for solutions in group A.

In their work, Telenyk et al. [119] have compared performances of K3s and MicroK8s (Group A) to native Kubernetes. All the tests were run on virtual nodes that had 2 virtual CPUs and 8 GB of RAM. As results, they noted that native Kubernetes was actually more performant compared to the other two. Lightweight solutions had slightly worse overall CPU and memory utilization compared to native, as would be expected from their smaller

size. K3s was shown having slightly better overall performance than MicroK8s, and it also showed much better disk utilization among all. From edge-native deployment perspective, this study does not give that much information, because the runtime nodes used in the tests were not representative of low-resource edge devices. It is to be expected that running native Kubernetes cluster on nodes that are well within Kubernetes requirements specs shows its best potential performance.

Kjorveziroski et al. [104] have made similar performance comparison as by Telenyk et al. for Kubespray[1], K3s and MicroK8s. They used OpenFaaS serverless platform as the test target, with more than ten different benchmarks to measure performance. Both lightweight Kubernetes solutions provided better overall performance than native-like Kubespray. The differences between K3s and MicroK8s were not significant. The host nodes used for testing all had fast CPU with 8 cores, and 8 GB of memory. The runtime environment was therefore not fully comparable to edge-native environment in this study either. While focus of this thesis is not serverless computing, this study still gives positive signal that lightweight solutions can be more performant than native Kubernetes.

4.5 Edge-native application runtimes

Containers are lightweight deployment units for applications and workloads, suitable for running in the edge. As discussed, Kubernetes abstracts the containers with its Pod concept. The Pods can then be orchestrated to run in edge devices, assuming the target devices have capacity and capability to do so. Kubernetes Container Runtime Interface (CRI) allows for using different container runtimes, in addition to default containerd. For example, CRI-O is more lightweight container runtime, originally targeted as the reference implementation of CRI. CRI-O uses runC by default as its low-level container run-

[1]Kubespray is another Kubernetes distribution that contains all standard native components. It has significant hardware requirements comparable to native Kubernetes, and therefore is not targeted for edge environments.

time, similar to containerd. However, the low-level runC isolates each container only in a separate kernel namespace, which brings security concerns in respect to isolation [50], as discussed in section 2.2.2. Since the edge is particularly vulnerable in many aspects of security, any edge-native solution needs more robust security mechanisms than standard cloud-native models currently provide.

Containers are also not fully agnostic to their environment in the sense that they could be built once and run everywhere. While edge devices typically host some type of Linux OS, it cannot be guaranteed. Therefore, container developers cannot fully trust the host OS interface running on any specific device will match with the runtime needs of their container instance [120]. Also, the edge has wide variety of hardware architectures in use, which include x86, amd64 and arm. The implied solution for these issues is that containers need to be built for all the possible combinations of operating systems and architectures, which considerably increases complexity for application development and deployment. The same heterogeneity has further consequence in terms of Kubernetes orchestrator, in that there is no guarantee that any particular device node can be used as deployment target. It can be seen that orchestration complexity increases in the edge context when using standard container solutions.

One solution could be to use a platform that already exists and can run applications everywhere. The Java Virtual Machine (JVM) is one such platform and runtime [120]. With JVM, it is possible to compile Java source code once to common binary byte code format, and then run the binary anywhere with JVM instance. However, JVM does not fill the requirement of lightweightness that is required in the edge, and is therefore more suitable in data centers and powerful server machines. However, there is another byte code language known as WebAssembly [15] that has recently emerged. It shares the beneficial attribute of JVM in that it can execute in heterogenous environments without recompilation. WebAssembly has also lightness comparable to containers. Before discussing WebAssembly in more detail, next section briefly introduces other alternative runtimes

that currently exist and may be suitable in edge context.

4.5.1 Alternative Kubernetes runtimes

Reference component runC is not the only low-level runtime that high-level Kubernetes runtimes, such as containerd and CRI-O, can use. Any OCI compliant low-level runtime can be used with Kubernetes, provided that the runtime is also supported by a CRI runtime.

For one, Kata containers [121] is a type of runtime that executes containers inside lightweight virtual machines, or in so-called microVMs. Each container runs over dedicated kernel in microVM, instead of running the container directly over the shared kernel. Kata container VMs provide extra security for running containers, although the new layer brings runtime performance overhead [122]. Another container runtime example, which is also security oriented, is gVisor's runsc [123]. Instead of wrapping containers inside VM, it acts as a system call proxy, or an application kernel, between the container and host kernel. This allows for tighter control and monitoring of application behaviour, which increases security. The downside is that the system call interception mechanism that it uses brings serious overhead [122].

Yet another type of container runtimes are ones that utilize unikernel approach for more isolated container environments. Unikernels can be more performant than containers in many aspects [122]. However, they have also many disadvantages that reduce their applicability [14], not only in edge context, but also in centralized cloud context. One of the most prominent is lack of standardization, but lack of development tools and debugging capabilities are other.

Lastly, there are container runtimes that are focusing on high-density, high-concurrency and secure serverless FaaS deployments. Here, RunD [124] is prime example. It is capable of launcing over 200 containerized function invocations concurrently per second, while achieving container density of over 2500 in single VM node. However, serverless

computing should be understood predominantly as public cloud offering, since it heavily relies on assumption of homogenous and uniform data center infrastructure [14]. These assumptions do not hold in the edge. Therefore, container runtimes and serverless methods in general may not be considered optimal solution for edge-native.

4.5.2 Standalone WebAssembly runtimes

Recently, another type of lightweight application runtime known as WebAssembly [15] has emerged. WebAssembly (Wasm) is a general-purpose virtual binary instruction set architecture (ISA) that is executed in stack-based virtual machine [15, 120]. While originally developed for the Web in mind, the current Wasm is designed to be executed both in web browsers and in server-side using standalone *Wasm runtimes*. Wasm runtime is the software component responsible for actually executing Wasm virtual binary instructions, using runtime specific execution models such as interpretation, just-in-time-compilation (JIT) or ahead-of-time compilation (AOT) [120]. Wasm runtimes integrate with the underlying OS through standard API known as WebAssembly System Interface (WASI) [125], as depicted in Figure 4.6.

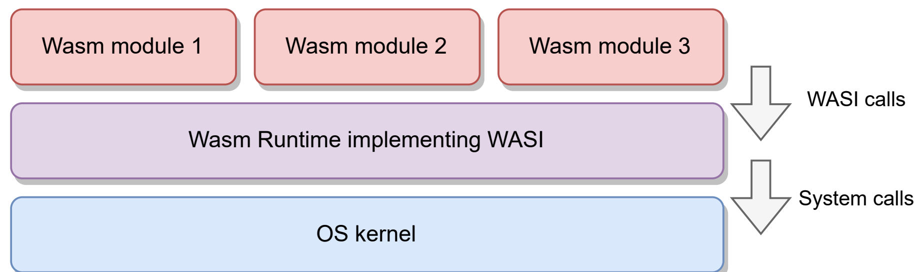

Figure 4.6: Wasm runtime that implements WebAssembly System Interface (WASI) translates functions, which Wasm module imports and calls, into kernel system calls. [14]

WASI is a modular API that allows creating Wasm runtimes outside browser context that provide common OS integration interface for Wasm modules. The core principles behind WASI are security and portability [126]. For portability, WASI provides access to OS capabilities and features through its API modules. For example, wasi-core module

provides standard POSIX-like interface with file access, networking, time functions and more. Wasm runtimes are expected to implement the WASI modules in a way that suits their specific use case. For security, WASI takes capability-based approach [126]. Instead of relying on kernel to manage access to system resources, Wasm runtime must be given explicit set of access privileges, which defines the level of access that any Wasm module can at most have. In other words, where as containers typically use allow-by-default model relying on standard OS security features, Wasm takes deny-by-default model.

As of now, many different Wasm runtimes outside the web context have been developed, which are interestingly often targeted for edge computing, embedded computing and IoT. Following briefly describes some of them in no particular order of importance:

- Wasmtime [127] is a Bytecode Alliance [128] project that presents fast, standards compliant Wasm runtime. It uses Cranelift as the compiler backend, therefore supporting several execution models. Wasmtime was one of the first non-browser runtimes that implemented WASI. It has CLI tool to run Wasm modules, and it can be used as a library module embedded in several programming languages. Wasmtime is written in Rust, but it has native C API, allowing for native embedding in these two languages. Bindings to C API exist for other languages.

- WebAssembly Micro Runtime (WAMR) [129] is another Bytecode Alliance project that provides a lightweight standalone runtime with small footprint and high performance for many use scenarios, including the edge. It supports interpretation, AOT and JIT as execution models. WAMR can be embedded into C, C++, Go and Python host programming languages, but it also comes with CLI tool to run Wasm binaries.

- WasmEdge is a lightweight, high-performance and extensible Wasm runtime targeted for the edge, cloud and distributed applications [130]. It is an official sandbox project of CNCF. It supports all standard WebAssembly features and it has also number of custom extensions, or plugins, for cloud-native and edge computing scenarios. It has

CLI tool, and it can be embedded as a library runtime for Wasm in several languages, including C, Rust, Go Python and Node.js.

- Wasmer [131] is runtime that enables lightweight containers to run anywhere from Desktop to the cloud, the edge and IoT devices. The runtime can be embedded in many different languages, which makes it possible to run Wasm binaries in wide range of ways.

- Wazero is a newly released Wasm runtime written purely in Go language, targeted for Go developers [132]. It is Wasm standards compliant and supports basic set of WASI features. It comes with standalone CLI tool to run modules, and the runtime libary can be embedded in other Go programs.

While the above list is by no means exhaustive, it can already be seen that Wasm runtimes are typically implemented as lightweight libraries that can be embedded in other tools that may use wide variety of programming languages. In Wasm terminology, the embbedding application process is referred as the *host*. For other terminology and details, the current WebAssembly Core Specification 1.0 standard is available at [133].

Wasm has several benefits when running applications and workloads in edge contexts. First, Wasm binaries can be compiled using wide variety of toolchains and high-level programming languages, including C, C++, Rust, Python and Go. More languages and toolchains are expected to be supported in the future [134]. Second, Wasm modules have near-native runtime performance [120], although some benchmarks indicate the performance can at times be clearly worse. Third, Wasm module startup times are significantly better compared to containers. Fourth, Wasm module binaries have very small size, with low memory footprint [14]. Fifth, Wasm modules are executed in sandboxed environment with particular consideration given to security, very much similar to how web applications are run in restricted browser context. Furthermore, many of the existing Wasm runtimes provide support for different hardware based Trusted Execution Environments (TEE). Fi-

nally, WASI is designed to be portable [14], meaning that it is easier to implement Wasm runtimes on wide variety of systems. Compiled Wasm binaries that import WASI features can be executed without modification on any environment where a WASI compatible runtime exists.

Works like from Ménétrey et. al and Vanõ et al. [14, 120] suggest that Wasm modules can be perfect fit for adapting cloud-native model to edge-native. However, Wasm is not going to replace current container based workloads. Instead Wasm is expected to complement containers, especially in low-resource edge devices. However, one challenge that remains to fully adapt Wasm in the edge relate to extending WASI host capabilities that runtimes expose for Wasm modules [120]. There is a clear trade-off between WASI portability and the surface area of the system calls that it exposes. While full system call compatibility would allow implementing and porting wider range of applications, it complicates portability between heterogenous host devices in the edge. This said, WASI does provide extension points for native code, to enable access to host hardware functions that are not otherwise exposed. This way the WASI common interface can be kept compact and portable, while still allowing for custom OS integrations by other means. Finally, the Wasm and WASI are trending topics in industry and academy, which means that the standards are still changing and evolving. At this point in time it is not clear which Wasm features, capabilities and toolsets will remain after the "hype" phase stabilizes.

4.5.3 Kubernetes as edge-native orchestrator

RuntimeClass is Kubernetes resource type that is used to define cluster runtime support for different workloads, such as Linux containers and Wasm binaries. Pods and Deployments can be bound to specific runtime type by setting their `spec.runtimeClassName` field to refer to `metadata.name` field of a RuntimeClass. The RuntimeClass has nested `scheduling.nodeSelector` field that contains label-based selector matchers that Kubernetes scheduler uses when choosing suitable nodes where to assign Pods. The Run-

timeClass resource has also `handler` field that defines the target runtime. As an example of using RuntimeClass, Pods could by default run as standard runC containers when no runtime was set for them, and as Kata containers inside lighweight VMs if their runtime was set as kata-container.

However, a third set of Pods could be configured to run through Wasm runtimes. Here, the enabling component for Kubernetes orchestration is OCI compatible low-level container runtime that supports running one or more of Wasm runtimes. For containerd, there is a shim library component known as *runwasi* [135] that supports WasmEdge and Wasmtime runtimes. Wasm binaries can be packaged as OCI compatible images, and if Kubernetes and containerd are configured to use the Wasm shim, the image payload gets executed as Wasm application instead of Linux container. Another option is to use *crun* [136], which is low-level OCI container runtime implemented in C language. It has native support for running WasmEdge, Wasmer and Wasmtime. By replacing default runC in containerd or CRI-O with crun, it is possible to run Wasm binaries in Kubernetes cluster.

This kind of support for hybrid Kubernetes cluster in respect to runtimes is clearly beneficial in edge-native context. Since the edge is highly heterogenous, use of different runtimes based on device capabilities brings flexibility.

4.6 Prospects of edge-native in cable access

Native Kubernetes is a heavyweight platform that assumes centralized and uniform cloud infrastructures. While certain level of uniformity is present in cable access edge networks, use of Kubernetes as such in these networks for edge-native transition is not practical solution. Three groups of suitable alternative lightweight Kubernetes solutions are recognized.

In the first group, K3s and MicroK8s are solutions targeted for the edge as lightweight

and fully compliant Kubernetes distributions. Both distributions provide simplified installation procedures. The common approach these solutions take is to strip away unnecessary Kubernetes features. K3s in addition modifies the native architecture to further improve performance. The downside of this K3s approach is that components of K3s, such as the node agent, are incompatible with native Kubernetes components. Both solutions have low worker node RAM footprint, and their binary sizes are small. Both solutions provide good support for variety of architectures. However, K3s is better suited for restricted edge deployments, since it has support also for arm32 architecture. K3s replaces etcd with SQLite database by default, which when taking case study preconsiderations into account may bring better performance on the cable edge. Analysis of performance comparisons found in literature shows that between the two, K3s has slightly better performance than MicroK8s, while the comparison to native Kubernetes gives mixed results. However, the analysed performance tests were all executed on powerful test hardware, which cannot be considered as representative of the edge. Overall, of the two solutions, K3s can be seen as the more mature and more edge-ready solution for cable access.

KubeEdge was the only analysed solution in the second group. The approach that KubeEdge takes is to leverage real Kubernetes cluster running in the centralized cloud as CloudCore, while having custom CloudEdge agent implementation that supports OCI workloads. An interesting feature of KubeEdge is that the edge agent can function autonomously without connection to its counterpart in the cloud. For diverse edge environments with unreliable cloud connections, such offline capability is useful. Furthermore, since each edge agent has its own local database, the performance issues that were noted regarding native Kubernetes etcd are not so prevalent. KubeEdge has also facilities to support edge devices through its inbuilt support for MQTT protocol. Although such capability may not be directly useful for cable access networks as such, the capability could be useful for MEC-like systems that allow third party applications to execute in EdgeCores as MEC applications that can leverage other available edge services. Compared to K3s

solution, KubeEdge overall is more complete edge solution for cable access. However, the choice between the two depends on how much support for other edge devices is needed. Furthermore, KubeEdge architecture distances the edge agents from native Kubernetes platform, which might become an issue if the EdgeD component starts to lag behind in development.

In the last group are alternative lightweight Kubelet implementations. The underlying prospect is that an existing lightweight Kubelet can be taken as a basis which to extend. The base can be used as a starting point to implement a custom Kubelet agent that fulfills any explicit requirements in cable access context, i.e. integration with real Kubernetes cluster. Virtual Kubelet is the most ready and mature solution within its own group. That said, the other solutions in the group can be used as reference implementations if the purpose was to implement Kubelet agent almost from the ground up.

In addition to alternative lightweight Kubernetes solutions, it is also necessary to adopt lightweight workload runtimes in edge-native cable access. Here, WebAssembly provides architecture agnostic, secure runtime environment. Wasm can be combined with all of the highlighted solutions, K3s, KubeEdge and Virtual Kubelet, which makes it an ideal solution to use as lightweight deployment unit in any cable access edge-native architectures.

5 Discussion

This chapter presents additional discussion points that this study may have revealed. First, extended considerations are given on cloud-native and cable access topics that were explored in Chapter 2 and Chapter 3. Next, any left over points that relate to case study conducted in Chapter 4 are covered. The chapter concludes with evaluation of study weaknesses and future work.

5.1 Cloud-native model evolution to edge-native

As discussed in Chapter 2, the cloud from its inception in 2005 has seen rapid evolution. Although tools and techniques used to provision virtual machines at IaaS layer have seen improvements, the role of IaaS as the fundamental base over which other platforms and services operate seem to have not really changed. The same is not true for the upper cloud layers PaaS and SaaS, which have seen many changes in their development and deployment models. In essence, the heavyweight VM model used to deploy monolithic applications as described in section 2.2.1, has seen transformation to more lightweight methods and tools. Now, it can be also seen that Kubernetes has taken the place as the de facto standard container orchestrator. This point was seen stated throughout the online and literature sources used for this thesis. Alongside Kubernetes, the containerization trend has been popularized by Docker. While Docker has had significant role to play in containerization evolution, the ecosystem today is larger, thanks to open-source runtime components and standards like OCI and CRI.

On top of issues noted in case study section 4.5, there are other issues and improvements to be made with cloud-native containers. For one, while containerized microservice architectures are scalable and elastic, they require some resources to be always provisioned, even if the services have no load. Serverless computing has been suggested as one solution by many authors, with FaaS being one of implementations offered by most cloud providers. Serverless was not covered extensively in this thesis, although it was seen there is an interest to move serverless architectures from public clouds to the edge. This interest aside, serverless has still inherent startup latencies and performance issues, which may make it non-optimal solution, unless recent runtime improvements like in RunD can be moved to the edge. Unikernels are another microservice oriented solution, which can be faster than containers. However, unikernels have not yet seen wide adoption, most likely because there are development and runtime transparency related problems with their use.

It is well known that centralized clouds are operated in large data centers built over homogenous infrastructures. This is the result of why cloud model has been succesful in the first place. With combination of hardware virtualization and uniform infrastructure, it is possible to achieve the greatest benefits through economies of scale. Whereas virtualization solves the underutilization problem, uniform infrastructure brings needed predictability. Therefore, it is possible to infer that the issues mentioned above, while problematic to an extent, are not so crippling that they could have stopped cloud-native paradigm from establishing itself in the way that it has.

Edge computing is another concept that has seen many evolutionary forms. Origins of edge computing are in the larger order vision of pervasive computing, which has been predicted to materialize many times over the years. Fog computing can be considered one later appearance of the same vision, with some practical implementations. Fog computing is interesting in that it includes the whole continuum from the central cloud to the edge in its architectural semantics. Still, fog computing can be seen more as a theoretical concept that aligns with pervasive computing. On the other hand, cloudlets model and its

later successor multi-access edge computing are edge architectures designed and targeted particularly for mobile devices.

As edge computing architectures and models are taking shape, some degree of integration with centralized cloud models is needed, since edge computing should be understood as only part of the larger cloud continuum. However, as this work noted, the edge environment is more diverse, insecure and unpredictable compared to the core clouds. These aspects bring difficulties. Consequently, there are still unsolved problems in the edge that relate to application migration and orchestration, communication reliability, mobility functions and security. Solutions to these problems will require dedicated methods, which are not found in centralized cloud. Nevertheless, as far as possible, the computing and development models that exist in the public cloud should be adapted and moved to the edge. As cloud-native model in central clouds has matured, an interest to move its characteristics to the edge has emerged. This new model can be called edge-native model. However, it can be seen this edge-native transition is still in the early stages.

5.2 Cable access future prospects

Against the backdrop of constantly increasing customer demand for more bandwidth, and the pressure to remain competitive against other access operators who employ different access technologies, cable industry has to look for ways how to increase their bandwidth capacity. Transition to distributed architectures has been one response, as it gives an opportunity to use more spectral efficient RF modulations, such as OFDM, in the last kilometers of cable network. Other methods are to extend the limits of usable downstream and upstream spectrum bands, and to use full duplex transmissions as discussed in section 3.5. In the first case, Extended Spectrum DOCSIS increases the upper limits for upstream and downstream bands. For the second case, Full Duplex DOCSIS aims to increase capacity by allowing use of overlapping upstream and downstream bands. All

the methods listed above are intended to realize symmetric multi-gigabit speeds for 10G vision that is being branded by cable access industry.

Visions aside, even with improvements, cable access technology will have difficulties to compete in raw capacity against fixed fiber optics like PON in new deployments. However, as the cost of replacing existing coaxial lines with fiber is still high, many cable operators see it more cost-efficient to update existing cable systems with new cable equipment that improves bandwidth capacity over existing coaxial lines. Here, the methods listed above, DOCSIS 4.0 extensions, and technologies like FMA will no doubt help operators to remain competitive to an extent.

Still, as the far future trend of cable technology can be seen on decline, cable operators are naturally forced to consider alternative options. Multiple system operators have a strong interest in unifying and harmonizing their access networks and service offerings as far as possible to achieve synergy benefits. Convergence of access networks is a general trend that cable industry has to take into account in its future plans.

5.3 Case study considerations

Although KubeEdge was the only analysed solution in group B of lightweight Kubernetes solutions, it must be noted there are other solutions that could have been studied. For the reason why no other solutions was chosen, one is that early in the case study, arguments were found for clear leading edge of KubeEdge compared to other solutions. It was seen there was no point to include other solutions, although it would have made sense from case validation perspective.

As hinted in introduction Chapter 1, the goals of this thesis were somewhat different in its original form. In original work, it was explicitly stated that the research goal was to leverage Kubernetes only for edge device monitoring purposes. In other words, the original study did not consider Kubernetes container workload support as a requirement

for lightweight Kubelet implementation. For this simplified purpose, Virtual Kubelet may be the most sensible project to use. All that would have been necessary to implement is Virtual Kubelet provider with empty functions as its extension points. However, as the original study proceeded, this simplification was eventually dropped, as there are other dedicated solutions for edge node monitoring purposes.

5.4 Study evalution and future work

The case study implemented in thesis did not include any experimental or practical work, in which lightweight Kubernetes solutions in edge contexts would have been measured and evaluated. Such evaluations are invaluable for several reasons. First, they give important feedback on strengths and weaknesses of the solutions that cannot be otherwise found from pure literature reviews. Second, practical evaluations challenge any vague presumptions, disproving them quickly, thus saving time and effort. Third, if the practical work is ambitious enough, it forces to understand the topic in greater detail than one might otherwise do. Fourth, it gives experience and new ideas in ways that cannot be predicted.

Therefore, the next research steps from here are to design, implement and evaluate practical edge-native case setup for the most promising looking solution canditates that were found in this thesis for RQ4. As the thesis background relates to cable access business, the natural choice for the edge case is to use an existing cable access device, such as RPD or RMD. The first step for the test case would be the simple integration with Kubernetes platform with basic support for workload scheduling, along with any supporting MEC-like features like telemetry and basic services. For this, Virtual Kubelet is good starting point, as it gives flexibility to integrate standard containers on low-resource devices. As WebAssembly seems like it could be the future of the edge, it also makes sense to include support for that runtime as part of the implementation right from the start.

Next, for more ambitious case setup that would also include support for IoT devices,

KubeEdge project seems like the most promising option. Similar to Virtual Kubelet, integration with WebAssembly runtime should be considered. The benefit of KubeEdge is that it already contains many features that would be good match in the edge-native context. These include autonomous operation when connections to core cloud are unreliable, and builtin support for MQTT. While the latter function may not seem like necessary in current cable access landscape, the prospects of MEC-like functionality close to subscribers, even in lightweight form, can be profitable feature to have. Finally, KubeEdge being CNCF incubating project gives it sense of credibility that other similar existing projects may not yet have.

6 Conclusion

This thesis presented a study on cloud-native transition to edge computing, where cable access networks work as its driving background. While seemingly separate topics, this work shows how the topics intersect in their future trends. Cloud-native and cable access topics were explored through informal literature review to answer three research questions RQ1, RQ2 and RQ3. Fourth research question RQ4 was formulated based on the results of the first three questions. Following sections summarize the results of this work.

6.1 Cloud-native, cable access, and future trends

For finding an answer to RQ1, Chapter 2 explored the cloud-native landscape through informal literature review, including historical perspective. As cloud-native model has matured in core clouds, it now takes direction towards edge computing. Several challenges can be seen on this new path, which need to be solved before edge-native model can be realized. First, containers are lightweight deployment units, but they are too resource intensive and insecure for many of the alluring edge use cases. Second, container orchestration as implemented by Kubernetes standard need to be adopted in the edge. However, Kubernetes is a resource heavy platform that assumes homogenous and predictable infrastructures. The edge environment is nothing like this. Therefore, for Kubernetes adoption in the edge computing, more lightweight versions of it are needed.

The cable access networks was studied in Chapter 3 to give an answer for RQ2. Against the backdrop of edge-native transition discussed above, cable networks are a type

of edge networks. In recent years, their architectures have seen evolutionary transition to distributed form. The same kind of development has been occurring for other access networks types. However, the next major trend in cable access can be seen to occur in form of technology and service convergence, in which different access network technologies will be harmonized and unified. In parallel to this convergence trend, cable access industry has another goal in virtualizing its existing platforms. This goal is expected to encompass adopting cloud-native principles. Therefore, as an answer for RQ3, it can be seen that convergence of cable access networks intertwines with more general trend that is happening in edge computing. The future of cable access is in edge-native model.

6.2 Case study results

Chapter 4 presented a case study that aimed to find answer to fourth research question that was formulated based on the results of the first three questions:

RQ4. What are the current options to leverage Kubernetes in context of edge-native cable access convergence?

The case study was conducted as a literature review on current landscape of Kubernetes compatible tools and runtimes targeting the edge. The results of RQ4 were used to analyse their prospects in edge-native cable access convergence. The original experimental work that was planned to complement the case study was moved to any future work.

As one important result for edge container-like workloads, the standalone WebAssembly, or Wasm, seems like the most promising alternative runtime environment that is also compatible with OCI runtimes and Kubernetes CRI runtimes. Compared to Linux containers, binaries of Wasm are more lightweight, start up faster, and are executed in secure sandboxed environments. Wasm binaries are also more portable, as they depend on underlying host environment only through the WASI interface exposed by their runtime component. While Wasm is relatively new technology, there seems to be great potential

to use Wasm binaries as the edge-native deployment units.

Next, it is found that lightweight alternative Kubernetes solutions can be categorized in three different groups. In the first group belong Kubernetes compatible distributions that are optimized in resource footprint, and targeted for resource constrained edge devices. Of the two distributions examined in this work, K3s seems more promising.

In the second group, KubeEdge takes another approach. It has specialized edge node component that is fully autonomous, while keeping overall platform controllers in central Kubernetes cluster. KubeEdge has builtin functions to manage fleet of edge devices through MQTT protocol. Compared to solutions of first group, KubeEdge seems like the more complete option for adopting edge-native model.

Solutions of third group aim to replace native Kubelet agent with custom implementation. This is the most direct and flexible method to integrate edge nodes in Kubernetes platform. It is also the most laborous method, as depending on which Kubelet base implementation is chosen for extension, much of existing workload management and routing functions must be reimplemented. Among the solutions examined for this group, Virtual Kubelet is the most suitable starting point, as its provider concept is natural extension point. Furthermore, there are reference projects that have successfully adapted it.

References

[1] A. Bhardwaj and C. R. Krishna, "Virtualization in cloud computing: Moving from hypervisor to containerization—a survey", *Arabian Journal for Science and Engineering*, vol. 46, no. 9, pp. 8585–8601, Sep. 1, 2021, ISSN: 2191-4281. DOI: 10.1007/s13369-021-05553-3. [Online]. Available: https://doi.org/10.1007/s13369-021-05553-3 (visited on 06/14/2023).

[2] P.-J. Maenhaut, B. Volckaert, V. Ongenae, and F. De Turck, "Resource management in a containerized cloud: Status and challenges", *Journal of Network and Systems Management*, vol. 28, no. 2, pp. 197–246, Apr. 1, 2020, ISSN: 1573-7705. DOI: 10.1007/s10922-019-09504-0. [Online]. Available: https://doi.org/10.1007/s10922-019-09504-0 (visited on 06/14/2023).

[3] "Cloud adoption statistics for 2023", WebTribunal. (May 23, 2023), [Online]. Available: https://webtribunal.net/blog/cloud-adoption-statistics/ (visited on 06/14/2023).

[4] "Amazon EC2", Amazon Web Services, Inc. (May 23, 2023), [Online]. Available: https://aws.amazon.com/ec2/ (visited on 06/14/2023).

[5] "Compute engine: Virtual machines (VMs)", Google Cloud. (May 23, 2023), [Online]. Available: https://cloud.google.com/compute (visited on 06/14/2023).

[6] "Azure infrastructure as a service (IaaS)". (2023), [Online]. Available: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/azure-iaas/ (visited on 06/14/2023).

[7] S. K. Tesfatsion, C. Klein, and J. Tordsson, "Virtualization techniques compared: Performance, resource, and power usage overheads in clouds", in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, Berlin, Germany: Association for Computing Machinery, Mar. 30, 2018, pp. 145–156, ISBN: 978-1-4503-5095-2. DOI: 10.1145/3184407.3184414. [Online]. Available: http://doi.org/10.1145/3184407.3184414 (visited on 06/14/2023).

[8] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review", *IEEE Transactions on Cloud Computing*, vol. 7, no. 3, pp. 677–692, Jul. 2019, ISSN: 2168-7161. DOI: 10.1109/TCC.2017.2702586.

[9] "Kubernetes". (2023), [Online]. Available: https://kubernetes.io/ (visited on 06/14/2023).

[10] "CNCF annual survey 2022", Cloud Native Computing Foundation. (Jan. 31, 2023), [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2022/ (visited on 06/14/2023).

[11] "7 major trends for cloud native in 2020: Kubernetes", Alibaba Cloud Community. (Mar. 3, 2020), [Online]. Available: https://www.alibabacloud.com/blog/7-major-trends-for-cloud-native-in-2020-kubernetes_595938 (visited on 06/14/2023).

[12] "Cloud native survey 2019", Cloud Native Computing Foundation. (Dec. 2, 2019), [Online]. Available: https://www.cncf.io/reports/cloud-native-survey-2019/ (visited on 06/14/2023).

[13] P. Kayal, "Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope : Invited paper", in *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, Jun. 2020, pp. 1–6. DOI: 10.1109/WF-IoT48130.2020.9221340.

[14] R. Vaño, I. Lacalle, P. Sowiński, R. S-Julián, and C. E. Palau, "Cloud-native workload orchestration at the edge: A deployment review and future directions", *Sensors*, vol. 23, no. 4, p. 2215, Jan. 2023, ISSN: 1424-8220. DOI: 10.3390/s23042215. [Online]. Available: https://www.mdpi.com/1424-8220/23/4/2215 (visited on 06/14/2023).

[15] "WebAssembly". (2023), [Online]. Available: https://webassembly.org/ (visited on 06/14/2023).

[16] Z. Jia and L. A. Campos, *Coherent Optics for Access Networks*. CRC Press, Oct. 28, 2019, 123 pp., ISBN: 978-1-00-073650-2.

[17] J. Andréoli-Fang and J. T. Chapman, "Cable and mobile convergence", presented at the SCTE Cable-Tec Expo, 2020. [Online]. Available: https://www.nctatechnicalpapers.com/Paper/2020/2020-cable-and-mobile-convergence (visited on 06/14/2023).

[18] A. Vladyka and A. Matatyaou, "Virtualization and edge compute evolution in cable - NCTA technical papers", presented at the SCTE Cable-Tec Expo, 2020. [Online]. Available: https://www.nctatechnicalpapers.com/Paper/2020/2020-virtualization-and-edge-compute-evolution-in-cable (visited on 06/14/2023).

[19] C. N. Höfer and G. Karagiannis, "Cloud computing services: Taxonomy and comparison", *Journal of Internet Services and Applications*, vol. 2, no. 2, pp. 81–94, Sep. 1, 2011, ISSN: 1869-0238. DOI: 10.1007/s13174-011-0027-x. [On-

line]. Available: https://doi.org/10.1007/s13174-011-0027-x (visited on 06/14/2023).

[20] A. J. Younge, R. Henschel, J. T. Brown, G. von Laszewski, J. Qiu, and G. C. Fox, "Analysis of virtualization technologies for high performance computing environments", in *2011 IEEE 4th International Conference on Cloud Computing*, Jul. 2011, pp. 9–16. DOI: 10.1109/CLOUD.2011.29.

[21] B. B. Gupta and D. P. Agrawal, *Handbook of Research on Cloud Computing and Big Data Applications in IoT*. IGI Global, 2019, ISBN: 978-1-5225-8407-0.

[22] A. Slominski, V. Muthusamy, and V. Ishakian, "Future of computing is boring (and that is exciting!) or how to get to computing nirvana in 20 years or less", *arXiv:1906.10398 [cs]*, Jun. 25, 2019. arXiv: 1906.10398. [Online]. Available: http://arxiv.org/abs/1906.10398 (visited on 06/14/2023).

[23] R. McHaney, *Cloud technologies: an overview of cloud computing technologies for managers*, First edition. Hoboken, NJ: Wiley, 2021, ISBN: 978-1-119-76952-1.

[24] Y. Mansouri and M. A. Babar, "A review of edge computing: Features and resource virtualization", *Journal of Parallel and Distributed Computing*, vol. 150, pp. 155–183, Apr. 1, 2021, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2020.12.015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731520304317 (visited on 06/14/2023).

[25] P. Mell and T. Grance, "The NIST definition of cloud computing", National Institute of Standards and Technology, NIST Special Publication (SP) 800-145, Sep. 28, 2011. DOI: https://doi.org/10.6028/NIST.SP.800-145. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-145/final (visited on 06/14/2023).

[26] N. Kratzke, "A brief history of cloud application architectures", *Applied Sciences*, vol. 8, no. 8, p. 1368, Aug. 2018. DOI: 10.3390/app8081368. [Online]. Available: https://www.mdpi.com/2076-3417/8/8/1368 (visited on 06/14/2023).

[27] *Cloud computing*, in *Wikipedia*, May 26, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Cloud_computing&oldid=1157135946 (visited on 06/14/2023).

[28] "Types of cloud computing - SaaS vs PaaS vs IaaS - AWS", Amazon Web Services, Inc. (2023), [Online]. Available: https://aws.amazon.com/types-of-cloud-computing/ (visited on 06/14/2023).

[29] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study", in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16, New York, NY, USA: Association for Computing Machinery, Mar. 28, 2016, pp. 1–13, ISBN: 978-1-4503-4300-8. DOI: 10.1145/2988336.2988337. [Online]. Available: http://doi.org/10.1145/2988336.2988337 (visited on 06/14/2023).

[30] N. Dragoni *et al.*, "Microservices: Yesterday, today, and tomorrow", in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds., Cham: Springer International Publishing, 2017, pp. 195–216, ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12 (visited on 06/14/2023).

[31] Datadog. "The state of serverless", The State of Serverless. (2022), [Online]. Available: https://www.datadoghq.com/state-of-serverless/ (visited on 06/14/2023).

[32] M. Villamizar *et al.*, "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS lambda architectures", *Service Ori-*

ented Computing and Applications, vol. 11, no. 2, pp. 233–247, Jun. 1, 2017, ISSN: 1863-2394. DOI: 10.1007/s11761-017-0208-y. [Online]. Available: https://doi.org/10.1007/s11761-017-0208-y (visited on 06/14/2023).

[33] N. Kratzke and P.-C. Quint, “Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study”, *Journal of Systems and Software*, vol. 126, pp. 1–16, Apr. 1, 2017, ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.01.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121217300018 (visited on 06/14/2023).

[34] *Cloud native computing foundation policy repo*, May 4, 2023. [Online]. Available: https://github.com/cncf/foundation/blob/d3c181735ae0478ca7b4fa charter.md (visited on 06/14/2023).

[35] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, “Microservices: The journey so far and challenges ahead”, *IEEE Software*, vol. 35, no. 3, pp. 24–35, May 2018, ISSN: 1937-4194. DOI: 10.1109/MS.2018.2141039.

[36] C. Puliafito, E. Mingozzi, F. Longo, A. Puliafito, and O. Rana, “Fog computing for the internet of things: A survey”, *ACM Transactions on Internet Technology*, vol. 19, no. 2, 18:1–18:41, Apr. 2, 2019, ISSN: 1533-5399. DOI: 10.1145/3301443. [Online]. Available: https://doi.org/10.1145/3301443 (visited on 06/14/2023).

[37] A. Yousefpour *et al.*, “All one needs to know about fog computing and related edge computing paradigms: A complete survey”, *Journal of Systems Architecture*, vol. 98, pp. 289–330, Sep. 1, 2019, ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2019.02.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762118306349 (visited on 06/14/2023).

[38] M. Satyanarayanan, “Pervasive computing: Vision and challenges”, *IEEE Personal Communications*, vol. 8, no. 4, pp. 10–17, Aug. 2001, ISSN: 1558-0652. DOI: 10.1109/98.943998.

[39] “Multi-access edge computing (MEC); framework and reference architecture”, ETSI, 2022. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/03.01.01_60/gs_MEC003v030101p.pdf (visited on 06/14/2023).

[40] B. Liang, M. A. Gregory, and S. Li, “Multi-access edge computing fundamentals, services, enablers and challenges: A complete survey”, *Journal of Network and Computer Applications*, vol. 199, p. 103 308, Mar. 1, 2022, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2021.103308. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804521002976 (visited on 06/14/2023).

[41] M. S. Bonfim, K. L. Dias, and S. F. L. Fernandes, “Integrated NFV/SDN architectures: A systematic literature review”, *ACM Computing Surveys*, vol. 51, no. 6, 114:1–114:39, Feb. 4, 2019, ISSN: 0360-0300. DOI: 10.1145/3172866. [Online]. Available: https://doi.org/10.1145/3172866 (visited on 06/14/2023).

[42] A. Pereira Ferreira and R. Sinnott, “A performance evaluation of containers running on managed kubernetes services”, in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec. 2019, pp. 199–208. DOI: 10.1109/CloudCom.2019.00038.

[43] M. Chae, H. Lee, and K. Lee, “A performance comparison of linux containers and virtual machines using docker and KVM”, *Cluster Computing*, vol. 22, no. 1, pp. 1765–1775, Jan. 1, 2019, ISSN: 1573-7543. DOI: 10.1007/s10586-017-

1511-2. [Online]. Available: https://doi.org/10.1007/s10586-017-1511-2 (visited on 06/14/2023).

[44] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures", *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974, ISSN: 0001-0782. DOI: 10.1145/361011.361073. [Online]. Available: http://doi.org/10.1145/361011.361073 (visited on 06/14/2023).

[45] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization", *ACM SIGPLAN Notices*, vol. 41, no. 11, pp. 2–13, 2006, ISSN: 0362-1340. DOI: 10.1145/1168918.1168860. [Online]. Available: http://doi.org/10.1145/1168918.1168860 (visited on 06/14/2023).

[46] I. Mavridis and H. Karatza, "Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing", *Future Generation Computer Systems*, vol. 94, pp. 674–696, May 1, 2019, ISSN: 0167-739X. DOI: 10.1016/j.future.2018.12.035. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18305764 (visited on 06/14/2023).

[47] R. Di Pietro and F. Lombardi, "Virtualization technologies and cloud security: Advantages, issues, and perspectives", in *From Database to Cyber Security: Essays Dedicated to Sushil Jajodia on the Occasion of His 70th Birthday*, ser. Lecture Notes in Computer Science, P. Samarati, I. Ray, and I. Ray, Eds., Cham: Springer International Publishing, 2018, pp. 166–185, ISBN: 978-3-030-04834-1. DOI: 10.1007/978-3-030-04834-1_9. [Online]. Available: https://doi.org/10.1007/978-3-030-04834-1_9 (visited on 06/14/2023).

[48] WMware, *Understanding full virtualization, paravirtualization, and hardware assist*, Mar. 11, 2008. [Online]. Available: https://www.vmware.com/

techpapers/2007/understanding-full-virtualization-paravirtuali
1008.html (visited on 06/14/2023).

[49] D. Ernst, D. Bermbach, and S. Tai, "Understanding the container ecosystem: A taxonomy of building blocks for container lifecycle and cluster management", p. 6, 2016.

[50] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security", *Concurrency and Computation: Practice and Experience*, vol. 32, no. 17, e5668, 2020, ISSN: 1532-0634. DOI: 10.1002/cpe.5668. [Online]. Available: http://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5668 (visited on 06/14/2023).

[51] "Linux containers - LXC - introduction". (2023), [Online]. Available: https://linuxcontainers.org/lxc/introduction/ (visited on 06/14/2023).

[52] "Docker: Accelerated, containerized application development". (May 10, 2022), [Online]. Available: https://www.docker.com/ (visited on 06/14/2023).

[53] "A practical introduction to container terminology", Red Hat Developer. (Feb. 22, 2018), [Online]. Available: https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction (visited on 06/14/2023).

[54] "Open container initiative". (2023), [Online]. Available: https://opencontainers.org/ (visited on 06/14/2023).

[55] 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, pp. 273–281, ISBN: 978-989-758-424-4. DOI: 10.5220/0009340402730281. [Online]. Available: http://www.

scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0009340402730281 (visited on 06/14/2023).

[56] *Runc GitHub*, 2023. [Online]. Available: https://github.com/opencontainers/runc (visited on 06/14/2023).

[57] T. Donohue. “The differences between docker, containerd, CRI-o and runc”, Tutorial Works. (Jan. 2, 2023), [Online]. Available: https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/ (visited on 06/14/2023).

[58] “Open source container-based virtualization for linux.”, OpenVz. (2023), [Online]. Available: https://openvz.org/ (visited on 06/14/2023).

[59] C. Pahl, “Containerization and the PaaS cloud”, *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, May 2015, ISSN: 2325-6095. DOI: 10.1109/MCC.2015.51.

[60] “CNCF annual survey 2021”, Cloud Native Computing Foundation. (Feb. 10, 2022), [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2021/ (visited on 06/14/2023).

[61] “Kubernetes: Concepts”, Kubernetes. (2023), [Online]. Available: https://kubernetes.io/docs/concepts/ (visited on 06/14/2023).

[62] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade”, *Queue*, vol. 14, no. 1, pp. 70–93, Jan. 1, 2016, ISSN: 1542-7730. DOI: 10.1145/2898442.2898444. [Online]. Available: https://dl.acm.org/doi/10.1145/2898442.2898444 (visited on 06/14/2023).

[63] “Cloud native computing foundation”. (2023), [Online]. Available: https://www.cncf.io/ (visited on 06/14/2023).

[64] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications", in *2019 IEEE Conference on Network Softwarization (NetSoft)*, Jun. 2019, pp. 351–359. DOI: 10.1109/NETSOFT.2019.8806671.

[65] "The cloud native wiki", Aqua. (2023), [Online]. Available: https://www.aquasec.com/cloud-native-academy/ (visited on 06/14/2023).

[66] "Cri-o". (2023), [Online]. Available: https://cri-o.io/ (visited on 06/14/2023).

[67] "Etcd", etcd. (2023), [Online]. Available: https://etcd.io/ (visited on 06/14/2023).

[68] "Introduction - the cluster API book". (2023), [Online]. Available: https://cluster-api.sigs.k8s.io/ (visited on 06/14/2023).

[69] N. Velayudhan. "A visual guide to kubernetes networking fundamentals | opensource.com". (Jun. 1, 2022), [Online]. Available: https://opensource.com/article/22/6/kubernetes-networking-fundamentals (visited on 06/14/2023).

[70] "Introduction - kubernetes gateway API". (2023), [Online]. Available: https://gateway-api.sigs.k8s.io/ (visited on 06/14/2023).

[71] L. Mirsky. "Why do kubernetes applications need a package manager?" (2023), [Online]. Available: https://www.opsfleet.com/articles/why-do-kubernetes-applications-need-a-package-manager (visited on 06/14/2023).

[72] "Helm". (2023), [Online]. Available: https://helm.sh/ (visited on 06/14/2023).

[73] "Go template package - text/template". (2023), [Online]. Available: https://pkg.go.dev/text/template (visited on 06/14/2023).

[74] S. Gorshe, A. Raghavan, T. Starr, and S. Galli, *Hybrid Fiber Access Technologies*, 1st ed., in collab. with S. Gorshe, T. Starr, A. R. Raghavan, and S. Galli. Chichester, United Kingdom: Wiley, John Wiley & Sons, Ltd, 2014, ISBN: 978-0-470-74180-1. DOI: 10.1002/9781118878774.ch06.

[75] M. Tornatore, G.-K. Chang, and G. Ellinas, *Fiber-wireless convergence in next-generation communication networks: systems, architectures, and management* (Optical networks). Cham, Switzerland: Springer, 2017, ISBN: 978-3-319-42822-2.

[76] "DOCSIS 3.1 physical layer specification", CableLabs, 2013. [Online]. Available: https://www.cablelabs.com/specifications/CM-SP-PHYv3.1 (visited on 06/14/2023).

[77] "DOCSIS 3.1 MAC and upper layer protocols interface specification", CableLabs, 2013. [Online]. Available: https://www.cablelabs.com/specifications/CM-SP-MULPIv3.1 (visited on 06/14/2023).

[78] "DOCSIS 4.0 physical layer specification", CableLabs, 2019. [Online]. Available: https://www.cablelabs.com/specifications/CM-SP-PHYv4.0 (visited on 06/14/2023).

[79] "DOCSIS 4.0 MAC and upper layer protocols interface specification", CableLabs, 2019. [Online]. Available: https://www.cablelabs.com/specifications/CM-SP-MULPIv4.0 (visited on 06/14/2023).

[80] "The digital HFC — a path to 10g". (Feb. 19, 2020), [Online]. Available: https://broadbandlibrary.com/the-digital-hfc-a-path-to-10g/ (visited on 06/14/2023).

[81] "OECD broadband statistics update - OECD". (Feb. 23, 2023), [Online]. Available: https://www.oecd.org/sti/broadband/broadband-statistics-update.htm (visited on 06/14/2023).

[82] M. J. Emmendorfer, "Cable operator's access architecture from aggregation to disaggregation and distributed", in *2019 IEEE Photonics Society Summer Topical Meeting Series (SUM)*, Jul. 2019, pp. 1–1. DOI: 10.1109/PHOSST.2019.8795043.

[83] "Flexible MAC architecture system specification", CableLabs, 2020. [Online]. Available: https://www.cablelabs.com/specifications/CM-SP-FMA-SYS (visited on 06/14/2023).

[84] W. Coomans, H. Chow, and J. Maes, "Introducing full duplex in hybrid fiber coaxial networks", *IEEE Communications Standards Magazine*, vol. 2, no. 1, pp. 74–79, Mar. 2018, ISSN: 2471-2833. DOI: 10.1109/MCOMSTD.2018.1700011.

[85] J. T. Chapman, H. Jin, T. Hewavithana, and R. Hillermeier, "Blueprint for 3 GHz, 25 gbps DOCSIS - NCTA technical papers", presented at the SCTE Cable-Tec Expo, 2019. [Online]. Available: https://www.nctatechnicalpapers.com/Paper/2019/2019-blueprint-for-3-ghz-25-gbps-docsis (visited on 06/14/2023).

[86] H. Jin and J. Chapman, "Echo cancellation techniques for supporting full duplex DOCSIS", presented at the SCTE Cable-Tec Expo, 2017, p. 24. [Online]. Available: https://www.nctatechnicalpapers.com/Paper/2017/2017-echo-cancellation-techniques-for-supporting-full-duplex-docsis (visited on 06/14/2023).

[87] B. Berscheid and C. Howlett, "Full duplex DOCSIS: Opportunities and challenges", *IEEE Communications Magazine*, vol. 57, no. 8, pp. 28–33, Aug. 2019, ISSN: 1558-1896. DOI: 10.1109/MCOM.2019.1800851.

[88] N. M. Gowdal, X. Si, and A. Sabharwal, "Full-duplex DOCSIS: A modem architecture for wideband (>1ghz) self-interference cancellation for cable modem termination systems (CMTS)", in *2018 52nd Asilomar Conference on Signals,*

Systems, and Computers, Oct. 2018, pp. 2202–2206. DOI: 10.1109/ACSSC.2018.8645538.

[89] M.-S. Baek, J.-H. Song, O.-H. Kwon, and J.-Y. Jung, "Self-interference cancellation in time-domain for DOCSIS 3.1 uplink system with full duplex", *IEEE Transactions on Broadcasting*, vol. 65, no. 4, pp. 695–701, Dec. 2019, ISSN: 1557-9611. DOI: 10.1109/TBC.2019.2897738.

[90] N. A. J. BAUMGARTNER, S. Editor, and L. Reading 10/1/2021. "Full duplex DOCSIS amplifier chatter heats up", Light Reading. (Oct. 1, 2021), [Online]. Available: https://www.lightreading.com/cable-tech/full-duplex-docsis-amplifier-chatter-heats-up/d/d-id/772466 (visited on 06/14/2023).

[91] P. Sowinski, "The impact of remote PHY on cable service convergence", presented at the SCTE Cable-Tec Expo, 2016. [Online]. Available: https://www.nctatechnicalpapers.com/Paper/2016/2016-the-impact-of-remote-phy-on-cable-service-convergence (visited on 06/14/2023).

[92] "Distributed access architecture (DAA). the challenges & benefits of DAA". (Sep. 10, 2019), [Online]. Available: https://www.viavisolutions.com/en-us/distributed-access-architecture (visited on 06/14/2023).

[93] "Remote PHY specification", CableLabs, 2015. [Online]. Available: https://www.cablelabs.com/specifications/CM-SP-R-PHY (visited on 06/14/2023).

[94] "DOCSIS modular headend architecture", CableLabs, 2008. [Online]. Available: https://www.cablelabs.com/specifications/modular-headend-architecture-technical-report (visited on 06/14/2023).

[95] T. Liu and J. Chapman, “R-PHY with remote upstream scheduler”, presented at the SCTE Cable-Tec Expo, 2019. [Online]. Available: https://www.nctatechnicalpapers.com/Paper/2019/2019-r-phy-with-remote-upstream-scheduler (visited on 06/14/2023).

[96] J. Rodriguez and J. Jansen, “Fixed-wireless convergence on a multi-access edge - NCTA technical papers”, presented at the SCTE Cable-Tec Expo, 2021. [Online]. Available: https://www.nctatechnicalpapers.com/Paper/2021/2021-fixed-wireless-convergence-on-a-multi-access-edge (visited on 06/14/2023).

[97] J. Chapman and T. Liu, “Unleash the power of cloud computing for CMTS - NCTA technical papers”, presented at the SCTE Cable-Tec Expo, 2021. [Online]. Available: https://www.nctatechnicalpapers.com/Paper/2021/2021-unleash-the-power-of-cloud-computing-for-cmts (visited on 06/14/2023).

[98] “Connexus”, CableLabs. (2023), [Online]. Available: https://www.cablelabs.com/connexus (visited on 06/14/2023).

[99] “5g wireless wireline converged core architecture technical report”, CableLabs, 2019. [Online]. Available: https://www.cablelabs.com/specifications/WR-TR-5WWC-ARCH (visited on 06/14/2023).

[100] A. Vladyka, A. Matatyaou, and H. Abramson, “Exploring multi-access edge compute in converging access networks - NCTA technical papers”, presented at the SCTE Cable-Tec Expo, 2021. [Online]. Available: https://www.nctatechnicalpapers.com/Paper/2021/2021-exploring-multi-access-edge-compute-in-converging-access-networks (visited on 06/14/2023).

[101] B. Hallahan, “Why, how, and where to converge fixed and mobile networks - NCTA technical papers”, presented at the SCTE Cable-Tec Expo, 2022. [On-

line]. Available: https://www.nctatechnicalpapers.com/Paper/2022/FTF22_CONV02_Hallahan_3736 (visited on 06/14/2023).

[102] D. K. Bainbridge, S. Barbarie, D. Fedorov, M. Naveda, and R. Ranganathan, "Implementing multi-layer infrastructure management for multi-access edge computing services using kubernetes - NCTA technical papers", presented at the SCTE Cable-Tec Expo, 2021. [Online]. Available: https://www.nctatechnicalpapers.com/Paper/2021/2021-implementing-multi-layer-infrastructure-management (visited on 06/14/2023).

[103] T. Goethals, F. DeTurck, and B. Volckaert, "Extending kubernetes clusters to low-resource edge devices using virtual kubelets", *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020, ISSN: 2168-7161. DOI: 10.1109/TCC.2020.3033807.

[104] V. Kjorveziroski and S. Filiposka, "Kubernetes distributions for the edge: Serverless performance evaluation", *The Journal of Supercomputing*, vol. 78, no. 11, pp. 13 728–13 755, Jul. 1, 2022, ISSN: 1573-0484. DOI: 10.1007/s11227-022-04430-6. [Online]. Available: https://doi.org/10.1007/s11227-022-04430-6 (visited on 06/14/2023).

[105] N. Marco, D. Fedorov, and R. Ranganathan, "Delivering cloud-native operations with edge compute enabled DAA: Implementing a kubernetes distributed edge - NCTA technical papers", presented at the SCTE Cable-Tec Expo, 2020. [Online]. Available: https://www.nctatechnicalpapers.com/Paper/2020/2020-delivering-cloud-native-operations (visited on 06/14/2023).

[106] A. Jeffery, H. Howard, and R. Mortier, "Rearchitecting kubernetes for the edge", in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, ser. EdgeSys '21, New York, NY, USA: Association for Computing

Machinery, Apr. 26, 2021, pp. 7–12, ISBN: 978-1-4503-8291-5. DOI: 10.1145/3434770.3459730. [Online]. Available: https://doi.org/10.1145/3434770.3459730 (visited on 06/14/2023).

[107] A. Abouaomar, S. Cherkaoui, Z. Mlika, and A. Kobbane, "Resource provisioning in edge computing for latency-sensitive applications", *IEEE Internet of Things Journal*, vol. 8, no. 14, pp. 11 088–11 099, Jul. 2021, ISSN: 2327-4662. DOI: 10.1109/JIOT.2021.3052082.

[108] "K3s". (2023), [Online]. Available: https://k3s-io.github.io/ (visited on 06/14/2023).

[109] "MicroK8s", microk8s.io. (2023), [Online]. Available: http://microk8s.io (visited on 06/14/2023).

[110] KubeEdge. "KubeEdge", KubeEdge. (2023), [Online]. Available: https://kubeedge.io/en/ (visited on 06/14/2023).

[111] "Virtual kubelet". (2023), [Online]. Available: https://virtual-kubelet.io/ (visited on 06/14/2023).

[112] "Rancher", Rancher Labs. (2023), [Online]. Available: http://www.rancher.com (visited on 06/14/2023).

[113] "The kubernetes operating system". (2023), [Online]. Available: https://k3os.io/ (visited on 06/14/2023).

[114] A. Lai. "KubeEdge and its role in multi-access edge computing", The New Stack. (Jun. 19, 2020), [Online]. Available: https://thenewstack.io/kubeedge-and-its-role-in-multi-access-edge-computing/ (visited on 06/14/2023).

[115] *Virtual kubelet GitHub*, 2023. [Online]. Available: https://github.com/virtual-kubelet/virtual-kubelet (visited on 06/14/2023).

[116] *KubeMark GitHub*, 2023. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scalability/kubemark-guide.md (visited on 06/14/2023).

[117] "Kind". (2023), [Online]. Available: https://kind.sigs.k8s.io/ (visited on 06/14/2023).

[118] "KWOK". (2023), [Online]. Available: https://kwok.sigs.k8s.io/ (visited on 06/14/2023).

[119] S. Telenyk, O. Sopov, E. Zharikov, and G. Nowakowski, "A comparison of kubernetes and kubernetes-compatible platforms", in *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 1, Sep. 2021, pp. 313–317. DOI: 10.1109/IDAACS53288.2021.9660392.

[120] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "WebAssembly as a common layer for the cloud-edge continuum", in *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, ser. FRAME '22, New York, NY, USA: Association for Computing Machinery, Jun. 27, 2022, pp. 3–8, ISBN: 978-1-4503-9310-2. DOI: 10.1145/3526059.3533618. [Online]. Available: https://dl.acm.org/doi/10.1145/3526059.3533618 (visited on 06/14/2023).

[121] "Kata containers - open source container runtime software". (2023), [Online]. Available: https://katacontainers.io/ (visited on 06/14/2023).

[122] X. Wang, J. Du, and H. Liu, "Performance and isolation analysis of RunC, gVisor and kata containers runtimes", *Cluster Computing*, vol. 25, no. 2, pp. 1497–1513, Apr. 1, 2022, ISSN: 1573-7543. DOI: 10.1007/s10586-021-03517-8. [Online]. Available: https://doi.org/10.1007/s10586-021-03517-8 (visited on 06/14/2023).

[123] "gVisor". (2023), [Online]. Available: https://gvisor.dev/ (visited on 06/14/2023).

[124] Z. Li *et al.*, "{RunD}: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing", presented at the 2022 USENIX Annual Technical Conference (USENIX ATC 22), 2022, pp. 53–68, ISBN: 978-93-91332-92-1. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/li-zijun-rund (visited on 06/14/2023).

[125] "WASI". (2023), [Online]. Available: https://wasi.dev/ (visited on 06/14/2023).

[126] "Standardizing WASI: A system interface to run WebAssembly outside the web - mozilla hacks - the web developer blog", Mozilla Hacks - the Web developer blog. (Mar. 27, 2019), [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface (visited on 06/14/2023).

[127] *Wasmtime GitHub*, 2023. [Online]. Available: https://github.com/bytecodeallia wasmtime (visited on 06/14/2023).

[128] "Bytecode alliance", Bytecode Alliance. (2023), [Online]. Available: https://bytecodealliance.org/ (visited on 06/14/2023).

[129] *WebAssembly micro runtime GitHub*, Jun. 4, 2023. [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime (visited on 06/14/2023).

[130] *WasmEdge GitHub*, Jun. 5, 2023. [Online]. Available: https://github.com/WasmEdge/WasmEdge (visited on 06/14/2023).

[131] *Wasmerio/wasmer GitHub*, 2023. [Online]. Available: https://github.com/wasmerio/wasmer (visited on 06/14/2023).

[132] "Wazero". (2023), [Online]. Available: https://wazero.io/ (visited on 06/14/2023).

[133] "WebAssembly core specification". (Dec. 5, 2019), [Online]. Available: https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/ (visited on 06/14/2023).

[134] V. Kjorveziroski, S. Filiposka, and A. Mishev, "Evaluating WebAssembly for orchestrated deployment of serverless functions", in *2022 30th Telecommunications Forum (TELFOR)*, Nov. 2022, pp. 1–4. DOI: 10.1109/TELFOR56187.2022.9983733.

[135] *Containerd/runwasi GitHub*, Jun. 2, 2023. [Online]. Available: https://github.com/containerd/runwasi (visited on 06/14/2023).

[136] *Containers/crun GitHub*, 2023. [Online]. Available: https://github.com/containers/crun (visited on 06/14/2023).