User-plane containerization in cloud RAN

Toni Kangas

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 2.10.2017

Thesis supervisor:

Prof. Ville Viikari

Thesis advisor:

M.Sc. Tuomo Eskola

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Toni Kangas

Title: User-plane containerization in cloud RAN

Date: 2.10.2017 Language: English Number of pages: 6+59

Department of Electronics and Nanoengineering

Supervisor: Prof. Ville Viikari

Advisor: M.Sc. Tuomo Eskola

The amount of devices connected through mobile networks has been growing rapidly. This growth will create a demand for network capacity that cannot be met with traditional methods. This problem could be solved by implementing a cloud radio access network (RAN), a new concept, to adapt cloud computing technologies, such as software containers, from the software industry to RANs. This adaptation will also create a need to modify working practices in order to better comply with these new cloud computing technologies.

While cloud RAN has recently received much research attention, the actual software implementations have not been widely discussed in the literature. Therefore, this thesis evaluates the feasibility of using software containers in the user-plane applications of cloud RAN in terms of networking and inter-container communications (ICC). This is accomplished by identifying potential approaches for ICC and for container networking as well as measuring the performance of these approaches.

Two approaches are proposed for ICC and container networking. The approaches were evaluated in terms of throughput and latency. These approaches were found to be suitable for use in cloud RAN user-plane applications. However, since the measurements were performed in a simplified environment, implementing the approaches into a cloud RAN component will require further work.

Keywords: Cloud RAN, Container, DPDK, Microservice, Mobile networks, Virtualization

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Toni Kangas

Työn nimi: Käyttäjätason ohjelmistokontittaminen pilviradioliityntäverkossa

Päivämäärä: 2.10.2017 Kieli: Englanti Sivumäärä: 6+59

Elektroniikan ja nanotekniikan laitos

Työn valvoja: Prof. Ville Viikari

Työn ohjaaja: DI Tuomo Eskola

Mobiiliverkkoihin liitettävien laitteiden määrä kasvaa nopeasti. Tämä kasvu tulee luomaan verkon kapasiteetille kysynnän, johon ei kyetä vastaamaan perinteisin menetelmin. Tämä ongelma voitaineen ratkaista implementoimalla pilviradioliityntäverkko (*Cloud RAN*), uusi konsepti, joka sovittaa ohjelmistoalalla vakiintuneita pilvilaskentateknologioita käytettäväksi radioliityntäverkoissa (*radio access network, RAN*). Tämä sovitusprosessi luo tarpeen mukauttaa myös työskentelytavat yhteensopiviksi uusien pilvilaskentateknologioiden kanssa.

Vaikka pilviradioliityntäverkkoa on tutkittu aktiivisesti viime aikoina, käytännön ohjelmistotototeutukset eivät juuri ole olleet esillä kirjallisuudessa. Tämä diplomityö arvioi ohjelmistokonttien (*software containers*) soveltuvuutta käytettäväksi pilviradioliityntäverkon käyttäjätason (*user-plane*) applikaatioissa verkottamisen (*networking*) ja ohjelmistokonttien välisen kommunikoinnin (*inter-container communications, ICC*) suhteen. Tämä arviointi suoritetaan identifioimalla mahdollisia toteutuksia ohjelmistokonttien väliselle kommunikaatiolle ja ohjelmistokonttien verkottamiselle sekä mittaamalla näiden toteutuksien suorituskyky.

Tässä diplomityössä ehdotetaan tutkittavaksi kaksi toteutusta ohjelmistokonttien väliselle kommunikaatiolle ja ohjelmistokonttien verkottamiselle. Nämä toteutukset arvioitiin välityskyvyn (*throughput*) ja latenssin suhteen. Näiden toteutuksien todettiin olevan soveliaita käytettäväksi pilviradioliityntäverkon käyttäjätason applikaatioissa. Kuitenkin, koska mittaukset toteutettiin yksinkertaistetussa ympäristössä, vaatii toteutuksien implementointi pilviradioliityntäverkon komponenttiin lisätyötä.

Avainsanat: DPDK, Mikropalvelu, Mobiiliverkot, Ohjelmistokontit, Pilviradioliityntäverkko, Virtualisointi

Preface

This thesis was conducted at Nokia Networks as a part of a project investigating the possibilities of containerization, microservices, and DevOps for cloud RAN software. I would like to thank my manager, Markku Niiranen, for providing me this thesis opportunity and my colleagues for supporting me throughout the thesis writing process.

I would like to thank my thesis advisor, Tuomo Eskola, for guidance, support, and interesting discussions throughout the thesis process. I would also like to thank my thesis supervisor, Ville Viikari, for providing guidance and valuable comments during the writing process.

Finally, I would like to thank my family and friends for supporting me in my studies and life in general.

Espoo, 2.10.2017

Toni Kangas

Contents

Symbols and abbreviations

Symbols

Δt_{ij}	Clock difference from instance i to instance j.
τ	Latency
τ_{ij}	Measured latency from instance i to instance j.
`$`	Shell prompt for non-root user
`#`	Shell prompt for root user

Abbreviations

API	Application Programming Interface
BBU	Base Band Unit
C	Container (Only used as an abbreviation in figures)
cgroup	Control group
DPDK	Data Plane Development Kit
ePC	evolved Packet Core
E-UTRAN	Evolved Universal Terrestial Radio Access Network
IaaS	Infrastructure as a Service
ICC	Inter-Container Communication
IPC	Inter-Process Communication
LTE	Long Term Evolution
MN	Mobile Network
NF	Network Function
NFV	Network Function Virtualization
NIC	Network Interface Card
OS	Operating System
OSS	Open Source Software
PaaS	Platform as a Service
PMD	Poll Mode Driver
RAN	Radio Access Network
RRH	Remote Radio Head
SaaS	Software as a Service
SaaS	Software-defined networking
TCP	Transmission Control Protocol
Telco	Telecommunications
UDP	User Datagram Protocol
UE	User Equipment
UI	User Interface
VM	Virtual Machine
VMM	Virtual Machine Monitor
VNF	Virtual Network Function

1 Introduction

The amount of mobile devices connected to the Internet has been increasing rapidly. This is expected to cause huge growth in data transmitted over mobile networks (MN) [1]. Traditional methods for increasing network capacity, such as adding more radio access points or implementing multi-user Multiple Input Multiple Output (MIMO) techniques, have proved to be problematic due to their high costs and inter-cell interference in the network [2]. In order to overcome these problems, Cloud Radio Access Network (RAN), a new concept, adapts cloud computing technologies already widely used in the software industry. This adaptation also requires a change in the software development practices in order to efficiently deliver cloud compatible software.

Recently, DevOps and microservices have become trending topics in the software developer community due to increased process efficiency and support for cloud native scalability [3],[4]. DevOps is a term referring to both the mindset of bringing developers and operations personnel closer together in software projects as well as the toolset applied in this process. Some of the goals in the DevOps process include delivering as well as deploying software more often and in smaller pieces than was possible using the old-fashioned waterfall model [5]. One of the enablers to achieve this is the usage of microservices. A microservice is an independent piece of software that focuses on completing a single task and is deployable and testable by itself [6]. This independency from other services is the key here to allow DevOps teams to test, deliver, deploy, and monitor their code more often.

One solution to provide greater independence for microservices is to use software containers, such as Docker [7],[8]. Software containers provide an environment that is isolated from both the host system and other containers [9]. Containers are light-weight, since they use the kernel of the host system and are packed to contain only the dependencies of the delivered software: software delivered in a container is ready to be deployed or run without any installations.

Virtual machines (VM) can be used to virtualize hardware if heavier virtualization is needed due to, for example, the need for different guest operating system (OS). Compared to containers, VM needs more resources, because of the included OS, but also provides better isolation from the host and other VMs [9]. The main difference between VMs and software containers is the level of virtualization: VMs virtualize the hardware whereas software containers virtualize the OS.

A modern way of obtaining the computing resources to run a software container or a VM is to rent the needed resources from a public or a private cloud provider [10]. This allows to avoid buying, setting up, hosting, and maintaining in-house physical servers. This management of in-house computing resources has generated a huge cost in traditional software business. Employing cloud computing technologies enables scaling the computing resources up and down with the demand, thus avoiding the need to buy computer resources to match peak demand and have them idling most of the time. However, this comes at a cost of not being able to physically access the servers, which could introduce problems in some applications.

Virtualization and cloud computing are currently also trending topics in telecom-

munications research [11]–[13]. Traditionally, there has been strong dependency between software and dedicated hardware in telecommunications systems causing additional challenges in keeping up with the rest of the software industry [2],[14]. Although previously meeting the high performance requirements of the telecommunications industry has not been possible without dedicated hardware and embedded software, recent developments in computing and virtualization technologies have enabled emergence of the cloud computing in the telecommunications industry [14],[15]. Especially cloud RAN has recently received much research attention [2].

However, the telecommunications industry is experiencing challenges in developing telecommunications cloud software [14],[16]. These challenges include performance issues with throughput and latency, as well as compatibility issues between cloud infrastructure and legacy software designed for dedicated hardware. Due to recent research interest, there is already available both open source and commercial solutions to overcome some of the challenges in telecommunications clouds, such as Data Plane Development Kit (DPDK), a library for fast packet processing [17]. Despite these solutions, there is a decrease in performance when moving from dedicated hardware to virtual hardware or software containers [18].

The aim of this thesis is to evaluate the current and future feasibility of adopting a combination of virtualization and containerization methods in a cloud RAN environment in terms of network performance and inter-container communications (ICC). In order to accomplish this, the thesis will combine open-source software (OSS) components and test their performance in terms of two main user-plane application performance indicators: latency and throughput. The thesis forms part of a larger project by the client to demonstrate the possibilities of DevOps, microservices, and containerization.

The rest of this thesis is divided into four chapters. Chapter 2 introduces the most common virtualization, containerization, and cloud computing techniques, as well as discusses their challenges and benefits. Chapter 3 describes the performance requirements of user-plane applications and the challenges these requirements introduce to virtualization, containerization, and cloud computing. Chapter 4 discusses two different approaches for ICC, describes the measurement setup for analyzing the differences in performance, and presents the measurement results. Finally, Chapter 5 concludes by summarizing the main outcomes of the thesis and suggesting directions for future work.

2 Virtualization and cloud computing

Virtualization and cloud computing are key enablers for modern DevOps practices and microservices architecture [6],[19]. They are expected to lower costs in computing businesses [10], thereby increasing the popularity of these technologies, especially software containers, such as Docker [7],[9]. In the telecommunications industry, cloud radio access networks (Cloud RAN) have recently received much research attention [2].

In order to understand how software containers can be used as the building blocks of cloud RAN, Section 2.1 describes the principles of virtualization technologies; Section 2.2 discusses general containerization principles, the usage of Docker containers, and the orchestration of containers with Kubernetes; Section 2.3 describes fundamentals of cloud computing and usage of cloud computing in modern software development; and finally Section 2.4 discusses the design and benefits of cloud RAN.

2.1 Virtualization

Virtualization is one of the key enabling technologies behind modern cloud computing [10]. Although virtualization as a technology dates back multiple decades, recent developments in computing resources and virtualization technologies have enabled more widespread usage of virtualization [20]. Virtualization technologies have been commonly employed to improve the utilization, manageability, and reliability of computing systems.

Example usages of VMs include user isolation, live migration, and live update [20]. To provide users safe complete control of their environment, each user can be isolated to their own VMs. Thus, changes made in one user's environment will not affect that of others. To migrate computing jobs from one hardware to another, a snapshot can be taken of the VM, move or copy this snapshot to different hardware, and continue from the exact state that the VM was in when the snapshot was taken. Finally, when updating the software, there is no need to reserve any downtime for the system, since a new VM can be booted in parallel to an existing one. When everything is up and running, connections are directed to the new VM, thus allowing the old one to be terminated.

While virtualization has been around for a long time, Intel's x86 central processing unit (CPU) architecture, which is the de facto industry standard at the moment, was virtualized for the first time in 1998 by VMware [21]. This long period between the first x86 processor and the first virtualization solution is a consequence of x86 architecture not originally being designed to be virtualized [22]. Since the x86 virtualization implementation of VMware, other companies and communities have also developed virtualization solutions for x86 architecture.

In x86 virtualization, a virtualization layer is added between the hardware and the OS [21]. The virtualization layer is usually called either hypervisor or Virtual Machine Monitor (VMM) [20]. In this thesis, the term VMM is used when referring to the virtualization layer. The role of the virtualization layer is to dynamically partition and divide the available physical resources between VMs. The virtualization

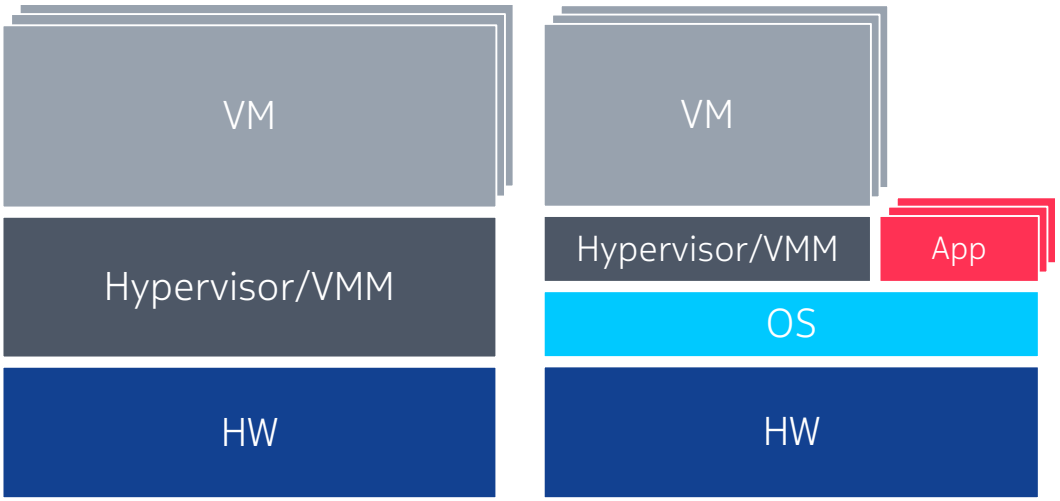


Figure 1: The two different virtualization types, bare-metal (a) and hosted (b), illustrated.

layer and virtualization in general introduces virtualization overheads which have a negative effect in the computing performance. This virtualization layer can be installed either directly on the hardware or on top of the host OS. Based on the installation location, virtualization is divided into bare-metal type and hosted type virtualizations. The bare-metal type virtualization is more efficient than hosted, as the virtualization layer has direct access to the hardware resources instead having to go through the OS. The difference between the two virtualization types is illustrated in Figure 1.

Virtualization can be implemented via three different methods: full virtualization, hardware assisted virtualization, and paravirtualization [20]. In full virtualization, all hardware resources are virtualized, and VM does not have access to any physical resources [21]. Full virtualization provides the best isolation of the three virtualization techniques. In hardware assisted virtualization, physical CPU grants VMM more privileges to allow more efficient virtualization. In both Full and hardware assisted virtualization, OS inside a VM, guest OS, is not aware that it is being virtualized and no changes are needed to guest OS. Finally, in paravirtualization, VMM communicates with guest OS to allow more efficient virtualization. Thus, modifications to the guest OS are required to enable paravirtualization. Nevertheless, in certain situations, paravirtualization provides better performance than full or hardware assisted virtualization [22],[23]. To clarify terms related to virtualization, an example of a system employing full virtualization is illustrated in Figure 2.

Regardless of the virtualization type or method, virtualization introduces multiple performance overheads when compared to the native OS [21]. These overheads come from virtualizing physical devices, such as the CPU, memory, network interfaces, and disk. The decrease in performance caused by virtualization ranges from zero to several dozen percent, depending on the computing task and the virtualization technology employed [18],[23]. Thus, performance losses can be optimized through

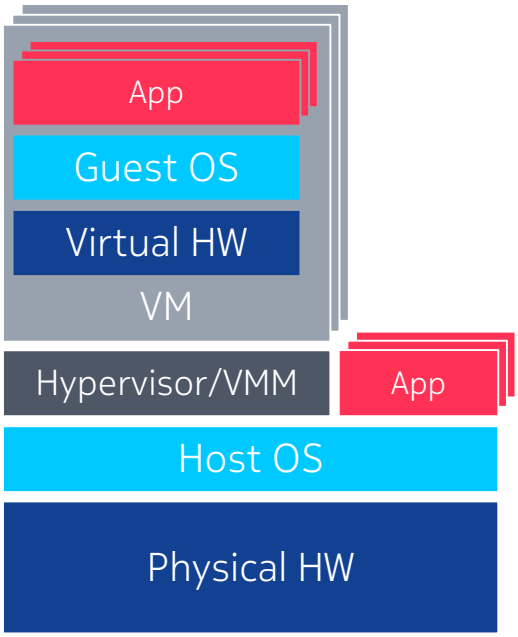


Figure 2: Example of a system employing hosted VMM and full virtualization.

proper choice of virtualization technology.

The state of a VM or physical computer can be saved to a disc image [24]. For example, to provide a functional environment to developers, the state of a VM could be saved after booting the VM and installing the OS as well as the dependency libraries. The image produced from the saved state of that VM could then be used to boot other VM or a physical machine, thus allowing the user to continue from the saved state. There are, however, limitations in the compatibility of the images when they are created and deployed in different systems.

Finally, it is possible to run VMM inside a VM on top of which we could run one or more VMs to create an environment for nested virtualization [25]. Such setup could be useful, for instance, in software development involving VMMs or much virtualization, or to provide additional security when operating in a public cloud [26]. In nested virtualization additional virtualization overhead could be as little as six to eight percent.

2.2 Containers

While VMs are commonly employed and effective technology, not always is such strong isolation or different guest OS required [27],[28]. In many cases a more lightweight solution, software container, is enough for the application. Software containers share the kernel of the host OS, which is the main reason for the lightweight nature of the software containers. Because of this shared kernel, the guest OS must be the same as the host OS. The shared kernel also results to lower level of isolation from the host OS and parallel containers. The lightweight nature of containers also introduces multiple benefits, such as increased portability, efficiency, and much faster boot time.

The main difference between VMs and containers is the level of virtualization: in

VMs hardware is virtualized, whereas in containers OS is virtualized [27],[28]. The difference between containers and VMs is illustrated in Figure 3. Note that there are now Bins/Libs layers visible in the figure. This layer represents the dependency binaries and libraries of the Application running in VM or container. Thus, instead of having to include whole guest OS, it is enough to only pack the dependencies of the application when using containers. Binaries or libraries from the host OS could also be exposed to the container, but generally it is considered better practice to pack everything in the container to keep it independent from the execution platform. The term containerization is used for this process of packing application with its dependencies to a container and isolating it from the host OS and other processes.

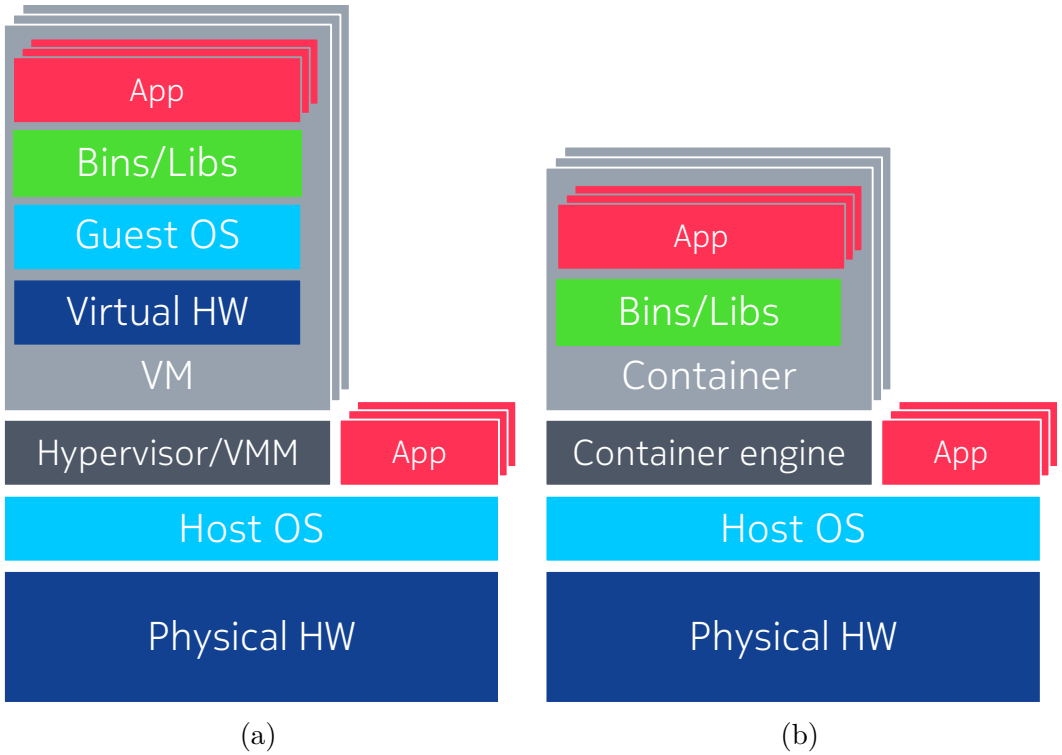
The isolation in the OS level virtualization is implemented with namespaces and control groups (cgroups). Namespaces are used to separate container resources from host resources. For example, the user namespace separates users in container from users in host system, thus making the host users invisible to container and vice-versa. Similar namespaces are used with the process tree and networking. Cgroups are responsible for limiting and monitoring resource usage of a container. These resources to be limited include memory, CPU time, and disk usage.

The performance overheads of software containers are smaller than with VMs due to more lightweight virtualization level. The performance of the application inside the container is in most cases the same or only slightly worse than that of the same application running natively on the host OS [18],[29],[30]. In most use cases, comparisons between VM and container performance show that containers have much smaller decrease in performance. There are also differences between different containerization technologies and system configurations: exposing host resources, such as storage volumes or network, will show increase in performance. Thus, the better performance of software containers compared to VMs comes at a cost of lesser isolation from the host OS and other containers.

Similarly than with VMs, disk image files are also employed with containers [28]. Containers require a more lightweight image, since OS is not included inside the image. There is also differences in the image creation process. The shared kernel with the host OS creates limitations on the type of images that can be on run on a given host OS.

When building an application from microservices, one approach is to run each microservice in its own container [28]. This allows scaling up or down the microservice by creating or terminating container instances. The scaling up or down can be automated with a container orchestration solution, such as Kubernetes [27]. The orchestration tool will often also provide most of the required operations and monitoring tools for the application. These tools include auto scaling, load balancing, and service discovery functions.

This section is further divided into two subsections. Section 2.2.1 describes Docker, a popular container type, and Section 2.2.2 describes Kubernetes, a tool for Docker container orchestration. These two are both open source systems that are enablers of containerization and are employed in the practical part of this thesis.



(a) (b)

Figure 3: A VM (a) and a container (b) compared [27],[28].

2.2.1 Docker

Docker provides tools to build, re-use, version, and share containers easily, which is exciting from DevOps point of view [31],[32]. Docker container images can be built either interactively or automatically based on instructions provided through a Dockerfile. Dockerfile is a set of commands that define what is installed to the container and what commands are run in the build phase. Dockerfile can be used to build the image from scratch or instructed to use any Docker image as base image to easily re-use existing images in build phase. Docker includes git [33] like version control to track different versions of a container. This allows efficient comparison of version and low-cost uploads and downloads as only the difference files have to be transferred. Docker encourages sharing images by providing public image registry where anyone can share or take into use images. It is also possible to create private registries.

One of the main benefits of Docker images and containers is the possibility of extensive sharing and re-using of the containers [31],[32]. The Docker images are built of layers, which can be stacked on top of any existing image. The image that layers are stacked top of, is called base image. Everything from base image to commands to install dependencies or application itself are considered layers. Layers can also contain meta-data about the images, such as maintainers and versions, environment definitions, and can be employed to add file or directory structures to images. The layered structure of an image also requires attention in the build phase as unoptimized build procedure will create larger than necessary images.

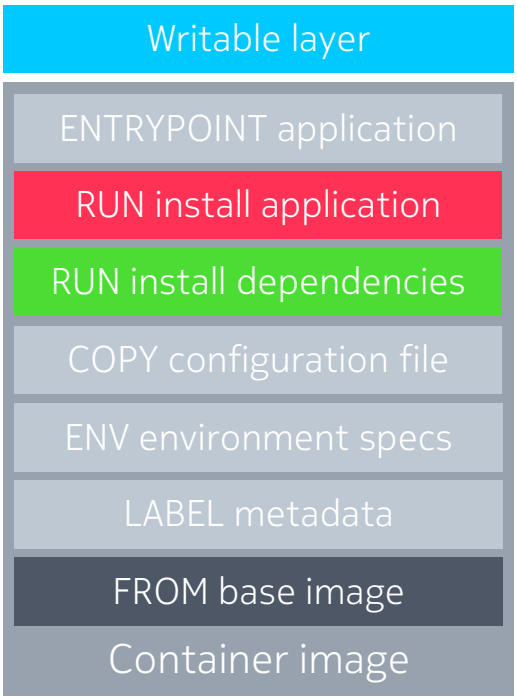


Figure 4: Layers of Docker images illustrated.

An example of a Docker container image is illustrated in Figure 4. In figure base image is on the bottom and would be the first command in the dockerfile to create this image. Layers progress towards the top of the figure, where as in dockerfile they would progress towards the end of the file and be separated with line breaks.

Docker images are executed with Docker engine. Each container instance executed with same image shares the layers of the image with writable layer added to the top [31]. This writable layer is unique for every container instance and all differences to image filesystem are written to this layer. The changes made to this writable layer can be committed to make a new image with executed image as the base image and changes made to writable layer as a new layer on top of that.

Docker has great potential from DevOps and microservices point of view. Docker provides a way to containerize an application [3],[31]: Dockerfile and Docker build tools provide an easy way to automate the building of container images, which are then pushed to private or public Docker registry from where test automation and deployment automation have access to the container to be tested or deployed. Since Docker provides a solution to pack the software to a container with its dependencies, different parts of an application can be build, tested, and deployed independently allowing building of an application from microservices.

However, all these benefits come at a cost of lesser isolation from the host and other containers [9]. This means that usage of Docker containers will cause security issues if the architecture is not well designed [34]. Especially in systems where additional host resources or operation capabilities need to be exposed to the Docker container, malicious user might be able to sabotage the host system or parallel containers: Container that is ran in privileged mode might allow root access to the

host system. Container orchestration tools, such as Kubernetes, provide solutions to some of the security issues as low-level control of the containers is hidden from the user.

2.2.2 Kubernetes

Kubernetes is an orchestration tool for Docker containers [35]. It has gained a lot of popularity in the cloud computing community due to providing extensive platform for running Docker containers in different cloud and datacenter environments [27]. Kubernetes provides tools required to efficiently run application built of Docker containers including load balancing, high availability, service discovery, and automatic upgrading as well as downgrading of software containers.

Portability is the key benefit provided by Kubernetes [35]. As a container platform, Kubernetes abstracts away implementation specific properties and tools offered by different cloud and datacenter providers; The same kubernetes configurations and recipes can be easily migrated from cloud to another, for example, from OpenStack to Amazon web services.

Kubernetes provides support for distributed multi-host environments [35]. When more than one host is used in same Kubernetes configuration, one host acts as a Kubernetes master node and others as worker nodes. Similar responsibility division is common in cloud computing systems: for example, OpenStack implements similar division in multi-host environments.

Kubernetes executes containers inside pods [36]. A pod is the smallest component orchestrated by Kubernetes and it can contain one or more containers. The containers inside a pod share resources extensively. Containers in same pod share by default the same network and the resource sharing can be extended, for example, by adding a storage volume common for all containers to a pod. Because of this extensive resource sharing inside a pod, all containers running inside the same pod are always run on a single Kubernetes node. Pod often comprises an independent component of an application, with a single task [37].

Kubernetes automatically assign pods to available host node, but preference can be passed to Kubernetes by using labels [36]. Labels can be used to describe worker nodes, for example host with memory priority could have memory label. Pods can be given required labels allowing pods to only be run on host that fulfills all label requirements.

2.3 Cloud computing

Cloud computing differs from traditional data-center based computing by five key characteristics [38]: The computing resources are available to be provisioned by customer manually through user interface (UI) or automatically through application programming interface (API). The computing resources are accessed over the network via standardized remote access protocols. The computing resources are provisioned from the cloud providers resource pool by customer with little or no control over the physical location of the computing resources or the other users sharing the same

physical hardware. The amount of provisioned computing resources can be rapidly scaled up or down to match the actual demand and appear to the customer as infinite. The usage of the computing resources is measured by the providers and the billing is based on the actual usage of the resources. For example, computing resources could be billed by provisioned hours and storage could be billed by provisioned gigabytes per day.

Clouds are typically divided in three categories based on their accessibility: public, private, and hybrid [10],[38]. Public cloud is offered by cloud provider for the use of general public and typically billed on usage based rates. Private cloud is a cloud available exclusively to limited user base, for example one corporation. The private cloud can either be offered by in-house provider or bought as a service from third party provider. Hybrid cloud is a combination of these two; hybrid cloud consists of two or more independent clouds that are connected through standardized interfaces allowing seamless portability between these clouds.

Clouds can be further divided into two categories based on their location in the network: core and edge clouds [39]. Core cloud has the features described earlier by the key characteristics and is suitable for most applications. Edge cloud is located at the edge of the network, as close to the end-user as possible, and is required for delay critical applications, such as augmented reality displays, virtual reality gaming, and high throughput content delivery. Edge cloud does not comply with all of the general cloud characteristics presented earlier, but is considered to be a part of the cloud as it is seamlessly connected to the core cloud. For example, video streaming application might have its UI and other not so delay critical components operated from core cloud and content synced to edge cloud to allow high throughput and low latency streaming of high quality video.

Cloud computing providers usually follow one of three typical service models: Software as a Service (SaaS), Platform as a Service (PaaS), or Infrastructure as a Service (IaaS) [10],[38]. In SaaS model provider offers an application, software, as a service to the end user. Couple examples of such services include Google Drive and Dropbox. In PaaS model service provider offers a platform, with the OS and possibly agreed dependencies included, where the customer can deploy their application. Example of such service is Heroku. Finally, In IaaS service provider offers, in most cases, virtualized computing infrastructure that the customer can control. Example of such service is Amazon EC2.

These service models and their providers are illustrated in Figure 5. Note that the providers can either own and manage the resources necessary to offer their services or buy these resources from other cloud providers or data-center providers. For example, IaaS and PaaS providers could offer their services for SaaS providers.

PaaS and IaaS are the most interesting service models from software development point of view, because the offered service is in the form of computing resources [10]. The difference between these two is not always clear, but generally PaaS provides easier solution for deploying the application in environment defined by the provider where as IaaS provides more control over the computing environment and a management tools to controlling services, such as computing instances, storage volumes and public IP addresses. PaaS providers might also offer services specific to certain scenarios,

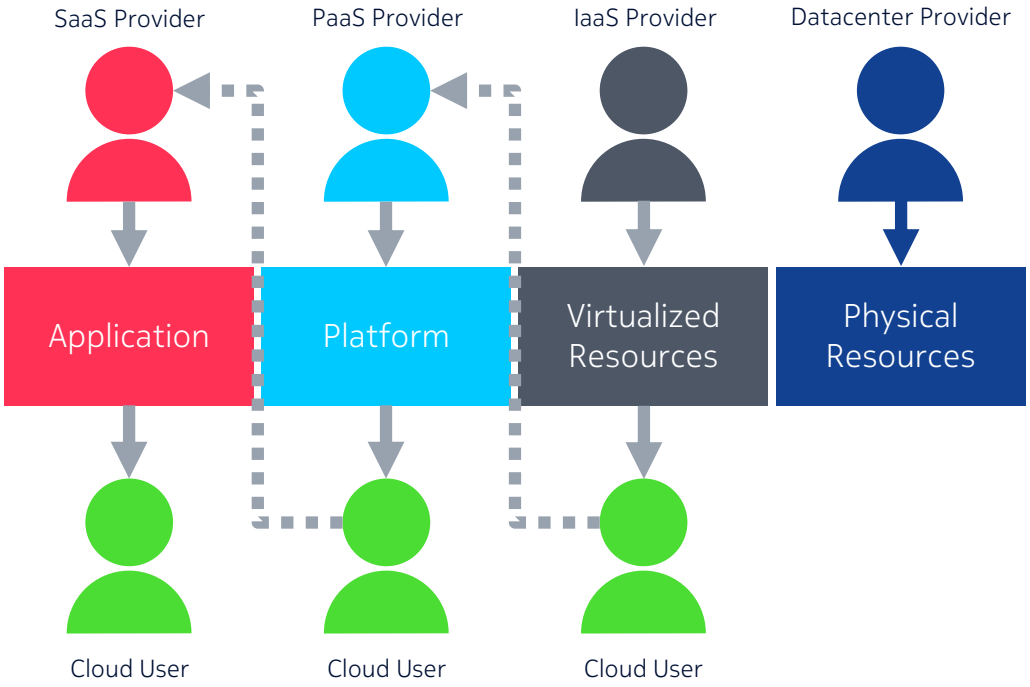


Figure 5: Illustration of different cloud providers, their products, and their relation to other cloud providers.

such as web applications, which might cause problems or opportunities depending on the planned usage.

There are many obstacles in adapting cloud computing, from cross-industry challenges to cloud specific ones [10]: challenges such as service availability, data lock-in, data confidentiality, slow or expensive data transfer, and licensing fall in the first category and solutions from other fields can be adapted to overcome problems: service availability and avoiding data lock-in can be ensured by using multiple independent services, data confidentiality can be secured with standard measures including encryption and firewalls, slow or expensive data transfer over public network can be in some cases avoided by shipping physical storage devices, and licensing of software can be negotiated to follow cloud native usage based rates.

In addition to these classic challenges there are also cloud specific challenges to overcome [10], including performance unpredictability, scalability, debugging of large distributed systems, and reputation fate sharing: Performance unpredictability is due to sharing physical resources with other users as their usage will affect the overall computing performance available. Scalability is a cloud native feature and efficient implementations might require significant refactoring of an application. Debugging of large distributed systems is problematic as errors might not occur in smaller deployments and production size resources are also needed in debugging. Reputation fate sharing will become a problem if physical resources are shared with ill-behaving users: resources such as IP addresses might be blacklisted by Internet service providers or email spam filters.

There exists solutions to these cloud specific challenges, but as cloud technology

is a relatively new concept many of these solutions are still in development phase [10]. The solutions being developed include concepts such as machine learning based predicting scalability and machine learning based debugging tools for large distributed systems. As cloud computing is currently receiving lot of attention in both academic community and the industry the development of new technologies is rapid and OSS gets more and more mature.

This section is further divided into two subsections: Subsection 2.3.1 discusses differences between traditional and microservice based software and gives examples of methods to convert traditional to microservice based software. Subsection 2.3.2 described DevOps practices in more detail and gives examples of DevOps mindset and toolset. These two technologies are fundamental in development of cloud native applications.

2.3.1 Microservices

Microservices architecture is a modern cloud-native way of building applications [3],[6]. Microservices have three key characteristics: the microservice should implement a single and clear task, it should be possible to independently develop, build, test, as well as deploy the microservice, and the microservice should have well defined outside interfaces. These characteristics allow building larger applications from independent pieces that communicate with each other through standardized interfaces and can be extensively re-used in other applications.

One of the key benefits microservices architecture has to offer is the possibility to develop software in small teams [40]. As applications are built of smaller independent pieces of code, personnel involved in a large software project can be divided into small teams focusing to one microservice at a time. This allows teams to push their changes without having to merge code produced by other teams, which might be time consuming, and software teams are able to work more efficiently.

As a lot of software has been developed before the cloud era and deployed on individual mainframe servers, process of migrating from monolithic software architecture to microservices based is going on in many organizations [3]: much research attention has been recently directed towards different aspects of microservices. However, as microservices are relatively new concept there are still open questions on adaptation process: there is a very wide range of possible combinations of microservices which causes problems in areas such as configuration management, performance against isolation trade-off, monitoring, and elastic scheduling [41].

Microservices are often deployed in software containers, most popular at the moment being Docker [42], as lightweight isolation is enough for most microservices. Containers provide or implement many cloud native features required by microservices architecture, such as abstraction of storage or networking services [43]. Containers and container orchestration also provide extensive automation possibilities for microservices. While containers and container orchestration have lot to offer for microservices, microservices can also be run on VM or directly on the host. The optimal solution for the environment to run microservices in depends greatly on the implemented service as there is not yet any all-around best practices available in the

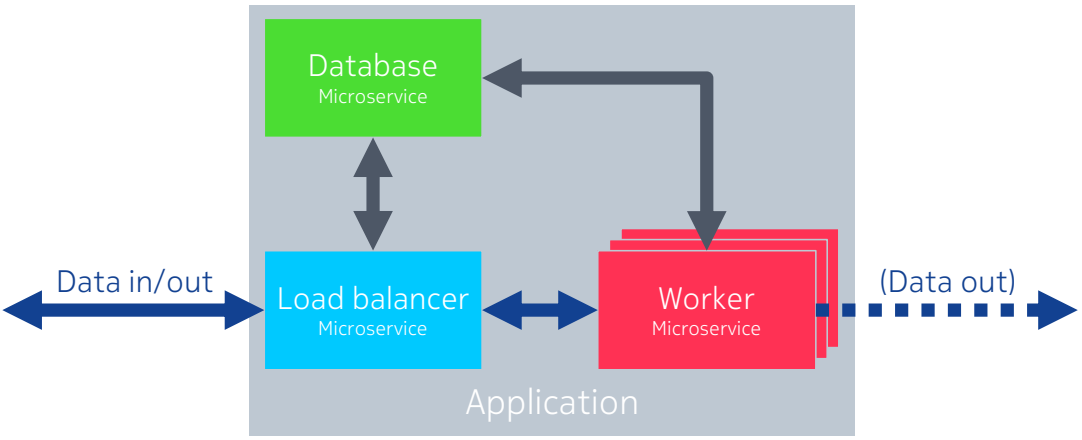


Figure 6: An example of a generic application built of microservices.

literature [42],[43].

An example of a simple and generic application built of microservices is provided in Figure 6. In the figure, colored boxes illustrate different building blocks of the application, microservices. These are isolated from the rest of the application, for example by the means of Docker containers. Gray arrows indicate database connections. These could be used for example to implement stateless microservices: load balancer writes upcoming tasks and reads completed tasks from database, whereas workers keep track of their status and upcoming tasks through the database. Blue arrows indicate data connections: typically, all incoming data is received by load balancer and directed to workers according to internal communication of the application. Data flow from load balancer to workers can be direct or via database depending on the needs of the application. Data flow from the application can be directed out via the load balancer or directly from the workers. The worker nodes are the ones actually implementing the business logic of the application.

Microservices offer a lot of benefits to all phases of large software projects, but it might not always be cost efficient to use microservices [40]: microservices require more code to perform similar task than a monolithic implementation, because of the more complex inter-process communication (IPC), but offer better flexibility, scalability, and agility. Generally, microservices architecture offers more benefits for bigger projects, but each software is unique and trade-offs have to be made.

2.3.2 DevOps

DevOps is a term referring to both mindset of bringing developer and operations personnel in software project closer together and toolset applied in this process [5]. Some of the goals in DevOps process include delivering and deploying software more often and in smaller pieces than before. One of the key enablers of DevOps process is microservices architecture: when software is developed in small independent pieces it can be built, tested, and deployed more often [3].

Building an application using microservices allows develop-build-test-deploy chains for each component instead of single chain for the whole application [3]. This allows

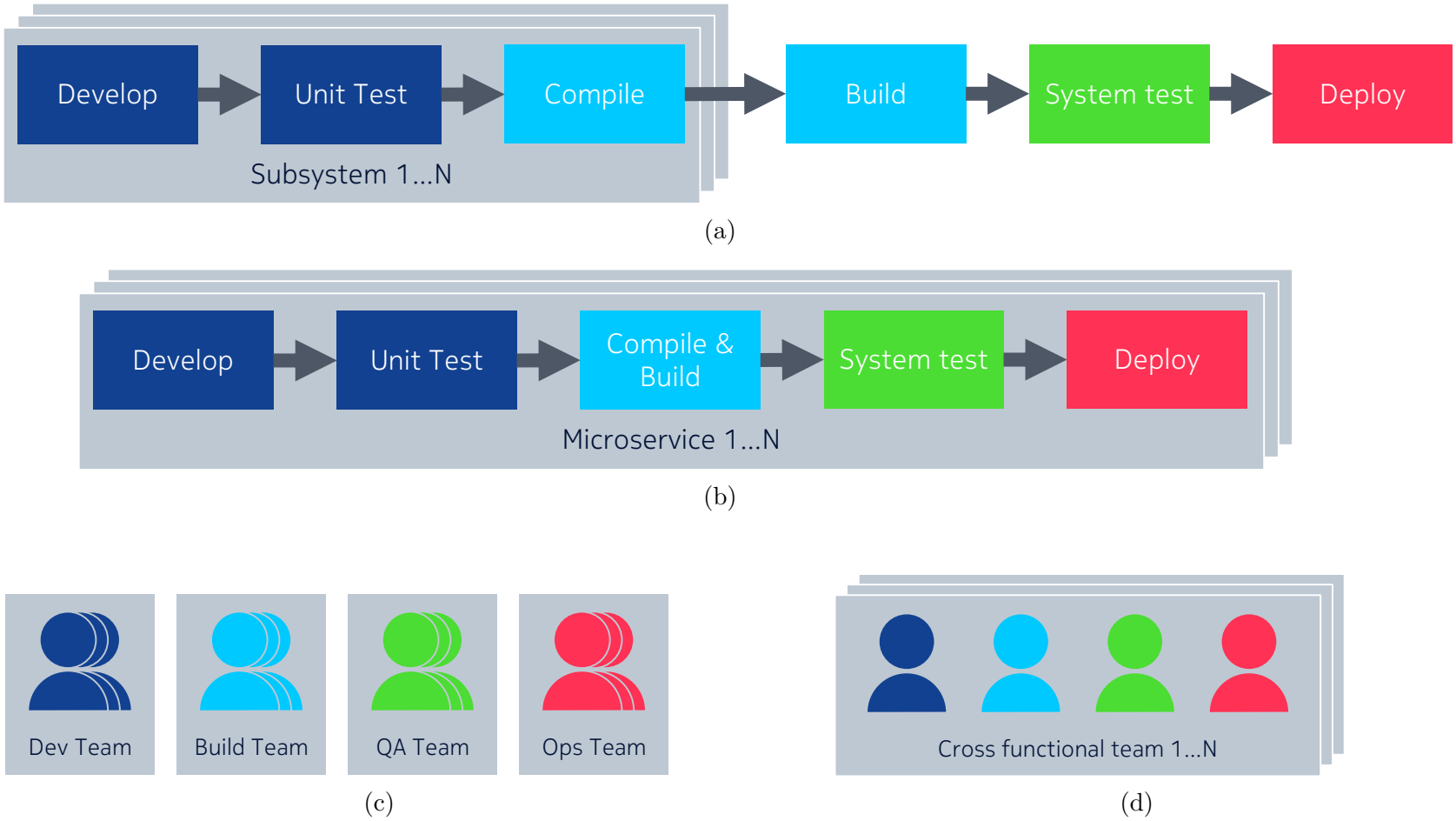


Figure 7: Examples of transformation from single chain (a) to independent development of microservices (b) and transformation from function specific (c) to cross-functional (d) teams illustrated [3].

independent development of the microservices so that there is no need to wait for each component in some phase of the chain. This transformation from single chain to independent development of microservices is illustrated in Figure 7a–b. There are differences in different project and the chains illustrated are only examples. The key enabler of this transformation is the independence of microservices and well defined external interfaces: as external interfaces are predefined there should not be strict version requirements between microservices and each microservice can be updated individually.

Traditionally software has been developed by function specific teams [3]: For example, when following the chain from figure 7a, development team would develop the source code and unit tests, build team would compile and build the software as well as ensure that the versions of different subsystems are compatible with each other, Quality Assurance (QA) team would take care of developing and managing of system level tests, and operations team would deploy and operate the application. In DevOps organization teams are ideally build such that every team is responsible for whole develop-build-test-deploy chain of a microservice [3]: This requires that each team is cross-functional. There might still be persons who specialize into some phase of the chain, but cooperation between team members should be on a level that allows switching or adapting different roles when needed. This transformation from function specific to cross-functional teams is illustrated in Figures 7c–d

DevOps practices promote extensive usage of automation to allow faster develop-build-test-deploy chains and more efficient work flow in cross-functional teams [3]. This extensive usage of automation can be built on top of processes such as continuous integration (CI) and continuous deployment (CD): CI is typically triggered by change pushed to the code base of an application and aims to automatically validate the quality as well as functionality of the code and merges the change to existing code base. CD takes the automation a step further by automatically deploying changes to production environment after successful CI execution. Both CI and CD typically notify the author of the change and stop further automated steps, if integration or deployment failed for some reason. This is to keep code base and production environment in functional state and means that human interaction is only needed in situations where change would cause problems according to a set of predefined rules.

The key benefits achieved through DevOps are faster and more straightforward develop-build-test-deploy chains, more efficient teams, and widespread automation of processes in the chain. The develop-build-test-deploy chain of a microservice is much faster than traditionally because of high level of automation, single responsible team, and no need to wait for depending subsystems. Teams are more efficient due to smaller size and high level of automation. Wide spread usage of automation is possible because of more straightforward chains and independent components.

2.4 Cloud radio access network (Cloud RAN)

RAN is the part of MN connecting core network and physical radio resources [44]. The relative position of RAN and other components of MNs are illustrated in Figure 8. In the figure, and also later in this section, Long Term Evolution (LTE)

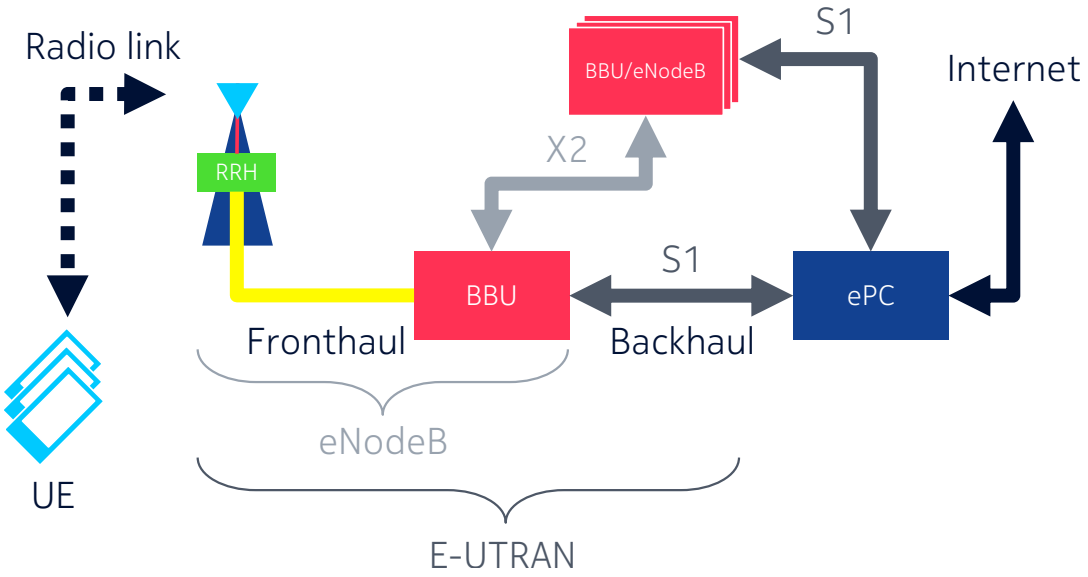


Figure 8: The relative position of RAN and other components of MNs illustrated [44].

terminology is used, though these components and technologies can be also generalized to match other generations of MNs. Figure simplifies MN to four key components: user equipment (UE), Base Band Unit (BBU), Remote Radio Head (RRH), and evolved Packet Core (ePC). BBU and RRH together form a base station (BTS, *base transceiver station*), eNodeB, and all elements between the UE and ePC comprise RAN, Evolved Universal Terrestial Radio Access Network (E-UTRAN).

Cloud RAN is a cloud implementation of traditional RAN. The transformation from traditional RAN to cloud RAN and the challenges of cloud RAN are discussed in this section. Cloud RAN architecture has received much research attention recently [2]. The reasons for this research attention include expected decreasing total cost of ownership in MNs, increasing maintainability, and decreasing power consumption.

Cloud RAN is a continuation of the development from single BTS element to separate RRH and base band unit BBU [2]. In traditional BTS, signal processing, conversions between analog and digital signals, power amplification, and filtering are all done in same element, which is connected to antenna with a short, max tens of meters long, coaxial cable. This implementation in illustrated in Figure 9a.

In base station with separate BBU and RRH, processing responsibilities are shared between these elements: BBU takes care of base-band processing and RRH handles RF processing. Similarly than with single element BTS, coaxial cable connecting RRH to antenna has to be short. Nevertheless, BBU and RRH are connected via optical fiber and the length of this cable can be extended to tens of kilometers. This setup is presented in figure 9b

Cloud RAN makes use of the possibility to have a significant distance between BBU and RRH by centralizing multiple BBUs to the same location as a cloud service [2]. In cloud RAN deployment, BBUs are virtualized and centrally located in a data center, or an edge cloud, as a virtual BBU pool. This BBU pool is connected to the

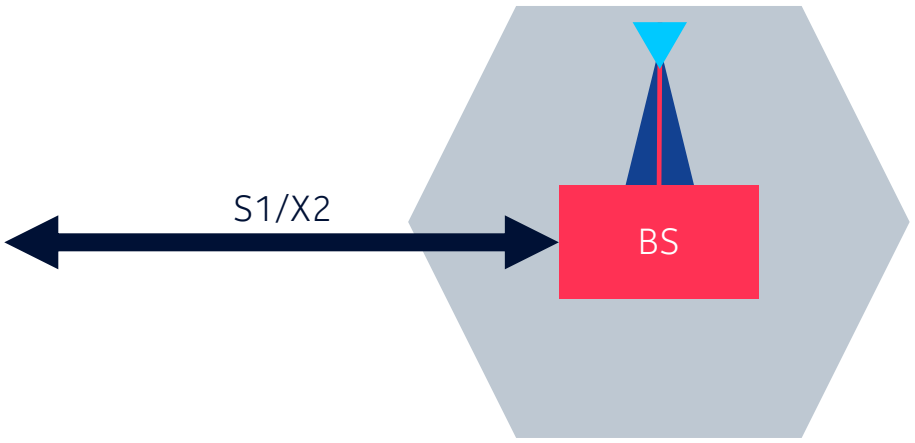
RRHs scattered around the area via optical fibers [45],[46]. This allows a cloud like scalability in three dimensions [47]: number of active RRHs, cell sizes via change in transmit power of RRHs, and size of BBU clusters assigned to RRH cluster. This implementation is illustrated in Figure 9c.

The main differences between traditional distributed RAN and cloud RAN are the cloud RAN's three scalabilities: number of active RRH, cell size of active RRHs, and number and size of active BBUs [2]. The number of active RRHs in the same area can be scaled up or down to meet the demand of peak and quiet hours [47]. In contrast, traditional BTSs are always on and often designed to meet peak demand. This causes much unnecessary power consumption in the network. The scalability in number of active RRHs is partly possible due to scalable cell sizes. By increasing or decreasing the transmit power of the RRH, cell size of individual RRH is scaled to meet the requirements of the cluster it belongs to: During peak hours, when more RRHs are required to handle the network load, transmit power and cell sizes are scaled down to allow denser network and larger quantity of active RRHs. During the quiet periods transmit power and cell size are scaled up to allow more sparse density and smaller quantity of active RRHs. Finally, number and size of virtualized BBUs is scaled up or down to match the number of active RRHs and computational resources required by the users served by BBU and RRH clusters.

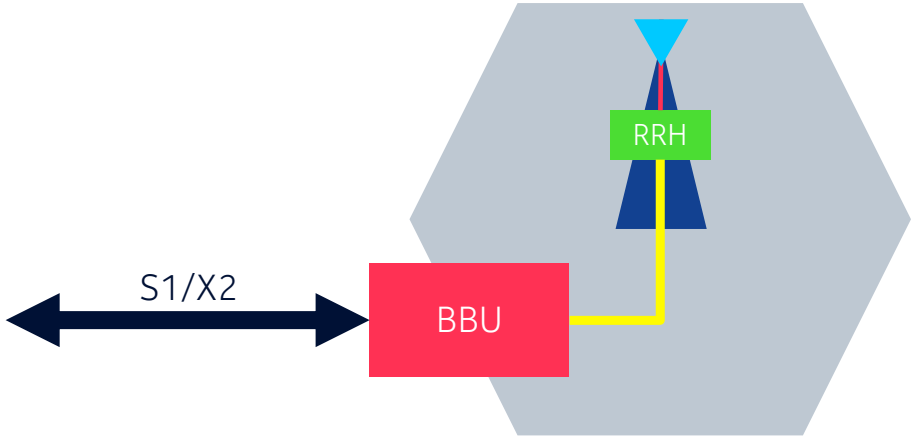
The scalabilities provide an opportunity for significant cost savings compared to a traditional distributed RAN [46]. As the number of active RRHs is scaled to match demand in network instead of keeping all of the BTS on all the time, lot of power is saved [47]. RRHs with cloud RAN also introduce cost savings in installation and operation of cell sites, because installation process of RRH is much more lightweight than that of traditional BTSs, the volume of equipment that needs to be located at rented property at cell site is much smaller, and the cooling of centralized computation equipment is much more effective.

Centralization of BBUs to an edge cloud as a virtualized pool provides additional benefits: as most of the computational equipment is located in a centralized data-center, cooling and maintaining physical equipment is much more effective than if they were distributively located. Depending on the scenario, the centralization can be done at different functional responsibility levels of BBU, thus creating flexible functional split between BBU and RRH [48]. This allows implementing a cloud RAN also in environments where there might be limitations in the fronthaul or the backhaul. Cloud implementation of virtual BBU pool allows effective software updates and virtual BBU with error state can be instantly replaced with new one. This allows the operators to offer better service level agreements to their customers. On demand scaling up and down of the BBU pool results to about 25% decrease in need for base band computational resources [49] and additional cost savings through the decrease in power consumption are available through advanced and dynamic optimization of the cloud RAN installation [50]. Cloud RAN could also serve as a single interface for multiple radio access technologies removing the need for multiple physical RANs in the MN [51].

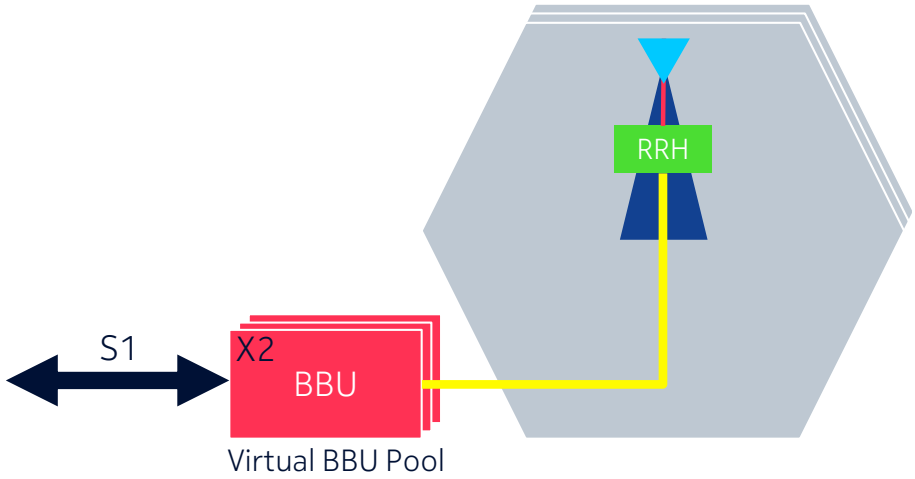
The individual savings described in the last two paragraphs contribute to a total power savings of cloud RAN are estimated to be around 70% [52],[53], and the



(a)



(b)



(c)

Figure 9: A traditional BTS (a), BTS with separate BBU and RRH (b) and a cloud RAN (c) compared [2].

total cost savings of 15% in capital expenditures (CAPEX) and 50% in operational expenditures (OPEX) are expected [53]. In addition to these cost saving, the cloud RAN is likely to create new business opportunities for virtual operators as they are not anymore tied to a single operator's infrastructure instead being able to acquire RAN resources on demand [48]. This might further increase the usage efficiency of the MNs through flexible pricing models.

The direct connection of BBUs and RRHs introduces requirements to the edge cloud that the virtual BBU pool is located in. As stated earlier, the maximum length of the optical fiber connecting BBU and RRH is some tens of kilometers [2]. This means that the individual edge clouds can only serve RRHs at relatively small area compared to the service area of core clouds. The optical fiber connections to the hardware introduces limitations to the choice of hardware and level of virtualization. This, in practice, means that private or hybrid clouds have to be used. The performance requirements for the cloud implementation and solutions to meet them are further discussed in Chapter 3. The rest of this section will discuss the modernization requirements for the user-plane applications, which are responsible for implementing the datapath for user traffic from BBU to core network, and vice-versa, in cloud RAN.

2.4.1 User-plane modernization

User-plane, like many other telecommunications applications, has traditionally been implemented as a monolithic piece of software that has been running on dedicated hardware [2]. However, this monolithic software architecture does not allow efficient scalability required by cloud RAN. To allow scaling of individual services the user-plane software has to be modernized. To comply with current state of the art software architectures and to be able to extensively re-use OSS, user-plane software modernization will progress towards microservices architecture and DevOps practices.

Out of the three scalabilities provided by cloud RAN, the user-plane containerization and modernization discussed in this thesis focuses mainly on the scalability in the virtual BBU pool. As BBUs are implemented as a pool and should be available on demand, individual BBUs should be isolated from the host platform and each other. However, some parts of this isolation might have to be bypassed to meet telecommunications grade performance requirements. Possible implementations for isolation of BBUs include Kubernetes pods and VMs. While Kubernetes itself offers scalability for pods, VM implementation requires some sort of cloud orchestration to implement scalability and such features.

The implementation of individual BBU, ignoring now the upper level isolation method, should provide scalability in computational resources to meet different size requirements of different BBUs. In microservices architecture this scalability is achieved via load balancing and by scaling the number of available worker units. This functionality is provided by orchestration, either container or VM depending on the implementation. To provide stateless functionality database services are required in BBU. However, since databases are most likely too slow for telecommunications traffic also a faster datapath must be established between load balancer and each

worker.

Other aspect of the user-plane modernization is the shift from usage of dedicated hardware to using general purpose hardware and OS. To maintain the real-time processing capabilities formerly provided by the hardware and real-time OS, libraries and kernel modules providing similar functionalities are added to the general purpose OS. One solution to enable better performance is to, instead of fair scheduling kernel, use low latency or real-time version of kernel. Although the performance gain is not visible when there is no other loads on the host, low latency and real-time kernels show significant performance boost when there exist load from other processes in the host [54]. This performance boost might be necessary in order to meet telecommunications grade performance requirements [55].

In addition to moving from dedicated to general purpose hardware, the transformation from traditional RANs to cloud RAN requires change from static to dynamic networks [13]. To allow the scalabilities introduced by the cloud RAN, networking should be also able to scale with the demand. The network scalability is enabled by two key concepts: software defined networking (SDN) and network function virtualization (NFV).

SDN is a network architecture where control-plane and data-plane are separated as well as are highly programmable through APIs. [56]–[58]. This means that network functions (NF), that traditionally also included the control interfaces, are controlled from external controller, thus simplifying the responsibilities of a NF to ones of a packet forwarder. The forwarding is implemented as flow based instead of destination based, to allow the forwarding of sequences of packets with the same rule. The network is programmable through APIs supported or provided by NFs and controllers. Although there have been concerns that SDN would decrease the network performance, it has been shown that SDN capable networks have quite promising performance when implemented with technologies such as DPDK [14],[59]. The main driver for SDN adaptations is that SDN is expected to increase efficiency of network management and to decrease the network operation costs [58].

NFV refers to virtualizing NFs, that have traditionally been run on dedicated hardware, so that they can be run on commodity servers with general purpose processors [14][60]. This means replacing the functionality of a traditional physical NF device, such as a switch or a firewall, with a software implementation. Similarly than with SDN, performance is expected to decrease with higher level of virtualization. Nevertheless, this decrease in performance can be compensated by using software containers instead of VMs and technologies such as DPDK in the virtual network function (VNF) implementation [61],[62]. The key benefits of NFV include increased efficiency, increased manageability, decreased power consumption, and decreased operational costs [60].

Although SDN and NFV are not strictly dependent of each other, these technologies are highly complementary when implemented together [56],[60]. While SDN could be implemented on top of physical NFs, VNFs provide additional scaling and dynamic networking possibilities for SDN. In addition, although NFV could be implemented without SDN, the gains in performance and manageability are further enhanced by implementing also SDN.

To summarize, the user-plane modernization by containerizing and by steering the implementation towards microservices architecture essentially equals developing a VNF. Thus, allowing to take advantage of extensive previous research, standardization, and OSS releases related to NFV. For user-plane modernization, SDN is essential in providing the support and scalability for highly dynamic networks of cloud RAN software implementations.

Chapter summary

This chapter introduced the key concepts of virtualization, containerization, and cloud computing. Compared to VMs, software containers provide a more lightweight solution to achieve a similar effect, as virtualization is done at the OS level. This allows more lightweight images and better performance. However, these benefits come at the cost of less isolation from other containers and the host OS. These two technologies are enablers of cloud computing, which offers the possibility to buy computing resources as a service. One application of these technologies is cloud RAN which is expected to provide significant cost and power savings to telecommunications operators by centralizing BBUs to an edge cloud. The transformation to cloud RAN requires changes in both software and network architectures, which are enabled by concepts such as microservices, DevOps, SDN, and NFV.

3 Performance indicators and enabling open source software

The performance overheads of virtualization and cloud computing are challenging in telecommunications applications, because of real-time processing requirements. This results to different requirements for the system compared to more common software types. Common OS kernels are not always suitable for telecommunications applications and libraries to modify OS kernel or bypass it are required to meet the performance requirements.

To provide basis for container architecture discussion and measurements, this chapter introduces performance indicators and requirements as well as describes OSS applied in the practical part of this thesis: Section 3.1 discusses two performance indicators applied to compare different containerization architectures, and Section 3.2 introduces OSS employed to boost the performance of x86 hardware in pursue towards telecommunications grade performance.

3.1 Performance indicators

This section describes two performance indicators, throughput and latency, and their importance for cloud RAN and MNs. The performance indicators will be used to compare different containerization architectures in Chapter 4.

Throughput is the rate of successful data deliveries over a network. Throughput is often measured either in packets per second (pps) or bits per second (bit/s). The relation between these is that bits per seconds equals to packets per second multiplied by the packet size. As packet size is not constant in different network, packets per second unit can cause confusions, if the used packet size is not provided.

High throughput is an important feature of modern MN, since smart mobile devices are increasingly used for applications that send and receive high amounts of data, such as video streaming. Current LTE networks support data throughputs of hundreds of Mbits/s between UE and BTS [44]. Deeper in the MN, where single element handles multiple UEs or BTS, the required throughput is much higher. Further, the target throughput in 5G MNs are set to range from 1 to more than 10 Gbits/s. In order for current microservices to be re-usable in the future, throughputs much higher than this should be supported.

In telecommunications, latency is the time delay between sending and receiving packets. As a time delay, latency is measured with seconds as the base unit. In telecommunications and other latency optimized systems latency and clock cycles of processors are closely related. This means that that latency is usually the most convenient to be measured in nanoseconds or microseconds.

One typical method used for latency measurements is time stamping of packets. This method is used also by iperf software which will be used for the basic latency measurements within this thesis [63]. This method provides accurate results provided that the clocks at transmitting and receiving ends are synchronized. If the clocks are not accurately enough synchronized the latency can be estimated by looking at the

latencies to both directions:

$$\begin{cases} \tau_{12} = \Delta t_{12} + \tau \\ \tau_{21} = \Delta t_{21} + \tau \end{cases}, \tag{1}$$

where τ_{12} is the measured latency from host 1 to host 2, τ_{21} is the measured latency from host 2 to host 1, Δt_{12} is the clock difference from host 1 to host 2, Δt_{21} is the clock difference from host 2 to host 1, and τ is the latency between the hosts. This equation group can be rearranged to

$$\begin{cases} \tau = \tau_{12} - \Delta t_{12} \\ \tau = \tau_{21} - \Delta t_{21} \end{cases}. \tag{2}$$

Since $\Delta t_{12} = -\Delta t_{21}$, clock difference between the hosts can be calculated from

$$\Delta t_{12} = \frac{\tau_{12} - \tau_{21}}{2} \tag{3}$$

and the actual latency can be then estimated with Equation (2). It has to be noted that this estimation is quite sensitive to random variations. Thus, the results are the most reliable when measurements to both directions are conducted simultaneously.

Latency has traditionally been an important metric of a telecommunications network. In the early days of MNs the traffic was mostly voice calls. In order for natural conversation to be possible, the latency has to be kept low with minimal variance. Although latency has not been as critical in traditional data connections, the latency in 5G MNs has been targeted to be few milliseconds [1]. This is due to requirements set by emerging technologies, such as augmented reality and self-driving cars.

In MNs latency can be defined to take into account different combinations of the components in the system. For example, radio latency is the propagation delay over radio channel from BTS to UE, or vice-versa, and fronthaul latency is the propagations delay in optical fiber from BBU to RRH, or vice-versa. End-to-end latency refers to latency between first source element and last destination element. For example, in MN UE could be the source element and internet server the destination element.

As this thesis focuses on cloud networking performance and ICC, the thesis uses term latency to describe end-to-end latency between source and destination processes which are being run in containers or VMs on top of cloud infrastructure. This definition is illustrated in figure 10.

These two indicators, latency and throughput, form the basis for performance analysis as they give fast overview of the achieved networking performance. There are also more parameters to consider in more detailed performance analysis, such as jitter which is the variation in latency. Those further parameters, however, are not discussed nor analyzed in this thesis.

3.2 Enabling open source software

Several open source technologies are being developed by the open-source community to allow more efficient virtualization and data plane prosessing with x86 hardware.

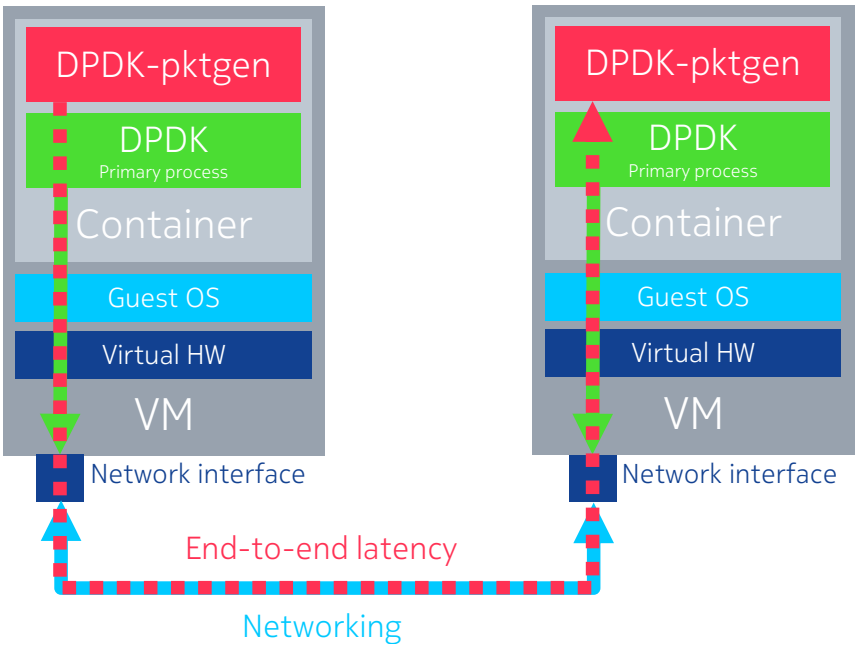


Figure 10: End-to-end latency between source and destination processes which are being run in containers and VMs on top of cloud infrastructure illustrated.

This section describes three technologies that are employed during the practical part of this thesis. Subsection 3.2.1 discusses DPDK, a set of open source libraries for fast path networking. Subsection 3.2.2 describes OVS, an open source implementation of virtual switch. Subsection 3.2.3 discusses OpenStack, an open source cloud computing software.

3.2.1 Data Plane Development Kit (DPDK)

DPDK [17] is a set of open source libraries and drivers for fast packet processing [17]. It was originally developed for x86 architecture, but the support has since been extended to also other CPU architectures, such as ARM. DPDK allows bypassing some kernel functions, thus providing more direct access to HW resources. Linux user-land applications gain more control on process with usage of DPDK: for instance, logical cores can be occupied for the use of a process or a network interface card (NIC) can be accessed without going through the OS [62]. This allows executing high speed data-plane applications on x86 architecture instead of dedicated HW.

Applications employing DPDK access the DPDK libraries through linking to Environment Abstraction Layer (EAL) [64]. EAL is created by DPDK with make and configuration files to match the physical environment. EAL gains access to low-level system resources and is the entry point to other DPDK libraries. In addition to initializing DPDK for use of the application, EAL is responsible for tasks such as assigning cores, reserving memory, providing debug functionalities, and providing access to PCI memory.

DPDK uses hugepages for memory allocations [64]. Hugepages are much larger, in x86 hardware typically 2 MB or 1 GB, than normal memory pages of 4 kB. This

usage of larger page size allows better performance, since less pages have to looked trough to find relevant memory section.

DPDK processes are divided into two categories depending on their memory usage: Primary processes, which are responsible for initializing memory, and secondary processes, which can attach to memory initialized by a primary process. Multiple primary processes can co-exist in a single system as long as they are using different hugepages. In this case different primary processes are not able to reach each other's resources. Secondary processes can only be started if there is primary process they can attach to. In case of crash of a primary process, secondary processes are still capable of accessing the memory allocated before the crash. However, secondary process cannot be converted to primary processes. Thus, no new memory allocations can be made as there is no primary process to allocate those.

One of the key features provided by DPDK, from container networking point of view, are poll mode drivers (PMD) for network interfaces, which provide significant performance boost compared to kernel drivers [17]. Kernel drivers are often based on interrupts, meaning that the interface sends an interrupt to the driver when new packets are available [65]. This causes overhead in network performance, but allows efficient computing from the scheduling point of view. To overcome this overhead, DPDK PMDs constantly poll the interface for new packets [64]. This allows better performance in terms of throughput and latency, but consumes much more computing resources [17]. PMD often consumes all processing time available at the core it is being run on.

DPDK libraries and drivers enable boosting the performance of an application running in Linux user-space. This allows implementing efficient VNFs and SDN on general purpose hardware and OS. An example of a software supporting performance optimization with DPDK is Open vSwitch (OVS).

3.2.2 Open vSwitch (OVS)

OVS is an OSS implementing a distributed multi-layer virtual switch [66]. The difference between OVS and standard linux bridge is the support for multi-server deployments; OVS provides cloud native implementation for VM or container networking [67]. OVS offers wide support for VM migrations from one host to another: both network configuration and live network state are migrated with the VM to the new host. OVS is widely used and the OVS kernel module has been part of Linux kernel since version 3.3.

The performance of OVS can be boosted by compiling and running it with DPDK [68]. This DPDK support has been available since OVS version 2.4.0. Before that there has been commercial solutions implementing similar features [16]. The performance of OVS with DPDK boosting shows an up to 12x speedup compared to the performance of native OVS [68].

OVS provides features for: security, monitoring, quality of service (QoS), and automated control [67]. Security features OVS implements include virtual local area network (VLAN) isolation and traffic filtering. Monitoring features include tools such as NetFlow. QoS service features include traffic queueing and shaping. Automated

control allows controlling of the OVS via OpenFlow and OVS database protocol.

A running OVS consists of two main processes: a daemon implementing the switch, ovs-vswitchd, and a database server responsible for providing the daemon the current state of the switch, ovsdb-server. In addition to these two processes, OVS provides tools for configuration and management of daemon as well as database processes and for building release binaries. OVS package also includes tools for management of OpenFlow switches and controllers. These tools are not only compatible with OVS but allow management of any OpenFlow compatible switch or controller, virtual or physical.

A typical use case for OVS, or any other virtual switch, is to provide networking for virtualized cloud infrastructure; virtual switches are typically used to divide limited physical resources of host machine for VMs that are deployed in it [69]. OVS is well suited for cloud networking because of its support for distributed switching. For example, OVS can be used to provide switching for OpenStack [70], an open source cloud computing software [71].

3.2.3 OpenStack

OpenStack [70] is an open source cloud computing software for controlling of computing, storage, and networking resources. It is used to convert datacenter resources into a cloud implementation. This allows controlling the resources in the datacenter through OpenStack dashboard or OpenStack APIs and allows exposing the resources to the user as a public, private or hybrid cloud.

When compared to DPDK and OVS, OpenStack is a higher-level solution; Whereas DPDK and OVS solve or implement quite simple tasks, OpenStack aims to provide a complete solution for cloud computing. OpenStack is built of many smaller projects, such as: Nova computing service, Cinder block storage service, Neutron networking service, Horizon web based management service, and many more [70]. In addition to these OpenStack based projects, many other OSS, such as OVS, is employed.

In addition to offering a cloud platform for general purpose computing, OpenStack provides a platform for VNFs [72]. The key enablers to allow this is the support for standardized interfaces and popular networking plug-ins as well as drivers. The OpenStack's capability for NFV has been proved by many companies in the telecommunications field [70].

Similarly than many other OSS, OpenStack is available both as a pure open source package released under Apache 2.0 licence and as a customized solutions offered by software vendors [70]. This thesis uses the pure open source version of OpenStack in the measurements.

OpenStack can be deployed to a wide variety of hardware setups, ranging from all-in-one installation to full-rack setup with multiple compute nodes [70]. There exists both open-source and commercial tools to deploy OpenStack automatically. In this thesis OpenStack is deployed with DevStack, a tool provided by OpenStack for developing and testing of OpenStack [71]. All-in-one setups are deployed for measurements with and without DPDK boosting in OVS. The OVS with DPDK datapath is deployed by enabling networking-ovs-dpdk plugin [73]. The configuration

files provided to DevStack scripts are presented in Appendix B.

Chapter summary

This chapter discussed the performance indicators used in evaluation of different containerization approaches to be proposed and introduced the main OSS that will be used in the measurements. The discussed indicators, latency and throughput, give good overview of the networking performance and can be measured with openly available software. The main OSS to be used in the measurements, in addition to Docker and Kubernetes that were already introduced in Chapter 2, consist of DPDK, OVS, and OpenStack.

4 Containerization architectures and measurements

To analyze the feasibility of adopting a combination of virtualization and containerization methods in a cloud RAN environment in terms of network performance and ICC this chapter proposes two containerization architectures with fast ICC capabilities. The two architectures are described in detail and their performance is measured in terms of throughput and latency.

The chapter is divided into two sections: First, Section 4.1 describes different containerization architectures proposed to implement isolation, scalability, and performance requirements. Second, Section 4.2 introduces the measurement setup and describes the measurement results.

4.1 Containerization architectures

One of the challenges when building user-plane application from containerized microservices is how to provide fast networking interfaces for each microservice. By default, Docker containers are connected to internet and each other via docker0 bridge, which is a normal Linux bridge [31]. This implementation depends on kernel networking stack, which is usually implemented with interrupt based drivers, and the performance is further affected by kernel scheduler.

One solution to increase the network performance is to use DPDK, which was described in previous chapter, and the PMDs it provides [68]. However, since containers share kernel with host OS, host's NICs and possibly other physical resources are also visible from inside the container. This will not be a problem if number of containers accessing the physical resources is strictly controlled. However, in cloud native applications one of the key design principles is to allow applications to scale up and down rapidly and freely. As briefly discussed in Section 2.3.1, load-balancing is often applied to divide input data from single source to multiple worker nodes. In DPDK accelerated container networking, this would be equivalent to one container being connected to physical NIC and providing networking or similar datapath for application containers.

Before going into the implementation, the general purpose system has to be prepared to be used with DPDK. This means optimizing the kernel parameters, reserving and mounting hugepages, as well as binding network interfaces to DPDK compatible drivers. First, kernel parameters have to be optimized for DPDK usage. This includes activating memory management tools, isolating physical cores from kernel scheduling, and reserving hugepages memory. For example, in CentOS this could be done with grubby tool:

```
# grubby --update-kernel=ALL --args="iommu=pt intel_iommu=on"
# grubby --update-kernel=ALL --args="isolcpus=2-7"
# grubby --update-kernel=ALL --args="default_hugepagesz=1G
   hugepagesz=1G hugepages=8"
# reboot 0
```

where the first command activates input output memory management unit (IOMMU) [64], the second command isolates cores 2–7 from kernel scheduling [14], and the

third command sets the default hugepage size to one Gigabyte as well as reserves eight 1GB hugepages at boot time [64]. Reboot is required to activate these changes. In addition to or instead of 1 GB hugepages smaller 2 MB hugepages can be reserved. The smaller hugepages can also be reserved at run time. The hugepages are then mounted to the system with the commands:

```
# mkdir -p /mnt/huge
# mount -t hugetlbfs nodev /mnt/huge/
```

where the first command creates the directory for the mounting point and the second command actually mounts the hugepage tables file system to the system [65]. Finally, the network interfaces to be controlled by DPDK PMDs are bound with dpdk-devbind.py utility:

```
# ./${RTE_SDK}/usertools/dpdk-devbind.py -b \
    ${driver_name} ${pci_address}
```

This command binds the specified interfaces to specified driver. Meaning that it can be used to both reserve or release interfaces to or from DPDK PMD usage.

Although these preparations are most natural to be done from the host environment as the root user, these commands could also be entered from a container with a privileged access to the host system. However, it is considered a bad practice to do modifications to the host system from inside a container as that would introduce an increased risk of security vulnerabilities to the product. After these preparations, a DPDK process running in a container with correctly configured access rights is able to occupy the resources required and to implement a containerized VNF with DPDK accelerated datapath. The access rights given to the container can be further limited with well-designed mounting points and extended usage of cgroups [74].

This section discusses two different approaches for DPDK accelerated communications: OVS networking with DPDK datapath and shared memory based IPC. OVS with DPDK datapath is commonly used in OpenStack networking, but its support for container networking through virtio-user virtual device is relatively new feature [64]. This virtio-user PMD is connected to vhost-user backend thus comprising a para-virtualized networking interface. The communication in this interface is based on shared memory. A pure shared memory based IPC in container communications provides more lightweight implementation for the datapath, but lacks the standardized networking interfaces. When the the processes communicating with each other are in different containers the IPC is often referred as ICC.

First, in OVS with DPDK datapath the fast ICC is based on networking over vhost-virtio port pairs. The implementations for these ports and their drivers are provided by DPDK libraries. The creation of vhost ports is done via OVS interfaces and creation of virtio port by DPDK libraries at the start time of DPDK process. Similarly to other DPDK drivers, vhost and virtio drivers are implemented as PMDs, meaning that the interface is constantly being polled for new data. This results to, ideally, 100% usage of a processor core. To allow efficient usage, PMD processes are usually mapped to specific cores and excluded from the kernel scheduler [64]. This approach is illustrated in Figure 11. Note that the figure is simplified so that only DPDK boosted connection and port related to these are visible. In actual

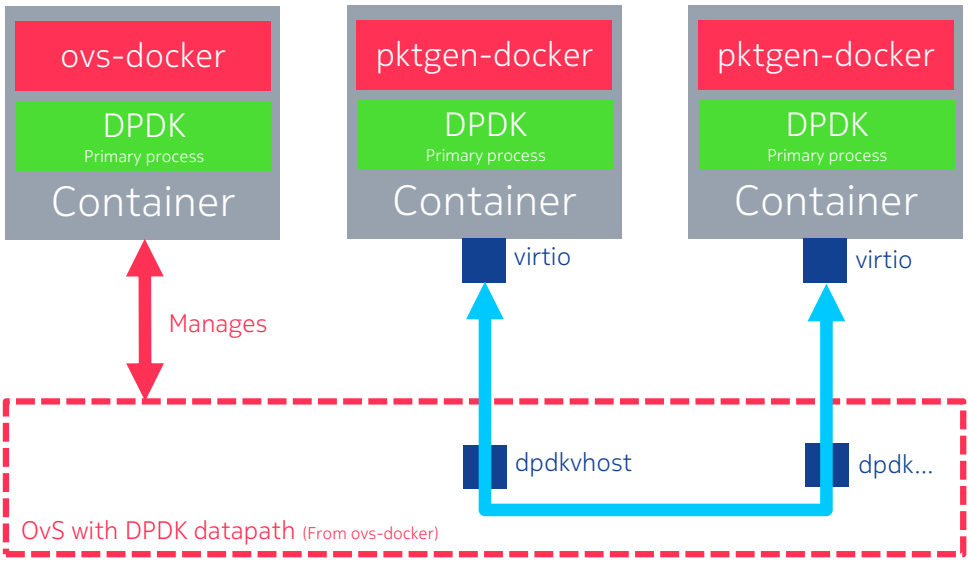


Figure 11: OVS networking with DPDK datapath illustrated.

implementation, other networking would likely also be implemented to allow remote management and monitoring of the application containers.

The first step in creating a set of containers communicating through OVS with DPDK datapath is to launch the OVS. The OVS can be launched as a native process or inside a container. OVS in a Docker container is launched with command:

```
$ docker run \
    --privileged \
    -v /mnt/huge:/mnt/huge \
    -v /usr/local/etc/openvswitch/:/usr/local/etc/openvswitch/ \
    -v /usr/local/var/run/openvswitch/:/usr/local/var/run/
       openvswitch/ \
    -v /usr/local/var/log/openvswitch/:/usr/local/var/log/
       openvswitch/ \
    --net=host \
    --name=ovs \
    -d ovs-docker;
```

where `--privileged` option allows running the container with privileged rights, `-v` flag mounts host directories to container, `--net` option selects the networking for container, `--name` option sets the name for container, and `-d` flag set the container to run in detached mode. The image used is ovs-docker, which Dockerfile is provided in Appendix A. This command mounts four directories to container: default hugepage mount, default OVS database location, default OVS socket location, and default OVS log location to container. In addition, if OVS is connected to network interface via DPDK PMD additional mounts might be required depending on the driver type. The command also sets the container to use host network to make it easier to create networks for other containers running on the host. The OVS could also be run directly on the host system.

The ovs-docker container contains a bash script, included in Appendix A, as an entrypoint to launch the ovsdb-server and ovs-vswitchd processes. The bash script is

responsible of passing further parameters, including the DPDK parameters, such as, core mask, socket memory, and file prefix, to these OVS processes. As the container is launched with host networking and mounted with OVS default directories, controlling of the OVS can be done from the host as if the processes were running natively on the host.

The Docker containers connected to the OVS can be connected via either network namespaces or via DPDK virtual device. However, only DPDK virtual device provides support for interface with DPDK PMDs. In both cases an OVS bridge has to be created in order to be able to provide OVS networking for the containers. This is done with the `ovs-vsctl` tool. For example, command to create bridge named br0 would be

```
$ ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev;
```

where the part of the command after `--` selects the datapath type for the bridge. If DPDK virtual devices are used to connect the containers, the bridge has to be configured to use user-space datapath instead of the kernel datapath. If the connection is made via networking namespaces, `ovs-docker` utility can be used to create connection from the bridge to the container and do all the necessary configurations. The main difference between these connection types is that networking namespaces approach can be used for general purpose networking while DPDK virtual device based connection is limited to usage of a single DPDK process only.

In order to use DPDK virtual device from inside a container, a port of dpdkvhostuser type has to be created to the bridge with `ovs-vsctl` tool. For example the command to add port vhost_user1 to bridge br0 would be

```
$ ovs-vsctl add-port br0 vhost_user1 -- \
    set Interface vhost_user1 type=dpdkvhostuser;
```

where the part of the command after `--` selects the type of the port. OpenFlow rules to direct packets from port to another can be created with `ovs-ofctl` tool. For example, command to create direct flow for all ethernet packets in bridge br0 from port one to port two would be

```
$ ovs-ofctl add-flow br0 \
    in_port=2,dl_type=0x800,idle_timeout=0,action=output:1;
```

where the port numbers used are OpenFlow port indexes associated to the ports. The unix socket associated with this port has to be mounted to docker container, and a virtual device of type virtiouser connected to this unix socket has to be created with `--vdev` option of the DPDK EAL. For example, assuming that one port of dpdkvhostuser type has been created, pktgen-docker with one DPDK virtual device port is started with command

```
$ docker run \
    --rm \
    --privileged \
    --net=none \
    -v /usr/local/var/run/openvswitch/vhost_user1:/var/run/vh1 \
    -v /mnt/huge:/mnt/huge \
    --name pktgen -it \
    pktgen-docker \
```

```
-c 0xC -n 4 -m 1024 --file-prefix=pktgen_ \
--vdev=net_virtio_user0,path=/var/run/vh1 -- \
-T -P -m "3.0";
```

where `--rm` option sets the container to be deleted after it is completed, the first mount, `.../vhost_user1`, is the socket file created by OVS at port creation time, `-c 0xC` is core mask defining which cores DPDK should occupy, `-n 4` defines number of memory channels used by DPDK, `-m 1024` defines how many megabytes of memory DPDK should reserve, `--vdev=...` defines virtual device for virtio driver with path to socket mounted to the container, `-file-prefix=pktgen_` defines file prefix DPDK uses for lock and configuration files of the process to be started, `--` splits the arguments to those passed to DPDK EAL and to those passed to pktgen, `-T` option enables colored output, `-P` enables promiscuous mode for ports, and `-m "3.0"` defines core to port mappings.

The DPDK virtual device implements similar interface that is used with VMs. This means that the level of the virtualization for the networking is higher than with the default networking namespaces based implementation. This increase in the level of virtualization might be visible in the performance. Nevertheless, usage of virtual devices and DPDK PMDs might provide possibilities for more simple bridge configurations or increased efficiency when working with physical NICs. In order to maximize performance of the ICC, implementation closer to IPC, such as shared memory, could be better than networking.

Second, DPDK shared memory based ICC relies on memory shared between primary and secondary DPDK processes. In this approach, memory allocation and data structures are provided by DPDK. Sharing memory, and other DPDK resources, between containers requires introducing common DPDK configuration and lock files for all containers accessing the same resources. This is easiest done by exposing same directory from host to all containers sharing same resources. Similarly than with DPDK network drivers, memory interfaces could be polled for new data constantly. However, the memory interfaces don't have common standards suitable for ICC the same way the network interfaces do. This means that, for example, available open source solutions might not be compatible with each other. This approach is illustrated in Figure 12. Note that here only one of the DPDK processes is a DPDK primary process. Only this primary process is able to reserve and release memory. Secondary processes, here mem-tester containers, are dependent of the primary process and cannot allocate new memory; if the primary process is lost, the secondary processes might be required to restart depending on the shared memory usage patterns.

Similarly as with OVS implementation, the container in responsibility of managing the memory has to be started first. There are two common approaches to implement the shared memory based ICC: peer to peer and client-server models. From Dockers perspective there is not much difference, both scenarios require similar mounts and docker options. The difference between these implementations is visible in the software implementation and parameters passed to the container entrypoint. An example container for shared memory ICC, shm-docker that is not implemented within this thesis, would be launched with a command similar to:

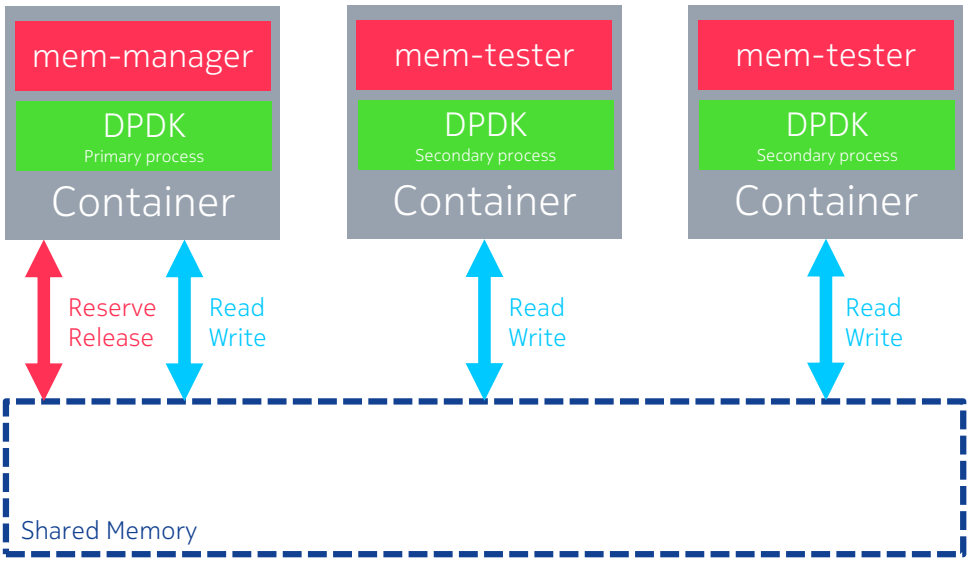


Figure 12: DPDK shared memory based ICC illustrated.

```
$ docker run \
    --privileged \
    -v /mnt/huge:/mnt/huge \
    -v /tmp/shm-docker/var/run/:/var/run/ \
    --name=shm \
    -d shm-docker ${shm-docker-arguments};
```

Again, similarly than ovs-docker, the container is launched with privileged rights and default hugepages mount location mounted. In addition, `/var/run/` directory, which is the default location for DPDK configuration and lock files, is mounted from host system to container. This same host directory should be mounted to all containers that require access to same shared memory resources as `/var/run/` to allow DPDK processes in the container to communicate with each other. If communication with host DPDK processes is not required any host directory can be mounted to containers as `/var/run/` to avoid mounting unnecessary files. Alternatively, DPDK files could be directed to other directory and this directory mounted to containers.

The clients or peers of the primary process are started similarly. The main difference in arguments is the process type passed to DPDK EAL with `--proc-type` argument. The client instance or joining peer instances have to have process type set to auto or secondary in order for them to be possible to attach to the primary process with the same file-prefix. The difference between peer and client programs comes from the software implementation. If the same binary is executed in multiple instances with detection of process type, the instances are peers of each other. If there is separate server process started first from different binary, the instances to be attached are clients of this server. The arguments passed to the process also depend on this process type.

The main difference between these approaches, OVS networking and shared memory based ICC, is the compatibility for multi-host setup. While OVS has great built-in support for creating networking spanning multiple physical machines, DPDK

shared memory based IPC is limited to providing communication within one physical machine at a time due to requiring containers to have access to same memory regions. This is not necessarily a limitation, since the expected performance boost provided by shared memory might be more important than possibility to directly communicate with containers on remote hosts. After all, these two approaches essentially solve different problems. While both provide solutions for ICC, shared memory is purely for IPC and OVS for communications via networking. For example, when implementing a microservice in a Kubernetes pod the communications to and from pod could be done with OVS and ICC within a pod with DPDK shared memory.

These approaches, OVS with DPDK datapath and shared memory based inter-process communications, have in common that much host resources has to be exposed to the containers for the solutions to work. Both solutions rely on hugepages to get better memory performance. The usage of hugepages requires running the container with privileged rights and access to hugepages mount directory on the host machine. Also, since DPDK is used in both approaches containers and processes have to be mapped to specific processor cores to allow efficient usage of computing resources. This core mapping will most likely cause additional work with Kubernetes and scaling implementations. This exposing of host resources and specific requirements for host resource utilization violate the idea of containers working as isolated and independent pieces of an application. Nevertheless, trade-off has to be made between isolation and performance in order to meet the performance requirements of user-plane applications with the current implementations of OSS available.

Technologies such as Kubernetes pods and VMs in OpenStack cloud can be used to overcome challenges caused by this decreasing isolation. The approaches and solutions provided by these technologies are different. While VMs and OpenStack would bring back the isolation by implementing stronger virtualization, Kubernetes would rely on containers and automated resource management. The trade-off is, again, between performance and isolation.

Both of these technologies, Kubernetes and OpenStack provide support for the approaches discussed in this section. However, with both technologies the usage of shared memory based ICC is somewhat limited. In OpenStack the memory can be shared only within as VM and in Kubernetes the memory can be shared only within a pod. Nevertheless, these limitations comply with the idea of isolating independent pieces of software. Both technologies support networking both to and within a pod or a VM with OVS. Openstack provides support for OVS networking with DPDK datapath through networking-ovs-dpdk plugin [73] and OVS provides support for Kubernetes through ovn-kubernetes plugin [75]. The plugins can be used to provide networking to the pod or the VM. If OVS networking inside a pod or a VM is required, an OVS can be run inside the pod or the VM as described earlier in this section. The actual implementations with Kubernetes or OpenStack are left for future work.

The containerization architectures presented in this section are designed based on several assumptions on the performance differences of the applied technologies. For networking, it has been assumed that OVS provides better performance and manageability than the default linux bridge and that the performance of OVS is

Table 1: The versions of the main OSS software applied in the measurements.

Software	Version
Ubuntu server	16.04 Xenial
Docker	1.12.6
DPDK	17.05.1
Open vSwitch	2.7.0 (with patches)
DevStack	stable/ocata

further increased by using DPDK PMDs. It has been also assumed that shared memory provides more efficient ICC and IPC at the cost of worse standardization of the interfaces. These performance related assumptions are to be verified through measurements.

4.2 Measurements

To support the analysis of Section 4.1, measurements are conducted. The goal of the measurements is to compare performance of the proposed networking and ICC methods in terms of the indicators presented in Section 3.1. Measuring the performance of a complete implementation or a product like system is not in the scope of this thesis.

4.2.1 Measurement setup

The measurements are performed with a VM operated in private cloud. The VM used has 16 virtual CPUs and 32 GB of memory. Further details of the VM are omitted from this thesis, as they are not strictly necessary in the process this thesis applies in the comparison of the proposed approaches. The measurements with physical NICs are performed with 4 core desktop computers that have 16 GBs of memory and two NICs: one for management connections and one Intel 82574L Gigabit Network Controller for performance measurements.

The OSS software applied in the measurements are listed in Table 1 with version information. The versions listed in table are used in both native and containerized applications. In physical machine Ubuntu server was installed via installation virtual media where as in VMs cloud base image was used. In both cases generic version of kernel is used. The Dockerfiles to build containers used in the measurements are provided in Appendix A.

Four types of measurements are conducted: unix bridge compared to OVS bridge in container networking, OVS networking with native datapath compared to OVS networking with DPDK datapath in VM networking, performance of OVS in container

Table 2: The versions of the measurement tools applied in the measurements.

Software	Version
DPDK-pktgen	3.3.9
iperf	2.0.9
iperf3	3.0.11

and VM networking, and DPDK boosted networking with physical interfaces from native application compared to containerized application.

First, performance of DPDK OVS bridge without DPDK PMDs boosting is compared to normal Linux bridge. Measurements are performed with iperf3 and iperf softwares [63] acquired in image from Docker hub [76] and build from Dockerfile listed in Appendix A, respectively. OVS bridge is created with `ovs-vsctl` tool and interfaces to Docker containers, started with no networking, are created and attached with `ovs-docker` tool. Unix bridge is created by Docker engine at Docker daemon start time and Docker containers are attached to this bridge at container instance start time when no networking settings are given. Throughput is measured with unlimited Transmission Control Protocol (TCP) connection and latency with 1 Mbit/s User Datagram Protocol (UDP) connection. In both cases the connections is open for 60 seconds to get a view on the average performance.

To continue step-by-step in evaluating the gained performance, OVS bridge performance is measured from VM to VM with and without DPDK boosting. The measurements are conducted with VMs as in containers the support for DPDK boosted connection would require DPDK process to connect to the DPDK interface. Thus, by using VMs instead of containers here same measurement tool can be used in both cases. The measurements are conducted in OpenStack installation booted with DevStack following instructions for single VM installation [71]. In DPDK boosted measurement case, installation is modified to comply also with networking-ovs-dpdk instructions [73]. In measurements two VMs are created to OpenStack cloud and connected to each other and to external networks via single internal network. The connection between VM and OVS in implemented with vhost-virtio pair. This measurement setup is illustrated in Figure 13. Note that here same network is used for both management of the instances and the performance measurements.

In these two first cases, the measurements are performed with iperf and iperf3 network performance measurement tools. Although both are based on original iperf, these tools provide support for different measurement scenarios. In these measurements iperf3 is used to measure throughput and iperf to measure latency. With both software one instance acts as a server and other as a client. In latency measurements two server-client pairs are executed simultaneously and latency is calculated as described in Equations (2) and (3) presented in Section 3.1. Servers are started with commands:

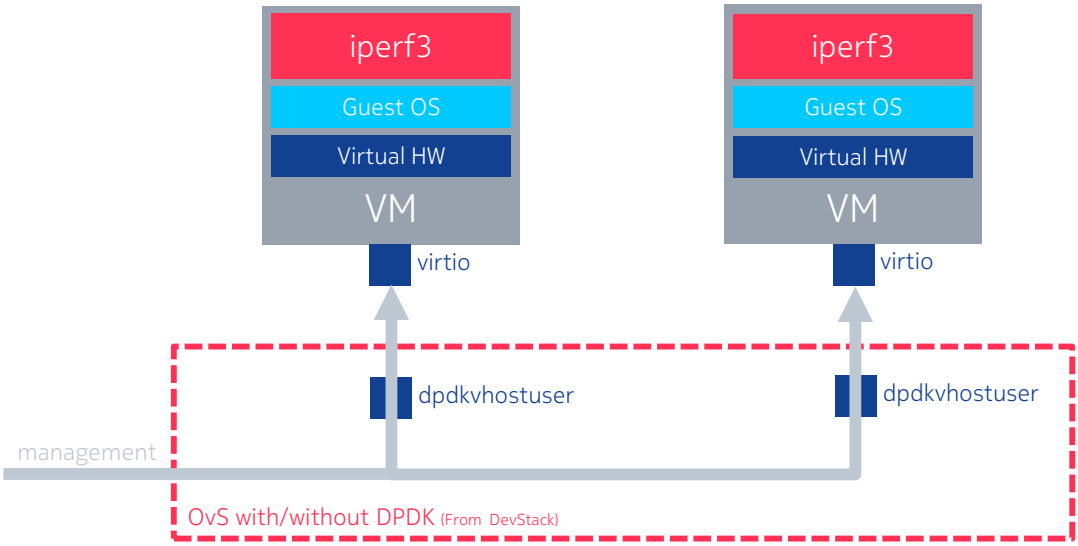


Figure 13: Measurement setup for VM to VM connection using OVS bridge to compare network performance with and without DPDK.

```
$ iperf3 -s;
$ iperf -sue;
```

where `-s` stands for server, `-u` for udp traffic, and `-e` for extended output. The clients are executed with commands:

```
$ iperf3 -t 60 -c ${server_ip};
$ iperf -ut 60 -c ${server_ip};
```

where `-t` stands for test execution time which takes time in seconds as a parameter, and `-c` for client which takes server ip as a parameter. When iperf or iperf3 are run from inside a container, the options for iperf and iperf3 stay the same. The containers are started as interactive containers, with default or no networking depending on the measurement case.

In order to obtain information on how DPDK PMDs affect the network performance when enabled on both sides of the interface, performance is measured for different container and VM networking cases. The cases to be compared are from container to container, from VM to VM, and from container in VM to container in VM. The containers are started as described in Section 4.1 and the VMs are started through OpenStack commandline interface. The measurement are conducted with DPDK pktgen. Two of these measurement cases are illustrated in Figure 14 and container to container case is similar to one illustrated in Figure 11. The illustrated cases are for throughput measurements. Latency measurements are conducted so that both sending and receiving is done from same container or VM. In this case VM or container will have two network interfaces bound to DPDK PMDs. This is because DPDK pktgen requires that same core is responsible for both sending and receiving the packets to avoid time synchronization issues. Note that in the figures VMs are connected to separate networks via two interfaces. Management and measurement networks have to be now separated, because after binding an interface

to DPDK it is no longer available to kernel and, for example, ssh connection via that interface is not possible to the VM. The management network and connections are not necessarily required as the measurement can be performed automatically, but the management connections are useful for debugging and development purposes.

Finally, to verify that containers can be used also to communicate with physical interfaces without significant decrease in performance, native application is compared to containerized one. The application used is DPDK pktgen, similarly than in the previous measurement case. In contrast to previous measurements, this measurement is performed with physical NICs. The measurement setup is similar to one illustrated in Figure 14 given that instead of VMs physical machines are used, instead of dpdkvhostuser ports physical NICs are used, and the networking isn't provided by DevStack or OVS.

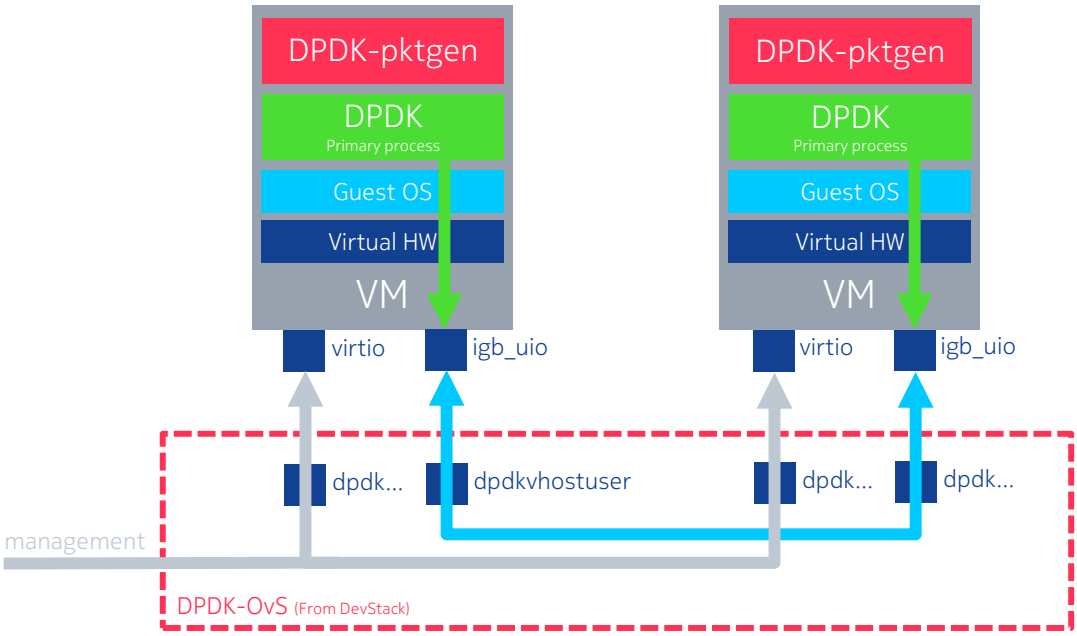
In these two last cases, the measurements are performed with DPDK pktgen software. The pktgen is used to both send and receive the packets. Both throughput and latency are measured with this tool. As DPDK pktgen used processor clock to measure latency, the latency measurements have to performed in such way that packet are sent and received from the same core. The pktgen software is started with commands:

```
$ ${pktgen} -c 0xC -n 4 -m 1024 --file-prefix=pktgen_ -- \
    -T -P -m "3.0";
$ ${pktgen} -c 0xC -n 4 -m 1024 --file-prefix=pktgen_ -- \
    -T -P -m "3.[0-1]";
```

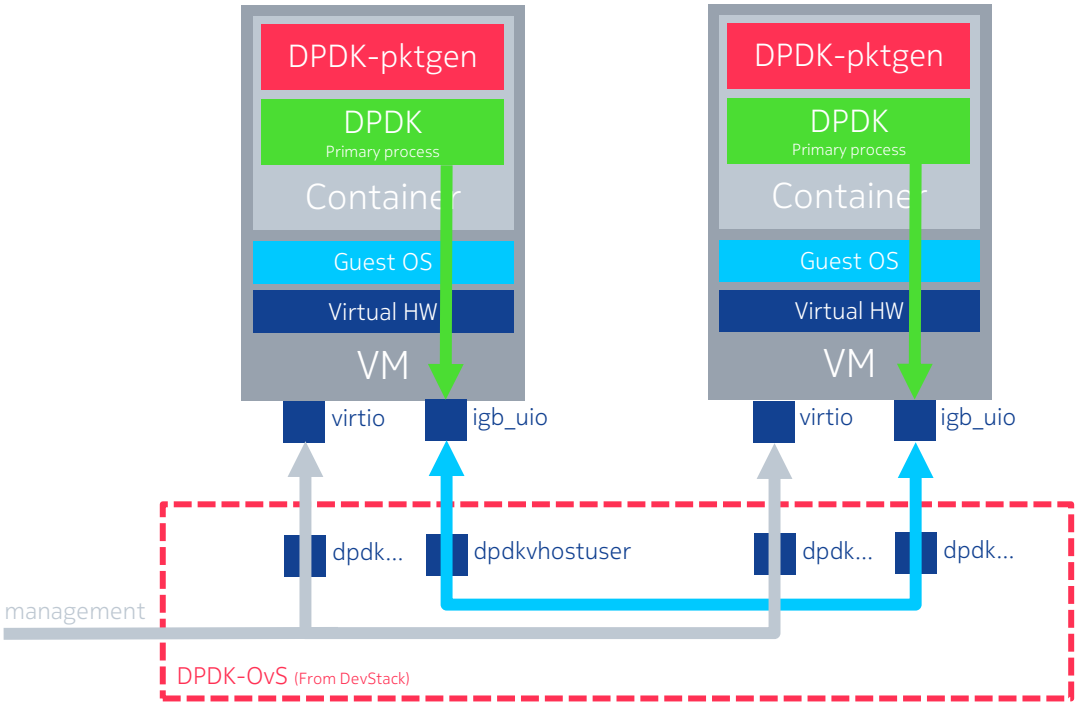
where `${pktgen}` is the path to pktgen binary. The first command is used for throughput measurements and second command for latency measurements. The software itself is controlled as described in [64]. When DPDK pktgen is run in a container the pktgen options stay the same. The container is started as interactive container with privileged rights, without networking, and with `/mnt/huge/` mounted for hugepages access.

To conclude, the measurements will proceed step-by-step from default Docker networking to comparing DPDK boosted communications in VM and container networking. It is expected that the performance will increase as the measurements proceed: OVS bridge should provide better performance than linux bridge, and OVS bridge should perform better with than without DPDK boosting. Finally, containerized application should not perform significantly differently when compared to a native application.

The measurements introduced here focus on ICC inside a single host. While from VM to VM and from container in VM to container in VM measurements provide some insight on how the performance will change when more networking and virtualization is introduced, more thorough measurements with multiple physical host involved would be necessary to evaluate the performance of an actual product. Those measurements, however, are not in the scope of this thesis. Instead they will be part of the further feasibility evaluations.



(a)



(b)

Figure 14: Two of the OVS with DPDK datapath measurements illustrated: VM to VM (a) and container in VM to container in VM (b).

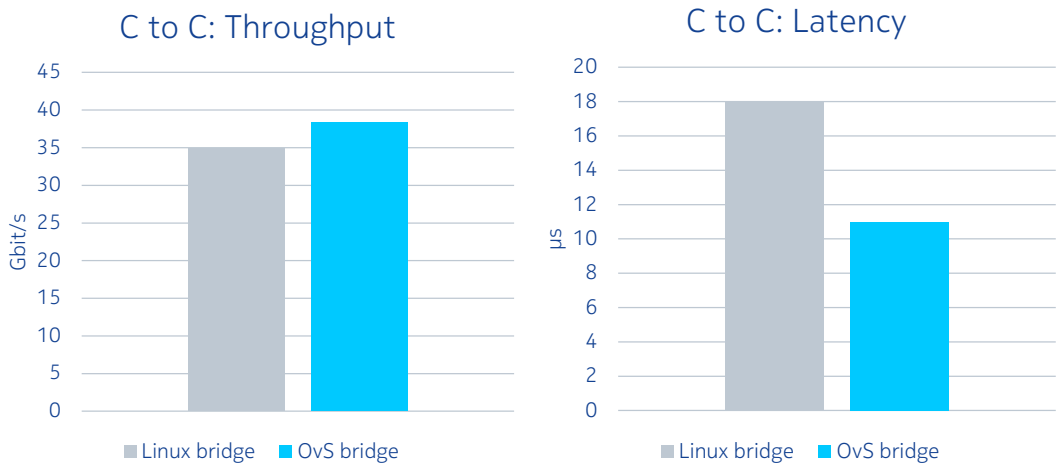


Figure 15: The results of performance, throughput (a) and latency (b), comparison of OVS bridge and Linux bridge in container (C) networking.

4.2.2 Measurement results

The measurement results are presented in the same order than the measurement cases were introduced: First, unix bridge compared to OVS bridge in container networking; OVS networking with native datapath compared to OVS networking with DPDK datapath in VM networking; performance of OVS in container and VM networking; and finally, DPDK boosted networking with physical interfaces from native application compared to containerized application.

First, the results from performance measurements of OVS bridge without DPDK PMDs and Linux bridge are illustrated in Figure 15. The figure shows that with OVS bridge, even without the DPDK boosting, throughput shows an improvement of about 10 percent and latency shows a decrease of about 60 percent. Furthermore, the additional benefits from taking OVS into use are the multi-host capabilities and increased configurability of OVS. However, for most applications the ease of use of the default Docker networking will be worth more than the improvement in performance.

The results from OVS bridge performance comparison with and without DPDK boosting are illustrated in Figure 16. From the figure, it can be seen that the throughput is significantly better and latency much shorter, with DPDK enabled. This is expected behavior as PMDs provided by DPDK libraries should provide better performance at a cost of larger CPU time and memory usage. However, at this point DPDK PMDs are only used at the OVS side of the interface; While vhost-port at the bridge is implemented with DPDK PMD, the virtio-port at the VM is using a driver provided by the guest kernel. Further increase in performance is expected when both sides of the interface are using DPDK PMDs in the ports. In addition, introducing an OpenFlow rule to create direct flow of traffic from sending to receiving port does not provide different results.

These two first measurement cases also illustrate the virtualization overheads

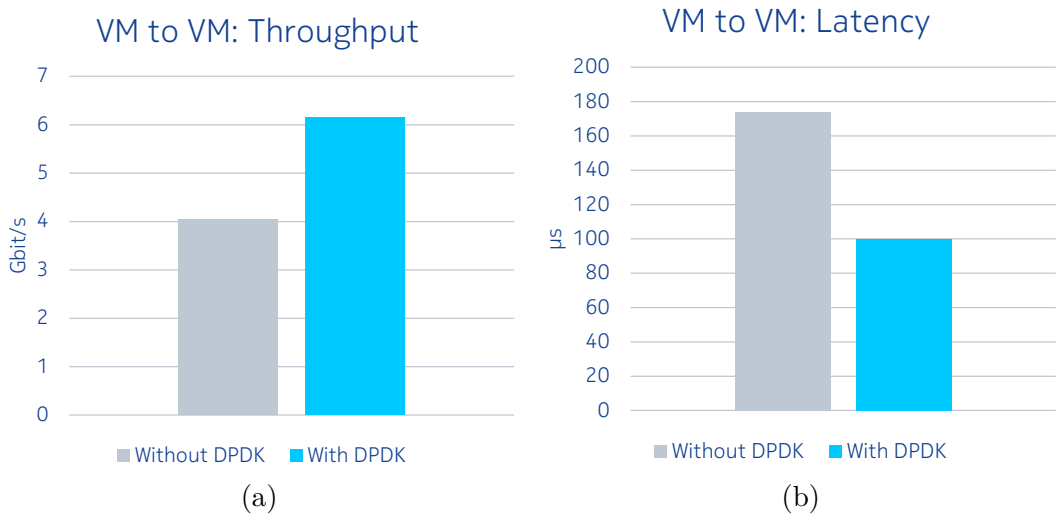
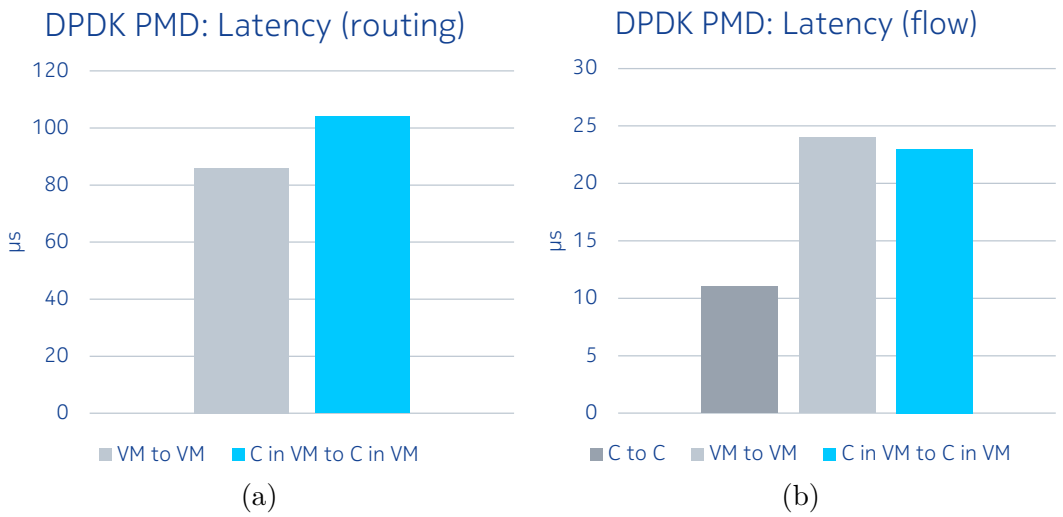


Figure 16: The results of performance, throughput (a) and latency (b), comparison of OVS bridge with and without DPDK boosting in VM networking.

present in OS level and hardware level virtualizations. While an increase in performance is seen for VM to VM connection when using DPDK boosting with OVS, the throughput is far from that of container to container measurements. This is due to heavier virtualization of the network interface in the VM. Similarly, the latencies are several times shorter with containers than with VMs.

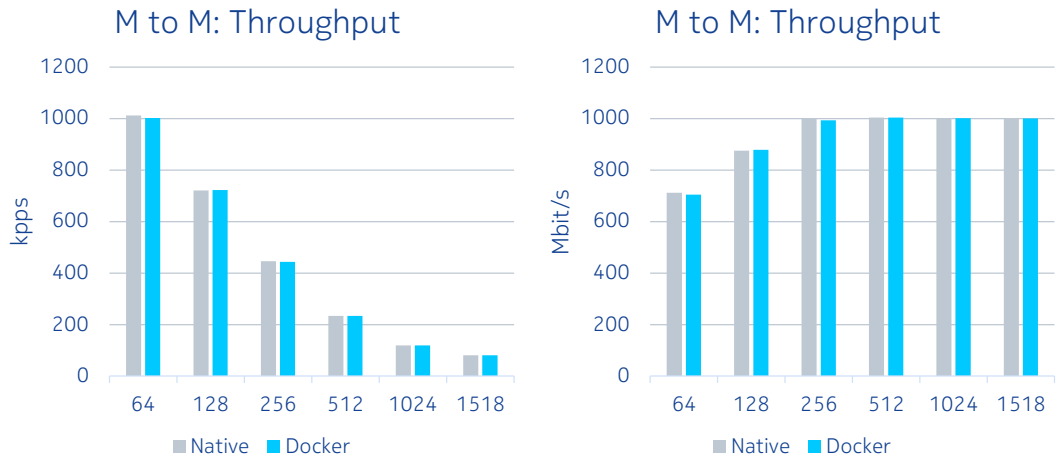
The results for OVS bridge networking in VM and container networking are illustrated in Figure 17. Unfortunately, the measurement connection was not stable enough for reliable throughput measurements due to problems most likely in the measurement configuration or the used drivers. The pktgen software showed a very high packet loss with many different configurations. For this reason, only latency results are presented for this measurement case. The Figure 17a shows that the guest side DPDK PMD does not provide considerable performance gain with VMs, compared to latencies shown in Figure 16b, when using routing provided by OVS to direct packets. Nevertheless, from the Figure 17b it can be seen that significant performance gain can be achieved with VMs by using a direct flow between the VMs. However, with containers the latency is the same than without the DPDK PMDs even when using direct flows. It seems that the heavier virtualization of the networking interface can not outperform the networking provided by OVS kernel datapath.

Finally, the results from comparison between native and containerized application for communication through physical NIC are presented in Figure 18. From the figure, it can be seen that there is no significant difference in performance between native and containerized DPDK application. Containerized application shows only slightly worse performance with smallest packet size. However, it has to be noted that in this case small differences could also be due to limitations in the measurement hardware. The used desktop computers are not able to reach the line rate of NICs with the smallest packet size. It will be left for future work to replicate these measurements



(a) (b)

Figure 17: The results of latency comparison, with routing (a) and flow configurations (b), of OVS bridge in VM and container networking.



(a) (b)

Figure 18: The results of throughput, in kpps (a) and Mbit/s (b), comparison of native and containerized application accessing a physical NIC.

with more suitable NICs and carrier grade servers.

The measurement results presented here confirm the assumptions made earlier with two exceptions: for inter-container communications the DPDK PMDs do not provide better performance in terms of latency, and for inter-VM communications the performance gain, in terms of latency, is only visible when direct flow is defined between the measurement ports. The measurement results are similar to those already available in the literature: container technologies perform better than virtualization technologies and the performance of the containers is close to that of a similar native process [18],[29],[30],[61].

It has to be noted that results from different measurements cases presented here are not directly comparable as the measurements have been conducted in slightly

different setups with different test tools. Nevertheless, the measurements results are suitable for evaluating the feasibility of containerization architectures and ICC approaches in user-plane applications. Further, the measurement results provide base line for making decisions on which architectures and approaches should be tested in more product like environment.

Chapter summary

This chapter proposed different approaches for ICC, implemented using OSS such as Docker, DPDK, OVS, and OpenStack. These approaches were analyzed and measured to effectively compare the performance achieved. Based on both analysis and performance measurements, it seems that the DPDK and OVS can be used to boost the networking and ICC performance. In addition, it is expected that performance could be further enhanced by implementing a shared memory based ICC. Combination of the proposed approaches will in the future be implemented to a more product like environment and its feasibility for usage in cloud RAN will be further evaluated.

5 Conclusions

This thesis summarized the fundamentals of virtualization, containers, and cloud computing, and discussed the transformation from traditional RAN to cloud RAN in terms of software containerization. The thesis introduced the OSS that have been commonly used to implement VNFs. This thesis proposed two approaches, OVS with DPDK boosted datapath and shared memory based ICC, for fast ICC based on the containerization discussion. These approaches were compared through discussion and measurements.

The proposed containerization approaches seem suitable to be used as building blocks of cloud RAN software. These proposed approaches are compatible with microservices architecture and with DevOps practices, thus offering a cloud native solution for cloud RAN software development. The performance measurements show promising results even with the issues in vhost-virtio connections. Nevertheless, the vhost-virtio interfaces for containers though DPDK virtual devices are relatively new technology and there are multiple organizations in the open source community developing approaches for DPDK boosted container networking. It is expected, that the DPDK virtual device support will be further developed.

The aim of this thesis was to evaluate the current and future feasibility of adopting a combination of virtualization and containerization methods in a cloud RAN environment in terms of network performance and ICC. From the measurements, it can be concluded that the proposed approaches are already suitable for most applications in terms of network performance in ICC and the network performance with physical NICs compared to native application. In measurements, containers showed significantly better performance than VMs with the performance of containerized application being similar or only slightly worse than that of native application. Further, although the performance of VMs was improved by using DPDK PMDs and OpenFlow rules, the performance provided by VMs was still worse than that of containers. However, the problems in implementing the networking with DPDK virtual devices through vhost-virtio interfaces might indicate that the technology is not yet mature enough to be used in applications with carrier grade availability requirements. Nevertheless, the active open source community around the related solutions should ensure future feasibility.

Since the aim of this thesis has been met and the inspected technologies are found to be feasible to be used in cloud RAN user-plane applications the work continues towards an actual implementation. The rest of this chapter will discuss the topics defined to be out of the scope of this thesis as well as items left for future work and suggest directions for future work.

Future work

This section discusses the items mentioned in this thesis but either defined to be out of the scope of this thesis or left for future work. In addition to the shared memory based ICC described in Section 4.1, these items include further containerization architecture designing with orchestration provided by Kubernetes or OpenStack; Implementing

more strict security and access rights for the containers; Further measurements to with these new architecture designs and high performance equipment; and implementing the proposed containerization architecture to an actual MN component as well as evaluating the performance of this containerized component against traditional one. Based on this discussion, this section suggests directions for future work.

To use containers as the building blocks of a user-plane application in cloud RAN, the orchestration of containers has to be well designed. While this thesis discussed the means of ICC between containers and container networking, there are still questions on how containers with DPDK processes can be orchestrated with Kubernetes. Problems might rise especially in automatic occupying of hardware resources, such as CPU cores or NICs. There are also decisions to be made on the role of VMs and OpenStack in the containerization architecture related to the trade-off between isolation and performance.

The access from to container to the host system should be limited to minimum. Within this thesis the containers have been run with privileged rights and with wide access to host hugepages and other resources. Similar approach cannot be used in a product because of the security risks involved. The containerization architecture and host system should be further developed to avoid any unnecessary access to the host resources.

Further measurements are required to further analyze the required level of performance boosting in the container networking. The single host and host to host measurements presented in this thesis should be extended to cover more combinations of possible container networking and ICC scenarios. The measurements should be combined and further developed to a single suite, possibly by using some test automation framework, that automatically analyzes multiple scenarios at once and provides concise performance report. This further development of the performance test would allow automated performance evaluation of current or future microservices.

Finally, the proposed architectures should be implemented to an actual MN component or similar system to verify the feasibility to use the proposed architectures in a user-plane application in cloud RAN.

References

[1] A. Osseiran *et al.*, "Scenarios for 5g mobile and wireless communications: the vision of the metis project," *IEEE Commun. Mag.*, vol. 52, no. 5, pp. 26–35, May 2014.

[2] A. Checko *et al.*, "Cloud RAN for mobile networks – a technology overview," *IEEE Commun. Surveys Tutorials*, vol. 17, no. 1, pp. 405–426, Firstquarter 2015.

[3] A. Balalaie *et al.*, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Softw.*, vol. 33, no. 3, pp. 42–52, May 2016.

[4] L. Bass *et al.*, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[5] J. Roche, "Adopting DevOps practices in quality assurance," *Commun. ACM*, vol. 56, no. 11, pp. 38–43, Nov. 2013.

[6] J. Thönes, "Microservices," *IEEE Softw.*, vol. 32, no. 1, pp. 116–116, Jan 2015.

[7] Docker - Build, Ship, and Run Any App, Anywhere. [Online] Available at https://www.docker.com/. [Accessed: 3.2.2017].

[8] J. Stubbs *et al.*, "Distributed systems of microservices using docker and serfnode," in *2015 7th Int. Workshop Sci. Gateways*, June 2015, pp. 34–39.

[9] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[10] M. Armbrust *et al.*, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[11] X. Zhiqun *et al.*, "Emerging of telco cloud," *China Commun.*, vol. 10, no. 6, pp. 79–85, June 2013.

[12] P. Bosch *et al.*, "Telco clouds and virtual telco: Consolidation, convergence, and beyond," in *12th IFIP/IEEE Int. Symp. Integrated Network Manage. and Workshops*, May 2011, pp. 982–988.

[13] J. Soares *et al.*, "Toward a telco cloud environment for service functions," *IEEE Commun. Mag.*, vol. 53, no. 2, pp. 98–106, Feb 2015.

[14] H. Masutani *et al.*, "Requirements and design of flexible NFV network infrastructure node leveraging SDN/OpenFlow," in *2014 Int. Conf. on Opt. Network Design and Modeling*, May 2014, pp. 258–263.

[15] Z. Zhu *et al.*, "Virtual base station pool: Towards a wireless network cloud for radio access networks," in *Proc. 8th ACM Int. Conf. Computing Frontiers*. New York, NY, USA: ACM, 2011, pp. 34:1–34:10.

[16] "Practical implementation of SDN & NFV in the WAN," Wind River, Tech. Rep., Oct. 2013.

[17] DPDK. [Online] Available at http://dpdk.org. [Accessed: 3.2.2017].

[18] J. P. Walters *et al.*, "A comparison of virtualization technologies for HPC," in *22nd Int. Conf. Advanced Inform. Networking and Applicat.*, March 2008, pp. 861–868.

[19] J. Wettinger *et al.*, "Standards-based DevOps automation and integration using tosca," in *Proc. 2014 IEEE/ACM 7th Int. Conf. Utility and Cloud Computing*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 59–68.

[20] R. Uhlig *et al.*, "Intel virtualization technology," *Comput.*, vol. 38, no. 5, pp. 48–56, May 2005.

[21] "Understanding full virtualization, paravirtualization, and hardware assist," VMware, Tech. Rep., Mar. 2008.

[22] P. Barham *et al.*, "Xen and the art of virtualization," in *Proc. 19th ACM Symp. Operating Syst. Principles*. New York, NY, USA: ACM, 2003, pp. 164–177.

[23] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 2–13, Oct. 2006.

[24] B. Le *et al.*, "Computer image capture, customization and deployment," Apr 2008, US Patent 7356679.

[25] M. Ben-Yehuda *et al.*, "The turtles project: Design and implementation of nested virtualization." in *12th USENIX Symp. Operating Syst. Design and Implementation*, vol. 10, Nov 2010, pp. 423–436.

[26] F. Zhang *et al.*, "Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proc. 23rd ACM Symp. Operating Syst. Principles*. New York, NY, USA: ACM, 2011, pp. 203–216.

[27] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sept 2014.

[28] C. Pahl, "Containerization and the PaaS cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, May 2015.

[29] W. Felter *et al.*, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE Int. Symp. Performance Anal. of Syst. and Software*, March 2015, pp. 171–172.

[30] M. G. Xavier *et al.*, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro Int. Conf. Parallel, Distributed, and Network-Based Process.*, Feb 2013, pp. 233–240.

[31] Docker Documentation - Docker. [Online] Available at https://docs.docker.com/. [Accessed: 13.2.2017].

[32] C. Anderson, "Docker [software engineering]," *IEEE Softw.*, vol. 32, no. 3, pp. 102–c3, May 2015.

[33] Git. [Online] Available at https://git-scm.com/. [Accessed: 18.9.2017].

[34] T. Combe *et al.*, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, Sept 2016.

[35] Kubernetes - Production-Grade Container Orchestration. [Online] Available at https://kubernetes.io/. [Accessed: 6.2.2017].

[36] Kubernetes Documentation - Kubernetes. [Online] Available at https://kubernetes.io/docs/. [Accessed: 24.3.2017].

[37] B. Burns *et al.*, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016.

[38] P. Mell *et al.*, "The NIST definition of cloud computing," *NIST Special Publication 800-145*, 2011.

[39] M. K. Weldon, *The future X network: a Bell Labs perspective*. CRC Press, 2016.

[40] A. Singleton, "The economics of microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 16–20, Sept 2016.

[41] M. Fazio *et al.*, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, Sept 2016.

[42] A. Sill, "The design and architecture of microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 76–80, Sept 2016.

[43] D. S. Linthicum, "Practical use of microservices in moving workloads to the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 6–9, Sept 2016.

[44] H. Holma and A. Toskala, *LTE for UMTS: Evolution to LTE-Advanced*. John Wiley & Sons, 2011.

[45] A. Pizzinat *et al.*, "Things you should know about fronthaul," *J. Lightw. Technol.*, vol. 33, no. 5, pp. 1077–1083, Mar 2015.

[46] J. Wu *et al.*, "Cloud radio access network (c-ran): a primer," *IEEE Network*, vol. 29, no. 1, pp. 35–41, Jan 2015.

[47] D. Pompili *et al.*, "Elastic resource utilization framework for high capacity and energy efficiency in cloud RAN," *IEEE Commun. Mag.*, vol. 54, no. 1, pp. 26–32, January 2016.

[48] P. Rost *et al.*, "Cloud technologies for flexible 5G radio access networks," *IEEE Commun. Mag.*, vol. 52, no. 5, pp. 68–76, May 2014.

[49] A. Checko *et al.*, "Optimizing small cell deployment by the use of c-rans," in *20th Eur. Wireless Conf.*, May 2014, pp. 1–6.

[50] X. Wang *et al.*, "Energy-efficient virtual base station formation in optical-access-enabled cloud-ran," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 5, pp. 1130–1139, May 2016.

[51] R. Wang *et al.*, "Potentials and challenges of c-ran supporting multi-rats toward 5g mobile networks," *IEEE Access*, vol. 2, pp. 1187–1195, 2014.

[52] "ZTE green technology innovations white paper," ZTE, Tech. Rep., 2011.

[53] "C-RAN the road towards green RAN," China Mobile, Tech. Rep., 2013.

[54] C. N. Mao *et al.*, "Minimizing latency of real-time container cloud for software radio access networks," in *2015 IEEE 7th Int. Conf. Cloud Computing Technol. and Sci.*, Nov 2015, pp. 611–616.

[55] N. Nikaein *et al.*, "Demo: Closer to cloud-ran: Ran as a service," in *Proc. 21st Annu. Int. Conf. Mobile Computing and Networking.* New York, NY, USA: ACM, 2015, pp. 193–195.

[56] D. Kreutz *et al.*, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan 2015.

[57] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 114–119, February 2013.

[58] K. Kirkpatrick, "Software-defined networking," *Commun. ACM*, vol. 56, no. 9, pp. 16–19, Sep. 2013.

[59] G. Pongrácz *et al.*, "Removing roadblocks from SDN: OpenFlow software switch performance on Intel DPDK," in *2013 2nd Eur. Workshop Software Defined Networks*, Oct 2013, pp. 62–67.

[60] "Network functions virtualisation – introductory white paper," ETSI, Tech. Rep., 2012.

[61] J. Anderson *et al.*, "Performance considerations of network functions virtualization using containers," in *2016 Int. Conf. Computing, Networking and Commun.*, Feb 2016, pp. 1–7.

[62] I. Cerrato *et al.*, "Supporting fine-grained network functions through Intel DPDK," in *2014 3rd Eur. Workshop Software Defined Networks*, Sept 2014, pp. 1–6.

[63] iPerf - The TCP, UDP and SCTP network bandwidth measurement tool. [Online] Available at https://iperf.fr/. [Accessed: 20.6.2017].

[64] DPDK documentation - Data Plane Development Kit 17.02.0-rc2 Documentation. [Online] Available at http://dpdk.org/doc/guides/. [Accessed: 9.2.2017].

[65] Linux Kernel Documentation. [Online] Available at https://www.kernel.org/doc/. [Accessed: 7.6.2017].

[66] Open vSwitch. [Online] Available at http://openvswitch.org/. [Accessed: 24.3.2017].

[67] Open vSwitch Documentation - Open vSwitch 2.6.0 documentation. [Online] Available at http://docs.openvswitch.org/en/latest/. [Accessed: 24.3.2017].

[68] "Open vSwitch* enables SDN and NFV transformation," Intel, Tech. Rep., 2015.

[69] B. Pfaff *et al.*, "Extending networking into the virtualization layer." in *8th ACM Workshop on Hot Topics in Networks*, Oct 2009, pp. 1–6.

[70] Home » OpenStack Open Source Cloud Computing Software. [Online] Available at https://www.openstack.org/. [Accessed: 3.4.2017].

[71] OpenStack Docs: Ocata. [Online] Available at https://docs.openstack.org/. [Accessed: 3.4.2017].

[72] "Accelerating NFV delivery with OpenStack," OpenStack Foundation, Tech. Rep., 2016.

[73] openstack/networking-ovs-dpdk: A Collection of Agents and Drivers to support managing DPDK accelerated Open vSwitch with neutron. [Online] Available at https://github.com/openstack/networking-ovs-dpdk. [Accessed: 4.7.2017].

[74] "Container and kernel-based virtual machine (KVM) virtualization for network function virtualization (NFV)," Intel, Tech. Rep., Aug. 2015.

[75] openvswitch/ovn-kubernetes: Kubernetes integration for OVN. [Online] Available at https://github.com/openvswitch/ovn-kubernetes. [Accessed: 4.8.2017].

[76] networkstatic/iperf3 - Docker Hub. [Online] Available at https://hub.docker.com/r/networkstatic/iperf3/. [Accessed: 20.6.2017].

A Dockerfiles

This appendix presents the four dockerfiles that were used to create the Docker container images that were employed in the measurements: First, dockerfile to create dpdk-docker image is presented in Listing A1; dockerfile to create ovs-docker image is presented in Listing A2, and scripts copied to the container in Listings A3–A5; dockerfile to create pktgen-docker image is presented in Listing A6; and finally, dockerfile to create iperf-docker image is presented in Listing A7.

The dockerfiles presented here are either based on commonly available base image, or the dpdk-docker image. Dpdk-docker is based on Centos Docker base image that is acquired from Docker hub. The Dockerfiles to create ovs-docker and pktgen-docker are based on dpdk-docker. The dockerfile to create iperf-docker is based on Alpine Docker base image from Docker hub.

The dockerfiles presented here are built into a docker images using the `docker build` command. All of the dockerfiles presented here allow passing software version, destination folder, and flag to either keep or remove compilers after installation. In addition, dpdk-docker can be configured to build shared libraries and to enable debug logging via build arguments. For example, command to build dpdk-docker with DPDK version 17.05.1, with shared libraries, debug logging enabled, and with compiler left to image, would be

```
$ docker build \
    --build-arg dpdk_ver=17.05.1 \
    --build-arg shared=y \
    --build-arg debug=y \
    --build-arg leave_compiler=y \
    .;
```

All of the build arguments to these four dockerfiles presented here have default values. This means that build command can be also executed without any build arguments.

Listing A1: `dpdk-docker/build/Dockerfile`

```
1  FROM centos
2
3  # Specify arguments for the Dockerfile:
4  #  - dpdk_ver: version to be installed. Exported to image as
       DPDK_VERSION.
5  #  - dest_dir: directory to install DPDK. Exported to image as
       RTE_SDK.
6  #  - debug=y:  enable or disable DPDK debug features. By default n.
7  ARG dpdk_ver
8  ARG dest_dir
9  ARG debug=n
10 ARG shared=y
11 ARG leave_compiler=n
12
13 # Specify DPDK version to be installed and destination folder.
14 ENV DPDK_VERSION=${dpdk_ver:-17.05} \
15     RTE_TARGET=x86_64-native-linuxapp-gcc \
16     RTE_SDK=${dest_dir:-/usr/src/dpdk} \
17     DPDK_BUILD=$RTE_SDK/$RTE_TARGET
```

```
18
19 # Fetch and untar DPDK sources.
20 RUN yum install -y --noplugins wget && \
21     wget http://fast.dpdk.org/rel/dpdk-${DPDK_VERSION}.tar.xz && \
22     tar xf dpdk-${DPDK_VERSION}.tar.xz && \
23     rm -f dpdk-${DPDK_VERSION}.tar.xz && \
24     mv dpdk-*${DPDK_VERSION} ${RTE_SDK} && \
25     yum remove -y wget && \
26     yum clean all
27
28 WORKDIR $RTE_SDK
29
30 # Disable compilation of kernel modules. These should be compiled
       and loaded from the host.
31 # Enable compilation of vhost-virtio virtual device requirements.
32 # Set Debug features on/off.
33 RUN sed -i s/CONFIG_RTE_EAL_IGB_UIO=y/CONFIG_RTE_EAL_IGB_UIO=n/ ${
       RTE_SDK}/config/common_linuxapp && \
34     sed -i s/CONFIG_RTE_KNI_KMOD=y/CONFIG_RTE_KNI_KMOD=n/ ${RTE_SDK
          }/config/common_linuxapp && \
35     sed -i s/CONFIG_RTE_LIBRTE_VHOST=n/CONFIG_RTE_LIBRTE_VHOST=y/ $
          {RTE_SDK}/config/common_base && \
36     sed -i s/CONFIG_RTE_LIBRTE_VHOST_NUMA=n/
          CONFIG_RTE_LIBRTE_VHOST_NUMA=y/ ${RTE_SDK}/config/
          common_base && \
37     sed -i s/CONFIG_RTE_LIBRTE_PMD_VHOST=n/
          CONFIG_RTE_LIBRTE_PMD_VHOST=y/ ${RTE_SDK}/config/common_base
           && \
38     sed -i s/CONFIG_RTE_LIBRTE_VIRTIO_PMD=n/
          CONFIG_RTE_LIBRTE_VIRTIO_PMD=y/ ${RTE_SDK}/config/
          common_base && \
39     sed -i s/CONFIG_RTE_VIRTIO_USER=n/CONFIG_RTE_VIRTIO_USER=y/ ${
          RTE_SDK}/config/common_base && \
40     sed -i "s/CONFIG_RTE_BUILD_SHARED_LIB=n/
          CONFIG_RTE_BUILD_SHARED_LIB=${shared}/" ${RTE_SDK}/config/
          common_base && \
41     sed -i "s/\(DEBUG.*=\)[yn]/\1${debug}/" ${RTE_SDK}/config/
          common_base
42
43 RUN yum install -y --noplugins make gcc gdb pciutils iproute sudo
       numactl-devel && \
44     make install T=$RTE_TARGET DESTDIR=install && \
45     make config T=$RTE_TARGET && \
46     # make clean && \
47     (test "$leave_compiler" == "y" || yum remove -y make gcc gdb)
          && \
48     yum autoremove -y && yum clean all
49
50 ENV LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$RTE_SDK/$RTE_TARGET/lib
```

Listing A2: ovs-docker/build/Dockerfile

```
FROM dpdk-docker

# Specify arguments for the Dockerfile:
#  - ovs_ver: version to be installed. Exported to image as OVS_VERSION.
#  - dest_dir: directory to install OVS. Exported to image as OVS_DEST.
ARG ovs_ver
ARG dest_dir
ARG leave_compiler=n

# Specify OVS version to be installed and destination folder.
ENV OVS_VERSION=${ovs_ver:-2.7.0} \
    OVS_DEST=${dest_dir:-/usr/src/ovs}

# Fetch and untar OVS sources.
RUN yum install -y --noplugins wget git && \
    git config --global https.proxy ${http_proxy} && \
    git config --global http.proxy ${https_proxy} && \
    git config --global url."https://".insteadOf "git://" && \
    git clone --depth 1 https://github.com/openvswitch/ovs.git && \
    mv ovs ${OVS_DEST} && \
    yum clean all

WORKDIR $OVS_DEST

# Configure OVS to use DPDK, build, and install.
RUN yum install -y --noplugins make gcc gdb perl numactl-devel libpcap-devel python-six openssl automake libtool && \
    ./boot.sh && ./configure --with-dpdk=$RTE_SDK/$RTE_TARGET && \
    make install && \
    make clean && \
    (test "$leave_compiler" == "y" || yum remove -y make gcc gdb) && \
    yum autoremove -y && yum clean all

COPY scripts/* /usr/local/bin/

ENTRYPOINT ovs-docker-start.sh
```

Listing A3: ovs-docker/build/scripts/ovs-docker-cleanup.sh

```
#!/bin/bash -x

# Remove any existing OVS processes and files
pkill -9 "(ovsdb-server)|(ovs-vswitchd)";
rm -f /usr/local/var/run/openvswitch/*;
rm -f /usr/local/etc/openvswitch/*;
mkdir -p /usr/local/etc/openvswitch;
mkdir -p /usr/local/var/run/openvswitch;
```

Listing A4: ovs-docker/build/scripts/ovs-docker-init-db.sh

```
#!/bin/bash -x

ovsdb-tool create /usr/local/etc/openvswitch/conf.db vswitchd/
    vswitch.ovsschema;
```

Listing A5: ovs-docker/build/scripts/ovs-docker-start.sh

```
#!/bin/bash -x

# Remove any existing OVS processes and files
ovs-docker-cleanup.sh;

# Init OVS DB
ovs-docker-init-db.sh;

# Start OVS DB server as detached process
export OVS_DB_SOCK=/usr/local/var/run/openvswitch/db.sock
ovsdb-server --remote=punix:$OVS_DB_SOCK \
    --remote=db:Open_vSwitch,Open_vSwitch,manager_options \
    --private-key=db:Open_vSwitch,SSL,private_key \
    --certificate=db:Open_vSwitch,SSL,certificate \
    --bootstrap-ca-cert=db:Open_vSwitch,SSL,ca_cert \
    --pidfile --detach --log-file;

# Modify dpdk params passed to ovs-vsctl here
ovs-vsctl --no-wait set Open_vSwitch . \
    other_config:dpdk-init=true \
    other_config:dpdk-lcore-mask=0xC \
    other_config:dpdk-socket-mem="1024" \
    other_config:dpdk-extra="--file-prefix=ovs_";
ovs-vswitchd unix:$OVS_DB_SOCK --pidfile --log-file;
```

Listing A6: pktgen-docker/build/Dockerfile

```
FROM dpdk-docker

# Specify arguments for the Dockerfile:
#  - pktgen_ver: to be installed. Exported to image as
    PKTGEN_VERSION.
#  - dest_dir: to install pktgen. Exported to image as PKTGEN_DEST.
ARG pktgen_ver
ARG dest_dir
ARG leave_compiler=n

# Specify DPDK version to be installed and destination folder.
ENV PKTGEN_VERSION=${pktgen_ver:-3.2.8} \
    PKTGEN_DEST=${dest_dir:-/usr/src/pktgen}

# Fetch and untar DPDK sources.
RUN yum install -y --noplugins wget && \
    wget http://dpdk.org/browse/apps/pktgen-dpdk/snapshot/pktgen-${
        PKTGEN_VERSION}.tar.gz && tar xf pktgen-${PKTGEN_VERSION}.
        tar.gz && rm -f pktgen-${PKTGEN_VERSION}.tar.gz && mv pktgen
```

```
        -${PKTGEN_VERSION} ${PKTGEN_DEST} && \
    yum remove -y wget && \
    yum clean all

WORKDIR $PKTGEN_DEST

RUN yum install -y --noplugins make gcc gdb libpcap-devel && \
    make && \
    (test "$leave_compiler" == "y" || yum remove -y make gcc gdb)
        && \
    yum autoremove -y && yum clean all

ENTRYPOINT ["./app/x86_64-native-linuxapp-gcc/pktgen"]
CMD ["--help"]
```

Listing A7: iperf-docker/build/Dockerfile

```
FROM alpine

# Specify arguments for the Dockerfile:
#  - iperf_ver: to be installed. Exported to image as IPERF_VERSION
    .
#  - dest_dir: to install iperf. Exported to image as IPERF_DEST.
ARG iperf_ver
ARG dest_dir
ARG leave_compiler=n

# Specify iperf version to be installed and destination folder.
ENV IPERF_VERSION=${iperf_ver:-2.0.9} \
    IPERF_DEST=${dest_dir:-/usr/src/iperf}

# Fetch and untar iperf sources
RUN apk add --no-cache wget ca-certificates && \
    if [ "${IPERF_VERSION}" \< "3.0.0" ]; then \
        wget https://iperf.fr/download/source/iperf-${IPERF_VERSION
            }-source.tar.gz && \
        tar xf iperf-${IPERF_VERSION}-source.tar.gz && \
        rm iperf-${IPERF_VERSION}-source.tar.gz; \
    else \
        wget https://github.com/esnet/iperf/archive/${IPERF_VERSION
            }.tar.gz && \
        tar xf ${IPERF_VERSION}.tar.gz && \
        rm ${IPERF_VERSION}.tar.gz; \
    fi && \
    mkdir -p $(dirname ${IPERF_DEST}) && \
    mv iperf-${IPERF_VERSION}*/ ${IPERF_DEST} && \
    apk del --no-cache wget ca-certificates;

WORKDIR ${IPERF_DEST}

# Compile and install iperf
RUN apk add --no-cache build-base && \
    sed -i 's/-pg//' src/Makefile* && \
    ./configure CXXFLAGS="-static-libgcc␣-static-libstdc++" && \
```

```
35      make && \
36      make install && \
37      if [ "${IPERF_VERSION}" \> "3.0.0" ]; then \
38          ln -s /usr/local/bin/iperf3 /usr/local/bin/iperf; \
39      fi && \
40      if [ "$leave_compiler" != "y" ]; then \
41          apk del --no-cache build-base; \
42      fi;
43
44  ENTRYPOINT ["iperf"]
45  CMD ["--help"]
```

B DevStack configuration files

This appendix presents the local.conf files used in setting up the DevStack. Configuration file for setup with OVS with DPDK datapath is presented in Listing B1 and configuration file for setup with default OVS installation is presented in Listing B2. In both listings `DEVSTACK_BRANCH` string is to be replaced with actual name of the used OpenStack branch, for example `stable/ocata`.

Listing B1: local.conf file to setup OpenStack with DPDK boosted OvS

```
[[local|localrc]]
ADMIN_PASSWORD=secret
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD

DEST=/ephemeral/stack
LOGFILE=$DEST/logs/stack.sh.log
enable_plugin heat https://git.openstack.org/openstack/heat
    DEVSTACK_BRANCH
disable_service tempest

enable_plugin networking-ovs-dpdk https://github.com/openstack/
    networking-ovs-dpdk DEVSTACK_BRANCH
OVS_DPDK_GIT_REPO=http://dpdk.org/git/dpdk-stable
OVS_DPDK_GIT_TAG=v17.05.1
OVS_GIT_TAG=add49d45bb7e87ad2ad8ccef8d6447ca8a57c89c
OVS_PATCHES=https://patchwork.ozlabs.org/patch/781100/raw/

OVS_DPDK_MODE=controller_ovs_dpdk
disable_service n-net
enable_service n-cpu
enable_service neutron
enable_service q-svc
enable_service q-agt
enable_service q-dhcp
enable_service q-l3
enable_service q-meta

OVS_NUM_HUGEPAGES=8192
OVS_CORE_MASK=0x2
OVS_PMD_CORE_MASK=0xC
OVS_DATAPATH_TYPE=netdev
OVS_LOG_DIR=/opt/stack/logs

[[post-config|$NOVA_CONF]]
[libvirt]
cpu_mode=host-passthrough
firewall_driver=nova.virt.firewall.NoopFirewallDriver
scheduler_default_filters=RamFilter,ComputeFilter,
    AvailabilityZoneFilter,ComputeCapabilitiesFilter,
    ImagePropertiesFilter,PciPassthroughFilter,NUMATopologyFilter
```

Listing B2: local.conf file to setup OpenStack without native OvS

```
[[local|localrc]]
ADMIN_PASSWORD=secret
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD

DEST=/ephemeral/stack
enable_plugin heat https://git.openstack.org/openstack/heat
    DEVSTACK_BRANCH
disable_service tempest

[[post-config|$NOVA_CONF]]
[libvirt]
cpu_mode=host-passthrough
```