Ville Kapanen

TCP/IP Acceleration for Telco Cloud

School of Science

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

Espoo 22.10.2018

Thesis supervisor:

Prof. Mario Di Francesco

Thesis advisor:

M.Sc. (Tech.) Tuomas Taipale

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Ville Kapanen

Title: TCP/IP Acceleration for Telco Cloud

Date: 22.10.2018 Language: English Number of pages: 8+51

Department of Computer Science

Professorship: Computer Science Code: SCI3042

Supervisor: Prof. Mario Di Francesco

Advisor: M.Sc. (Tech.) Tuomas Taipale

Mobile traffic rates are in constant growth. The currently used technology, long-term evolution (LTE), is already in a mature state and receives only small incremental improvements. However, a new major paradigm shift is needed to support future development. Together with the transition to the fifth generation of mobile telecommunications, companies are moving towards network function virtualization (NFV). By decoupling network functions from the hardware it is possible to achieve lower development and management costs as well as better scalability.
Major change from dedicated hardware to the cloud does not take place without issues. One key challenge is building a telecommunications-grade ultra-low-latency and low-jitter data storage for call session data. Once overcome, it enables new ways to build much simpler stateless radio applications.
There are many technologies which can be used to achieve lower latencies in the cloud infrastructure. In the future, technologies such as memory-centric computing can revolutionize the whole infrastructure and provide nanosecond latencies. However, on the short term, viable solutions are purely software-based. Examples of these are databases and transport layer protocols optimized for latency. Traffic processing can also be accelerated by using libraries and drivers such as the Data Plane Development Kit (DPDK). However, DPDK does not have transport layer support, so additional frameworks are needed to unleash the potential of Transmission Control Protocol/Internet Protocol (TCP/IP) acceleration.
In this thesis TCP/IP acceleration is studied as a method for providing ultra-low-latency and low-jitter communications for call session data storage. Two major frameworks – namely, VPP and F-Stack – were selected for evaluation. The major finding is that the frameworks are not as mature as expected, and thus they failed to deliver production-ready performance. Building robust interface for applications to use was recognized as a common problem in the market.

Keywords: TCP/IP Acceleration, VPP, F-Stack, Network Function Virtualization, Cloud Infrastructure

AALTO-YLIOPISTO
PERUSTIETEIDEN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Ville Kapanen

Työn nimi: TCP/IP kiihdytys pilvipohjaisessa mobiiliverkossa

Päivämäärä: 22.10.2018 Kieli: Englanti Sivumäärä: 8+51

Tietotekniikan laitos

Professuuri: Tietotekniikka Koodi: SCI3042

Valvoja: Prof. Mario Di Francesco

Ohjaaja: DI Tuomas Taipale

Mobiiliverkon datamäärät ovat jatkuvassa nousussa. Nykyisin käytössä olevaa neljännen sukupolven matkapuhelintekniikkaan (4G) tehdään enää pieniä päivityksiä. Jotta tulevaisuuden datamääriin pystytään vastaamaan, täytyy tekniikan ottaa seuraava suuri harppaus. Siirryttäessä viidennen sukupolven matkapuhelintekniikkaan (5G), siirtyvät yritykset myös kohti verkon funktioiden virtualisointia. Erottamalla verkon funktiot laitteistosta pystytään saavuttamaan entistä matalammat kehitys- ja hallintakustannukset, sekä parempi skaalautuvuus.
Siirtymä erityislaitteistosta pilveen on haasteellinen. Yksi keskeisimmistä ongelmista on matalaan ja tasaiseen viiveeseen pystyvän tietovaraston rakentaminen yhteyksien käsittelyyn. Jos tähän haasteeseen pystytään vastaamaan, mahdollistaa se uudenlaisten yksinkertaisten tilattomien radioapplikaatioiden suunnittelun. Matalaa viivettä pystytään tavoittelemaan monella tapaa. Tulevaisuudessa muistikeskeinen laskenta saattaa mullistaa koko infrastruktuurin ja mahdollistaa nanosekuntien viiveet. Tämä ei kuitenkaan ole mahdollista lyhyellä mittakaavalla, joten ratkaisuja pitää etsiä ohjelmistoratkaisuista. Tämä tarkoittaa esimerkiksi tietokannan tai kuljetuskerroksen protokollan optimointia viivettä ajatellen. Liikenteen prosessointia voi myös kiihdyttää erilaisilla kirjastoilla ja ajureilla, kuten Data plane development kitillä (DPDK). DPDK ei tue kuljetuskerroksen kiihdytystä, joten tähän joudutaan käyttämään erillisiä ohjelmistoja.
Tässä diplomityössä tutkitaan, pystyvätkö TCP/IP-kiihdytystä tarjoavat ohjelmointikehykset lyhentämään viivettä riittävästi yhteyksien tilannedatan varastoinnin tarpeisiin. Kahden yleisimmän ohjelmiston, VPP ja F-Stack, suorituskyky mitataan. Tutkimuksen tuloksena havaittiin, että kumpikaan ohjelmisto ei ole riittävän valmis tuotantokäyttöön. Yhteinen ongelma kaikissa tutkituissa ohjelmistoissa oli rajapinta, jota tarjotaan applikaation käytettäväksi.

Avainsanat: TCP/IP kiihdystys, VPP, F-Stack, verkon funktioiden virtualisointi, pilvi-infrastruktuuri

Preface

I would like to thank my thesis supervisor Professor Mario Di Francesco for all the guidance I received during the process. His excellence in academic writing and both calming and supportive attitude towards my stress truly made my thesis something it would not have been without his support.

This thesis was conducted at Nokia Oyj in Espoo, Finland. I would like to thank my advisor Tuomas Taipale and our technical lead Jussi Mäki-Äijälä for all their support and guidance. These two individuals truly know what they are talking about. At Nokia I would also like to thank Markku Niiranen and Osmo Kaukanen for the opportunity to work on this project. Thanks also belong to my colleague and old friend Samu, who is solely the reason I found my way to Nokia.

For me this thesis is much more significant than half a year I spend working on it. In its way, this is the ending of a very special part of my life. During this time I have met more awesome individuals than I could have ever imagined. I have learned about so many topics I didn't even know are taught in the university. I have also learned that quite a bit of learning can also happen outside of the lecture halls in the active student community of Otaniemi. I'm grateful for the community SIK has been able to offer me. It's hard to even think how much I have experienced with the peers over there. Especially I want to thank boards of '14 and '16 for the years I will never forget. In addition I want to thank FTMK14, ITMK13 and all the other committees I got to take part at AYY. Thanks go also to my second family Joutomiehet, with whom I have experienced some of the most unexpected events in my life.

Last but not least, I want to thank my actual family who have always supported me on everything and made sure I have what I need to pursue my dreams.

Espoo, 22.10.2018

Ville O. Kapanen

Contents

Symbols and abbreviations

Symbols

Opetators

Abbreviations

4G	Fourth generation of mobile telecommunications
5G	Fifth generation of mobile telecommunications
ACK	Acknowledgment
ADC	Analog-to-Digital Converter
API	Application Programming Interface
ARPA	Advanced Research Projects Agency
ARPU	Average Revenue Per User
BIC	Binary Increase Congestion
COTS	Commercial Off-The-Shelf
CPU	Central processing unit
DAC	Digital-to-Analog Converter
DB	Database
DCTCP	Data Center TCP
DDoS	Distributed Denial-of-Service
DMA	Direct Memory Access
DMM	Dual Mode, Multi-protocol, Multi-instance
DPDK	Data Plane Development Kit
FASP	Fast and Secure Protocol
FMO	Future Mode of Operations
FPGA	Field-Programmable Gate Array
Gbps	Gigabits Per Second
HFT	High Frequency Trading
HPE	Hewlett Packard Enterprise
IaaS	Infrastructure as a Service
IEEE	Institute of Electrical and Electronics Engineers
iLO	Integrated Lights-Out
IOMMU	Input–output Memory Management Unit
IoT	Internet of Things
IP	Internet Protocol
KVM	Kernel-based Virtual Machine
L2	Layer 2
L3	Layer 3
L4	Layer 4
LKM	Loadable Kernel Module
LTE	Long-Term Evolution
LTS	Long Term Support
MIMO	Massive Multiple-Input Multiple-Output

MME	Mobility Management Entity
MSS	Maximum Segment Size
NF	Network Function
NFV	Network Function Virtualization
NIC	Network Interface Controller
NIST	National Institute of Standards and Technology
NNTP	The Network News Transfer Protocol
NVM	Non-Volatile Memory
OFP	OpenFastPath
OPEX	Operating Expense
OS	Operating System
OvS	Open vSwitch
PCI	Peripheral Component Interconnect
PMD	Poll Mode Driver
PMO	Present Mode of Operations
PoC	Proof of Concept
PRR	Proportional Rate Reduction
QUIC	Quick UDP Internet Connections
RDMA	Remote Direct Memory Access
RFC	Request for Comments
RNC	Radio Network Controller
RTT	Round-Trip Time
SDN	Software-Defined Networking
TCP	Transmission Control Protocol
Telco	Telecommunications
TLDK	Transport Layer Development Kit
TTM	Time to Market
UDP	User Datagram Protocol
UIO	Userspace I/O
VCL	VPP communications library
VFIO	Virtual Function I/O
VMM	Virtual Machine Monitor
VPP	Vector Packet Processing
vSwitch	Virtual Switch

1 Introduction

Internet traffic trend has been growing both rapidly for years, and this trend is not expected to end in the foreseeable future [27]. Traffic growth is especially fast in the mobile market, where both the number of connected devices and data rates per device have been quickly growing. The increasing amount of connected devices and applications will demand more throughput and less latency in the future [70]. In addition, the number of connected devices and Internet of Things (IoT) solutions in the industry will grow and expand in the following years. These industry applications will also put other demands on radio technology, such as lower latency and ultra-reliability. The fifth generation of mobile telecommunications (5G) is developed to answer to these needs in the close future.

As these requirements tighten, the need for improvements in every part of the radio network rises. Modern radio solutions are built on top of cloud infrastructure, which gives plenty of benefits over dedicated solutions: scalability, cost savings and ease of maintenance to name a few [35]. Radio applications can significantly benefit from these properties. For example, the load on the network can be highly varying, so scaling is a very useful feature. Using the cloud infrastructure also makes deployment easier and new features can be released at a faster pace. The use of commercial off-the-self (COTS) hardware can increase the flexibility even more, since solutions from different providers can be combined. Of course, using the cloud also has its disadvantages. Especially with COTS hardware, it can be harder to achieve the highest possible performance and additional levels of abstraction bring more complexity to the systems. [35]

Compared to other cloud solutions, telecommunications (telco) cloud has some specific requirements [72]. Especially with the coming of IoT and industry applications, latency and reliability requirements of cloud are much stricter. There is pressure to get end-to-end latency in 5G network close to 1 ms and maintaining ultra-reliability [8]. These requirements put also pressure to the core network and its solutions. One important part of it is information sharing between radio applications inside the cloud. Future radio applications need low-latency, low-jitter and reliable data storage to ensure correct behavior of the service.

Even inside the cloud infrastructure there are many variables which affect latency and jitter between the application and data storage. These could be divided in three main parts: physical network, database (DB) and operating system (OS) network stack. There are also multiple ways to approach the problem. In most COTS solutions, the focus is on selecting the database most appropriate to the needs. In addition to DB selection we can try to accelerate the traffic processing at both the application and the database server. This can be done by changing the communication protocol of the database to something more suitable to low-latency and low-jitter processing. However, this usually comes with its trade-offs in reliability and development effort. Traffic processing can also be accelerated by using libraries and drivers such as the Data Plane Development Kit (DPDK) [3], which provides faster processing than the native Linux networking stack. However, also DPDK has its limitations by not offering the Transmission Control Protocol (TCP).

Several other frameworks offer this functionality on top of the DPDK. Lastly, for the trade-off of losing flexibility offered by COTS hardware there are also dedicated hardware like memory-centric solutions. One example is The Machine [52], where all the compute nodes are connected to shared memory via a proprietary interface.

1.1 Research scope and goals

Low-latency and low-jitter communications in telco cloud can be facilitated with multiple technologies. In the scope of this research only Transmission Control Protocol/Internet Protocol (TCP/IP) acceleration is studied in detail. Other methods are briefly investigated to allow proper comparison between technologies. One of the main reason for targeting TCP/IP acceleration is finding method which could be utilized in a quick pace.

Two major frameworks offering TCP/IP acceleration are selected for the experimental part of the thesis. These are Vector Packet Processing (VPP) and F-Stack. We focus on these and evaluate what other benefits these technologies could give us in addition to pure guest-side TCP/IP bypass. In particular the scope of the thesis is as follows:

- *Measurement tools:* selecting, building and modifying tools for measuring TCP latency. Due to the novelty of considered frameworks, there are not many tools which are integrated with those frameworks. Integration effort is not in the scope of the research, so tools are selected from a limited pool and multiple tools are needed instead of a "do-it-all" solution. The main question is to find a trade-off between versatility and support.

- *Evaluating performance of frameworks:* analyzing VPP and F-Stack in an environment which is similar to a production telco cloud. The scope includes determining the parameters which impact on performance, and optimizing those with traffic which resembles actual traffic in the radio cloud. These results are also compared to a baseline when TCP/IP acceleration is not used.

- *Drawing conclusions:* performance is not the only factor dictating TCP/IP acceleration's suitability to solve the problem of latency in the telco cloud. We draw conclusion on the suitability of TCP/IP acceleration in telco cloud through research data on maturity, deployment effort and performance of different frameworks.

1.2 Contributions

The contributions of the study are the following:

- This study is a starting point for the research of TCP/IP acceleration in telco cloud. It provides a clear view of the current market of the solutions and their features, which has not been done as one coherent study. This includes classifying which of the claims by developers of the frameworks actually hold true.

- A methodology to compare the considered frameworks was developed and supporting software was realized to manage the test environment and run experiments.
- A benchmarking tool for the Redis database was modified to provide more useful output in terms of latency-oriented testing.
- The study ends in a clear analysis and recommendations of how the research in this area could be continued.

1.3 Structure

This chapter has provided the introduction to the topic of the thesis. The rest of this thesis is divided into five chapters. First of all, Chapter 2 details in the root problem and introduces the technology behind 5G and telco cloud. After that, Chapter 3 discusses TCP and its Linux implementation in detail. The same chapter also introduces the major TCP/IP acceleration frameworks, along with their common components (e.g. DPDK).

Chapter 4 explains the experiment methodology and the setup used for the tests. It also individually explains each test performed. Chapter 5 introduces and analyzes the measurement results in depth. This chapter also includes a broader analysis which takes the results of the literature review into account to give a complete picture on TCP/IP acceleration frameworks. Lastly, Chapter 6 provides a summary of the considerations as well as directions for future work.

2 Technology Enablers of 5G

5G is becoming reality sooner than anticipated before. The Institute of Electrical and Electronics Engineers (IEEE) has published the 5G roadmap where they expect the fifth generation of mobile telecommunications to land consumer market in the early 2020's [45]. However, first telco operators have started to launch 5G as early as late 2018 [24]. The first 5G-capable smartphones are expected to hit the market in the beginning of the 2019 [49]. At the same time all major networking companies are pushing telco cloud products to support future networks.

This chapter aims to give a proper background on why TCP/IP acceleration should be researched for 5G. In section 2.1 5G and its technical requirements are described. The section continues with an explanation on how network function virtualization (NFV) and cloud computing benefit 5G. Section 2.1 wraps up with defining cloud computing. In section 2.2 the specific problem of call session storage round-trip time (RTT) is introduced and studied in the three main components of latency. Finally in section 2.3 different solutions for the problem are introduced and researching TCP/IP acceleration is justified.

2.1 Telco Cloud: An Overview

Telco cloud is effort to bring benefits of modern cloud architecture to highly demanding industry of telecommunications. In this section huge effort of architecture change is rationalized and peculiarities of the industry are presented.

2.1.1 5G

Mobile traffic rates and number of users are constantly growing and there seems to be no end for it. Nokia Bell Labs forecasts over 31-fold increase in control plane traffic between years 2016 and 2025 due to growing number of devices. Bell Labs also forecasts bearer traffic to grow 61 to 115 times in the same timeline [70]. Ericsson is expecting similar growth by predicting yearly global mobile data traffic to have an eightfold increase between years 2017 and 2023 [44]. The currently used technology, long-term evolution (LTE), is already mature and receives only small incremental improvements, which simply are not enough to provide future proof performance. To support future development of mobile data rates, a new major paradigm shift is needed, as the four previous generations of cellular technology also have been [28].

As for each major paradigm, the technical requirements for 5G are very high in comparison to 4G. Main engineering requirements for 5G are high data rate and low latency, energy and cost [28]. However, all the requirements do not have to be fulfilled to their peak performance at the same time. Most applications only rely on some of the requirements. For example, autonomous driving needs very reliable, low-latency connection, but the data rates are very moderate. Streaming services are at the other end of spectrum, where the data rate is crucial, but latency does not really matter. Figure 1 overviews the requirements in 5G, the most important of which are detailed next.

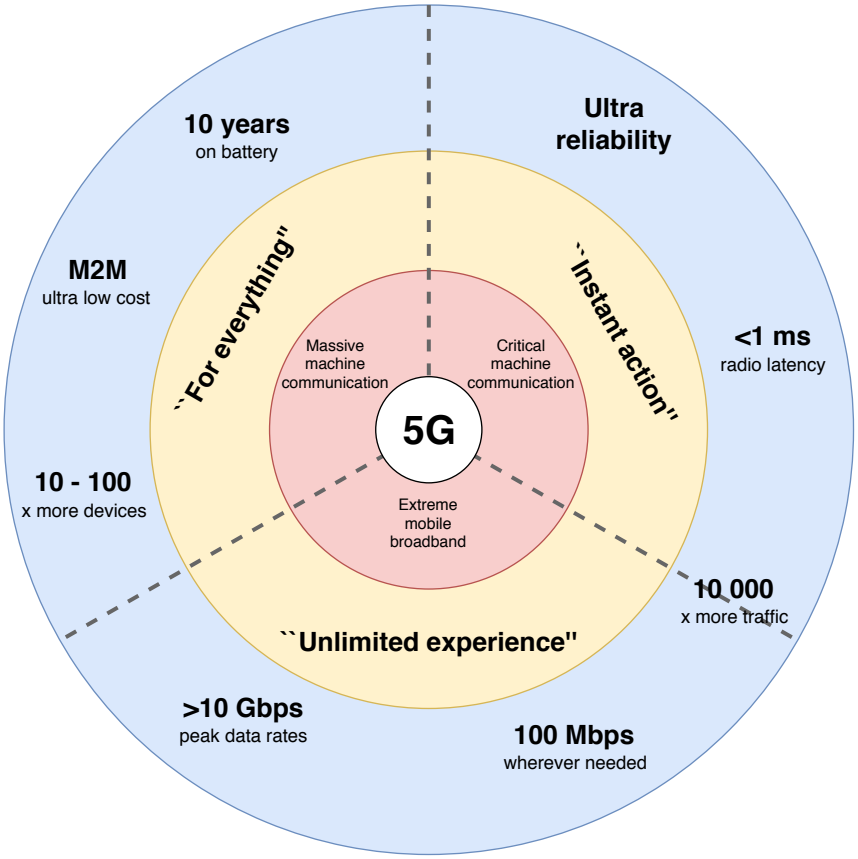


Figure 1: 5G requirements [8].

- *Data rate* can be defined in many ways. Peak data rates of 5G should be measured in tens of Gigabits Per Second (Gbps) . Edge rate, which is defined as 95th percentile data rate, should range from 100 Mbps to as high as 1 Gbps. Both of these requirements are about 10–100 times higher than in 4G. In addition to data rate, area capacity, i.e. how much data can be transferred in given area, should increase significantly. It should increase around 1,000 times in comparison to 4G.

- *Latency.* End-to-end latency of 5G network should be below 1 ms. In comparison, 4G has a requirement of latency below 15 ms. Tighter latency requirements support applications like autonomous driving, online gaming and augmented reality. Tight end-to-end latency also gives strict constraints for core network performance.

- *Energy and cost.* When network performance grows in multiple orders of magnitude, it is needless to say that the energy consumption can not follow the trend. Just to maintain the current energy consumption, we need to decrease the energy consumption per bit over 100-fold. One key element to this development is small cells, which are more cost- and energy-efficient than macro cells [28]. Another important factor is finding cost and energy savings from cloud computing.

Growing performance in orders of magnitude can not be done by just tweaking current technologies. There is a need for new innovations which can carry out the performance increase. The three key technologies are ultra-densification, mmWave and massive multiple-input multiple-output (MIMO). Densification of network is not a new technology, but it has to be taken to the next level. In the early 1980s cell sizes were as large as hundreds of square kilometers. In the modern day 4G network base stations can be as close as few hundred meters apart from each other. In the future 5G networks this can be taken even further by introducing cells as small as the current WiFi nodes. Increasing density has very distinct advantages and disadvantages. Increasing density requires more base stations, which may create additional costs and energy consumption. However, smaller cells can be built to be more cost- and energy-efficient. A beneficial side effect is that new base stations affect network performance in many ways, and the network capacity grows more than the number of base stations. More base stations mean that spectrum can be reused more efficiently, as the same frequency can be used for different transmissions in proximity. It also decreases the number of users per node, which leads to more resources for everyone. This is especially important in high-traffic areas, where edge data rates can be low. High density of base stations also decreases the distance a signal needs to travel, so higher frequencies can be used without problems [28].

mmWave, or Millimeter Wave, is the frequency range from 3 GHz to 300 GHz [33]. In this range the wavelength varies from 1 mm to 100 mm. mmWave is often used to describe only the higher end of the frequency range (30-300 GHz), as that is the part of the range which is not yet utilized as much and has thus less interference. mmWave can be a huge opportunity for 5G, but it definitely has its challenges. The challenges are worth solving, since current used spectrum is around 600MHz wide and re-purposing spectrum from other purposes can only free around 80MHz more. In the millimeter spectrum there can be even tens of gigabits of spectrum free for 5G. Even though there is a large amount of millimeter spectrum available, it is not trivial to put it into use. Increasing frequency means also increasing problems with propagation, blocking and absorption. These can be fought with adequate placement of base stations and directional antenna arrays [28]. In addition to signal characteristics, both analog-to-digital converters (ADCs) and digital-to-analog converters (DACs) for wide frequency bands are problematic. With current technology they are hard to manufacture and take a significant amount of energy [33].

Massive MIMO is closely related to introducing higher frequencies of mmWave. MIMO does exactly what the name tells: it enables communication via multiple antennas per base station. MIMO itself is not a new technology in the radio domain, but massive MIMO is set to be leap of improvement in comparison to old implementations. Multi-user MIMO used in 4G generally uses approximately the same number of antennas and terminals and is limited to few users at the time. It works well, but is not scalable at all; Massive MIMO was indeed developed to solve that problem. Massive MIMO uses arrays of directional antennas, and the number of antennas is intended to be large in comparison to amount of active terminals. This way energy can be focused to the areas where it's needed, which leads to sig-

nificant improvements in energy consumption. With an increased number of served terminals and enabling use of higher frequencies, Massive MIMO can increase the capacity by a order of magnitude [54].

To comply with the 5G requirements, telco operators and manufacturers face several challenges. The problem is not only to get enough throughput and latency from the network, but that it also needs to be done in a cost-effective manner, which is troublesome with present mode of operations (PMO). Telecommunications services are based on Network Functions (NF). Network functions implement clearly defined components of networks such as the Mobility Management Entity (MME) or the Radio Network Controller (RNC). In the traditional model (i.e., without cloud), NFs would run on dedicated hardware, which is a poorly scalable way to operate. Network function virtualization, which is introduced in the next section, offers a solution to this by using a virtualization technology to allow network functions to share resources, be more easily manageable, run on COTS hardware and scale significantly better [35].

2.1.2 Network Function Virtualization

Network function virtualization means decoupling the network functions from the underlying hardware, namely taking telecommunications to the cloud. Instead of running radio services directly on dedicated hardware, networks functions run on top of a virtualized platform. These platforms consist of three main components: physical servers, a virtual machine monitor (VMM or Hypervisor) and virtual machines (VMs). Network functions are installed into virtual machines instead of regular physical machines. A virtual machine abstracts the physical machine and its functionalities. Hypervisor is a software which manages the virtual machines and physical resources. It determines which physical resources are given to the virtual machine and provides a platform for managing it. Physical machines themselves are not very different from traditional network function deployment. In the NFV framework physical machines are located in data centers, network nodes or end-user facilities. Machines with very high capacity and performance are mostly used since the resources are shared and distributed dynamically [43]. The major difference in comparison to traditional deployment is the lack of proprietary hardware such as acceleration cards. In virtualized environment proprietary hardware is avoided, because all the network functions should be designed to work on COTS hardware. This way hardware and software are truly decoupled, and NFs can run on any server.

The lack of proprietary hardware, such as acceleration cards, is indeed one of the main challenges of NFV. However, this problem can be minimized with modern software. Hardware compatibility problems are not limited to special cards either. Even generic servers from different manufacturers can have minor differences, which can cause unexpected problems. This can be solved with standardization. When hypervisors are developed further, hardware manufacturers gain a better understanding of what is expected from their hardware. There are also many challenges on the software side. Virtualizing a telecommunications service is a complex task and there is the risk of making an already complex platform into something that is

even more complex and hard to understand. While developing NFV, it is crucial that everything is automated so operating expense (OPEX) benefits can be utilized. Security also needs to be considered in a virtualized platform. Application logic is not always drastically changed when migrating to NFV model, but the hardware is shared by multiple VMs in NFV, so the security of the hypervisor must be ensured [35].

There are many challenges associated with NFV, but the benefits do outweigh them. The most important factor is the lower cost, which makes it possible to respond to the growth in users and bandwidth while keeping the current revenue model. Due to the poor scalability of the traditional telecommunications service model, the cost of PMO is increasing faster than Average Revenue Per User (ARPU). This trend leads to the PMO exceeding ARPU, which is not sustainable for telco operators. This is illustrated in Figure 2 where PMO outgrows APRU. With scalable network function virtualization operators can match the cost of their Future Mode of Operations (FMO) to the growth of APRU, and thus be able to respond to the growing data usage. However, as shown in Figure 2, the initial set up costs for the cloud are higher than those for dedicated hardware [72].

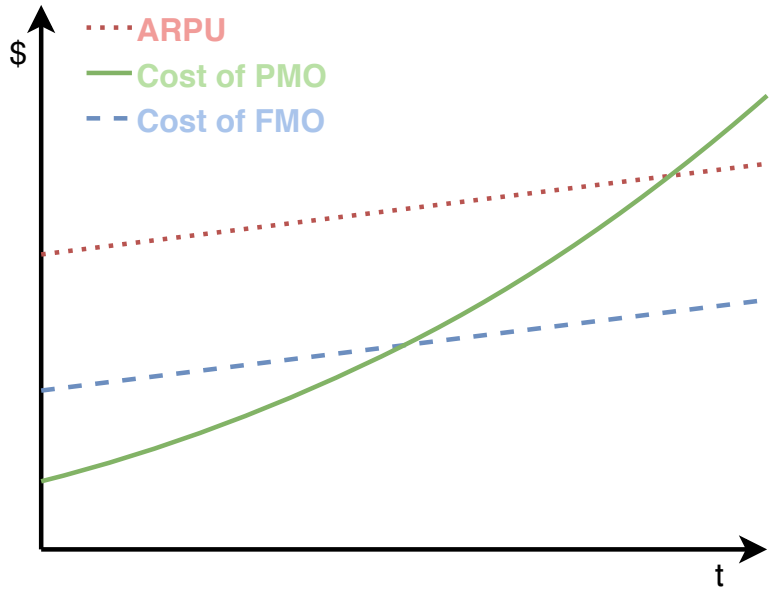


Figure 2: Cost reduction by introducing cloud to telco [72].

Using NFV and cloud instead of dedicated hardware brings a wide range of cost savings. First, telco cloud can be built with commercial off-the-shelf hardware. COTS hardware offers operators more flexibility, as they can buy their solution from multiple vendors instead of buying the whole stack from hardware to software from a single vendor as a package. In addition, cloud hardware is usually based on x86 architecture servers [72], which are cost-effective compared to more specialized hardware. Not only the hardware is potentially cheaper, but it is also easier to deploy the needed amount of resources in the cloud. As shown in Figure 3, the cloud model supports adding capacity just in time, which leads to better average

utilization and cost-efficiency. The cloud can also be scaled up and down for daily or seasonal fluctuations.

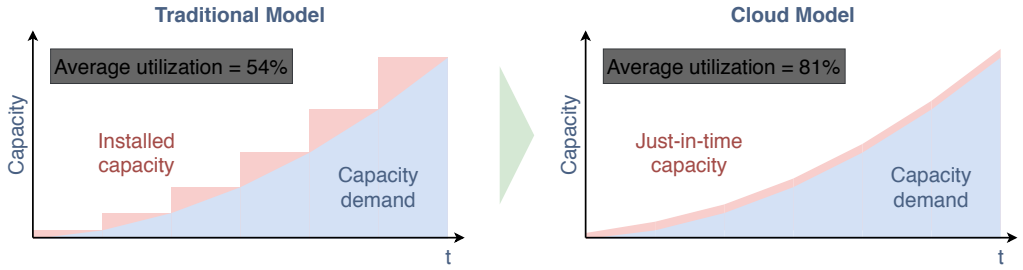


Figure 3: Average utilization with traditional and cloud model. [62]

Cloud deployments are easier to realize in comparison to dedicated deployments. This makes the costs in planning and deploying stay lower. Automation handles most of the work which would have been done by personnel in the traditional model and lowers OPEX. Together these factors allow a lower Total Cost of Ownership (TCO), because less servers and labor are needed. Scaling the cloud also enables energy savings, as during the off-peak hours it is possible to consolidate the workload on a subset of servers and set the rest in energy saving mode [35].

The switch to cloud gives plenty of direct cost-savings, but there are also other advantages. Since the cloud is easily accessible and COTS hardware, software development is easier and faster. New functionalities can be shipped in a significantly faster pace, as there is no need for new hardware deployments. This reduces the time to market (TTM) significantly. [35] This allows more innovation, which leads to many other benefits. Grown flexibility does not only give cost savings, but it also makes it easier to provide a reliable service, which leads to customer satisfaction.

2.1.3 Cloud Computing

Cloud computing has existed for over ten years now. Amazon published world's first public pay-as-you-go computing with Elastic Compute Cloud in 2006 [29]. After this we have seen many players from large technology companies like Microsoft and Google to newcomers like DigitalOcean entering the market as public cloud providers. In addition, large companies have been building huge private clouds to support their business.

Definition of cloud computing has not always been clear, and during the early days of cloud computing the term was often misused. Some of the technologies were not new either, which caused some frustration in the industry. In 2009 the former CEO of Oracle Larry Ellison gave an interview stating "The interesting thing about cloud computing is that we've redefined cloud computing to include everything that we already do .... I don't understand what we would do differently in the light of cloud computing other than change the wording of some of our ads." [30, 58]. However, the industry moved forward fast and cloud computing became well defined in a few years.

In short, cloud computing offers easy provision of hardware resources and allows users to quickly ramp up and scale down their usage. In 2011 the United States National Institute of Standards and Technology (NIST) defined cloud computing with five essential characteristics [59]:

- *On-demand self-service.* User can provision computing capabilities without human interaction. For example, in private cloud this would often be OpenStack dashboard which is used to manage instances in most popular open source Infrastructure as a Service (IaaS) platform [67].
- *Broad network access.* Service is accessible through network and can be used by variety of clients from mobile phones to workstations.
- *Resource Pooling.* Computing resources are pooled and serve multiple tenants dynamically.
- *Rapid elasticity.* Customer can rapidly ramp up and scale down the resources they use.
- *Measured service.* Use of resources is measured automatically and can be transparently monitored by both the provider and the user. Billing is often based on these measurements (Used core hours, active user accounts, used storage etc.)

2.2 Call Session Data Storage

One key element in moving telco to cloud is shared storage for call session data. Traditionally this information has been stored in each node itself but moving the data to the cloud makes it possible to build shared data storage for network functions. If the shared call session storage can meet very strict latency and jitter standards it gives significant advantage over old solutions. Low-latency shared storage allows building stateless radio applications, which is considerably simpler, more efficient and faster to build and innovate. Stateless network functions also scale better and are readier to be deployed in a container-based micro-service architecture [51]. Currently most of the telco clouds are still build on top of traditional virtual machines, but containers have clear benefits over them. For example, deployment times can be reduced and software updates are easier to deploy [69].

Figure 4 overviews how network functions and data storage might communicate through the telco cloud. Radio applications are deployed in virtual machines, which are located on cloud host machines. Data storage services can be on same or different host computers with radio network applications. The host machines are connected with modern data center networking, which means 10 Gigabit Ethernet (GbE) at minimum. On the host side, a virtual switch is deployed to route traffic to and from the virtual machines. In this scenario, the RTT of a network function communicating to the data storage and back can be divided in three main components. These are network processing on both computers, physical transmit network and time spent in data storage.

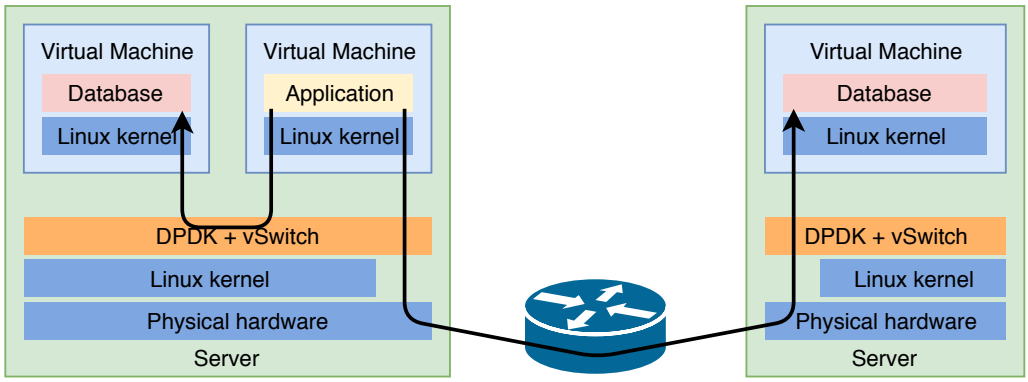


Figure 4: Overview of the network.

Here is a concise overview of the current achieved latency. According to previous studies it is possible to estimate the RTT to around 120μs [36, 53]. This estimate is broken down in Figure 5 and detailed in Sections 2.2.1 - 2.2.3. This estimate is to be taken with a grain of salt. It has been consolidated from multiple sources and there are a lot of contributors to the varying latency from exact hardware to software and measurement methodology. Other significant shortcoming of the estimate is the lack of taking tail latency in account. Low jitter is a very essential part of building stateless applications. However, the estimate should be adequate and give a good picture of which parts of the stack contribute to the latency the most and could thus be improved.

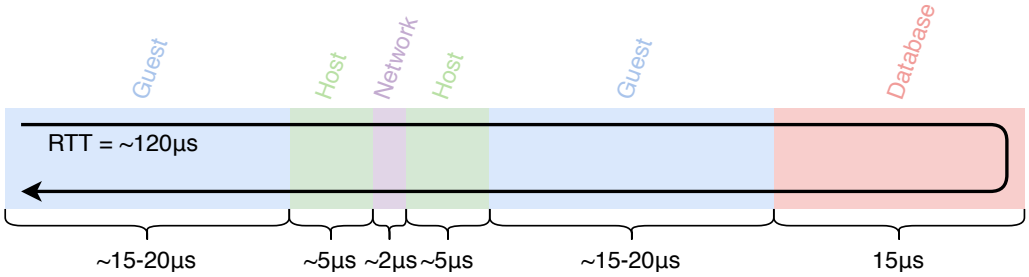


Figure 5: Estimated breakdown of the latency between radio application and call session storage.

2.2.1 Data Center Networking

Data center networking architecture is a broad topic which we briefly consider here for the sake of completeness. Both selected hardware and network topology have a significant impact on performance metrics such as latency and throughput. The only topology considered in this thesis is racks with top of the rack (TOR) switch. The topology outside the rack is also not considered, as even one full-sized rack can handle significant radio cloud processing. The current market has a huge amount of specialized hardware to optimize performance for specific metrics and use cases.

These include solutions like Infiniband and Myrinet, which have lower latency and better throughput than Ethernet [53]. There are also solutions which do not replace Ethernet, but use proprietary hardware like Field-programmable gate array (FPGA) Network Interface Controllers (NICs). Like Infiniband or Myrinet, these solutions are proven to offer very good performance. In FGPA solutions some of the software network stack is often offloaded to the FPGA itself. For instance, the high-frequency trading (HFT) world uses FPGAs to get as low as 1 μs wire-to-wire TCP latencies, which is one to two orders of magnitude faster than the best recent software implementations [57]. However, it is good to keep in mind that HFT is a very different use case than telco cloud.

In such as use case, the specialized low-latency networking hardware is not an option. As explained in Section 2.1.2 telco cloud significantly benefits from COTS hardware. In addition to that network latency and jitter are both low in comparison to other parts of the transfer between NF and data storage. According to Larsen's testing in 2007 TCP packet spends around 2 μs between the NICs, when software is not involved [53]. This includes around 3 ns per meter of cable, 300 ns in the switch, 670 ns in sending and 1.1 μs at the receiving end. It is important to remember that Larsen's tests are over 10 years old, and parts of the data are estimates. Nevertheless, even 2 μs is such a low latency that it is not significant compared to the total latency.

2.2.2 Database

Data storage and its features are a key element in building high-performance telco cloud. The choice of data storage does not only define the time spent on queries, but it can also determine which transport layer protocols can be used or how much work is required to integrate acceleration technology in the storage. In more specialized cases the storage selection can even replace Ethernet with proprietary interfaces in between servers to minimize the latency.

Telco cloud sets different requirements for databases as opposed to regular use cases. Many databases offer quite extensive features, but in case of call session data storage, a very simple key value storage with get and set operations is enough. Another peculiarity in selecting data storage is defining the meaning of "fast". Most of the especially fast databases are built with throughput in mind. In case of telco cloud only latency and jitter matter. Data rates are not very high, but packets should be handled as fast as possible. In telco cloud, a shared database should be also capable of serving multiple clients at the same time without significant performance degradation.

Redis [15] is an example of a database which could be suitable for telco cloud. It is a fast and reliable in-memory database and has done well in tests against other fast databases [26, 50]. Even though Redis is one of the fastest databases using Ethernet as a communication channel it largely undermines its strengths without any tweaks. In fact, Redis only supports TCP as a transport protocol, which does impact on latency. As a matter of fact, TCP is built for long and unreliable connections, while those in a data center are not. This creates overhead which could be avoided. [1]

[1] Additional details on TCP will be provided in Section 3.1.1

In Figure 5 the latency caused by the database is found to be 15 μs. This estimate is based on doing 1024 byte set operations on a local Redis-server through an UNIX socket application programming interface (API) [2]. In the test 90$^{\text{th}}$ percentile latency was consistently around 16μs. However, tail latencies were quite spread out, and even 95$^{\text{th}}$ percentile latencies are considerable worse.

2.2.3 Operating System Network Stack

Last part affecting the RTT between the application and the data storage is the kernel network stack. In the worst case scenario, the packet has to be processed through the kernel stack four times per direction, as it moves through both host kernel and guest kernel in both servers.

Processing done in host and guest systems is not identical. The host system only handles layer 2 (L2) and layer 3 (L3) since the application is located on a guest machine. In addition to handling L2 and L3, the guest system also needs to carry out layer 4 (L4) processing. In most cases L4 is TCP, but this could also be some other protocol, for example User Datagram Protocol (UDP), Quick UDP Internet Connections (QUIC) or Fast and Secure Protocol (FASP).

A different option is to use the DPDK's datapath at the host to provide direct communication between the NIC and the virtual switch (vSwitch). DPDK[3] is a well-established solution and it provides solid low-jitter processing with less than 5 μs latency [56]. However, DPDK does not have TCP termination implemented so we rely on Linux kernel L4 processing at the guest where the application and the data storage are located.

Chuanxiong and Shaoren [36] have measured and broken down the latency of the TCP/IP stack of the Linux kernel. They concluded that sending a TCP packet took 22.5 μs, which of 15.8 μs was spent before the IP layer. Receiving packets is a little bit more expensive: 36 μs, 22.9 μs of which used after the IP layer. In their testing they used payloads of 1024 bytes, which is close to real-life loads. From these results, we can estimate that TCP termination in Linux kernel takes around 15-20 μs per end, which would mean four times that number when calculating the RTT. This means that it is a significant part of the RTT and improving it could provide noteworthy results in the total RTT. TCP/IP acceleration is the main candidate for reducing latency in a call session storage, as discussed in Chapter 3.

2.3 Reducing Latency

As shown in Figure 5, two main latency overheads in the RTT between the NF and the data storage are the TCP processing in the Linux network stack and the data storage itself. There is a wide range of solutions which can improve these. Assessing the solutions is quite complex, as they all introduce different type of costs, variety of constraints and huge differences in time to develop and deploy. We detail these next.

[2]The environment used for testing is described in more detail in Chapter 4

[3]Detailed in Section 3.1.3

Memory-Centric Computation

First, there are proprietary data storage solutions, which can provide ultra-low latency even in comparison to fastest of in-memory databases. In the fastest end of selection is memory-centric computation [39]. In memory-centric computation memory becomes the center of computing instead of central processing units (CPUs). In practice this means placing boxes of shared Non-Volatile Memory (NVM) in racks or even having racks of shared memory to be used by the compute nodes. Applications write directly to the memory via special interfaces, as Ethernet is not fast enough for the speed that memory-centric computing can achieve. One example of such interface is Gen-Z, which is used for example by Hewlett Packard Enterprise (HPE) in their The Machine [52].

At the moment, memory-centric computation comes at a high cost. Data centers need to be accommodated with memory racks and servers need to have Gen-Z interfaces. Also, the cost of wiring can be surprisingly high. At the moment memory-centric hardware could be considered to be quite far from COTS hardware. However, this might change in the future if the market starts to adopt such a new technology. Also, the Gen-Z interface is an open standard, so it could become an essential part of the telco data centers.

Remote direct memory access

Direct memory access is also possible without expensive memory-centric hardware. Remote direct memory access (RDMA) means making direct changes to another computers memory. There are multiple different implementations of RDMA. Some of the implementations use special hardware for networking, but there are also solutions which use Ethernet [65]. RDMA could provide a similar solution compared to memory-centric computation, but with a much lower cost. However, direct memory access is still a relatively new technology and has to be studied more for the use case.

Replacing TCP

Without going to technologies requiring special hardware and huge architecture changes, there is still much more that can be done to reduce latency even when using standard in-memory databases. One of the main latency concerns of current in-memory databases is TCP transport. In short, TCP has many features which keep track of packets and optimize throughput over latency. Tweaking those can help up to a certain extend but replacing TCP with a more suitable protocol is a definite option. In a cloud environment lighter protocols work well, as it is very unlikely to lose packets on data center-grade networking. The most known alternative for TCP is UDP, which is very latency oriented at cost of reliability. Both UDP and TCP are old protocols, and recent development has also offered many protocols like QUIC and Data Center TCP (DCTCP), which offer functionality between these two. Replacing TCP requires development and maintenance work on both the server and the client side but could be a simple and cost-effective way to reduce latency.

TCP/IP Acceleration

As an alternative to removing TCP there is also the possibility to accelerate TCP further than just making tweaks to its functionality. There are some frameworks which do the TCP termination in Linux user space and most of them are built to be used with DPDK. These techniques are usually referred as TCP/IP acceleration or kernel bypass and they are detailed in Chapter 3. This thesis concentrates on TCP/IP acceleration, because from all the options, it has the most potential to give fast results with low cost and development work. However, frameworks which offer TCP/IP acceleration are quite new and built for very specific tasks, so their suitability for telco cloud environment needs to be carefully analyzed.

3 Accelerating TCP

In most of the cases TCP/IP acceleration is done by replacing the Linux kernel space TCP processing with user space stack. There are multiple open-source frameworks which provide either their own TCP stack or one borrowed, for example, from FreeBSD. This thesis focuses on these open-source options. In addition to TCP/IP acceleration, the term kernel bypass can also be used.

This chapter begins with an explanation of essential technologies to explain TCP/IP acceleration. These include an introduction of TCP, Linux kernel TCP stack and DPDK. After introducing the native Linux stack, section it describes different frameworks used to accelerate TCP/IP.

3.1 Background

Before proceeding further, we introduce the necessary background to understand TCP Acceleration. Accordingly, we start by reviewing how TCP works and which are the main problems in terms of latency. After that, the TCP/IP stack of the Linux kernel is detailed as a term of comparison for accelerated solutions. Lastly, DPDK is introduced, since it is the most widely used software for IP stack acceleration. This is also because of its reputation and wide support for hardware.

3.1.1 TCP

The transmission control protocol has been used for a long time. It was introduced in a request for comments (RFC) dated 1981 and the development of TCP started before that in the Advanced Research Projects Agency (ARPA) [23]. After many years most of the traffic in the Internet are still TCP: Around 95%, according to [55]. The original RFC specification is still valid, but that does not mean that TCP is exactly the same as forty years ago. For example congestion control was first introduced by V. Jacobson in 1988 [47]. The related TCP variant is known as TCP Tahoe [25] and has been followed by tens of different congestion control algorithms.

TCP is designed to be a reliable connection-oriented protocol. As a consequence it works very well in unreliable networks and guarantees in-order delivery of packets. It is also an end-to-end protocol, meaning that it only establishes a connection between two endpoints, i.e, does not support multicast. The ability of TCP to work with unreliable networks is both a strength and a weakness. In its original use in the military, or even in a modern commercial network, reliable communication is extremely important, but the additional functionality to achieve this can restrict the performance of the network in some cases. For example, data center networks are highly reliable and have precise requirements on latency and throughput. In this environment TCP is not optimal, but there are many ways to improve the performance on such settings.

Fundamentals of TCP

A connection must be established before sending or receiving TCP packets. Even this fundamental feature of TCP affects latency. If a connection is not already established, it takes considerably more time to send a packet. Luckily, there are many different ways for overcoming this problem. For example, a connection can be reused instead of terminating and re-establishing a new one. After a connection has been established, every packet sent has to be acknowledged (ACK) by the receiver. Also, this can cause problems in some situations. For example, if sent packets are small and each packet is acknowledged, a very significant portion of the connection is used for ACKs and headers [34]. Each sent packet has a sequence number and the receiver is responsible for reordering packets and discarding duplicates.

Simply sending packets and acknowledging them as fast as possible is obviously not enough. TCP has several features to adjust the flow to be as optimal as possible. The main contributors are flow control and congestion control. Flow control is a receiver-side feature, which is used to adjust the sending speed, so that the receiver can keep up with incoming packets. Flow-control data is sent in ACK packets and sequence numbers are used to tell the sender which packets can be sent. This method is called the sliding window, since the range of allowed packets advances overtime accordingly [23].

Congestion Control

Congestion control aims to adjust a TCP flow according to the network conditions. The standard way to carry out congestion control takes place only at the sender. However, there are many different algorithms, some of which also require modifications to the receiver or even the routers. A window is also used in congestion control, and it is usually adjusted by the data gathered from ACKs. For example, many algorithms use the RTT and lost packets as input. Most of them have three main phases at least in some form: slow-start, congestion avoidance and retransmit. The goal of *slow-start* is to get from zero packets to a rate of equilibrium of the path. Even though the name is slow-start, it is often an exponential or a very fast process. *Congestion avoidance* is the default sending mode of TCP and its behavior is very dependent on the specific algorithm used. Congestion avoidance is interrupted by packet loss. Usually at this point algorithm does some kind of *retransmit* and goes back to congestion avoidance with a lower data-rate which starts to increase slowly [47].

Tahoe and Reno are few of the first congestion control algorithms. Reno has also been updated with small changes in TCP NewReno [41]. The operation of both algorithms is based on two main variables: a congestion window ($cwnd$) and a slow-start threshold ($ssthresh$). When sending starts, TCP is in the slow-start phase and $cwnd = 1$ packet. Every time an ACK is received, $cwnd$ is increased by one and sending continues. This leads to an exponential growth of $cwnd$. At some point either a timeout or a duplicate ACK occurs and the algorithms need to enter the fast retransmit phase. An ACK timeout means that packet is lost, which hints that the line is congested. Duplicate ACKs signify incorrect order of packets.

Both of these situations are dealt as loss in Tahoe. All unacknowledged packets are retransmitted, *ssthresh* is set to half of *cwnd* and *cwnd* back to one. This leads to a new slow-start, but this time the slow-start turns to congestion avoidance when *cwnd* grows past *ssthresh*. In the congestion avoidance mode *cwnd* is increased by $1/cwnd$.

Tahoe has an obvious issue in its implementation. When loss occurs, both bandwidth and latency are significantly affected. Even though slow-start recovers very fast, latencies grow more than what is allowed in a telco cloud. Reno builds from Tahoe but tries to solve this obvious problem with simple changes. Reno handles timeouts similar to Tahoe, but in case of duplicate ACKs there it does not perform slow-start if packets are still moving. In this case both *ssthresh* and *cwnd* are reduced to $cwnd/2$ so congestion avoidance can continue without slow start. Reno also uses fast recovery mode for duplicate ACKs. A packet which is missing the ACK is resent and the algorithm waits for one RTT. If the ACK is received, it can continue straight to congestion avoidance, but otherwise a timeout occurs. Reno has less problems with latency and bandwidth, since duplicate ACKs do not trigger slow-start. Still, Reno is not ideal for low-latency and low-jitter scenarios. New Reno improves Reno by sending packets to the transmit buffer during fast recovery [41]. This significantly increases performance under high error rates.

Tahoe and Reno built the foundations for TCP congestion control but they are not very relevant for modern cloud scenarios. However, they demonstrate very well the relation between congestion control and lost/reordered packets. In their study of gaming traffic Chen et al. found that packet reordering and loss are a major component in latency and significantly affect jitter [34]. In their measurements, lost packets increased jitter by 60%, so everything that can be done to improve congestion control is beneficial to the telco cloud.

When diving deeper in the latency and jitter of modern cloud, new congestion control algorithms should be introduced. While there are many specialized congestion control algorithms build with data centers in mind, there are few even more relevant options. For example, recent versions of Linux kernel have used many different default congestion control algorithms.

Causes of Latency Overhead

Even though congestion control is an important part of TCP flow management, there are many other details which cause latency problems, but can also be addressed to some extent. Minshall et al. [60] they present a case where co-existence of Nagle's algorithm, delayed acknowledgements and The Network News Transfer Protocol (NNTP) create significant latency problems. Even though their case concentrates on how these functions do not work together, they can also be problematic alone.

Nagle's algorithm is implemented in TCP to reduce the number of small packets in the network. It prevents sending small packets unless everything sent previously is ACKed. Small packet is one lower than the Maximum Segment Size (MSS). In a normal network Nagle gives a significant benefit, as the number of small packets decreases notably and the impact on latency is low. This only causes problems

when combined with delayed ACKs, i.e., the receiver does not send ACK instantly but waits to see if it could piggyback the ACK into some data packet going in the other direction. This creates the situation where a packet waits for an ACK and an ACK waits for something to be sent to the other direction [61]. However, Nagle and delayed ACKs are both problems in latency and jitter sensitive networks even if they do not interfere with each other. Nagle's algorithm can introduce extra latencies of one RTT while waiting for an ACK. This is very significant on a low-latency application. Luckily, Nagle's algorithm can be turned off by means of the TCP_NODELAY flag. In low latency applications it might be also a good idea to reduce the delayed ACK timer so ACKs will not wait too long on the receiver side [68].

Since the most used congestion control algorithms are created for the Internet, there is room for improvement. In a data center environment the RTT is very predictable and much lower than in the Internet. TCP has timers for timeouts such as connection, retransmission and probing for a lost packet. These timers are set to relatively conservative values and performance on low-latency environment can be improved by setting these to lower values. This helps especially with tail latencies, while problems are detected and corrected faster [68].

3.1.2 The TCP/IP stack in the Linux Kernel

Linux is used everywhere and in a wide variety of scenarios. As a consequence, its network stack needs to cover very different use cases. Among these, ultra-low-latency networking has not received much attention. In this chapter, we take a latency-oriented look to the Linux networking stack and especially its TCP implementation.

Overview

The receiving and sending side of the Linux TCP/IP stack are obviously different, but most of the components are still same. This section refers to the receiving side as example, as it is a bit more complex, uses more time and introduces more severe latency problems [36, 71]. The Linux TCP/IP stack can be divided into three main parts: device driver, kernel space and user space [71].

The kernel networking stack mainly relies on two data structures: *sk_buff* and *sock*. *sk_buff* is used to save packet data during processing. Almost all functions of packet sending and receiving are called with the *sk_buff* passed as a parameter. If any packet field is updated, a change is made in *sk_buff*. The *sock* structure is used to keep information on a connection. *sock* is created every time when a socket is created in the user space. The *sock* structure is extended with *tcp_opt* where all TCP-specific data is saved. [64]

Device Driver and IP Processing

The packet receiving process is illustrated in Figure 6. To receive a packet an empty *sk_buff* has to be already initialized. Empty *sk_buffs* are located in *rx_ring* ring buffer. When a packet arrives to the NIC, the device driver will invoke the Direct

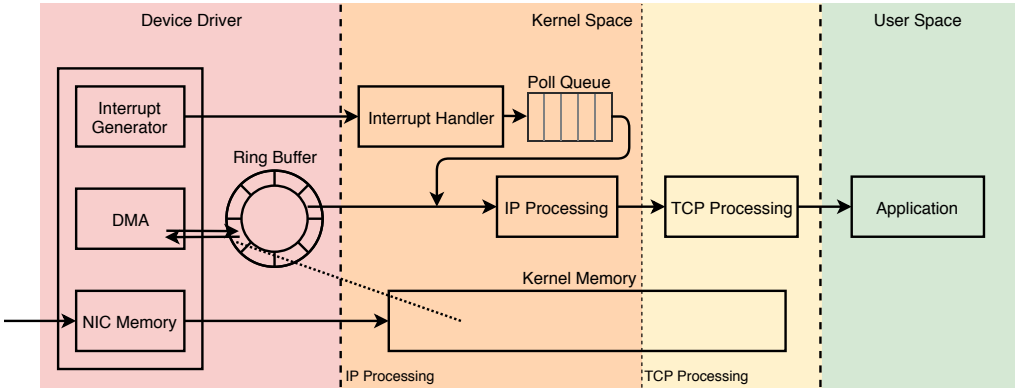


Figure 6: Overview on receiving a packet in Linux. [64, 32]

Memory Access (DMA) engine, which tries to save a packet descriptor in a free *sk_buff* and the full packet directly into kernel memory. As soon as the packet is completely transferred, interrupt handler is called from the NIC. The interrupt handler has multiple tasks, but first it will add the device into the poll queue of the CPU. This means that soon the CPU will poll the device to get the packets which are in *rx_ring*. After a packet is passed further, the packet descriptor in *rx_ring* can be reinitialized and used again. If packets arrive when *rx_ring* is full they are dropped. Because of this *rx_ring* also defines an upper limit for the connection window and limits the throughput if *sk_buffs* is set to too low [32, 71].

Every time a packet is pulled from *rx_ring*, function *ip_rcv()* is called. It does the basic checking of the packet. If the packet is not corrupted *ip_rcv_finish()* is called to handle routing. If the packet is set to be delivered to the current hop, IP fragment reassembly is handled. After that, the IP header is trimmed and layer 4 processing can start. In case of TCP stack this means that *tcp_(v4/v6)_rcv()* is called [71].

TCP Processing

In the Linux kernel TCP packets can be processed in either process context or interrupt context. Considering latency, all packets would ideally be processed by interrupt context, since that is considerably faster. The difference is significant when the processor is heavily loaded. The interrupt context gets still high priority and is executed almost instantly. However, process context is a user level process and has lower priority, thereby ends up waiting for processor time. Packet handling by interrupt or process context depends on a few variables related to the status of receiving end [71].

The first step of TCP preprocessing is checking the TCP header. After checking the header *tcp_v4_lookup()* is run to check if the packet has a corresponding socket. If there is no socket, the packet is dropped. Otherwise processing aims to get packet to user space application as fast as possible. This is illustrated in Figure 7. As seen in the figure, the first matter to confirm is that if the target socket is locked or open. If the socket is locked, the packet is placed in the backlog queue. If the

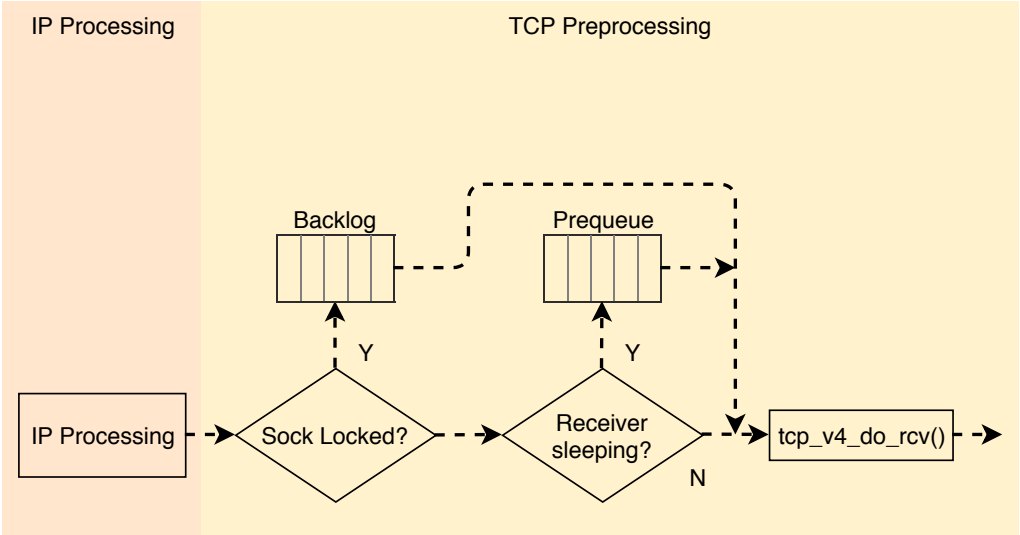


Figure 7: TCP packet preprocessing in interrupt context. [64]

socket is open, processing confirms the status of the data receiving. If processing is sleeping, the packet is put to prequeue. Both backlog and prequeue are emptied by the process context later. Only in case of prequeue overflow the interrupt context is used, because it need to be processed as soon as possible. If the packet does not end up in prequeue it is immediately passed to the main TCP processing function *tcp_v4_do_rcv()* [71].

The first step in TCP processing is to check if the packet can be delivered directly to the user space through the fast path. The fast path uses header prediction technology introduced in RFC1323 [48] to process the packet ultra-fast if the sequence number of the packet is expected [64]. After this, the packet can be directly sent to the user space if the right process has packet receiving turn and there is enough space in the memory location provided by the application. If these conditions are not met, the packet is sent to the slow path which is illustrated in Figure 8. The first action in the slow path is to check if the packet is in sequence; if not, it is sent to out of the sequence queue. Packets from this queue are transferred to the receive queue when missing packets come later. If a packet is in sequence in the slow path check, it can be still copied to the user space directly if the requirements for the interrupt context are still met. Otherwise the packet is sent to the receive queue to wait for the process context [71].

Packets which are not delivered rapidly by the interrupt context are delivered by the process context. The process context starts by locking the socket, which also means that the interrupt context cannot push packets during the process context and packets are not mixed. After locking the socket, the process context empties all queues except the out of sequence queue. It first copies data from the receive queue, which is trivial since packets are already in the right order. After that, the prequeue and backlog are emptied, and packets are processed. When all queues are empty, the socket is unlocked. The application has defined how much data it should receive. If it did not receive enough, it will remain waiting for data. Then it wakes

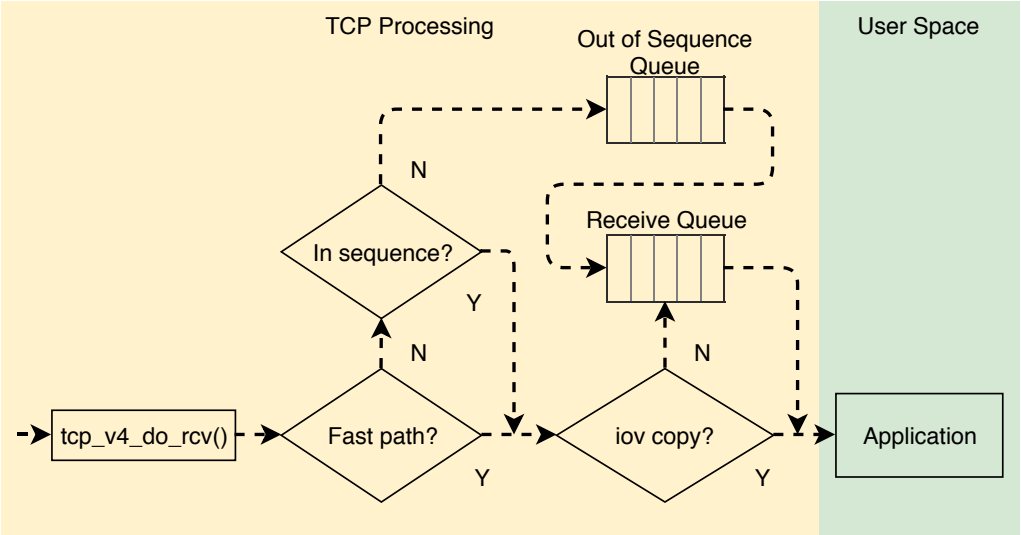


Figure 8: TCP Packet processing in interrupt context [64].

up and starts all over again by locking [71].

TCP processing in Linux has few weaknesses latency-wise. In a worst case scenario, a packet might end up waiting in the backlog for more than one second [71]. This is already enough to trigger retransmits and cause significant problems even in the context of a normal user. This happens due to the load of the server keeping the process context waiting for its turn. When analyzing the stack from an ultra-low-latency standpoint, even the number of queues in the process context affect performance. The stack is quite efficient with its processing, but in certain use cases extra wasted processor cycles do not matter if latency can be reduced.

3.1.3 DPDK

DPDK [3] is a set of libraries and drivers for fast packet processing. It runs mostly in Linux user space and supports a wide variety of hardware. DPDK was first introduced in 2010 by Intel under an open source license and has now gathered a wide open source community around it. Nowadays it is a Linux foundation project and very well recognized in the world of accelerated packet processing. One of the key ideas in DPDK is poll mode driver (PMD). In its essence, PMD means moving from traditional interrupt-centric way of sending to constantly polling for new packets instead.

Polling is not a new invention, but only recent advancements on hardware has made it a plausible candidate for sending and receiving traffic. To do efficient polling-based transmit, at least one CPU core has to be fully allocated to PMD only. Any interrupt by other applications or kernel slows DPDK down and causes jitter. With PMD packets are directly pulled to the user space from the *rx_ring* buffer of the NIC. Also transmit is done directly from the user space to *tx_ring* buffer. Figure 9 shows the PMD receiving flow and can be compared to Figure 6 which refers to processing in the Linux kernel.

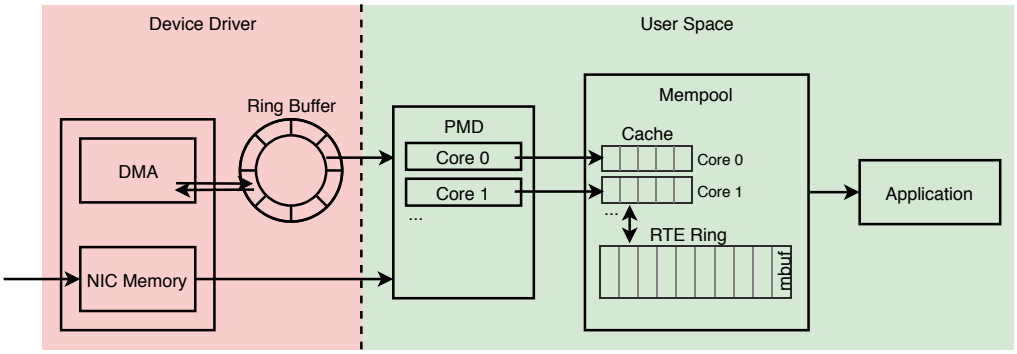


Figure 9: DPDK PMD packet processing.

DPDK includes three major libraries: Mbuf, Mempool and Ring. The mbuf library provides the structure to save packets to in DPDK. It can be generalized to be corresponding to the *sk_buff* structure of the Linux kernel. Mbufs are stored in mempool which does allocation of the space from memory. Mempool handles are usually based on a ring buffer implemented in the ring library. Mempool also implements a small cache which has a table of pointers for each core. This way, recent packets are even faster to process further. Multiple DPDK cores can share mempool, but caches are only accessed by the core which owns the cache table. If there is only one interface which DPDK is receiving data from, it only needs one core. However, if there are multiple interfaces or NIC rx/tx buffers, each needs its own processing core [4].

Memory for the DPDK is allocated in hugepages. Hugepages are crucial for the performance of DPDK. Normally memory is allocated in 4KB pages, which are enough when there is only a limited amount of data or no performance constraints. Normal pages are slow due to memory. The more memory is used, the harder it is to figure out which of the 4k pages has the right data. For this reason if the application needs a lot of memory, it is better to allocate the memory for that application in hugepages. In Linux hugepage sizes are usually 500-fold, so the normal pages are either 2MB or 1GB [11]. The hugepage cannot be shared with multiple applications, so too large pages should be avoided. For example, it might sound appealing to reserve a 1GB page for 500MB worth of data so it would fit in one page. However, in 2MB pages 500MB of memory is saved, and the lookup for 250 pages is still very fast.

Another important factor in DPDK is the number and availability of processing cores. PMD requires at least one core in the use of DPDK. Without any setup there is nothing which would prevent other applications from using that core other than kernel scheduling, which would run other applications on other cores if they are available. However, this does not take into account two things. First of all, if the whole processor is running at a 100% load, the scheduler is going to assign work for the PMD core. At that point the performance, and especially the jitter will take a great hit. This can be prevented by pinning the core for DPDK. In Linux core pinning is done by modifying the grub command line [9].

As DPDK is in direct contact with NIC over a Peripheral Component Intercon-

nect (PCI) lane, PCI drivers are a crucial part of the performance. The ideal driver for DPDK is Virtual Function I/O (VFIO). VFIO uses the input–output memory management unit (IOMMU) for secure and robust communication. VFIO requires support from kernel, BIOS and hardware, which brings some limitations to its use. In addition, the IOMMU groups can cause problems with some hardware. For example, if the hardware is a NIC with multiple ports and the ports happen to be in the same IOMMU group, all or none of the ports must be bound to VFIO. Alternative for VFIO is Userspace I/O (UIO). Performance of UIO is similar to VFIO in an ideal case, but it lacks features and is less secure. If UEFI secure boot is in place instead of BIOS, Linux can disable UIO completely [9].

DPDK provides ultra-fast communication from the NIC to the user space, but it is not enough for regular applications. In fact it does not provide TCP termination, so it cannot process packets directly to the applications like database servers or clients. This is also not on the roadmap of DPDK, since the use cases for DPDK are different and community expects L4 to be handled by some other project. The main use case for DPDK is delivering traffic to a virtual switch, which can then route traffic forward. Since DPDK provides an ultra-fast way from the NIC to the user space, basically all frameworks for TCP/IP acceleration are using DPDK for fetching the packets.

3.2 TCP/IP Acceleration

DPDK together with vSwitch can handle traffic up to Layer 3 (L3), but to fully bypass the kernel and use applications like database servers, TCP acceleration is also needed. In this chapter, the two most well-known acceleration frameworks are introduced, i.e., VPP and F-Stack. In addition, few smaller projects are briefly overviewed. TCP acceleration frameworks take TCP processing out of the Linux kernel and to the user space. Most of them use DPDK to receive packets directly from the NIC.

3.2.1 Vector Packet Processing (VPP)

VPP is a packet processing framework which offers modular vSwitch and vRouter. It originates in Cisco's VPP which is widely used in Cisco's products, but is now available as an open source project. VPP is a key component of FD.io - The Fast Data Project of Linux Foundation [21]. VPP's name comes from the fundamental way of processing packets. Regular scalar packet processing processes one packet at the time; in contrast VPP processes packets in vectors, which are essentially lists of packets. This reduces overhead and increases cache performance. VPP is especially attractive for call session data storage applications, since it also has an user space TCP stack. VPP is also promising, since it is one of the largest projects providing user space TCP and has an active developer community [22].

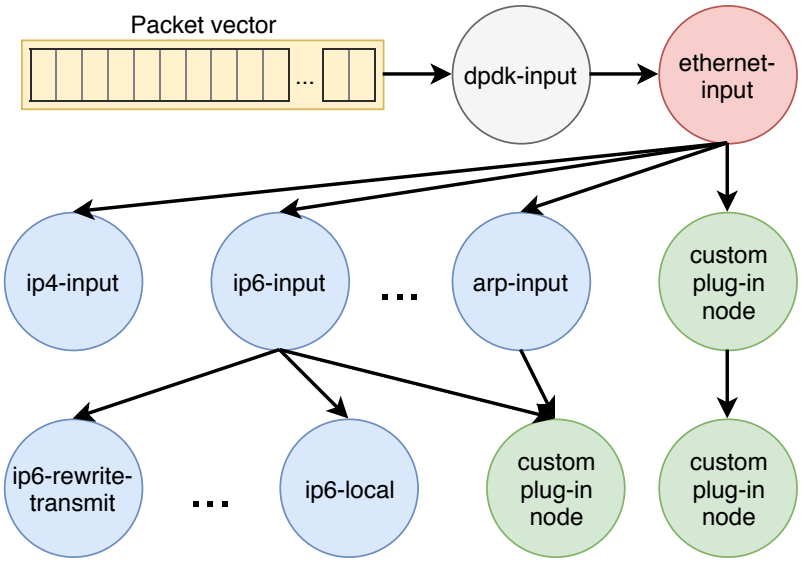


Figure 10: Example of VPP graph processing.

How VPP works

The core of VPP is the packet processing graph illustrated in figure 10. Vectors of packets come into the system and are processed according to the graph. If packets are not all the same, the vector is split in different nodes. All the basic processing in VPP is done in vSwitch/vRouter, but due to the high modularity users can create their own nodes to process the packets. As is apparent from the Figure 10, DPDK is common way to attach the NICs in the VPP.

The communication between VPP and DPDK is detailed in [38]. Let us consider the receiver, as described in Section 3.1.3 DPDK busy loops the CPU and gets packets from the NIC. In particular, DPDK fetches packets in bursts of size b, which currently defaults to a maximum of $b_{max} = 32$. This default is based on the current tests on modern hardware. Similarly, VPP has vector size of v, which is limited to a maximum of $v_{max} = 256$ packets. Around hundred packets is the best setting for high performance in VPP. Note that VPP does not start vector processing every time it receives packets from DPDK. If DPDK delivers b_{max} packets, there is a chance that it will bring another burst immediately, so the VPP waits and fills the vector. The vector processing starts each time the DPDK brings $b < b_{max}$ packets or when $v = v_{max}$. The first case means that there are no more packets at that given instant, so there is no reason to wait. The second case means that the system is achieving the limits of the VPP performance since that is the point where VPP cannot run any faster and the receive buffer of the NIC can fill up easily, resulting in dropped packets.

As mentioned earlier, one of the main performance gains of the VPP comes from better use of caches. Modern CPUs have multiple caches, which differ in latency and size. The L1 cache is the smallest and fastest, so ideally it is used as much as possible while running a program. The L1 cache is divided into instruction cache and data cache. In case of packet processing the instruction cache is the most important

factor. If a packet is processed in a scalar manner, so many instructions are used for processing the packet that when the next one starts the needed instructions are not in the L1 cache anymore. This is an example of cache trashing. With vector packet processing, where processing is done for all packets in the vector, one node at the time, L1 cache usage is much more efficient, which leads to better performance [38].

VPP Hoststack

VPP is a project still developing very fast. In version 18.07, the latest one at time of writing, there is a functional hoststack which can be used through many different APIs, but it is also under refactoring and the APIs might be changing in the process. In the current VPP version there are three main ways to communicate with the hoststack from a third-party application. Lowest level of these is a binary API, which requires tedious integration work for it to be used with the application, and it is not intended to be used directly for that. VPP communications library (VCL) offers a higher-level API to build software against. VCL is currently the recommended way of using the VPP hoststack. The last option is using VCL with LD_PRELOAD. This method does not require integration work, but does have a very limited compatibility with applications [37].

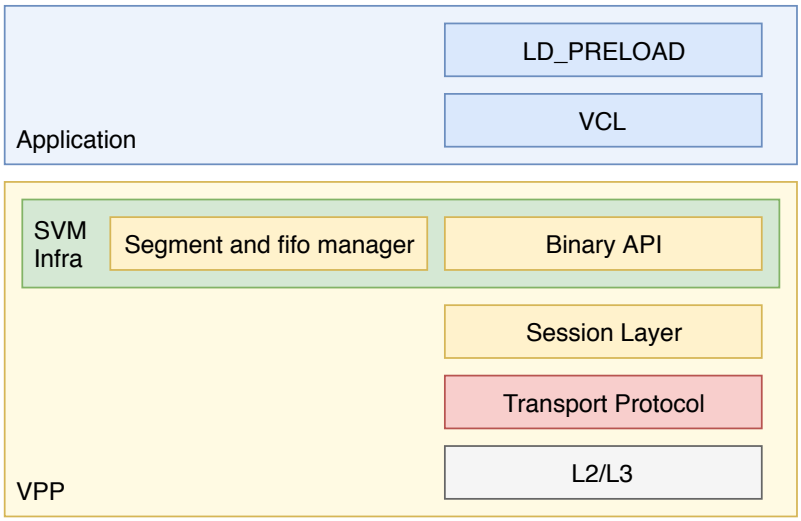


Figure 11: Overview of the VPP hoststack [37].

Raw session layer binary API is the lowest level API provided in the VPP hoststack. It does not provide support for asynchronous communication [37]. It is not meant to be used for integrating applications to VCL. This can be done if the limited functionality is not a problem from the standpoint of the application, but it is not recommended and there are no guarantees on stability of the API.

VPP Communication Library (VCL) API is the most modern and recommended way to leverage the VPP hoststack [40]. It does provide its own implementations of Linux I/O event notification facility epoll which provides more support for integration work [37]. Despite the fact that the VCL API is the recommended way

of accessing the hoststack, VPP's own socket test program seems to be the only publicly available application which uses the VCL API. However, the VCL API is still quite new and has not yet stabilized, so its support is expected to be better in the future.

VCL also has a POSIX API which can be used through LD_PRELOAD [37]. LD_PRELOAD allows a user to load any libraries before others when starting applications in Linux. This enables the user to replace any function provided by other libraries. This way LD_PRELOAD can be used to link an application to VCL without changing the application binary as long as it uses the POSIX API. This could be one of the key selling points of the VPP as it is the only acceleration framework which supports integration without changing the software. There is very limited information available on the use of LD_PRELOAD in VPP, so its actual status will be examined in testing part of the study.

3.2.2 F-stack

F-Stack is an open source high-perforce network framework [6]. Like VPP it is based on DPDK and it has its roots in a major technology company, as Tencent Cloud has originally developed F-Stack as a countermeasure to distributed denial-of-service (DDoS) attacks. Due to its very specific original use case, it is a much more limited tool, which can be both a strength and a weakness. It ensures that development is not targeted to irrelevant parts of the framework but, on the other hand, it might mean that out of scope use cases might be left unsupported.

Like VPP, F-Stack provides an API to link applications against it. In comparison to VPP, they only have one API, which has good documentation [5]. The API does provide a standard Kqueue/Epoll interface for ease of the integration. F-Stack also has more applications which are already supported by the development team. The list is not long, but both Nginx and Redis are widely used applications and makes testing the platform notably easier as both of them also have multiple already-existing testing tools. However, the main question is how well maintained these interactions are, as the supported Redis version is quite old (i.e. version 3.2.8, which has been outdated by another stable release in 2017).

F-stack has an interesting approach to their user space TCP stack. Building and maintaining a full TCP stack with all of its functionality can be costly, so they have integrated the TCP stack used in FreeBSD 11.01 to cut on development effort. However, the FreeBSD stack is not directly copied, but a large amount of irrelevant features are removed to provide better performance [6]. This makes a detailed comparison of the F-Stack TCP stack hard, as there is no extensive documentation on the optimizations made.

3.2.3 Other Solutions

There is also a handful of other frameworks providing similar functionalities as VPP and F-Stack. These other frameworks were left out of our study after a short evaluation of capabilities and activity of the projects. However, it is still meaningful to shortly summarize what these applications do.

The transport Layer Development Kit (TLDK) is definitely one of the most well-known projects in this category. Like VPP it is also an FD.io project [16]. Its idea is to provide an easy-to-use host stack for VPP [17]. TLDK is implemented with VPP functionality like graph nodes and plugins, but it aims to hide that from applications so it could use TLDK without knowing about VPP. Like F-Stack, TLDK also has already been integrated to Nginx. TLDK was left out from the closer look since it seems to be not active anymore.

Like F-Stack, OpenFastPath (OFP) leverages the FreeBSD network stack to provide high performance user space networking [12]. OFP is a very promising project and provides a wide variety of functionality. However, the project is still young, and as stated in their Technical Overview, the TCP implementation is only functional and not optimized for performance.

The accelerated network stack (ANS) is a DPDK-native TCP/IP stack. Like F-Stack and OFP, it also uses the FreeBSD network stack to provide its own stack in the user space [1]. As many others, DPDK-ANS provides an API with epoll implementation. DPDK-ANS does also include the longest list of software which has already been ported to work with it. This includes Nginx, Redis and Iperf, though not the newest stable versions. At the point of scoping the thesis DPDK-ANS seemed like a small and inactive project. However, it has potential for growth, as it aims to provide very narrow functionality instead of complete networking suite.

Dual Mode, Multi-protocol, Multi-instance (DMM) is yet another FD.io project [2], which could solve the problem of this study from another angle. DMM is a very young project started after scoping of the thesis. It tries to provide a framework between the application and the transport layer. The main idea behind DMM is not to lock on to a single implementation of the networking stack, but to provide an abstraction layer which can be used by both the application and the network stack.

4 Evaluation

This chapter details the experiment methodology. It first starts with an introduction of the measurement tools followed by a description of the experiment setup. Lastly, the chapter details the methodology used in the experiments.

4.1 Measurement tools

The selection of measurement tools is a key aspect in assessing different acceleration frameworks. Fair tool selection is challenging for multiple reasons. First, all the frameworks require applications to be modified in order to be used with their interface. This means that if a common testing tool is selected or a new tool is created, it has to be integrated to work with all the frameworks separately. Unfortunately, this was not in the scope of this research. Other option is to select tools from a pool offered by frameworks or other third parties. Unfortunately, all tested frameworks are so young that the catalogue of supported applications is very narrow.

While performing measurements, it is very important to identify what is to be measured and only focus on that. In this study the two main factors considered are latency and jitter. Unfortunately, most network performance testing tools are very much throughput-oriented. As a consequence, they often do not measure or report latency and jitter.

Another key aspect to successfully measure performance of TCP acceleration is to avoid any bias, namely, to ensure that only latency and jitter caused by the acceleration framework is measured. For example, databases like Redis can be used to conduct measurements where we can draw a conclusion that using acceleration provided a positive or a negative effect on performance. However, it can be hard to conclude if for example the jitter was due to the framework or the database server having problems. This can also happen outside of the testing software. For example, if other applications running on the server take CPU resources, this can manifest as jitter in the measurements.

It is impossible to eliminate or even acknowledge all factors affecting the measurement, however, the factors with the largest impact we tried to take into account. With the limitations in mind Redis and platform specific software were selected to be used in the evaluation to give a comprehensive picture of the frameworks. Unfortunately, there is no tool which works directly with all APIs of both considered frameworks, so the obtained results are compared to a baseline represented by the Linux kernel TCP stack.

Platform specific tools

As explained in Section 3.2.1, the VCL API provides the most suitable way to link the application to VPP. However, VCL API is still quite a new technology and there is not much support for it in the industry. The only publicly available testing tool for the VCL API is a socket test shipped with VPP and it focuses on bandwidth measurements. However, a higher data-rate in the test indicates better per packet

latency. The VCL testing tool supports both the kernel TCP stack and VCL, so it gives a good idea if VPP could offer any benefit over kernel.

Redis

Redis is a mature in-memory database, which can serve a significant amount of requests per second, thus is one of the most suitable databases for low-latency applications. Redis could very well be used in the call session data-storage use case. In addition, Redis is supported by both TLDK and F-Stack and can be tested with VPP's LD_PRELOAD.

Redis also has its own benchmark utility, which can be used for low-latency testing with some modifications. Such a tool, called redis-benchmark [66] offers back-to-back packet transfer, which means that it sends the next packet as soon as it gets a reply for the previous packet. The software is used from the console and can be configured through command-line options. Those used in this work are listed in Table 2. From the standpoint of latency measurements, number of clients and the data size are the most important ones. If both these values are too large, the redis-server may become the bottleneck of the system, and the tests do not tell anything about the latency in the TCP stack. However, if both the amount of clients and the payload size are low enough, redis-benchmark is a very good latency measurement tool since back-to-back transfers limit the bandwidth it can use. One important note is that increasing the number of clients does not increase the amount of sent packets: the number of packets is rather divided for sending clients.

Option	Function
-h <hostname>	Server hostname (default 127.0.0.1)
-p <port>	Server port (default 6379)
-s <socket>	Server socket (overrides host and port)
-c <clients>	Number of parallel connections (default 50)
-n <requests>	Total number of requests (default 100000)
-d <size>	Data size of SET/GET value in bytes (default 3)
-t <tests>	Only run the comma separated list of operations.
	The operation names are the same as the ones produced as output.

Table 2: Subset of redis-benchmark options.

In this study, some modifications were made to the C code of redis-benchmark. The main weaknesses of the tool were related to its way to report latencies. It does report latencies in milliseconds, which is not enough for a low-latency study. In addition to this, the tool reports latencies only by grouping the results by time, which makes analyzing the data difficult. Both of these problems were solved by around 70 lines of code in total.

The resolution of the latency measurements was changed to microseconds. This makes reading the output of the tool even harder, because the time groups are now divided in resolution of a microsecond. This is fixed by introducing 3 new options.

If none of these are used, the benchmark leaves latency reporting completely out. The new features are following:

- *--report_time* does keep the old functionality of latency reporting if that is ever needed. Only change to original is that resolution used is micro seconds.
- *--report_latencies* reports tail latencies of the test. These are grouped in 90th, 95th, 99th, 99.9th, 99.99th and 100 percentile latencies. This option is for initial testing and gives a good overview on the performance with a quick glance.
- *--report_full* is used to get per packet latencies in sending order. This tool is used to get all the data about a transmit so graphs and other in-depth analysis can be done.

In addition to changes in redis-benchmark, a tool was written to analyze the output of redis-benchmark with the *--report_full* option. This tool was written in Python and it calculates simple statistical numbers and draws graphs from the data. With multiple changing variables, some of the tests created a substantial amount of data, so an efficient handling was important.

4.2 Experimental setup

During the study, experiments were conducted in multiple different environments, but all the results presented in the thesis were run on the same setup. The test setup consists of two dedicated servers and a switch between them. This provided more control to the host side than using a cloud infrastructure. However, if any changes were to be found that would be required on the host side, all of the changes can be made also on a slightly different cloud architecture.

4.2.1 Hardware testbed

The servers used in the evaluation are two identical HP ProLiant BL460c Gen9s. Each server has two Intel Xeon E5-2680 v3 CPUs, which have 12 cores each and run at 2.5GHz. Both servers have 96GB of memory per processor which totals 192GB per server. The last important component in the setup is a HPE Ethernet 560M 10Gb network interface controller which uses a 10GbE Intel 82599 controller. The NICs in both servers are connected to a HPE 6125XLG Ethernet Blade Switch with a 10GbE downlink ports. Having two servers makes it possible to perform all the experiments in both single host and multi-host environments. In the single host environment, traffic only stays within one server, while in the multi-host environment the endpoints are located in different machines.

HP Proliant servers have many proprietary tools, like Integrated Lights-Out (iLO) which is used to manage the servers. Other than those, HP servers still are practically COTS hardware. However, there are some unfortunate limitations. In the terms of the TCP/IP acceleration study, the most severe problem is their VFIO

implementation. In fact, HP uses Reserved Memory Region Reporting (RMRR) to communicate management data of the server in some of their Proliant servers and that conflicts with VFIO [10].

4.2.2 Host

Three main tasks of the host system include starting the virtual machines, switching traffic to them and passing hardware resources like CPU cores or NICs. Virtual machines are used instead of containers, because that corresponds better to current telco cloud implementations. In cloud host machine could run some specific use case related to the operating system, but in this test the host system runs Ubuntu 16.04 with Linux kernel 4.4.0-134 which is also run on the virtual machines. In addition to the operating system the host needs to have vSwitch software and a machine emulator to run VMs. The whole host setup presented in this chapter is illustrated in Figure 12.

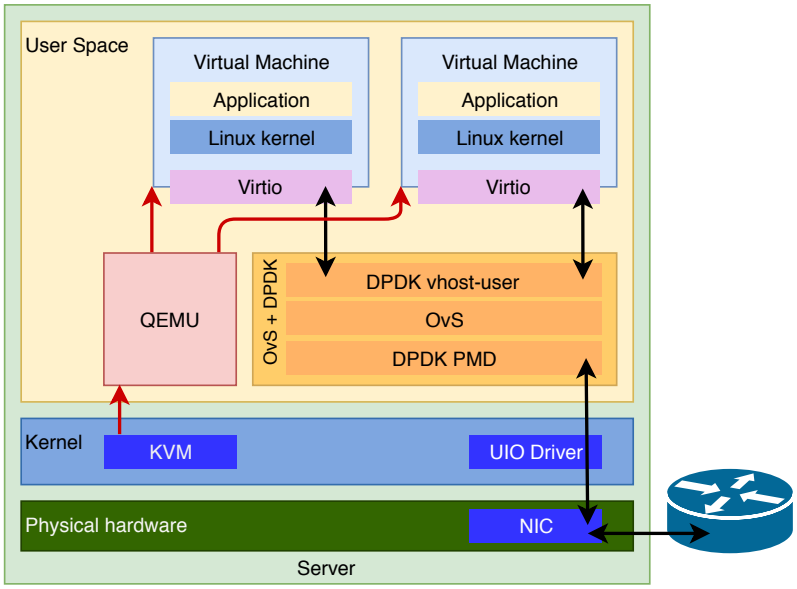


Figure 12: Overview of the host of the experiment setup.

vSwitch

Directing traffic to the virtual machines is facilitated with vSwitch. There are many different virtual switches, but in this study only Open vSwitch (OvS) [63] is used. OvS was released in 2009 and it has had enough years to develop into production-quality vSwitch. It is also open source and is well-recognized by software-defined networking (SDN) and NFV developers [46]. There are two types of OvS ports used in the test setup. Physical Ethernet ports of the NIC are connected to DPDK datapath. This way the packets are forwarded from the NIC directly to OvS without kernel intervention. The DPDK poll mode drivers take care of catching the packets nearly immediately, so the latency of the host is very low. Connection from the

host to the virtual machines is handled with DPDK vHost User. vHost User is a remarkably new technology released in 2014. It is based on an idea of sharing file descriptors between a guest and the host via Unix domain sockets [18].

Section 3.1.3 explained the most important performance tuning factors. In our test setup, we had plenty of extra processing power and no other applications running on the server, so CPU time was not a problem as it could be with poll mode drivers. Also the OvS DPDK setup guide [13] suggests that modern Linux kernels are so efficient that CPU pinning does not have a huge effect.

Since there was plenty of memory available, 10GB were allocated to OvS DPDK in 2M pages. All hugepage allocations of the setup were done in the startup by reserving pages upon booting. This was achieved in grub boot (etc/default/grub) with:

```
GRUB_CMDLINE_LINUX_DEFAULT=
    "default_hugepagesz=2M hugepagesz=2M hugepages=5000"
```

In addition to reserving memory, hugepage also needs to be mounted, which was done in fstab by adding the following line:

```
nodev /mnt/huge hugetlbfs pagesize=2M,size=10G 0 0
```

After covering hugepages and CPU, the last part of the OvS DPDK setup is the NIC in the PCI slot. As mentioned earlier, the DPDK datapath is used to connect the NIC to OvS, but the NIC must be used with DPDK-compliant drivers. As seen in Section 3.1.3, VFIO is the best option to use with DPDK and NIC. As the HPE Proliant servers do not fully support VFIO, an UIO driver was used instead. First, the loadable kernel module (LKM) has to be loaded in the kernel. This can be done with the modprobe utility. After loading the module the dpdk-devbind tool was used to bind the PCI device to the driver in the following way:

```
dpdk-devbind -b uio_pci_generic 00:04.0
```

Hypervisor

There are multiple options when choosing a hypervisor for cloud or the experiment setup. VMware is one of the first and most well-known virtualization platforms launched in 1998 [20]. Platforms like OpenStack still support VMware, but there are new contestants which take the majority of users. In 2005 QEMU user space emulator was launched [31]. It provides wide functionality and is open source. However, VMs run with only QEMU do not reach native processor speeds. In 2008 Kernel-based Virtual Machine (KVM) open source project was launched [42]. KVM is developed as a kernel module and provides virtually native performance. Both QEMU and KVM recognized their advantages and using a combination of QEMU and KVM is currently one of the most used virtualization techniques. QEMU and KVM are also used in the test setup.

The hypervisor is used to provide everything virtual machine needs. Configuring resources can be done in many ways, but in this setup simply the command

qemu-system-x86_ 64 is run inside bash script and configurations are done by giving options to that command. Most important of the options is *-enable-kvm*, which puts KVM into use.

Starting from the computing resources, memory is provided for the VM in hugepages. For the VM memory 1GB hugepages are used and 16GB of memory is provided per VM. Setting up the memory goes through same process as allocating memory for DPDK. This time "hugepagesz=1G hugepages=64" is added in the end of the grub command line and an own row is added in fstab, so it is easy to use both 2M and 1G pages and allocate them to different applications.

There are many options when emulating the CPU for the virtual machine. QEMU supports emulating many different processors with diverse feature sets. However, if there is no need to change the processor on the host side or run different processors in the cloud, the best performance is achieved with the *-cpu host* option which passes the host processor features to the VM. Other important CPU related option is Symmetric multiprocessing (SMP). In QEMU it can be used to determine how many processors, cores and threads are passed to the VM. In this test setup simple *-SMP 10* is used. This creates 10 unique CPUs [14]. Creating 10 CPUs in comparison to creating for example a 5-core machine with 10 threads has a difference in performance since the scheduler handles them differently, but the difference is expected to be low, since there are plenty of extra resources in the test setup.

Networking of the test setup was already covered to the edge of OvS. OvS DPDK vHost User has created the socket, which is now passed to the VM with QEMU. Also the receiving end in the VM is configured with QEMU. In this setup virtio-net-pci device is used to provide the networking device in the guest machine. Virtio [19] is the main IO virtualization platform of KVM and provides a very good performance. In addition to the link between the virtio device and vHost User also the device management link is configured with port forwarding. This way there is an easy connection to internet via the host machine's extra NIC and only the test traffic uses OvS and the 10GbE NIC, so there is absolutely no extra traffic going through the testing link. The configuration of test link looks like this:

```
-chardev socket,id=char1,path=${SOCK_PATH} \
-netdev type=vhost-user,id=mynet1,chardev=char1,vhostforce \
-device virtio-net-pci,mac=${MAC_ADDR},netdev=mynet1 \
```

Virtio is not only used to handle the network connectivity. As it is IO virtualization, it can also handle the hard drive of the VM. In the test setup there are multiple image files which have different guest configurations. These are loaded to the VM with the virtio interface. As a part of this thesis there were two bash scripts created totaling in around 200 lines of code. These scripts are used to easily launch wanted virtual machines with the intended options. They also include a possibility to change OvS to VPP by changing just one option.

4.2.3 Guest

The guest side of the VM includes two images, one for VPP and one for F-Stack. They both have different requirements for the accompanying software and their versions. For the majority of experiments CPU pinning and interrupt optimizations were turned off. An exception to this was an experiment solely conducted to evaluate the impact of those optimizations. Hugepages were allocated in 2M pages in a similar manner that was used in the host machines.

The software setup on both images is minimal. In the VPP image there are installations of VPP, Redis and DPDK. VPP was cloned from the Git repository [7]. From the repository version 18.07 stable was selected, as it is the most current stable code. DPDK can be built together with the VPP straight from the VPP repository. The newest DPDK version supported by VPP is 18.05, and it was the version used in our evaluation. Redis is installed separately and version 4.0.9 is used. redis-benchamrk.c was changed to the modified version before building as previously discussed.

F-stack is a less dynamic project and supports a bit older software versions. F-Stack comes with both DPDK and Redis. The DPDK version is latest long term support (LTS), 17.11. Redis version is 3.2.8. Our modified redis-benchamrk.c also works with this redis-server version without any problems.

As a part of the study, there were scripts created for the guest VMs to setup the machines for testing. This includes the setup for VPP which binds the NICs to the drivers, starts VPP and runs VPP testing. For F-Stack there is a very simple script which does the initial configuration of F-Stack, so it is ready to use after a reboot. Lastly, there is a simple script for redis-benchmark which loops packet sizes and number of clients, so everything can be tested on a single run. It also prints the results to text files which are ready to be processed with the Python script which calculates statistics and draws plots.

4.3 Methodology

This section outlines the experiments performed and the specific methodologies used therein. There were two main experiments, both consisting of multiple measurement runs. The obtained results are presented in Section 5.1.

4.3.1 VPP, F-Stack and Linux kernel with Redis-benchmark

The first experiment aims to compare the performance of VPP, F-Stack and the Linux kernel with a close to real-life software setup. The test is performed by running redis-benchmark over each of the three stacks in both single and multi-host environment. This is the main measurement of the thesis and other tests are planned to support the results gained. The major limitation in this experiment setup is the interface used for testing the performance of VPP. Using VCL directly would be the best option for VPP, but VCL does not have direct support for redis. Due to this, VCL is linked with LD_PRELOAD which might introduce performance and

stability issues. The second experiment is conducted to mitigate this problem in testing.

The experiment consists of six runs. The first two runs are conducted with both redis-server and redis-benchmark running on top of the Linux kernel TCP/IP stack for baseline. In the first run the test applications are on separate VMs but on the same machine (i.e., single host). In the second run VMs are running on different physical servers. Tests three and four represent single and multi-host tests with F-Stack. During these test runs F-Stack was only running on server side, since there is no implementation to work with the redis-benchmark software. In the last two runs, the client side is set to use VPP by linking redis-becnhmark to VCL with LD_PRELOAD. On the server side there is the Linux kernel TCP/IP stack since LD Preload was not stable with redis-server during initial testing.

Each test round consist of running series of SET and GET operations with a varying amount of clients and payloads. During each sequence 10,000 database operations are run with a constant client count and payload size. The amount of clients is varied from 1 to 24 and the size of the payload is varied from 4 bytes to 8 kilobytes. Even though Redis supports a wide variety of database operations, tests are only run with SET and GET, since those are the most relevant for the use case. These selected values should provide a wide enough view to cover possible real-life scenarios and indicate if different frameworks have specific weaknesses and strengths.

4.3.2 VPP and Linux kernel with VPP socket test application

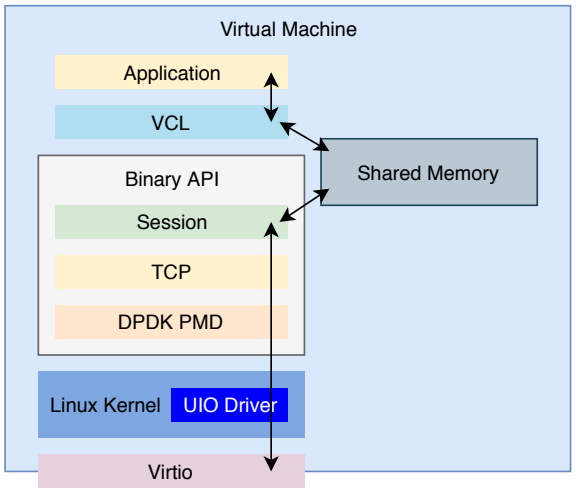


Figure 13: VCL. [40]

This experiment aims to give comparable results between direct VCL and Linux kernel. This test is conducted to provide more information alongside test 1, since LD_PRELOAD is the best way to link applications against VCL. The VPP socket_test tool is not accurate for drawing conclusions about the latency and jitter of the connection, but because of limited software support for VCL it still provides interesting

data about the possible performance of directly using VCL. This test is performed in both single host and multi-host environments. Figure 13 illustrates how packets reach application through the VCL API.

All the runs consists of 10 identical series with 10 million packets each. From these series, the mean is calculated when the results are processed. The payload size is set to 8KB. The experiment is conducted in an unidirectional mode, which means that the payloads are only transferred in one direction. socket_test has support for multiple simultaneous clients, but during the initial testing this feature was found broken on both kernel stack and VCL. In this test only one client is used.

5 Experimental results

This chapter presents the results of the experiments carried out to evaluate TCP/IP Acceleration. In section 5.1 there are three subsections which represent experiments introduced in section 4.3. In this section the results are presented and explained. Further analysis of implications of all the tests is in section 5.2. In this section technical results are also connected to other observations done during the research, such as the maturity of the frameworks.

5.1 Results

Both of the defined tests had their own peculiarities. In this chapter the results are presented in a manner that both takes the constrains into account and tries to highlight the interesting observations from the data. The first test contained 502 series of 100,000 operations run against the database server. Because of the high amount of data, only selected parts of it will be visualized in this thesis. On the other hand, in the second test the amount of sent packets was even higher, but due to the limitations of the tools, the data received is very limited.

Test 1: VPP, F-Stack and Linux kernel with Redis-benchmark

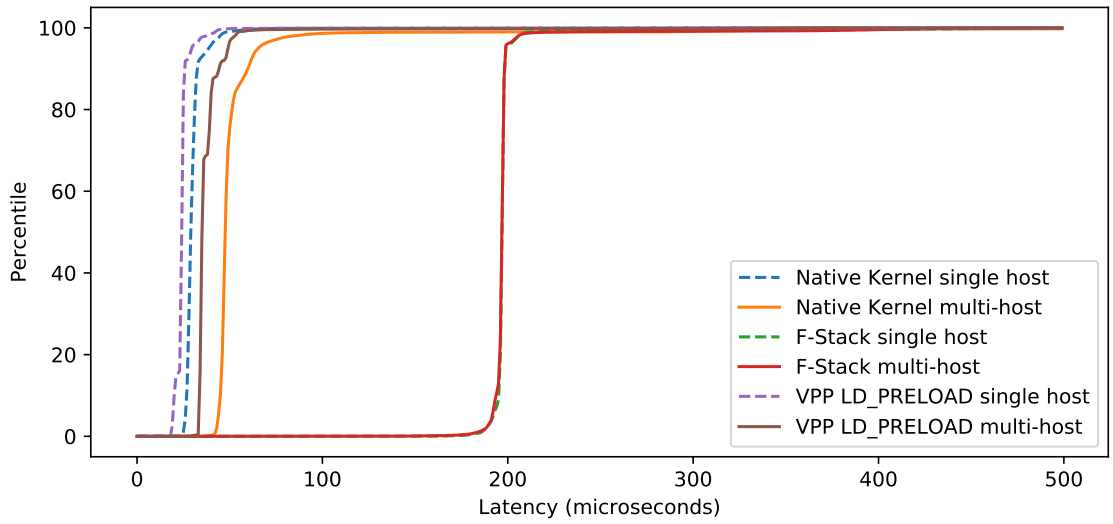


Figure 14: Latency distribution of a client sending 100,000 SET operations with a 1024 byte payload in six different environments.

Figure 14 represent the closest simulation of performance of the frameworks in a real-life scenario. In this particular test run the payload was set to 1024 bytes and packets were sent by one client. The 1024 byte packet size was selected as it is realistic in these kind of systems. Only one sending client is considered closest to real-life, because of how the redis-benchmark is implemented. It is sending packets in back-to-back mode, so the load is already considerably high with one client. The

operation used in the test was SET. All the tests were also run with GET operation, but because the results were very similar, only the results of SET operations are presented.

Test	Statistics			Percentile			
	Min	Max	σ	90%	95%	99%	99.9%
LD_PRELOAD single host	19	516	8.3	25	29	42	65
LD_PRELOAD multi-host	29	1553	15.0	44	49	54	302
Native Kernel single host	24	936	11.9	32	39	48	166
Native Kernel multi-host	36	1000	36.0	60	65	280	539
F-Stack single host	139	1042	21.4	198	199	212	533
F-Stack multi-host	128	1064	21.4	198	198	227	447

Table 3: Statistics of one client sending 10000 SET operations with a 1024 byte payload in six different environments.

In this particular test VPP with VCL used through LD_PRELOAD was performing the best with a short margin. In a single host environment native Linux kernel TCP stack was very close to same performance, but generally behind 5-10 μs as seen in Table 3. In multi-host environment the gap grew closer to 10-20 μs. In addition to slower transfer times the Linux kernel TCP also delivered higher standard deviation (σ), which shows also well in the 99.9$^{\text{th}}$ percentile results where Linux obtains over 100 μs worse transfer times in both categories.

The performance of F-Stack was surprising as it delivered a majority of the packets in slightly under 200 μs, which is considerably worse than below 99$^{\text{th}}$ percentile of the results achieved with other stacks (65 μs). Another peculiarity was the fact that F-Stack delivered the packets in a very similar speed both over the physical switch and from VM to VM in a single machine. Because of these experiments involving anomalies, F-Stack tests were re-run multiple times, but the results stayed within the margin of error, so the first ones were used.

When taking a look on the other runs it can be seen that the gap between LD_PRELOAD and native kernel is even narrower. Figure 15 illustrates the 95$^{\text{th}}$ percent times of each test run. Test runs are presented in the running order, which is running the payloads in an increasing order for each amount of clients in an increasing order. The amount of clients is presented with c in the figure and the size of the payload with b. 95$^{\text{th}}$ percentile latency provides a solid view on the latencies the majority of packets are experiencing.

According to this metric, LD_PRELOAD provides the best performance with a small margin until higher data rates are reached. However, with 4 sending clients and a 8192 byte payload, the Linux kernel TCP reaches a better latency in both multi-host and single host environments. In runs with more than 4 clients, the Linux kernel constantly reaches better results excluding the heaviest runs with 16 and 24 clients with a payload of 8192 bytes. When the data rate is high enough, VPP with LD_PRELOAD is failing. In the test runs this shows in the single host test with 24 clients and a 8192 bytes payload where LD_PRELOAD does not have

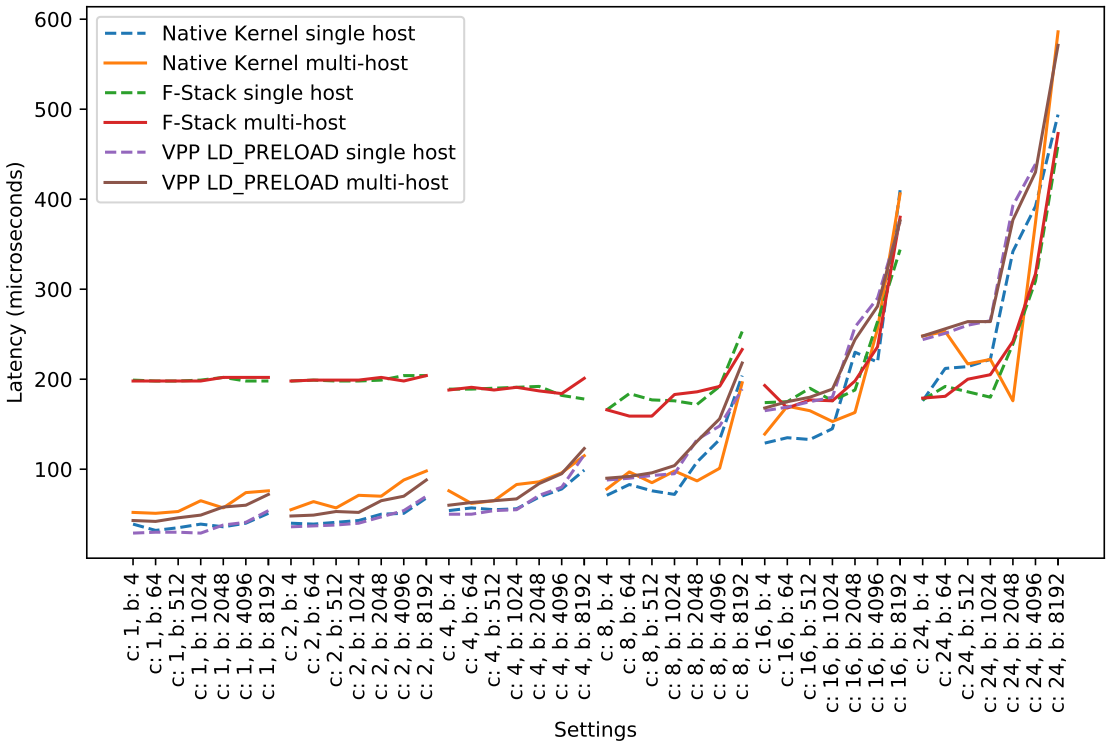


Figure 15: 95th percentile latencies of sending 100,000 SET operations with varying a payload size and number of clients in six different environments.

any results. The problem occurs a few seconds after the start of the run. The number of operations per second manages to reach a steady rate for a while, but then it drastically drops and starts to approach zero semi-asymptotically. Additional investigations show that the problem is indeed related to the data rate. For example if the payload is doubled and the number of clients is reduced in same proportion, the problem still occurs. It also appears on other operations. For example MSET can be used to trigger the problem with a lower payload and number of clients, because it carries the payload M times to set multiple values at once.

Interestingly, F-Stack gets closer to the other frameworks on 95^{th} percentile metric when the load gets higher. In scenarios with 24 clients it manages to achieve the best latencies for multiple lower payload runs in both environments. It also achieves its minimum 95^{th} percentile latency with 8 clients instead of the lowest load one client experiment.

Until this far mainly general latency is considered. 95^{th} percentile tells a lot about the general latency, but it does not describe jitter very well. Jitter can be quantified in many ways like the standard deviation, but in this study the best way is tail latencies like the 99^{th} percentile or the 99.9^{th} percentile. These tell exactly how poor latencies do the worst performing packets get. In this case these two have surprisingly different results. Figure 16 shows that VPP with LD_PRELOAD can perform up to the tail latencies. In the same figure multi-host performance cannot provide reliable, under 100 μs, latencies anymore. However, in a single-host mode

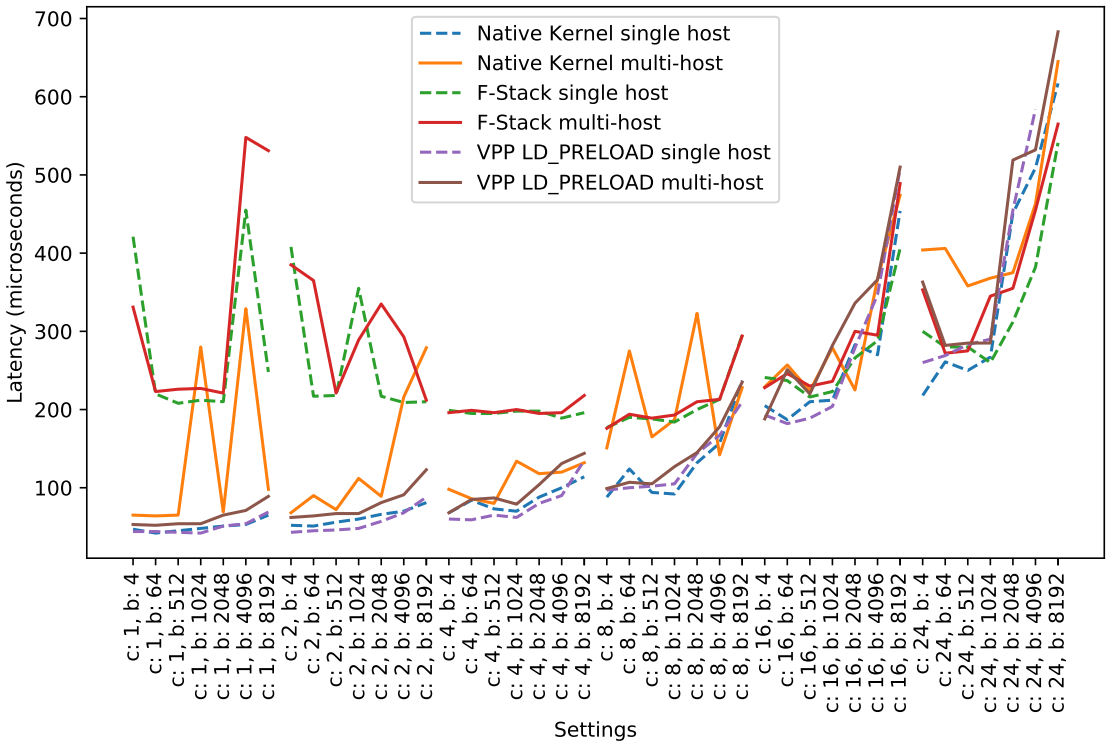


Figure 16: 99th percentile latencies of sending 100,000 SET operations with a varying payload size and number of clients in six different environments.

Linux kernel still holds up.

To call any framework reliable in this test, it should be able to achieve even reliable 99.9$^{\text{th}}$ percentile latencies. In the Figure 17 it shows that almost all of the measurements are far from reliable and almost any of the test series can throw 99.9$^{\text{th}}$ percentile latency easily to over 100 μs category. However, in a single host environment VPP used via VCL and LD_PRELOAD still hold up in tests where the payload and number of clients is moderate.

Analyzing the reasons for tail latencies further, a histogram timeline of the received packets reveals some potential reasons. In the timeline each individual packet is drawn as a dot to the figure, where time runs on x-axis and latency in y-axis. This can, for example, reveal if the latencies follow some kind of pattern. By drawing this kind of a figure for each of the 252 runs with SET operations, a few patterns were recognized.

Figure 18 presents timeline of multi-host native Linux kernel TCP with payload of 1024 bytes and one sending client. This is the same run which was used as part of Figure 14 in the beginning of this section. In this timeline there is a pattern which can be seen in the most of series run. There is a visible group of packets with latency of bit over 300 μs and again around 500-600 μs. This pattern is quite visible with all tested frameworks in both single host and multi-host environments. It does not show as well in the series with higher load, but it is present at least in all of the results with 4 clients or less. Since the same phenomenon is found

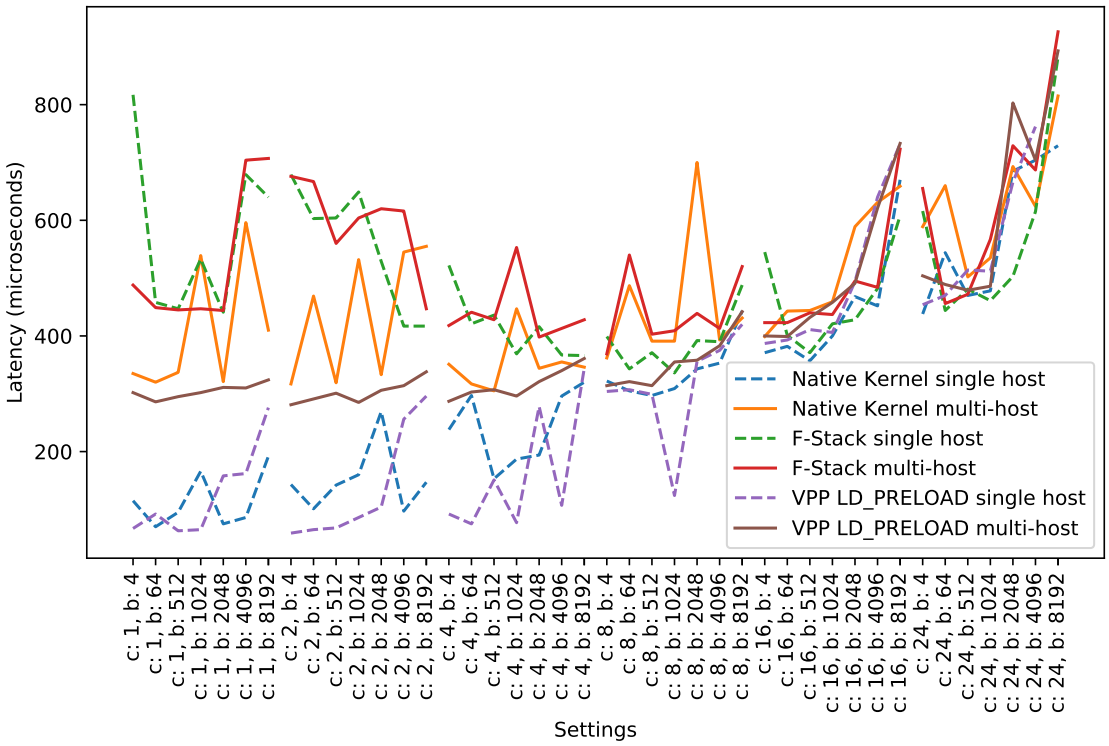


Figure 17: 99.9th percentile latencies of sending 10000 SET operations with a varying payload size and number of clients in six different environments.

from a wide variety of tests, the reason probably is in some common nominator. In this case most probably TCP is the root cause. The pattern, where most of the packets have similar extra delay of around 150 to 250 μs, would point towards TCP re-transmissions.

In addition to possible TCP retransmissions, there is also another interesting pattern found in the histograms. In Figure 19 there is a comparison of two runs done with F-Stack in a single host environment. In both tests the payload was 1024 bytes as in other examples. The only difference between the runs is the amount of used clients. For an unknown reason 4 and more clients split the majority of the packets into two categories. The most interesting detail is that the faster half of the packets in 4 client mode is significantly faster than F-Stack in one client mode. This hints that F-Stack is not performing up to its capabilities, and there is a problem with either the API to redis, the tested F-Stack version or the configuration. This was retested multiple times and the results were similar. This anomaly is not limited to only these parameters. The same results show also on different payloads and number of clients. However, the effect is best shown with four clients, as with eight clients there is four groups and with more clients groups are not as recognizable.

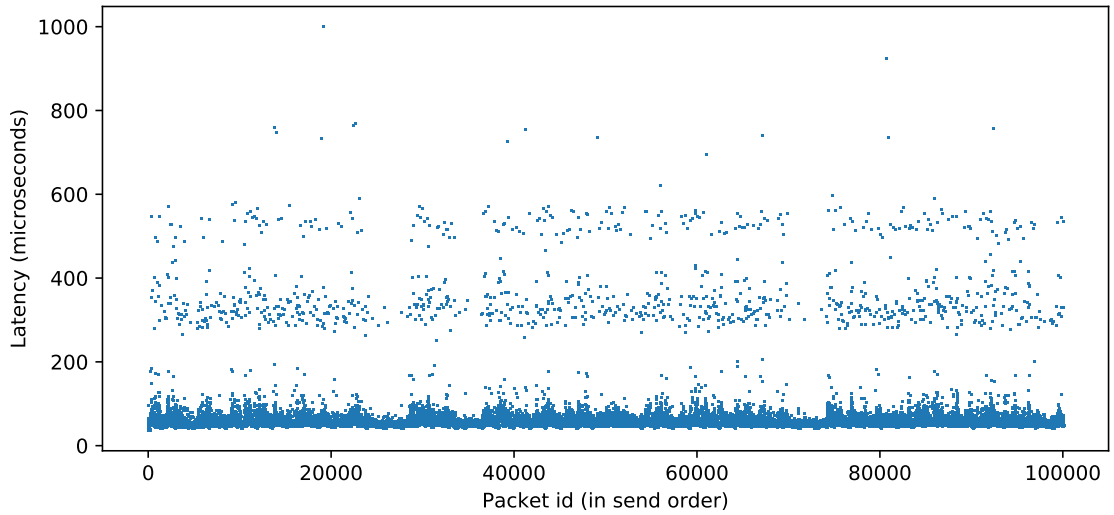
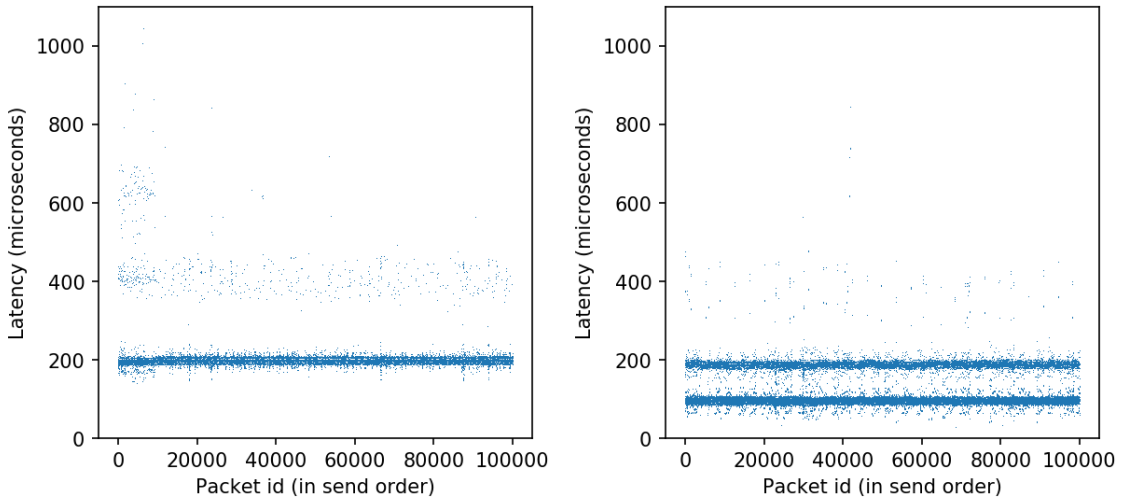


Figure 18: Per packet latency of every send packet sent in Linux kernel multi-host series with a payload of 1024 bytes and one sending client.



(a) (b)

Figure 19: Per packet latency with F-Stack in a single host environment for (a) 1 client and (b) 4 clients. Both runs have a payload of 1024 bytes.

Test 2: VPP and Linux kernel with VPP socket test application

In the VPP and Linux kernel TCP stack comparison there were 5 tests runs in the end. First both VCL and kernel TCP performance was measured in both single and multi-host environments. After this VCL was optimized further and run again on a single host environment. This test was not run on a multi-host environment, since the switch between the physical machines was already limiting the throughput of the first test. This optimization was done by pinning ten CPU cores for VPP. By pinning CPU cores VPP does not get interrupted by other applications. Amount of

cores is less critical, and eight of the ten cores reserved were idle.

In both single host and multi-host scenarios VPP through VCL API performs considerably better than the Linux kernel TCP stack. In a single host environment VCL provided 2.04 times better performance. With pinned CPUs the performance was 2.17 times better than the one provided by the Linux kernel TCP stack. In a multi-host environment VPP performed only 1.82 times better. The exact measurement data is presented in Table 4. The most likely reason for this difference in the performance is hitting the limits of 10 Gbps networking. The test measured the data rate from the transferred payload, so with overheads from packets the throughput is close to 10 Gbps.

	Kernel	VCL	VCL optimized
single host	4.859 Gbps	9.905 Gbps	10.555 Gbps
multi-host	4.781 Gbps	8.740 Gbps	-

Table 4: Average throughput through ten socket test runs in each environment.

Even though the socket test application demonstrates how VPP can achieve twice as good data rates, it is not possible to draw direct conclusions based on this test. Better data rate does not directly imply better latency and does not have any connection with jitter. Socket test also sends data in a pattern that is not realistic in a telco cloud. In the socket test all data is pushed through the link as fast as possible. VPP benefits a lot from this kind of scenario, because it can get full vectors which virtually means free packet processing for a majority of the packets. However, this data pattern would not be encountered in a real telco cloud and thus VPP's performance would be hindered.

5.2 Analysis

There is no single part in testing or background research which would point at specific implementation of single framework and prove that it solves the problem. However, the results seen in the section 5.1 and the background research done also leaves a possibility that these frameworks could be used to solve the problem.

F-Stack did not perform well on the test series which were considered to simulate real-life scenarios best. For example, with one client and a 1024 byte payload, F-Stack had over four times slower 99^{th} percentile than VPP on both environments. Also the native Linux kernel left it clearly behind in most of the statistics. However, F-Stack had very interesting anomalies and patterns which are described in detail in section 5.1. Part of these were division of the latencies in a manner where it actually was able to deliver half of its packets with considerably low latency when there were many clients. These kind of anomalies led us to believe that there might be something wrong with the used version of either F-Stack or the API used to connect to Redis. It is also possible that the light documentation led to misconfiguration which caused the problems. However, the tests were rerun multiple times and the results repeated on all tests.

F-Stack is originally developed by Tencent Cloud, which hints that it should be production mature code. On the other hand, as stated in the description of the project it is developed to a very different use case [6]. It is also possible that the Redis API is more of an unmaintained demo than an actual part of the product. However, in the light of the results in the literature and the experiments, VPP is showing more consistent results.

VPP did not deliver the desired results, but from the different pieces it is possible to draw a conclusion that VPP could be used to solve the problem with reasonable resources. Prior to any testing LD_PRELOAD was one of the top features of VPP. In comparison to any other framework providing TCP/IP acceleration VPP was the only one to promise to do it without extra work on developing the API for libraries or the application. However, after initial tests and communication with VPP development community, it is clear that the support for LD_PRELOAD was only an early demo for VCL library and is not supported or intended to work with just anything connected to it. In the initial tests VCL was very unstable and did only handle some special scenarios. However, it was able to handle the redis-benchmark client, which provided some results of the VPP in real action.

In the first test VPP proved that it can outperform the native Linux kernel TCP stack with a very narrow margin even with a very poorly optimized setup. In the second test VPP outperformed the native Linux kernel TCP stack clearly on a bandwidth oriented test. It is clear that a direct conclusion cannot be drawn from this test, but it seems that with proper API VCL can provide a solid performance. If the first test could be ran with VPP with a proper API in both ends, there could be better conclusions drawn and there is a high change that the results would speak for VPP.

VPP is developed by quite an active community. The application and its libraries are constantly changing and for example whole hoststack was receiving a major rewrite during the writing of the thesis, so the status of the project from the view angle of TCP/IP acceleration might change drastically to better or to worse in the near future. In some areas of the project there is also a lack of coordination and information, as the marketing for LD_PRELOAD functionality shows. It also seems that there are many features which are so new that they might not be in wide production usage. For example, a suggestion of the community is to use hoststack via VCL, but there is not a single software ported to it and publicly available. VPP is a large project which will definitely be around for years, but it seems to experience some lack of maturity and growing pains currently.

As the last point, the native Linux kernel TCP stack surprised by its performance in the testing. If the frameworks do not provide better results in future tests, it is very much possible that optimizing the native TCP stack could provide a solid start. However, this could also tell about a lacking testing setup. Even though the route of the packet was very similar to a possible real-life scenario, the setup does not represent actual production cloud with all of its extra layers of complexity and unexpected traffic patterns.

6 Conclusions

One of the premises for this study was the idea of using TCP/IP efficiently to reduce latency and jitter in call session data storage. It is safe to say that for multiple reasons the goal was not achieved and this study does not end in a simple proposal of a simple way to solve the problem completely. However, both of the major frameworks in the field provided some results which can be used as foundation for future studies.

The major finding was that the maturity of VPP was not as high as expected. The best example from this was the lack of compatible software for the VCL API and inconsistent functionality achieved with LD_PRELOAD. However, VPP is in a quickly developing state and changes made to host stack can quickly change it to be a more suitable solution. F-Stack seems to be in a more stable condition, but latency-oriented TCP/IP acceleration does not seem to be the core of the project.

User space TCP/IP acceleration frameworks domain are in the early stages. Even the major players have not been available for longer than a few years and there are many novel solutions started every year. It seems that the interface between the transport layer and the application is the common problem which no one has not been able to properly address. Mostly, this has been approached by porting each application individually, which has led to a few unmaintained frameworks only supporting old versions of popular applications. On the other hand, projects like VPP have not yet been successful in defining a more universal API.

Based on the literature review as well as the performed experiments there are a few recommendations to take into account for future research. First of all, VPP showed adequate results to justify putting the effort towards linking either Redis or another database solution directly into the VCL API. The VPP community should be monitored in the meantime, so the APIs could be made to work after major changes in the hoststack. If VPP keeps its promises, there are few aspects to consider while planning the deployment. As VPP has a wide set of functionality, it could also be used in the host as a vSwitch. In this deployment, a future study could investigate if VPP could benefit from its position on both the guest and the host by having only the VCL library on the guest and linking that to the VPP application running on the host.

If VPP does not provide a suitable solution, also work with F-Stack can be continued. With a very reasonable effort results provided in this study could be analyzed further, which could give a answer on how far from usable F-Stack is. F-Stack did show some strengths, and it can be just one bugfix away from providing the intended performance. As a different option TCP stack optimization in the Linux kernel could also be examined.

References

[1] ANS: TCP/IP stack for DPDK. https://github.com/ansyun/dpdk-ans. Accessed: 14.10.2018.

[2] DMM. https://wiki.fd.io/view/DMM. Accessed: 14.10.2018.

[3] DPDK. https://www.dpdk.org/. Accessed: 10.8.2018.

[4] DPDK docs: Programmer's guide. https://doc.dpdk.org/guides/prog_guide/index.html. Accessed: 1.9.2018.

[5] F-Stack api reference. https://github.com/F-Stack/f-stack/blob/master/doc/F-Stack_API_Reference.md. Accessed: 14.10.2018.

[6] F-Stack readme. https://github.com/F-Stack/f-stack. Accessed: 14.10.2018.

[7] FD.io gerrit: VPP. https://gerrit.fd.io/r/vpp. Accessed: 1.9.2018.

[8] Get ready for 5G. https://networks.nokia.com/5g/get-ready. Accessed: 3.9.2018.

[9] Getting started guide for Linux. https://doc.dpdk.org/guides/linux_gsg/index.html. Accessed: 2.9.2018.

[10] HPE support communication - customer advisory c04781229. https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c04781229&sp4ts.oid=5249566. Accessed: 27.8.2018.

[11] Hugepages. https://wiki.debian.org/Hugepages. Accessed: 2.9.2018.

[12] OFP: Technical overview. https://openfastpath.org/index.php/service/technicaloverview/. Accessed: 14.10.2018.

[13] Open vSwitch with DPDK. http://docs.openvswitch.org/en/latest/intro/install/dpdk/. Accessed: 28.8.2018.

[14] Qemu version 3.0.0 user documentation. https://qemu.weilnetz.de/doc/qemu-doc.html. Accessed: 29.8.2018.

[15] Redis. https://redis.io. Accessed: 9.8.2018.

[16] TLDK. https://wiki.fd.io/view/TLDK. Accessed: 14.10.2018.

[17] TLDK project proposal. https://wiki.fd.io/view/Project_Proposals/TLDK. Accessed: 14.10.2018.

[18] Vhost-user protocol spesification. https://git.qemu.org/?p=qemu.git;a=blob;f=docs/specs/vhost-user.txt;h=7890d7169;hb=HEAD. Accessed: 28.8.2018.

[19] Virtio. https://www.linux-kvm.org/page/Virtio. Accessed: 29.8.2018.

[20] VMware media resources. https://www.vmware.com/company/news/media-resources/press-kits.html. Accessed: 27.8.2018.

[21] What is the fast data project (fd.io)? https://fd.io/about/. Accessed: 6.9.2018.

[22] What is VPP? https://wiki.fd.io/view/VPP/What_is_VPP%3F. Accessed: 3.9.2018.

[23] Transmission Control Protocol. RFC 793, September 1981.

[24] Elisa first in world to launch commercial 5G. Jun 2018.

[25] Ghassan A. Abed, Mahamod Ismail, and Kasmiran Jumari. Exploration and evaluation of traditional TCP congestion control techniques. *Journal of King Saud University - Computer and Information Sciences*, 24(2):145 – 155, 2012.

[26] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. Experimental evaluation of NoSQL databases. *International Journal of Database Management Systems*, 6(3):1, 2014.

[27] Cisco affiliates. The zettabyte era: Trends and analysis. 06 2017.

[28] Jeffrey G Andrews, Stefano Buzzi, Wan Choi, Stephen V Hanly, Angel Lozano, Anthony CK Soong, and Jianzhong Charlie Zhang. What will 5G be? *IEEE Journal on selected areas in communications*, 32(6):1065–1082, 2014.

[29] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. 01 2009.

[30] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[31] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[32] Helali Bhuiyan, Mark McGinley, Tao Li, and Malathi Veeraraghavan. TCP implementation in Linux: A brief tutorial. *http://www. ece. virginia. edu/cheetah/documents/papers/TCPlinux. pdf*.

[33] Federico Boccardi, Robert W Heath, Angel Lozano, Thomas L Marzetta, and Petar Popovski. Five disruptive technology directions for 5G. *IEEE Communications Magazine*, 52(2):74–80, 2014.

[34] Kuan-Ta Chen, Chun-Ying Huang, Polly Huang, and Chin-Laung Lei. An empirical evaluation of TCP performance in online games. In *Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology*, page 5. ACM, 2006.

[35] Margaret Chiosi, Don Clarke, Peter Willis, Andy Reid, James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, Chunfeng Cui, Hui Deng, et al. Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action. In *SDN and OpenFlow World Congress*, volume 48. sn, 2012.

[36] Guo Chuanxiong and Zheng Shaoren. Analysis and evaluation of the TCP/IP protocol stack of LINUX. In *WCC 2000 - ICCT 2000. 2000 International Conference on Communication Technology Proceedings (Cat. No.00EX420)*, volume 1, pages 444–453 vol.1, Aug 2000.

[37] Glorin Coras. (vpp/hoststack). `https://wiki.fd.io/view/VPP/HostStack`. Accessed: 11.9.2018.

[38] Christian Dumitrescu. Introduction to DPDK. `https://www.youtube.com/watch?v=ewsrzoKwwz0`. FD.io /dev/boot.

[39] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan S Milojicic. Beyond processor-centric operating systems. In *HotOS*, 2015.

[40] Dave Barach Florin Coras. VPP host stack. `https://wiki.fd.io/images/1/15/Vpp-hoststack-kc-eu-18.pdf`. Accessed: 1.9.2018.

[41] Sally Floyd, Tom Henderson, and Andrei Gurtov. The NewReno modification to TCP's fast recovery algorithm. Technical report, 2004.

[42] Irfan Habib. Virtualization with kvm. *Linux Journal*, 2008(166):8, 2008.

[43] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal. NFV: state of the art, challenges, and implementation in next generation mobile networks (vepc). *IEEE Network*, 28(6):18–26, Nov 2014.

[44] Niklas Heuveldop. Ericsson mobility report november 2017. 2017.

[45] IEEE. Ieee 5G and beyond technology roadmap white paper. 2017.

[46] Intel. Open vSwitch enables SDN and NFV transformation. 2017.

[47] V. Jacobson. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 314–329, New York, NY, USA, 1988. ACM.

[48] V Jacobson, R Braden, and D Borman. RFC 1323: TCP extensions for high performance, may 1992. *Obsoletes RFC1072, RFC1185 [12, 13]. Status: PROPOSED STANDARD.*

[49] Fredrik Jejdling. Ericsson mobility report june 2018. 2018.

[50] Abdullah Talha Kabakus and Resul Kara. A performance evaluation of in-memory databases. *Journal of King Saud University - Computer and Information Sciences*, 29(4):520 – 525, 2017.

[51] Murad Kablan, Blake Caldwell, Richard Han, Hani Jamjoom, and Eric Keller. Stateless network functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '15, pages 49–54, New York, NY, USA, 2015. ACM.

[52] Kimberly Keeton. Memory-driven computing. In *FAST*, 2017.

[53] S. Larsen, P. Sarangam, and R. Huggahalli. Architectural breakdown of end-to-end latency in a TCP/IP network. In *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07)*, pages 195–202, Oct 2007.

[54] Erik G Larsson, Ove Edfors, Fredrik Tufvesson, and Thomas L Marzetta. Massive MIMO for next generation wireless systems. *IEEE communications magazine*, 52(2):186–195, 2014.

[55] DongJin Lee, Brian E Carpenter, and Nevil Brownlee. Media streaming observations: Trends in UDP to TCP ratio. *International Journal on Advances in Systems and Measurements*, 3(3-4), 2010.

[56] J. W. Lockwood and M. Monga. Implementing ultra low latency data center services with programmable logic. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 68–77, Aug 2015.

[57] John W Lockwood, Adwait Gupte, Nishit Mehta, Michaela Blott, Tom English, and Kees Vissers. A low-latency library in FPGA hardware for high-frequency trading (hft). In *2012 IEEE 20th annual symposium on high-performance interconnects*, pages 9–16. IEEE, 2012.

[58] S. Marston, Z. Li, S. Bandyopadhyay, and A. Ghalsasi. Cloud computing - the business perspective. In *2011 44th Hawaii International Conference on System Sciences*, pages 1–11, Jan 2011.

[59] Peter Mell, Tim Grance, et al. The NIST definition of cloud computing. 2011.

[60] Greg Minshall, Yasushi Saito, Jeffrey C. Mogul, and Ben Verghese. Application performance pitfalls and TCP's Nagle algorithm. *SIGMETRICS Perform. Eval. Rev.*, 27(4):36–44, March 2000.

[61] Jeffrey C Mogul and Greg Minshall. Rethinking the TCP Nagle algorithm. *ACM SIGCOMM Computer Communication Review*, 31(1):6–20, 2001.

[62] Nokia. The value of telco cloud. 2016.

[63] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vSwitch. In *NSDI*, volume 15, pages 117–130, 2015.

[64] Miguel Rio, Mathieu Goutelle, Tom Kelly, Richard Hughes-Jones, Jean-Philippe Martin-Flatin, and Yee-Ting Li. A map of the networking code in Linux kernel 2.4. 20. *Technical Report DataTAG-2004–1*, 2004.

[65] Allyn Romanow, Jeffrey C Mogul, Tom Talpey, and Stephen Bailey. Rfc 4297: Remote direct memory access (RDMA) over ip problem statement. Technical report, Technical report, IETF Network Working Group, 2005.

[66] Salvatore Sanfilippo. Redis-benchmark. `http://download.redis.io/redis-stable/src/redis-benchmark.c`, 2009–2012.

[67] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[68] D. Sidler, Z. István, and G. Alonso. Low-latency tcp/ip stack for data center applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Aug 2016.

[69] Samu Toimela. Containerization of telco cloud applications. 2017.

[70] Marcus K. Weldon. *The Future X Network: A Bell Labs Perspective*. CRC Press, Inc., Boca Raton, FL, USA, 2015.

[71] Wenji Wu and Matt Crawford. Potential performance bottleneck in Linux TCP. *International Journal of Communication Systems*, 20(11):1263–1283, 2007.

[72] Xu Zhiqun, Chen Duan, Hu Zhiyuan, and Sun Qunying. Emerging of telco cloud. *China Communications*, 10(6):79–85, 2013.