h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Hochschule Darmstadt

- Fachbereich Informatik -

High performance packet processing in the Linux kernel

Abschlussarbeit zur Erlangung des akademischen Grades
Master of Science (M.Sc.)

vorgelegt von Marius Gerling

Referent:	Prof. Dr. Lars-Olof Burchard
Korreferent:	Prof. Dr. Ronald Moore

Ausgabedatum:	12.04.2018
Abgabedatum:	12.10.2018

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 12. Oktober 2018

Marius Gerling

Abstract

High performance packet processing in the Linux kernel

Cloud providers are trying to optimize the energy efficiency of the computing resources which are needed to provide the services to their customer. With regards to new Platform as a Service (PaaS) and Functions as a Service (FaaS) solutions like serverless platforms the provider has more influence on the machine usage and resource optimization than for providing Infrastructure as a Service (IaaS) to their customer. This work will show how the overhead of the Linux network stack can be reduced and therefore more compute resources can be utilized to provide execution time to the customer.

Linux has a general-purpose network stack integrated into the kernel. It is capable of most use cases but falls shortly when high performance networking is needed. For theses specific use cases there are third party user space technologies available which provide higher performance and flexibility in the network path. With technologies like Data Plane Development Kit (DPDK)[1] and pf_ring[2] the data doesn't traverse the kernel space. However, there are also downsides when using these technologies like the need of special drivers and the need to disable all the energy efficiency settings on the machine.

eXpress Data Path (XDP)[3] and the underlying technology Enhanced Berkeley Packet Filter (eBPF)[4] are new technologies available in the kernel space to improve the performance and minimize the overhead of the networking stack. Today these technologies are mostly used for blocking unwanted traffic or providing tracing in the kernel.

With AF_XDP, which was released with Linux kernel 4.18, the data path for network traffic can be optimized. Packets received or transmitted via AF_XDP omit kernel traverses in the data path. This technology is used in this work to provide efficient network connectivity and provide services like serverless function execution at line speeds of 10 Gbit/s.

[1] `http://www.dpdk.org/` (downloaded on 2018-10-11)
[2] `https://www.ntop.org/products/packet-capture/pf_ring/` (downloaded on 2018-10-11)
[3] `https://www.iovisor.org/technology/xdp` (downloaded on 2018-10-11)
[4] `https://www.iovisor.org/technology/ebpf` (downloaded on 2018-10-11)

Abstract

Hochperformante Netzerkpaketverarbeitung im Linux Kernel

Das Internet und die Nutzung von Internetdiensten ist fester Bestandteil des Alltags vieler Menschen. Viele Internetdienstleistungen werden aus Rechenzentren von Cloud Dienstleistern bereitgestellt. Um profitabel und effizient zu arbeiten muss der Cloud Dienstleister die Last der Anfragen auf möglichst wenig Server verteilen ohne dabei Einbußen in der Performanz der angebotenen Dienste hervorzurufen. In dieser Arbeit wird AF_XDP, eine neue Technologie im Linux Kernel, eingesetzt um die Paketverarbeitung von Systemen zu optimieren. Durch die Optimierung werden weniger Ressourcen eines Servers für die Paketverarbeitung verwendet und Dienstleister können ihren Serverbedarf reduzieren.

AF_XDP setzt bei der Verarbeitung von Netzwerkpaketen darauf, den umfangreichen Netzwerkstack des Linux Kernels zu umgehen und die Pakete direkt in den Speicherbereich der empfangenden Anwendung weiterzureichen. Ähnliche Netzwerkarchitekturen sind bislang mit alternativen Netzwerkstacks wie Data Plane Development Kit (DPDK)[5] oder pf_ring[6] möglich. Durch die Verfügbarkeit von AF_XDP kann die Netzwerkarchitektur zukünftig auch ohne den Einsatz solcher Netzwerkstacks verwendet werden.

Der Einsatz von Enhanced Berkeley Packet Filter (eBPF)[7] und darauf aufbauenden Technologien wie eXpress Data Path (XDP)[8] ermöglicht es Pakete bereits vor dem Eintreffen im Linux Kernel zu verarbeiten und das Paket direkt an die richtige Applikation weiterzuleiten. AF_XDP stellt ein Framework für dieses Szenario bereit, welches von der Applikation eingesetzt werden kann um die Netzwerkperformanz zu erhöhen.

Diese Arbeit nutzt AF_XDP und evaluiert, wie groß die Performanzverbesserungen für einen Anwendungsfall ausfallen. AF_XDP ermöglicht es, die Datenverarbeitung auf die Netzwerkgeschwindigkeit von 10 Gbit/s zu beschleunigen.

[5] `http://www.dpdk.org/` (downloaded on 2018-10-11)
[6] `https://www.ntop.org/products/packet-capture/pf_ring/` (downloaded on 2018-10-11)
[7] `https://www.iovisor.org/technology/ebpf` (downloaded on 2018-10-11)
[8] `https://www.iovisor.org/technology/xdp` (downloaded on 2018-10-11)

Foreword

This thesis was written as part of the dual master's degree in computer science in cooperation with *UNIBERG GmbH*. It is also the final thesis of the degree course.

I want to thank Mr. Andreas Möller and Mr. Moritz von Keiser of *UNIBERG GmbH* for supporting me in the decision of doing a master's degree and giving me the possibility to write this master thesis.

I also want to thank Prof. Dr. Lars-Olof Burchard of *Hochschule Darmstadt* for supporting my master thesis and being the examiner of this work. Last but not least I want to thank Prof. Dr. Ronald Moore for taking the co-examiner role of this thesis.

Contents

List of Figures

List of Tables

Listings

1. Introduction

1.1. Motivation

Energy efficiency is a key parameter for optimization of computing systems. With virtual machines, Infrastructure as a Service (IaaS) and cloud computing, there are already optimizations in resource allocation and energy efficient computing. With Platform as a Service (PaaS), Software as a Service (SaaS) and Functions as a Service (FaaS) solutions cloud providers have even more influence on resource allocation and optimization of energy efficiency.

The goal of the cloud provider should be to provide all the services and compute resources the clients require and only use as many physical compute nodes as needed. The other nodes should stay off for energy efficiency.

This work tries to reduce the overhead of packet processing for workloads which require network access. Today more people are on the internet and consume internet services than ever before. Network speeds of 10 Gbit/s and more are available for server systems. The computing power might be a bottleneck in scenarios with high network speed available. Reducing the overhead of packet processing in this scenario leads to more packets and thus more workload processed by a single server. AF_XDP is a new feature available in the Linux kernel which can be used to provide better network performance. It will be evaluated if a network intense workload performs better on a system that employs AF_XDP. If the workload performs better, less server systems have to be used for the same workload and less energy is consumed.

1.2. Problem and goal

IT and cooling equipment are the two main drivers for energy consumption in data centers. To reduce the costs related to cooling and energy consumption, cloud providers are building new data centers in regions with low temperatures and low energy prices.[9]

[9] `http://www.datacenterdynamics.com/content-tracks/power-cooling/googles-finland-data-center-pioneers-new-seawater-cooling/32932.fullarticle` (downloaded on 2018-10-11)

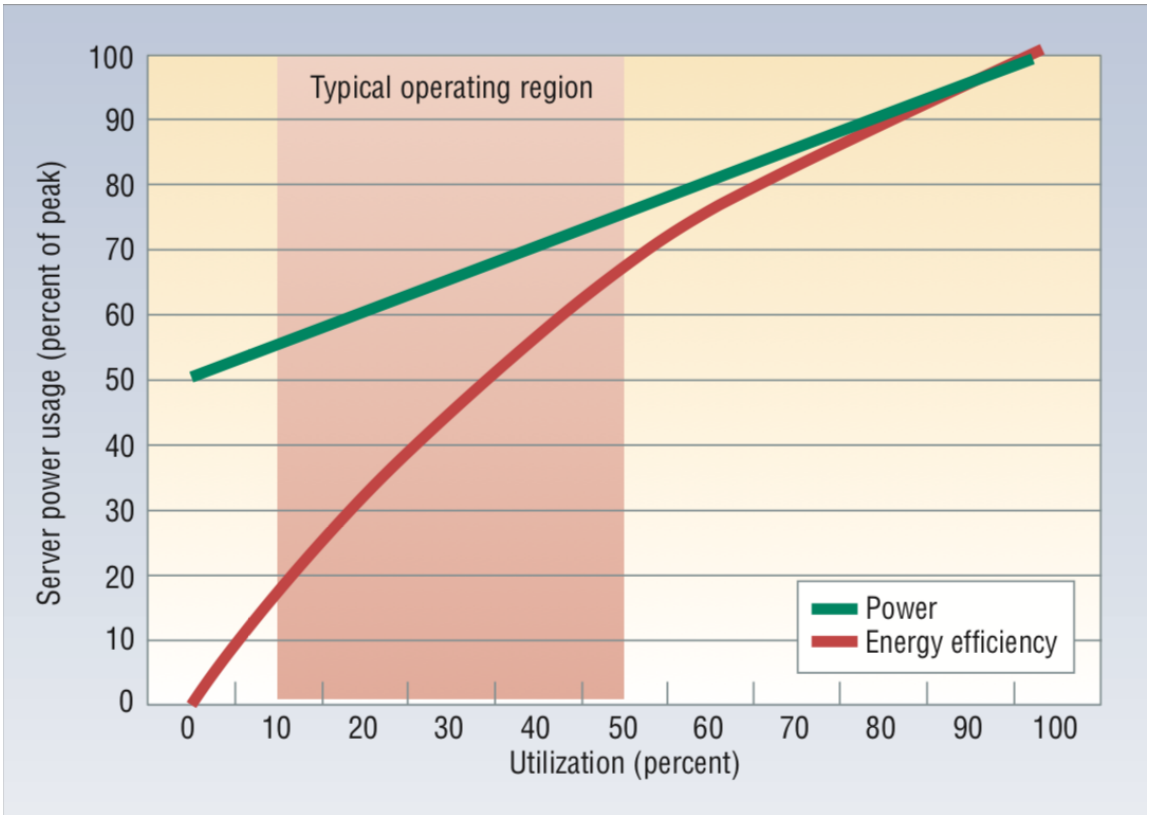

Figure 1: Server power usage and energy efficiency at varying utilization levels, from idle to peak performance. [BH07, p. 35]

Further reductions of energy consumption of IT equipment can be achieved by optimized allocation of the workload. With Virtual Machine (VM), IaaS and cloud computing the physical equipment can be used by multiple customers and multiple workloads. However, the usage of CPU-cycles may still be low if all the VMs are mostly idling. As [BH07] stated, most systems are either idling with lower than 10 percent CPU utilization or work between 20 and 50 percent of CPU utilization. Figure 1 shows that server systems are most energy efficient with utilizations of 80% or more. Cloud providers can use resource allocation algorithms and relocate running VMs on other hosts to improve the utilization.

With PaaS and SaaS there is less overhead in resource allocation than for VMs. The cloud provider has more configuration options to tune and provide the best customer experience regarding price and availability. To remain profitable, the service needs to be scaled according to the usage. Each system which can be turned off will reduce the power bill and therefore increase profitability if the service remains available.

The systems are not only executing customer workloads on the nodes but also need to run an operating system, monitoring and other applications to operate properly. If the CPU utilization is between 80% and 100%, the compute nodes are working energy efficient. Further optimizations can be achieved if the platform is optimized and less CPU cycles are used per request.

This work focuses on the optimization of the network stack in such systems. The Linux network stack is a general-purpose network stack which is sufficient for most use cases. However, since the network stack is built for general-purpose computing, a lot of CPU cycles are wasted for features not used in this specific use case of providing PaaS, SaaS or FaaS.

The goal is to minimize the CPU consumption needed for the network related part of the workload and increase the request throughput. If every single server can handle more requests, less servers are needed to perform the workload. Thus, the energy efficiency increases. AF_XDP is a new feature in the Linux kernel. It addresses the need of high performance packet processing by omitting the Linux kernel out of the data path. This work employs AF_XDP into a FaaS environment and evaluates how AF_XDP can be used to accelerate packet processing.

1.3. Environment

This work is used in a well-defined environment and under certain conditions. The setup and the environment are defined in this subsection.

1.3.1. Use case

The use case is defined as a Functions as a Service environment. A FaaS environment is used to provide an execution environment for functions defined by customers. The customers don't have access to any low-level information of the system and cannot influence the placing of the functions. Thus, the provider of such an environment has the full control and a change of the network stack can be performed without customers involved. The functions defined by a customer is invoked by the serverless platform for each request, which is a single packet in this work. Further information about a serverless platform are available in section 2.4 on page 16.

This work optimizes the single node performance, communication across multiple data centers is not relevant. It is assumed that the packet loss in the data center network is very low and the links between the systems are not overprovisioned and can communicate with full 10 Gbit/s. The network condition is possible if the network infrastructure is scaled without overprovisioning or the server and client are connected to the same switch with enough backplane bandwidth available. [AFLV08, p. 64]

With this network setup the use of TCP would mean more overhead and many small packets on the line. Also it would only be of little use. Therefore, this work relies on UDP only. Typical packets in the network are between 200 and 1500 bytes of size. The prototype will be designed to work with packets sized between 200 and 1500 bytes.[BAM10, p. 272f]

1.3.2. Hardware setup

The hardware available for this work is limited to two test systems and a management system. One test system is used to execute the workload of the simulated FaaS platform, the other is used to generate the requests against the platform. Both test systems are connected via a separate 10 Gbit/s network link and use this link exclusively for the payload of the tests.

All three devices are connected to a management network. The management host is used to monitor the test systems and has Prometheus[10] and Grafana[11] installed to collect metrics and show those metrics in graphs, which are also used in this work.

The metrics consist of the system monitoring of the test systems with Prometheus node_exporter[12]. This software exports CPU-load, memory consumption, network and disk I/O and more data points, which are collected and saved by Prometheus. There are also metrics available from the load generator application. It counts the number of requests, responses and the time between request and response. Also failed requests and responses are counted for the evaluation.

The hardware of both the test systems is the same and defined in Table 1, the

[10] `https://prometheus.io` (downloaded on 2018-10-11)
[11] `https://grafana.com` (downloaded on 2018-10-11)
[12] `https://github.com/prometheus/node_exporter` (downloaded on 2018-10-11)

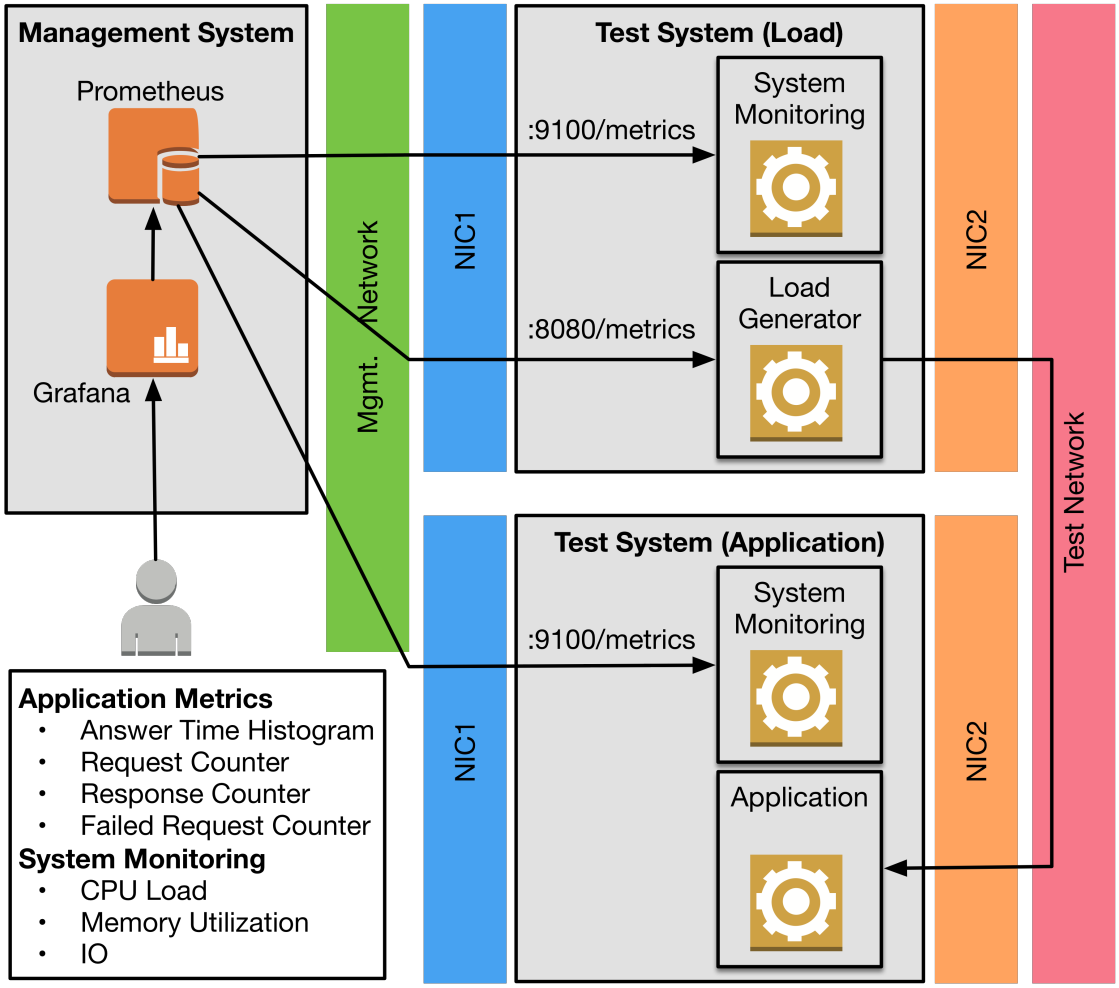

Figure 2: Test environment

operating system is Fedora[13] 28 with custom built Linux kernel version 4.18.6 with enabled AF_XDP support.

Installation instructions and configuration files are available together with the source code of this work in a git repository.[14]

[13] `https://getfedora.org/` (downloaded on 2018-10-11)
[14] `https://github.com/gerl1ng/afxdp-packet-processor` (downloaded on 2018-10-11)

Device	HP Z420
CPU	Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz 6 Core, 12 Threads
Memory	32GB DDR3 ECC
Network Card	Mellanox ConnectX-2 10GigE (MT26448)
Hard Drive	TRANSCEND SSD 2.5" 240GB

Table 1: Hardware of the test systems

1.4. Structure of the work

In the next chapter the prerequisite technologies, which are used in this work, are described. Related work in regards of performance improved networking are shown in section 3.

In chapter 4 the performance of the Linux kernel network stack is measured for the use case. Afterwards a new application concept using AF_XDP is introduced in chapter 5 and implemented in chapter 6. Chapter 7 contains the performance evaluation of implementations. The work ends with a conclusion in chapter 8 and a gives an overview of future work topics regarding the technologies in this work.

2. Technologies

In this section the technologies used for this work are introduced and described.

2.1. Data center networks

Data center networks are the networks which are used in the data center to connect the computing systems with other computing systems and connect the outer world to the computing systems. Usually, a large data center hast the network designed in a three tiered topology as shown in figure 3.

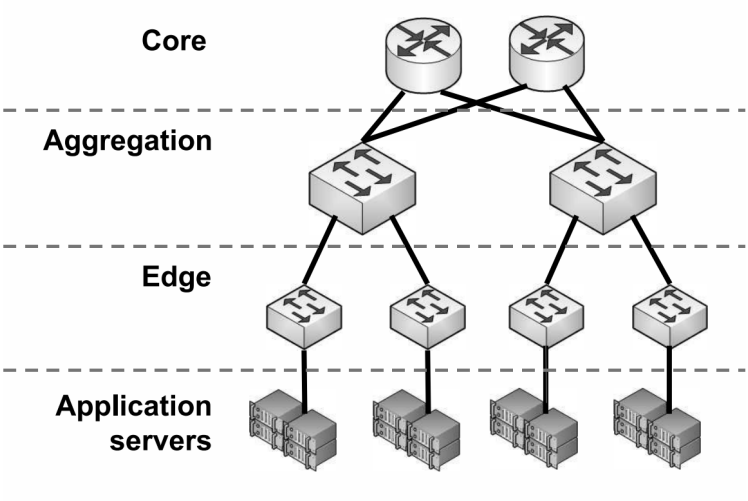

Figure 3: Three tiered data center network topology. [BAM10, p. 270]

The edge tier is connecting the computing systems to a first switch. These switches are usually 1 Gbit/s or 10 Gbit/s with 10 Gbit/s or 40Gbit/s uplink ports as of 2018. The aggregation tier is used to aggregate multiple edge switches to the core network and is full 10 or 40 Gbit/s for the connection to the edge switches and might have uplink speeds from 40 to 100 Gbit/s to the core network. With a setup like this, most of the large data centers are able to provide network access to many computing systems with redundancy and multi-path routing. To lower the cost of such a network setup, the connections are usually overprovisioned. If all the computing systems are using the full link, the connections to the aggregation and core network are not fast enough to handle all the data and packets might get lost.

[BAAZ09] shows that links with packet loss ranges from 1.6% at the edge over 2.78 % at the aggregation to 3.78% at the core network. [AFLV08]

For this work is is assumed that the requests to the performance optimized server system are able to use the full link throughput of 10 Gbit/s without any packet loss. This may be achieved with intelligent system placing to the same edge switches or without overprovisioned network links in the data center.

[BAM10, p. 272f] shows that the packet size distribution in data center networks is usually between 200 and 1500 bytes and averages around 850 bytes. The test application of this work uses this information and is able to generate packets between 200 and 1500 bytes.

2.2. Linux TCP/IP stack

Linux and Unix powers the web with more than 68 % of the internet facing web servers being Linux or Unix based.[15] This is especially true for the use case of serverless computing in which software like Apache OpenWhisk[16] is used and only available for Linux.[17] This work focuses on performance optimizations on the networking part of a workload. Linux abstracts the network connectivity from the application with a socket interface and system calls like recv and send to the file descriptor of the socket. This section shows how the standard Linux TCP/IP stack processes packets from the Network Interface Card (NIC) to the application or vice versa.

The monolithic Linux kernel provides network connectivity with drivers for most network cards out of the box. It also provides the ability to use either Transmission Control Protocol (TCP)[18] or User Datagram Protocol (UDP)[19] over Internet Protocol Version 4 (IPv4)[20] or Internet Protocol Version 6 (IPv6)[21].

[15] `https://w3techs.com/technologies/overview/operating_system/all` (downloaded on 2018-10-11)
[16] `http://openwhisk.apache.org` (downloaded on 2018-10-11)
[17] The software is available for Mac and Windows, but is provided in a Docker image, which itself is a Linux machine.
[18] `https://tools.ietf.org/html/rfc793` (downloaded on 2018-10-11)
[19] `https://tools.ietf.org/html/rfc768` (downloaded on 2018-10-11)
[20] `https://tools.ietf.org/html/rfc791` (downloaded on 2018-10-11)
[21] `https://tools.ietf.org/html/rfc8200` (downloaded on 2018-10-11)

If an application wants to receive data over the network, it creates a socket and binds it to an IP-address (IPv4 or IPv6), port (0-65535) and protocol (TCP or UDP) combination. After a successful bind the application can try to read data from the socket. The application provides a chunk of memory to the recv system call, which is used as a buffer. If data is available it is copied into this buffer. The application can reply to the sender through the socket with the send system call. The data provided to this system call will be sent to the address, port and protocol combination, which the application also provided to the system call. On the way to the NIC the packet is encapsulated with a TCP or UDP header and in case of TCP, the packet is adapted to the stream and maybe assembled together with other packets on the same stream. It may also be split into multiple packets based on the TCP window size.

On the next step the TCP or UDP packet is encapsulated with an IP header. If a packet is to large to be transported over the line due to the Maximum Transmission Unit (MTU) size, it will be split into multiple packets in this stage. Now there can be additional more layers like VLAN[22] or Virtual Private Network (VPN) between the IP layer and the Ethernet layer. The kernel encapsulates the packet as much as needed. Finally, the packet is encabsulated again to provide Ethernet header and copied to a Transmitter (TX)-ringbuffer, which is used by the NIC and send onto the physical connection. All those headers (UDP, TCP, IP, VLAN, Ethernet) are defined in network byte order, which happens to be big-endian. The TCP/IP stack converts the host endianness (little-endian for x86 architecture) to the network byte order if necessary.

On the receiving side the data is read from the Receiver (RX)-ring after an interrupt by the NIC informs the kernel that data arrived. The layers are removed one after the other. For the TCP-layer the stream is reassembled and separated into single packets. If the TCP data arrives out of order or is late even a retransmit is handled in this layer. In the end the data can be received by the application and is copied into the provided buffer. The fact that the kernel is used in the I/O Path means, that we have context switches while packets are sent or received and the data is copied multiple times on the way. While sending and receiving network data over a Linux socket, the structore sk_buff[23] is used. Currently this structure is defined by

[22] `https://standards.ieee.org/getieee802/download/802-1Q-2014.pdf` (downloaded on 2018-10-11)

[23] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/skbuff.h#n665` (downloaded on 2018-10-11)

194 lines of code and represents the complexity these actions require.

Additionally the Linux kernel provides further features to manage the network I/O. Counters are increased, tracepoints and dumping infrastructure is also included, connections are tracked and firewall rules are applied to the data path. [Ker10, p. 1165ff][Chi05]

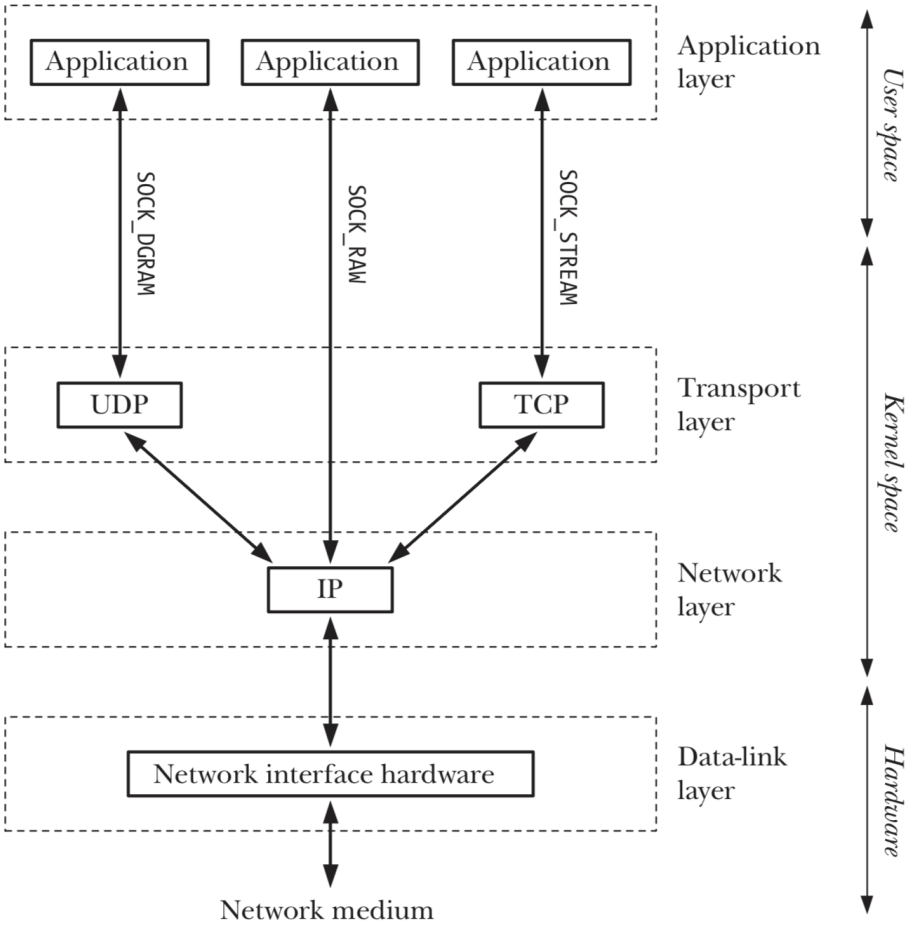

Figure 4: TCP/IP stack in the Linux kernel (minimalistic view) [Ker10, 1181]

The TCP/IP stack is under active development and receives updates and new features every release. One example of enhancements is the Receive Packet Steering (RPS)[24] functionality, which allows packet processing in the Linux kernel with multiple CPU cores. This improves the performance and allows higher CPU utilization. This technology can also be used by single queue NICs since it is integrated

[24] `https://lwn.net/Articles/370153/` (downloaded on 2018-10-11)

in the kernel. With Receive Flow Steering (RFS)[25] this functionality has been improved by pinning a network flow to a CPU and reduce cache misses.

This subsection shows that there are a lot of features integrated into the Linux TCP/IP Stack. It is a powerful tool feasible for most workloads running on Linux servers. For specific use cases this TCP/IP stack is not optimized because it offers to many features which are not used and may consume unnecessary CPU cycles. These CPU cycles could be used by the application to provide service to the customers.

2.3. BPF and eBPF

This work strongly relies on a feature of the Linux kernel called Enhanced Berkeley Packet Filter (eBPF). eBPF is the extended successor of Berkeley Packet Filter (BPF). BPF was first introduced in 1993 by Steven McCanne and Van Jacobson. It was the approach of a high performance standardized packet filter in BSD and UNIX kernels. This technology was also build into the Linux kernel and is widely used. Programs like tcpdump[26] use it extensively. The concept of BPF is that there is a programmable pseudo-machine in the kernel space which can be accessed and programmed by other programs. Because of a limited instruction set and optimized mapping to the CPU architecture, BPF outperformed other filter mechanisms.[MJ93]

eBPF extends the now called classical Berkeley Packet Filter (cBPF) functionality with shared data structures. It extends the instruction set of the in-kernel pseudo-machine and introduced new system calls for accessing the pseudo-machine or the shared data structures. eBPF was introduced with Linux kernel version 3.18 in December 2014[27]. The eBPF pseudo-machine is only available in combination with a Linux kernel and not interoperable with other UNIX Systems at the time of this work. Together with eBPF there is a whole collection of management tools like bpftool[28] and integrations like iproute2[29].[Ker18][Cil18]

[25] `https://lwn.net/Articles/381955/` (downloaded on 2018-10-11)
[26] `http://www.tcpdump.org/` (downloaded on 2018-10-11)
[27] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tag/?h=v3.18` (downloaded on 2018-10-11)
[28] Available in the kernel source code repository at tools/bpf/bpftool
[29] `https://wiki.linuxfoundation.org/networking/iproute2` (downloaded on 2018-10-11)

The basic concept of eBPF is that there is user generated bytecode for the pseudo-machine invoked on specific events. One event is for example the ingress of a packet on a NIC. This event is configured on the file descriptor of the socket and is also used in this work. The bytecode is executed for each incoming packet. Other events are defined in the kernel and are for example used for tracing purpose. eBPF programs are usually written in C and compiled to eBPF bytecode by the LLVM Compiler Infrastructure[30], which is the official compiler. Before the bytecode can be attached to an event, it has to be loaded via a bpf() system call to the Linux kernel. An in-kernel verifier checks if the bytecode terminates in a reasonable number of instructions for every possible combination of parameters and a Just-in-Time (JIT)-Compiler translates the bytecode to the host architecture. This is necessary to provide a stable computing performance while using eBPF programs. In general an eBPF program has to terminate within 4096 BPF instructions. However, this limit can be extended by using BPF tail calls. These call another BPF program which itself can use up to 4096 instructions. There is also a limit of 32 tail calls for the whole program structure. If the bytecode is valid and running on the host, shared maps can be accessed by other programs via file descriptors. Figure 5 shows the workflow of the activation of an eBPF program and the relationship of it to other components. [Ker18][Cil18]

Since eBPF uses a limited instruction set and is usually mapped to a subset of the host architecture instruction set, the execution of eBPF programs is fast. There are also SmartNICs, NICs with the ability to execute programs on them, which support eBPF. If the host has such a SmartNIC installed, the JIT-Compiler will compile it to the SmartNIC instruction set and offload the program to the SmartNIC. This allows further reduction of CPU consumption on the host system and execution of the eBPF program at line-speed. [Mon18]

2.3.1. eXpress Data Path (XDP)

XDP is a low-level packet processor using eBPF as the underlying technology. It provides the ability to react on network events directly out of the device driver. Therefore, it works without the allocation of sk_buffs in the Linux kernel. Thus, XDP is able to perform tasks on raw packet information with high performance. It is used to filter or forward packets. XDP also introduces new invocation points for

[30] `https://llvm.org` (downloaded on 2018-10-11)

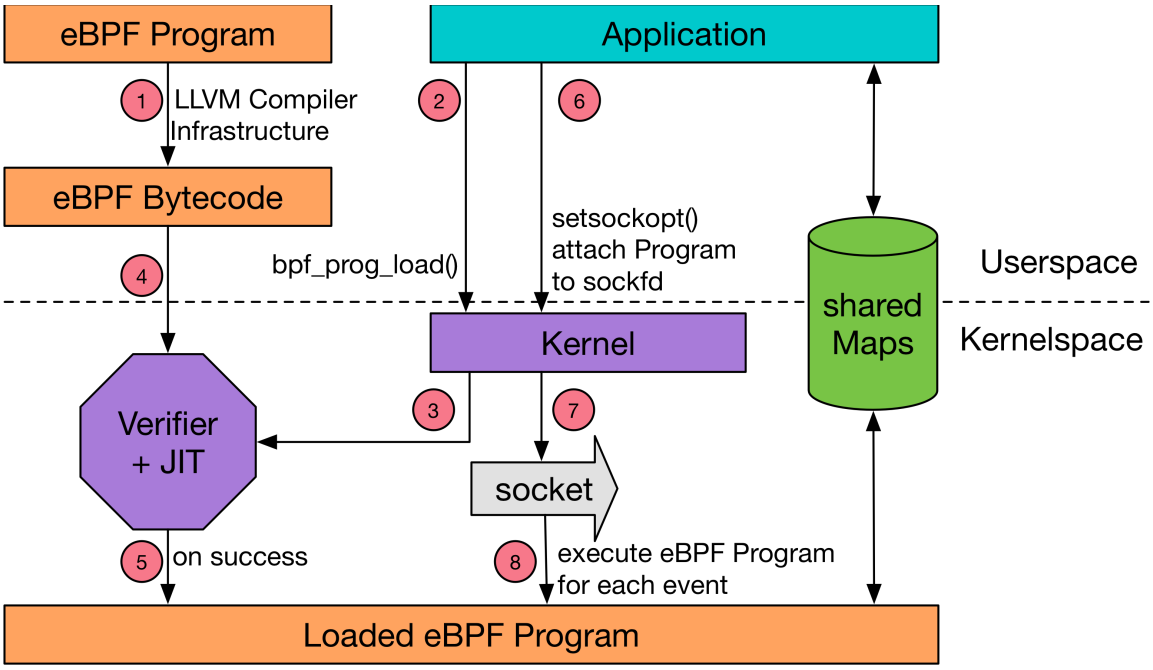

Figure 5: Workflow and relationship of eBPF programs while loading

eBPF programs if the driver of the NIC supports those. These invocation points are earlier on the network receive processing than the otherwise used earliest kernel invocation point. XDP is available in the Linux kernel since version 4.8[31] from October 2016.

Figure 6 shows a XDP packet processor, which is attached to the driver and acts on incoming packets. The packet processor might either drop the packet, which results in a page recycle by the driver, forward it to the TCP/IP stack of the Linux kernel or change the packet locally and forward it via the driver. [IOV16][HS16]

2.3.2. AF_XDP

AF_XDP is the technology which is used by this work to accelerate the packet processing. It is new to the Linux kernel and was released with version 4.18[32] in August 2018. AF_XDP is a new address family in the Linux kernel which optimizes

[31] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tag/?h=v4.8` (downloaded on 2018-10-11)

[32] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tag/?h=v4.18` (downloaded on 2018-10-11)

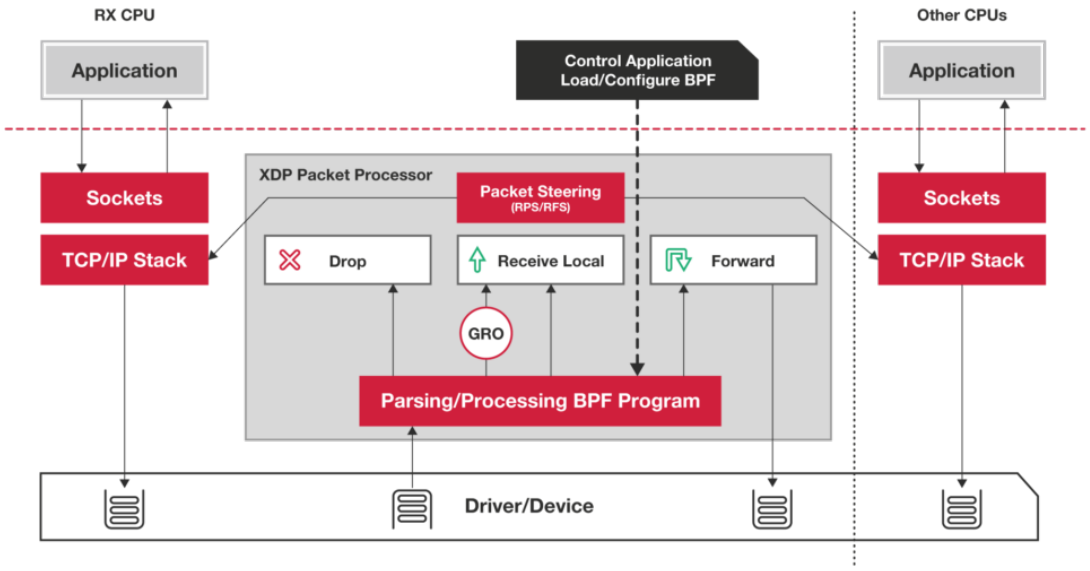

Figure 6: XDP packet processor[IOV16]

the data path from the NIC to the application by bypassing the TCP/IP stack and providing the data to the application in a zero-copy manner. This means that the application doesn't get a copy of the data by the kernel. The application accesses the data directly.

Programs that use AF_XDP allocates a chunk of continuous memory which will be used by the AF_XDP socket and accessible by the eBPF program. This memory is called UMEM and is used to store the plain packet data. Additional to the UMEM an AF_XDP socket consists of multiple single producer and single consumer rings, which are used to transfer descriptors to UMEM frames from the socket to the application and vice versa. An AF_XDP socket may be used for ingress, egress or both directions of data flow.

The Fill ring is used to transfer UMEM frames from the user space to the kernel-space. Frames that are transferred via the Fill ring are used by the RX ring for incoming packets. The RX ring is used by the application to read incoming data from the socket. The application may also send frames via the TX ring. This ring is consumed by the XDP part and sends the frame. After the frame is sent, the descriptor is transfered to the application again by the Completion ring. A single UMEM can be used by multiple sockets. It will still use just one Fill and one

Completion ring in this case, but for every socket there will be a RX ring and a TX ring. Figure 7 shows the different descriptor rings which point to frames of the UMEM. [Cor18][TK18a]

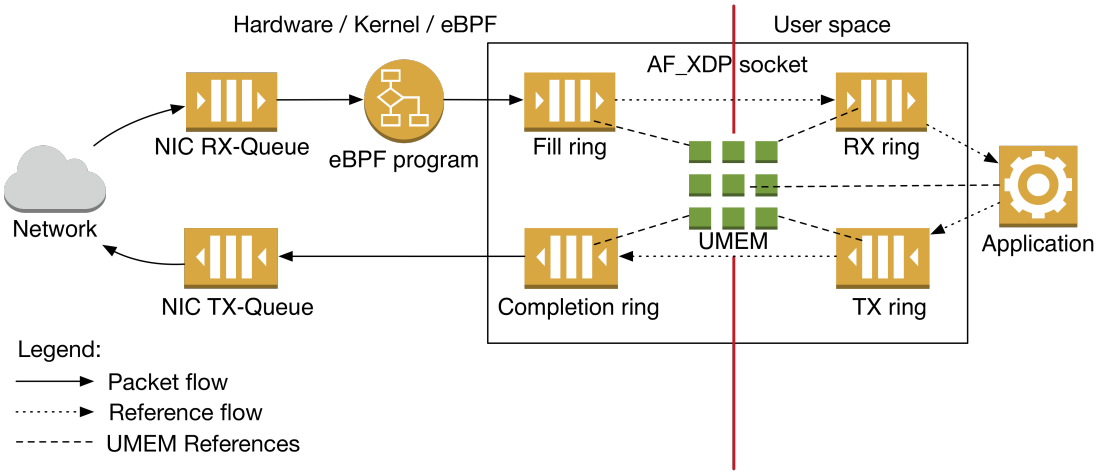

Figure 7: AF_XDP architecture showing the shared UMEM and the rings

Since AF_XDP is working with raw packet data in the UMEM frames, one needs to handle Ethernet, IP, UDP and possibly other additional headers oneself. If you want to use TCP you also need to implement the TCP protocol on top of AF_XDP within one's application. This work is using UDP over IPv4 over Ethernet and doesn't integrate any further protocols like VLAN.

AF_XDP has two operation modes with the release in kernel v4.18. The XDP_SKB mode uses sockets and the generic XDP path. This is up to five times faster than the previous best Linux integrated solution AF_PACKET V3. The second mode is the XDP_DRV mode which works on network devices with XDP support and provides even faster speeds. There will be a third operation mode available within the next kernel releases which will provide true zero-copy mechanisms to AF_XDP. This operation mode is called XDP_DRV + ZC and needs support by the device driver. [TK18b]

2.4. Serverless computing

Serverless computing is a computing architecture that provide Functions as a Service to the customer. This architecture is special because the customer doesn't order specific compute resources and manages them for their application. The customer provides the business logic defined as functions and the serverless computing provider executes the function. The serverless computing platform scales the number of instances for a function up or down according to the current request rate of the function. This reduces the operational costs for customer and increases the flexibility because the customer pays for each request. If there isn't any request, there won't be any cost that could incur.

To be provided by a serverless platform, an application needs to be designed stateless. All the state and data the application requires needs to be provided by other services like databases or object stores. Also standard functionality is used by Backend as a Service (BaaS) providers. One usually don't use one's own authentication service in this application architecture, but the one provided by the serverless provider. The application depends on the serverless provider, which defines how the application needs to be programmed in order to be used as a serverless function in their environment. AWS Lambda provides a platform for functions in the programming languages Java, Node.js, C# and Python.[33] There are also open source serverless platforms like Apache OpenWhisk[34] and serverless.com[35] which provide flexibility and self-hosting solutions for your serverless functions. With these solutions one trades higher operational costs for the flexibility and provider independence.[Rob18]

Figure 8 shows the architecture of the Apache OpenWhisk serverless platform. Nginx[36] is used as the user-facing component and is used as a reverse proxy to the platform controller. The controller extracts the function identifier from the request and looks up the function in the CouchDB[37]. It will also be checked if the client is authenticated and if the client is authorized to request this function. If the request is authorized, the controller publishes the request into a Apache Kafka[38] message queue. Kafka is used as a publish and subscribe messaging system in OpenWhisk. There is an *ActivationId* attached to the request which is used by the client to

[33] `https://aws.amazon.com/de/lambda/features/` (downloaded on 2018-10-11)
[34] `https://openwhisk.apache.org` (downloaded on 2018-10-11)
[35] `https://serverless.com` (downloaded on 2018-10-11)
[36] `https://nginx.org/` (downloaded on 2018-10-11)
[37] `https://couchdb.apache.org` (downloaded on 2018-10-11)
[38] `https://kafka.apache.org/` (downloaded on 2018-10-11)

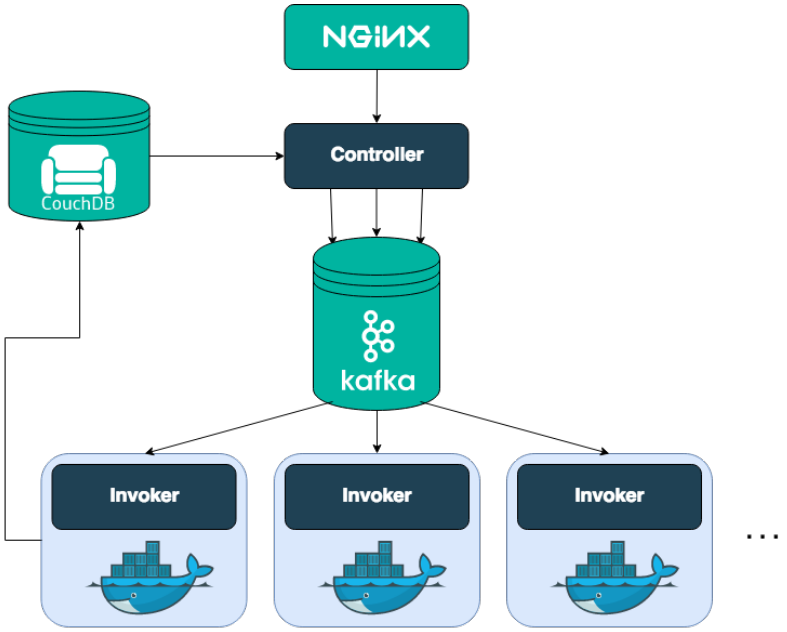

Figure 8: Architecture of Apache OpenWhisk [IBM18]

request the answer from the controller asynchronous.

There might be several hosts active in the serverless platform which are executing the functions. These hosts have an *Invoker* instance running, which subscribed to the Kafka messaging platform and execute the requests in the messaging queues. After the function was executed, the Invoker writes the result into the CouchDB. Now the client is able to request the result with the *ActivationId*.[IBM18]

Due to the fact that the serverless provider has the freedom of managing and scaling the serverless platform, it is possible to compute the workload of many customers by a few machines. This section showed that there are many networking functionalities involved in a serverless platform. The servers involved may provide better performance by an optimized packet processing. This work uses the application of the function invoker as an example application.

2.5. Multi-threading and multi-processing

Today servers provide computing power from multiple CPUs installed in the server. Each CPU usually has multiple CPU cores. With simultaneous multi-threading, each physical CPU core may have multiple logical CPU cores. The term CPU core in this work is defines as a logical CPU core if not stated otherwise. An application which wants to use more computing power than one CPU core provides, has to occupy multiple CPU cores and execute instructions in parallel. This can either be realized by multiple processes in the same process group or by multiple threads within a single process. The work of Inoue and Nakatani [IN10] shows that the performance of multi-threaded processes is higher than multiple processes. Another benefit of threads is that the allocation and start of a new thread is faster than starting a new process for multi-processing. Threads start faster because they live in the same program and share the same global memory. Processes on the other hand result in new memory allocations and data initializations. These reasons lead to the decision, that this work focuses on multi-threaded application designs and doesn't use multi-process applications. However, the performance improvements introduced by this work will also be usable by multi-process applications.[Ker10, p.513ff, p. 616ff]

Multi-threaded applications are able to occupy multiple CPU cores in simultaneously. Optimal performance can be achieved if the threads are allocated on specific CPU cores and no other application is running on the same CPU core. On Linux the tool *taskset*[39] can be used to set the CPU affinity of a process and thus define which CPU cores are usable by the process. Listing 1 shows how to use taskset. Bad performance by using a CPU, which has to handle a lots of interrupt requests, can be avoided by setting the available CPUs for the process accordingly.

Listing 1: Setting the used CPU cores of an application

```
# start the application pktgen only on CPU core 0
taskset --cpu-list 0 ./pktgen
# bind process 1234 to CPU core 1 and 2
taskset --cpu-list 1,2 -p 1234
```

Linux, which is used by this work, provides multi-threading functionality via POSIX threads, called Pthreads. The Pthread Applicatin Programming Interface (API)[40]

[39] `http://man7.org/linux/man-pages/man1/taskset.1.html` (downloaded on 2018-10-11)
[40] `http://man7.org/linux/man-pages/man7/pthreads.7.html` (downloaded on 2018-10-11)

provides the ability to create threads, monitor and interact with them or terminate threads if needed. Listing 2 shows a simple example of a multi-threaded application. The main procedure starts a thread by calling the *pthread_ create* system call. This thread executes the function hello with the given parameter "World".

Listing 2: Definition of a function and execution of the function in a thread.

```
static void* hello (void* arg)
{
    char *var = (char *)arg;
    fprintf(stdout, "Hello %s from Thread\n", var);
}

int main (int argc, char **argv)
{
    pthread_t thread;
    int s = pthread_create(&thread, NULL, hello, "World");
    if (s != 0) {
        fprintf(stderr, "Thread creation failed");
        exit(EXIT_FAILURE);
    }
    sleep(1); //Wait to give the thread time to run
    return EXIT_SUCCESS;
}
```

3. Related work

High performance systems and optimized data paths in the operating systems and applications make usage of the higher network speeds available today. This chapter provides an overview of related work in the NIC and operating systems over the last years.

3.1. Network Interface Card improvements

Current NIC speeds for servers are at 10 Gbit/s and faster. The fastest available NICs provide speeds for up to 200 Gbit/s[41] and 400 Gbit/s were demoed already.[42]

With network cards providing faster network speeds every few years, the server and operation system needs to be able to handle more and more packets, if we assume that the packet sizes stay the same. There are technologies build into the operating system and the network cards to be able to handle these packets. This subsection introduces these technologies.

3.1.1. Receive Side Scaling, multi-queue NICs and hardware offload

Interrupts are one of the glueing components between the operating system and the NIC. On the receiving side the NIC requests an interrupt to tell the operating system that a packet has arrived and is receivable. This hardware interrupts have the highest priority in the operating system and the interrupt handler is executed instantaneously for each interrupt request. Each hardware interrupt source has a handler attached to it which is specified on a specific core of the CPU. If the NIC is under load, the CPU core executing the interrupt handler may be overwhelmed by the amount of interrupt requests and packets get lost.

Receive Side Scaling (RSS) is introduced to distribute the interrupt requests of a single network card over multiple CPU cores. The basic principle is, that a NIC provides multiple RX-queues. The number of queues used by the NIC is configurable

[41] e.g. Mellanox ConnectX-6 `http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-6_EN_Card.pdf` (downloaded on 2018-10-11)

[42] `https://www.eetimes.com/author.asp?section_id=36&doc_id=1331526` (downloaded on 2018-10-11)

and defaults to the number of cores of the CPU, if the NIC provides this number of queues. For each incoming packet the NIC computes a hash over the packet header. The hashing algorithm may be specific to the NIC model but usually includes IP and / or transport header information. Packets belonging to the same connection share the same hash value. The hash value is used to identify which RX-queue is to use for this packet and therefore which CPU core has to handle the interrupt request. Figure 9 shows this in a simple diagram.

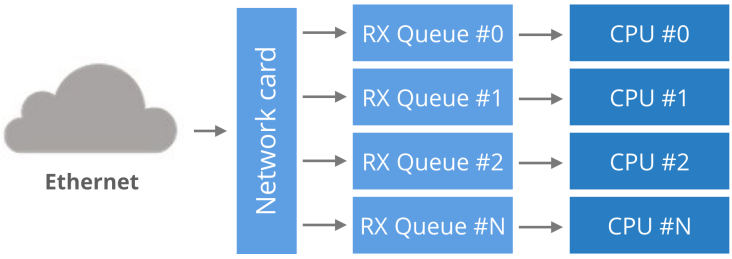

Figure 9: Multi-queue NIC[43]

Muti-queue is not only available on the RX side of the NIC but also on the TX side. If the NIC provides sufficient TX-queues, each CPU core of the system is attached to its own TX-queue. This allows multiple applications to write to through the network stack to the NIC simultaneously. The NIC checks each queue and sends the packet on the line. There might be operating system specific extensions to the TX multi-queue usage. In subsection 3.2.1 Transmit Packet Steering, an extension in the Linux kernel is presented. The commands listed in listing 3 can be used to configure multi-queue NICs and RSS on Linux based operating systems. [HB18]

Listing 3: RSS configuration

```
#show queue configuration of interface ens4
ethtool --show-channels ens4
#set the number of rx and tx queues for interface ens4 to 4
ethtool --set-channels ens4 rx 4 tx 4
#show interrupt sources and interrupt requests by cores
cat /proc/interrupts
#assign core 3 (bitwise 0100) to interrupt source 8
echo 004 > /proc/irq/8/smp_affinity
```

[43]From `https://blog.cloudflare.com/how-to-receive-a-million-packets/` (downloaded on 2018-10-11)

Recent improvements on NICs are not only limited to the network speed and the multi-queue support. The NICs also got smarter. Most of the recent business grade network cards provide feature offloads to the operating systems. The NIC used for this work provides multiple feature offloads like checksumming for IPv4 and IPv6 packets, TCP segmentation and more. A complete list of enabled features in this work is shown in listing 4.

Listing 4: Enabled NIC features in this work

```
> ethtool --show-features ens4 | grep ": on"
rx-checksumming: on
tx-checksumming: on
        tx-checksum-ipv4: on
        tx-checksum-ipv6: on
scatter-gather: on
        tx-scatter-gather: on
tcp-segmentation-offload: on
        tx-tcp-segmentation: on
        tx-tcp6-segmentation: on
generic-segmentation-offload: on
generic-receive-offload: on
rx-vlan-offload: on
tx-vlan-offload: on
receive-hashing: on
highdma: on [fixed]
rx-vlan-filter: on [fixed]
rx-vlan-stag-hw-parse: on
rx-vlan-stag-filter: on [fixed]
rx-udp_tunnel-port-offload: on
```

NIC features like this take some functions from the TCP/IP stack and integrate them on the NIC hardware. By this the TCP/IP stack doesn't need to check or compute specific checksums or perform TCP segmentation. This reduces the system load generated by the TCP/IP stack and more resources are usable by the user space.

The feature *receive-hashing* is enabled on the machine used for this work. This is used for RSS and queue selection. The used NIC doesn't provide the feature *ntuple-filters* which allows programmable filters. One filter could forward all packets with destinatio port 80 to a single queue or provide quality of service functionality for specific services. [HB18]

3.1.2. Smart Network Interface Cards

A programmable NIC which provides more than the basic functionality of TCP/IP stack function offloading is called a SmartNIC. The biggest difference to other network cards is, that SmartNICs provide a software defined data plane on the network card. Netronome, one of the leading hardware manufactures of SmartNICs, defines that a SmartNIC must:

> 1) Be able to implement complex server-based networking data plane functions, including multiple match-action processing, tunnel termination and origination, metering and shaping and per-flow statistics, for example.
>
> 2) Support a fungible data plane either through updated firmware loads or customer programming, with little or no predetermined limitations on functions that can be performed.
>
> 3) Work seamlessly with existing open source ecosystems to maximize software feature velocity and leverage.

[Tau16]

Currently there is no standard programming language available for every SmartNIC. Different vendors introduced different programming interfaces for their SmartNICs. The SmartNICs of Netronome can be programmed with P4[44] to accelerate the specific use case and integrate new functions directly on the NIC. Netronome also supports eBPF offload with some of its SmartNICs. The eBPF code is executed on the NIC in this case.[45]

Kaufmann et al. [KPS+16] show in detail how a flexible programmable NIC can be used to accelerate specific use cases. They build a match plus action pipeline with programmable NICs and use them to optimize the data flow from the NIC to the desired application. In case of a key-value store use case, they identify get and set requests to the key-value store and generate a hash over the requested key. With this hash they use Direct Memory Access (DMA) to push the request directly into the request memory buffer of the application. Since the hash value is already calculated by the SmartNIC, the application doesn't need to calculate it again. FlexKVS, their

[44] `https://p4.org` (downloaded on 2018-10-11)
[45] `https://www.netronome.com/technology/ebpf/` (downloaded on 2018-10-11)

modified Memcached[46] key-value store is able to handle 16 times more requests than a standard Memcached instance over the same number of cores. The processing time of each request was also reduced by 60% compared to an installation with a high performance user-level network stack. Figure 10 shows the architecture and the match and action steps used for this work.

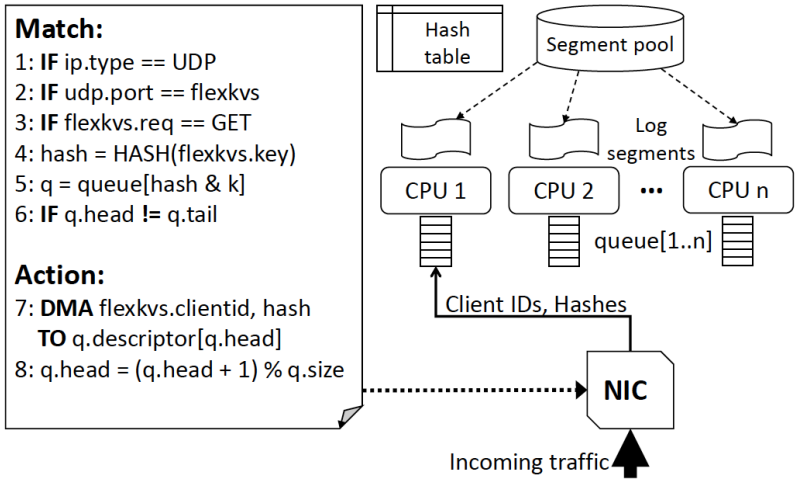

Figure 10: FlexKVS application architecture and match plus action pseudocode [KPS+16, 71]

Other use cases have also been topic of the work. The throughput of a real-time analytic platform has been increased by a factor of 2.3 and the throughput of an intrusion detection system by 60%. This shows the high potential in performance optimizations of applications by using a SmartNIC. [KPS+16]

3.1.3. Infiniband and Remote Direct Memory Access

Another development in the NIC marked doesn't particular use the network. Infiniband is used to provide a low latency connection between servers in a high performance cluster. With Remote Direct Memory Access (RDMA) the network stack is bypassed completely and packets are stored in a memory region where an application can consume the data. [LWP04] shows that a message passing interface build on top of RDMA with Infiniband reduces the latency by 24% and increases the bandwidth by 104%. Simultaneously the overhead of the network interactions are reduced by 22%.

[46] `https://www.memcached.org` (downloaded on 2018-10-11)

3.2. Operating systems

Operating systems received new features and functions to provide better network performance in the last years. Some changes also resulted in new operating systems. This subsection provides an insight in network performance related work in the Linux operating system and in micro-kernel operating systems. An overview of user space network stacks and applications currenty using eBPF is also provided.

3.2.1. Linux kernel

The Linux kernel is a monolithic kernel which abstracts input and output operations via system calls. If an application wants to send or receive data over the network, the application uses the system calls *send()* or *recv()* provided by the kernel. The kernel itself has a TCP/IP stack included. Some details about the datapath in the Linux kernel TCP/IP stack are presented in section 2.2 on page 8. Most of the Portable Operating System Interface (POSIX)[47] system calls are available on Linux based operating system. Thus, applications using this interface are not only usable on Linux machines but also on other operating systems supporting the POSIX standard.

The kernel does not only provide a simple interface for the application to send and receive data. Since many applications share the usage of the Linux TCP/IP stack, security concerns are present. No other user-space application on the same host should be able to intercept the data flow and receive the data from another application. Special tools like tcpdump which are executed by the root user are able to have low level access, but other applications don't. Data provided to or received by the system call is copied to or from the user space memory of the application performing the system call.

Providing an abstraction layer to applications isn't only used for the network input and output but also for storage access and other resources. With this implementation the kernel provides easy usage of those resources and the performance of this implementation is sufficient for must use cases. Since the kernel also provides the device driver and sets up the devices, the application is completely independent of the system and portable to use on other hardware.

[47] `https://publications.opengroup.org/standards/unix/t101` (downloaded on 2018-10-11)

RPS, RFS and Transmit Packet Steering (XPS)

If an use case is in need of higher performance than provided by the Linux system defaults there are some optimizations configurable by the system administrator to tweak the performance. One such optimization is RSS which provides performance improvements for multi-queue NICs. Detailed information on this feature are given in section 3.1.1 on page 20.

Further optimizations in regard to parallel data processing by the TCP/IP stack are RPS, RFS and XPS. RPS is similar to RSS but is implemented on the software level. Typically the CPU core which received the interrupt from the network device is also processing the packet through the TCP/IP stack. If the network card has less queues than the system has CPU cores, RPS can be enabled and configured to allow multiple CPUs to process packets by the network card. The interrupts are still handled by a single core.

RFS on the other hand is an extension to the RPS functionality. While RPS is distributes incoming packets across the enabled cores, RFS provides a flow-table for the network flows used by the network card. If a flow is already known, the packets will be forwarded to the same CPU core. By this the cache hit ratio is improved and the throughput can be optimized. With activated ntuple-filters[48] on the network card, it is also possible to accelerate RFS.

XPS is the counterpart of RSS on the transmit side. The default configuration is, that each CPU core gets its own transmit queue attached, if the NIC provides sufficient queues. Under certain circumstances the packets may be sent out of order if they use different CPU cores. XPS provides also flow based transmit queue steering. This can be used to guarantee that the packets of a flow are sent via the same queue and that the packets are transmitted in order. [HB18]

Table 2 provides the performance test results provided by the developer of RPS and RFS. It shows the processed Transactions per second (TPS) and the CPU utilization. Figure 11 shows the general architecture of the data path in Linux, ranging from NIC-queues through the kernel and to the applications. Listing 5 provides the configuration commands to set up RPS, RFS and XPS.

[48] see section 3.1.1 Receive Side Scaling, multi-queue NICs and hardware offload on page 20

Setting	Transactions per second	CPU utilization
No RFS or RPS	104 K	30 %
RPS	290 K	63 %
RFS	303 K	61 %

Table 2: RFS performance on e1000e NIC with 8 core system[Her10]

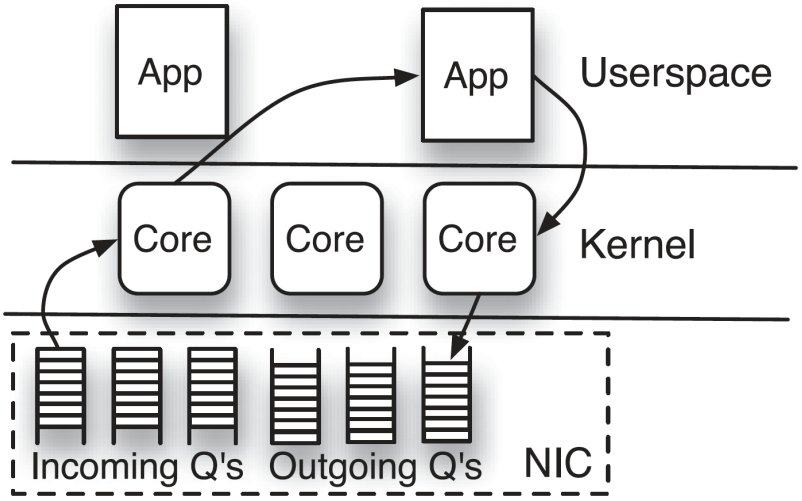

Figure 11: Linux network I/O architecture [PLZ+15]

Listing 5: RPS, RFS and XPS configuration

```
#assign core 2 and 1 to the RPS group of rx-0 on interface ens4
echo 003 > /sys/class/net/ens4/queues/rx-0/rps_cpus

#enable RFS with 32768 flow table entries available
echo 32768 > /proc/sys/net/core/rps_sock_flow_entries
#bind half the flows to rx-queue 0 on interface ens4
echo 16384 > /sys/class/net/ens4/queues/rx-0/rps_flow_cnt

#allow cores 1,2,3 and 4 to use tx-queue 0 on interface ens4
echo 00f > /sys/class/net/ens4/queues/tx-0/xps_cpus
```

The SO_REUSEPORT socket option

There are also optimizations an application is able to use in combination with the Linux kernel TCP/IP stack. First the SO_REUSEPORT socket option, which is available with Linux kernel version 3.9 and newer. This option is used to bind multiple sockets to the same IP-address and port combination. This allows multiple instances of an application to receive and send packets on the same port. To enable this feature for an socket, every instance of the sockets needs to have the socket option set. For security reasons all sockets must be opened by the same user. With a further optimized setup it is possible to attach a cBPF or an eBPF program to the socket. This program can be used to distribute the requests across all the instances of the sockets. The default procedure is to assign a configuration to a socket instance by hashing the IP-address and port combination of the sender.[Ker13]

Listing 6: Enabling SO_REUSEPORT on a socket [Ker13]

```
int sfd = socket(domain, socktype, 0);

int val = 1;
setsockopt(sfd, SOL_SOCKET, SO_REUSEPORT, &val, sizeof(val));

bind(sfd, (struct sockaddr *) &addr, addrlen);
```

Reduce system calls with multiple message receive and send

Another optimization usable by the application is to use specific system calls which are able to receive or send multiple messages. The system calls are called *recvmmsg()*[49] and *sendmmsg()*[50] accordingly. To use this feature an array of *mmsghdr* structures must be allocated and configured. Each mmsghdr structure contains an *msg_hdr* structure which itself can hold multiple*iovec* structures. Each iovec structure contains a base address of a data structure and a length. For receiving this length is used to know how large the buffer is, in case of sending the length defines how many bytes to send. Additionally the msg_hdr structure contains the *msg_name* which contains the IP-address of the other side of the connection. Listing 7 shows how to use the sendmmsg and recvmmsg system calls. For UDP the use of sendmmsg there is a performance gain of 20% in throughput.[51]

[49] `http://man7.org/linux/man-pages/man2/recvmmsg.2.html` (downloaded on 2018-10-11)
[50] `http://man7.org/linux/man-pages/man2/sendmmsg.2.html` (downloaded on 2018-10-11)
[51] `https://lwn.net/Articles/441169/` (downloaded on 2018-10-11)

Listing 7: Multi message send and receive echo server

```
#define MMSGLEN 64
struct sockaddr_in server, clients[MMSGLEN];
struct mmsghdr msgs[MMSGLEN];
struct iovec iovecs[MMSGLEN];
char buffer[MMSGLEN][BUFLEN];
// Initialize data structures
memset(buffer, 0, sizeof(buffer));
memset(msgs, 0, sizeof(msgs));
for (int i = 0; i < MMSGLEN; i++) {
    iovecs[i].iov_base = buffer[i];
    iovecs[i].iov_len = BUFLEN;
    msgs[i].msg_hdr.msg_iov = &iovecs[i];
    msgs[i].msg_hdr.msg_iovlen = 1;
    msgs[i].msg_hdr.msg_name = &clients[i];
    msgs[i].msg_hdr.msg_namelen = sizeof(clients[i]);
}
// recv at least one message and up to MMSGLEN
int recv_len = recvmmsg(sock, msgs, MMSGLEN, MSG_WAITFORONE, 0);
if (recv_len < 0) { // Length negative if recvmmsg failed
    exit(EXIT_FAILURE);
} else if (recv_len > 0) {
    for (int i = 0; i < recv_len; i++) { // Read every packet
        // Set the return message size
        msgs[i].msg_hdr.msg_iov[0].iov_len = msgs[i].msg_len;
    }
    // Respond to the requests
    if (sendmmsg(sock, msgs, recv_len, 0) != recv_len) {
        exit(EXIT_FAILURE); //not all messages sent
    }
}
```

3.2.2. Micro-kernel

Beside Linux with its monolithic kernel there are other operating system architectures available. Micro-kernel architectures provide only a subset of the functionality of a monolithic kernel and may be extended by modules to provide further function-

ality. The networking functionality could by provided by a module or implemented directly into the application if needed.

Arrakis is an operating system which uses a micro-kernel architecture and is specialized for low latency and high performance network and disk I/O. The kernel provides the functionality to control access to the storage and the network devices but the kernel is not integrated in the data plane. If data is send by or send to the application the data doesn't have to traverse the kernel and the overhead of copying data through the kernel is omitted.

An application that wants to send or receive data has to request a virtual device from the Arrakis micro-kernel. The Single Root I/O Virtualization technology is used to provide multiple virtual NICs with only a single physical NIC. Afterwards the virtual NIC can be used by the application. Arrakis provides a library for the applications which contains device driver and some basic abstraction usable by the application and even allows usage of the POSIX API for the application. However, the application could also implement the driver and I/O functionality directly into the program if needed. This mode allows the application to read the packet without a single copy instruction and reduces the overhead. If an application can access a packet without a single data copy, the term zero-copy is used. [PLZ+15]

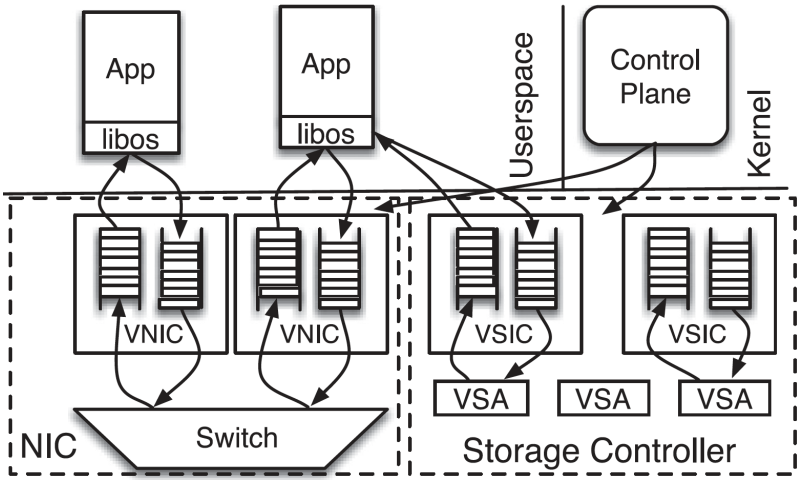

Figure 12: Arrakis I/O architecture [PLZ+15]

Figure 12 shows the architecture of the I/O path in the Arrakis operating system. If one compares the architecutres of Arrakis with the I/O path in Linux, presented in figure 11 on page 27, differences are easy to spot. The kernel got removed from

the data path for the application.

By getting the operating system out of the data path and thus eliminating system calls the I/O performance improves. The Arrakis whitepaper [PLZ+15] provides test results for different applications. A UDP echo server was used to measure the throughput of the data path with a simple echo of the payload. The POSIX API calls to the Arrakis library available for the applications achieves 2.3 times more throughput than the Linux socket implementation. With the zero-copy implementation on Arrakis the throughput is 3.9 times the throughput of Linux.

Arrakis shows that the elimination of the kernel in the data path is able to accelerate the network performance of applications. The Arrakis operating system is based on the Barrelfish[52] operating system. The contributions made to the Arrakis kernel have been merged back into the Barrelfish operating system. The system is under active development and works on optimizing the performance on multi-core systems.

3.2.3. User space network stacks

The last section already showed that I/O performance can be improved by taking the operating system kernel out of the data path. The Linux kernel is monolithic and provides its own network functionality. Beside this, the kernel also provides the functionality to provide hardware access to user space programs. User space network stacks are able to bypass the Linux kernel and use the NIC directly. Data Plane Development Kit (DPDK)[53], which is developed by Intel, is one of the most famous user space network stacks.

DPDK unbinds a network card from the Linux network stack and configures the network device directly by setting the parameters in the correct /dev location of the device. DPDK needs to support the NIC and integrate the driver for the device to be able to configure it and make use of the NIC. The packets received by the NIC are stored into a memory chunk available via DMA from the NIC. If DPDK is used, it allocates the memory and configures the NIC to use the allocated memory.

While the Linux network stack uses interrupt handlers to react on new packets, DPDK uses a descriptor ring. The NIC writes a reference to the head of the ring

[52] `http://www.barrelfish.org` (downloaded on 2018-10-11)
[53] `https://www.dpdk.org` (downloaded on 2018-10-11)

and DPDK checks the ring if new data is readable. Furthermore the applications using DPDKs network stack also use descriptor rings to consume data. By only passing the reference to a chunk of memory to the application, there are no data copies involved in the data path. Thus, DPDK provides zero-copy network access to applications on Linux.

Since the architecture of the network stack differs that much to the Linux kernel network stack, applications need to be developed especially with DPDK in mind. PF_RING[54], developed by ntop, is another alternative network stack implementation available targeting high performance networking. Both of the implementations are capable of network processing at line speeds of 10 Gbit/s and more. The downsides of user space network stacks is, that they only work with a subset of NICs and applications need to be developed especially for these network stacks. Thus, the applications are not portable anymore. [Sch14]

3.2.4. Enhanced Berkeley Packet Filter

eBPF is a new technology in terms of kernel functionality and the XDP functionality is even newer. XDP and eBPF in general can be employed to react on incoming packets before they hit the Linux kernel data path and before the kernel invests unnecessary CPU cycles on the data arrived. The programmability of the eBPF pseudo-machine in the Linus kernel allows simple use cases like packet forwarding to bypass the kernel and don't allocate resources in the kernel beside the eBPF pseudo-machine. This subsection contains information about projects which employ eBPF and XDP.

Open vSwitch[55] a software based network switch used by virtualization platforms like XenServer[56], Proxmox[57] and OpenStack[58]. The software provides virtual switching functionality between the VMs and the physical NIC or other VMs. Open vSwitch not only provides data plane functionality but also provides security features like traffic filtering, monitoring and Quality of Service (QoS). Since Open vSwitch takes part in the data path of every packet sent by the VMs, performance is very important. Currently there is ongoing work in the Open vSwich development for

[54] `https://www.ntop.org/products/packet-capture/pf_ring/` (downloaded on 2018-10-11)
[55] `https://www.openvswitch.org` (downloaded on 2018-10-11)
[56] `https://www.xenproject.org` (downloaded on 2018-10-11)
[57] `https://www.proxmox.com/en/` (downloaded on 2018-10-11)
[58] `https://www.openstack.org` (downloaded on 2018-10-11)

performance improvements by utilizing eBPF in the data path. By providing a flow table directly into the eBPF pseudo machine packets can be forwarded or dropped faster. A first request for change has been submitted to the development mailing list of Open vSwitch in June 2018.[59] [TSP17]

Another project utilizing the eBPF pseudo-machine is Katran[60], a C++ library and eBPF program developed by Facebook. Facebook uses Katran as a layer 4 load balancer. Since the eBPF pseudo-machine handles request much faster than the traditional TCP/IP stack, Katran doesn't occupy designated machines but lives together with other services on the same hardware. Katran announces virtual IPs to other network elements and acts on requests coming to this IP. It computes a consistent hash over the 5-tuple of source and destination IP, source an destination port and the protocol. Based on the hash the actual server for processing of the request is chosen. The consistent hash is used to guarantee, that a request handled by another Katran instance will be forwarded to the same server responsible for the 5-tuple. [DS18]

These two examples show that the eBPF is employed by some projects already. One other ongoing project employing eBPF is the change of the kernel part of iptables to eBPF. Iptables is the primary tool for packet filtering or firewalling on Linux and has been for many years. Third party applications use the iptables interface to set rules and the packet filtering is done by iptables. Performance wise iptables has some downsides when it comes to multiple thousands of rules. Setting up 20 thousand services on Kubernetes leads to 160 thousand iptables rules. If one appends a rule to the iptables ruleset, all the existing rules are replaced with a new copy. Setting up the 160 thousand rules take 5 hours.[61] First changes have been introduced with Linux kernel version 4.18.[62] The performance improvements by some early tests in the development phase are displayed in figure 13. In combination with the hardware offload, bpfilter, the eBPF enabled implementation of iptables, is able to provide traffic filtering at line speed. [Gra18]

[59] `https://mail.openvswitch.org/pipermail/ovs-dev/2018-June/348521.html` (downloaded on 2018-10-11)
[60] `https://github.com/facebookincubator/katran` (downloaded on 2018-10-11)
[61] `https://www.slideshare.net/LCChina/scale-kubernetes-to-support-50000-services` (downloaded on 2018-10-11)
[62] `https://lwn.net/Articles/747504/` (downloaded on 2018-10-11)

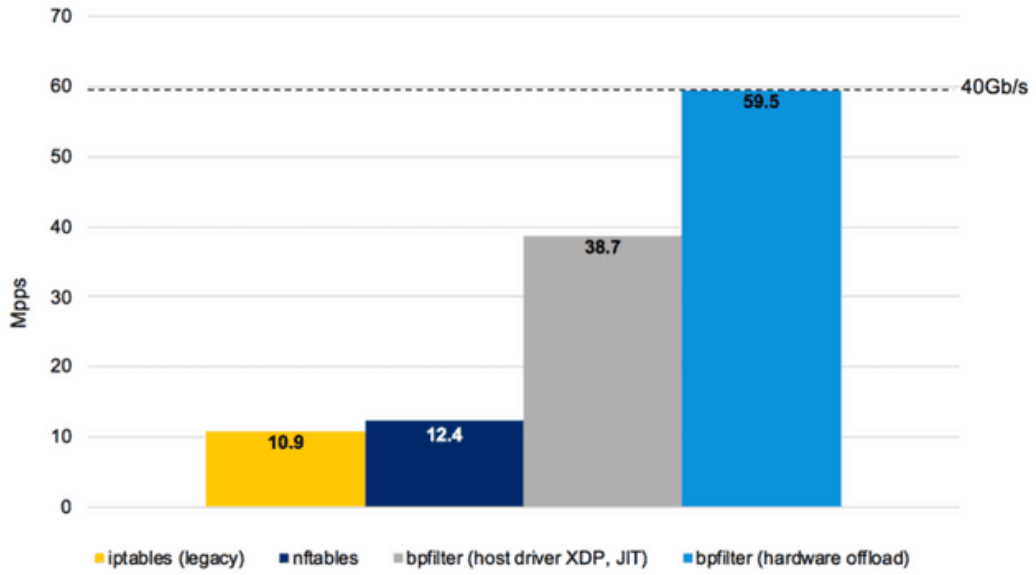

Figure 13: Performance comparison of iptables vs nftables vs bpfilter [Gra18]

The implementation of new eBPF and XDP features doesn't stop. This work is build with AF_XDP, an extension to the kernel introduces with Linux kernel version 4.18. Currently the AF_XDP functionality is extended to support zero-copy mechanisms in combination with supported NICs.[63] This allows even faster and more efficient packet processing. Not only AF_XDP is under active development but there are new features introduced to the eBPF system every kernel release. [Gra18] states, that implementations like bpfilter are only the beginning of the utilization of the eBPF pseudo-machine. There is great potential in the utilization of eBPF by applications.

3.3. Summary

This section introduced the current state of network processing such as NIC improvements like multi-queue RSS or Linux kernel improvements like RFS. Faster network speeds need such parallel packet processing mechanisms to share the load across multiple CPU cores. Optimized technologies like programmable NICs or the hardware offload of eBPF programs are able to accelerate packet processing up to line speeds of 40 Gbit/s and more.

[63] `https://lwn.net/Articles/754659/` (downloaded on 2018-10-11)

Alternative approaches like user space network stacks have been around some time and were needed for high performance workloads and for processing millions of packets per second. eBPF provides a programmable data layer for Linux. An overview of some projects employing the eBPF system for higher performance was given and shows that the technology is accepted by the developers. Employing eBPF programs increases the opportunity to process packets at line speed without the need of user space network stacks. AF_XDP which is build on top of eBPF is able to bypass the Linux TCP/IP stack and thus allow packet processing with lower overhead. This work uses AF_XDP to process packets more efficient and to increase the throughput of user space applications to be able to work at line speed.

4. State analysis

This work is based on the use case of a serverless function invoker with the goal of improving the efficiency of the platform. The solution which is able to handle more packets and thereby execute more functions in the same time is more efficient since less servers are needed to provide the same service to the customers.

A performance evaluation with the different implementations will be performed. Each of the candidates taking part in the evaluation has to invoke a function with incoming packets. To simulate the serverless function invoker use case accordingly, there is a function repository available which can define multiple functions. The function invoker selects the function based on the destination port of the UDP packet. All the implementations use the same function repository to execute user defined functions on the payload of the message. For the performance tests the defined function reverses the payload. The last byte is exchanged with the first byte and so on. Listing 8 shows the definition of the function interface and the specific implementation for the function attached to port 1232. If the function changes the length of the payload, the parameter length can be changed in the defined function. It has to be used by the implementations in the send path.

Listing 8: Function repository withthe function reversePayload used in this work

```
typedef bool (*function)();

// Function 1232 reverses the payload
bool func_1232_reversePayload(
        char *pkt, unsigned int *length,
        const unsigned int header_length)
{
    const int iterations = (*length - header_length) >> 1;
    char c = 0;
    for ( int i = 0; i < iterations; i++ ) {
        c = pkt[header_length + i];
        pkt[header_length + i] = pkt[*length - 1 - i];
        pkt[*length - 1 - i] = c;
    }
    return true;
}
```

```
function get_function(const unsigned int port)
{
    switch(port) {
    case 1232:
        return func_1232_reversePayload;
        break;
    default:
        return NULL;
        break;
    }
    return NULL;
}
```

In this section two implementations build with the Linux TCP/IP stack and with Linux sockets are implemented and compared in terms of performance. First there will be a simple implementation which creates one socket, reads from the socket, executes the function and responds back to the origin of the packet. The second implementation will optimize the Linux socket implementation with features Linux already provided out of the box.

Measurements in this section are taken as described in chapter 7 on page 71. The serverless function invoker application is started and a packet generator generates requests to the application. A test run only uses packets of one size and is executed for 20 minutes. within this time a 15 minute time period of throughput measurements is collected and the average throughput is used as the result of the test.

4.1. Simple Linux socket implementation

The simple socket implementation creates a single UDP socket and binds it to the defined port on any interface of the machine. Afterwards the program starts a loop and tries to read data from the socket with the *recvfrom* system call. If data is available, the system call stores it in the provided buffer. The application can access the data now and invoke the function for this packet. Since the socket doesn't provide the Ethernet, IP and transport header the payload starts with the first byte and the function is called with a header length of 0. If the execution of the function is successful, the now modified payload in the buffer is sent back to the origin with the *sendto* system call. Listing 9 shows the actual simple socket implementation.

Listing 9: Simple Linux socket implementation

```
#define PORT 1232
struct sockaddr_in server, client;
int sock, recv_len;
unsigned int addr_len = sizeof(client);
char buffer[BUFLEN];
memset(buffer, 0, sizeof(buffer));
function func = get_function(PORT); // Get the function
if (func == NULL){
    fprintf(stderr, "No Function defined...\n");
    return NULL;
}
// Create Socket
sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (sock < 0)
    error_exit ("Socket creation failed\n");
fprintf(stdout, "Socket created\n");
// Set Server Connection
memset(&server, 0, sizeof(server));
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port = htons(PORT);
server.sin_family = AF_INET;

if (bind(sock, (struct sockaddr*)&server, sizeof(server)) < 0)
    error_exit("Socket bind failed\n");
fprintf(stdout, "Socket bound\n");
    for (;;) {
    recv_len = recvfrom(sock, buffer, BUFLEN, 0,
                        (struct sockaddr*) &client, &addr_len);
    if (recv_len < 0) {
        error_exit("recvfrom() failed\n");
    } else if (recv_len > 0) {
        if (!(func)(buffer, &recv_len, 0)) { // Execute function
            error_exit("Function failed\n");
        }
        //Answer
        if (sendto(sock, buffer, recv_len, 0,
                (struct sockaddr*)&client, addr_len) != recv_len) {
            error_exit("sendto() failed\n");
```

```
            }
        }
    }
close(sock);
```

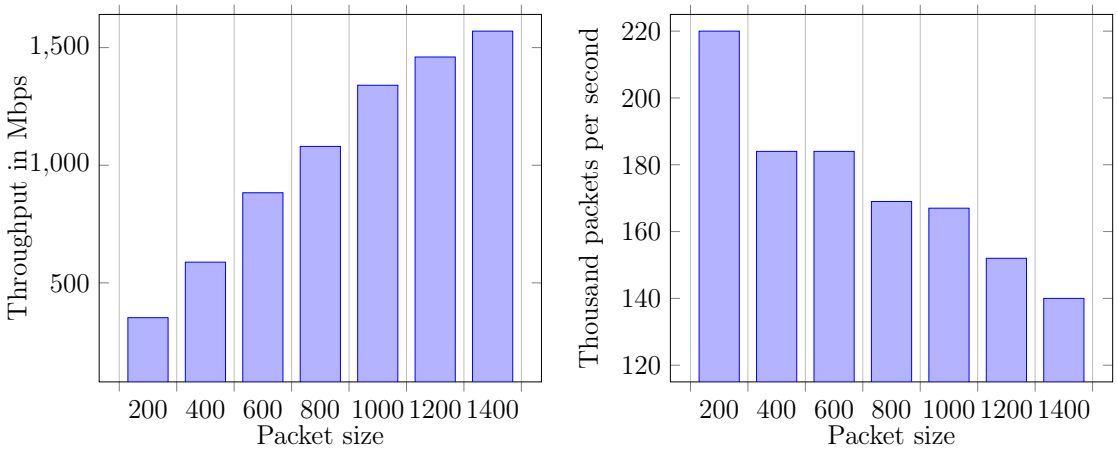

Figure 14: Throughput performance of the simple socket implementation

This application was then executed on one test system while the other test system generated load on the network. Multiple tests with packet sizes from 200 to 1400 bytes were executed and the throughput and the processed packets per second monitored. Figure 14 shows the average performance values of these tests. The timespan of each test is 15 minutes and the measurement started after at least one minute of load. As one can see in the charts, the smaller the packets the more packets can be processed per second. However, the Throughput is the highest with fewer but larger packets. Regarding the throughput we are far from saturating the 10 Gbit/s connection between the systems. In the next subsection the performance will be tuned by optimizing this implementation.

4.2. Optimized Linux socket implementation

After the first performance test with the standard Linux socket implementation in the previous subsection, the implementation gets modified to optimize the performance. The technologies used are described in section 3.1.1 *Receive Side Scaling, multi-queue NICs and hardware offload* (p. 20) and 3.2.1 *Linux kernel* (p. 25).

In detail the changes to the implementation from the previous section are a optimized

Setting	Throughput	Packets per second
8 Queues	1.08 Gbit/s	169.000
4 Queues	1.26 Gbit/s	197.000
2 Queues	1.39 Gbit/s	218.000
1 Queue	1.45 Gbit/s	227.000

Table 3: Throughput for different RSS settings with packet size 800 and one thread reading with one socket

RSS configuration, the ability to use multiple sockets on the same port with the socket option SO_REUSEPORT and the reduction of system calls by using the multiple message send and receive system calls.

The test system for this work provides multiple RX and multiple TX queues. RSS is activated by default. The measured throughputs in table 3 show, that this setting might not be optimal for some applications. If an application uses only one worker thread with one socket the throughput is higher with just one RX queue handling all requests. When multiple threads and sockets consume the data, RSS has to be enabled for optimized performance.

Further performance increases have been achieved by reducing the system calls with multiple message receive and send system calls. The application has been adapted to use recvmmsg and sendmmsg with up to 1024 messages received or sent by each system call. While reading the option MSG_WAITFORONE is set to reduce the latency added by the recvmmsg. With this option enabled the recvmmsg call blocks till the first packet is readable. After the first packet, MSG_DONTWAIT is set and the system call returns with one or more packets.[64] Without the option recvmmsg would read till the maximum amount of messages arrived or after a defined timeout. During tests for this work it was observed that enabling the timeout has a negative impact on the recvmmsg performance. Thus, the timeout was disabled.

Figure 15 shows the improvements in throughput with the optimizations of RSS and multiple message receive and send system calls. For packets with a size of 200 bytes the performance has more than doubled from 352 Mbit/s to 741 Mbit/s. Larger packets result in more CPU cycles used by the function and thus the over all performance improvement shrinks. For packets with the size of 1400 bytes, the results increased from 1.57 Gbit/s to 2.23 Gbit/s.

[64] `http://man7.org/linux/man-pages/man2/recvmmsg.2.html` (downloaded on 2018-10-11)

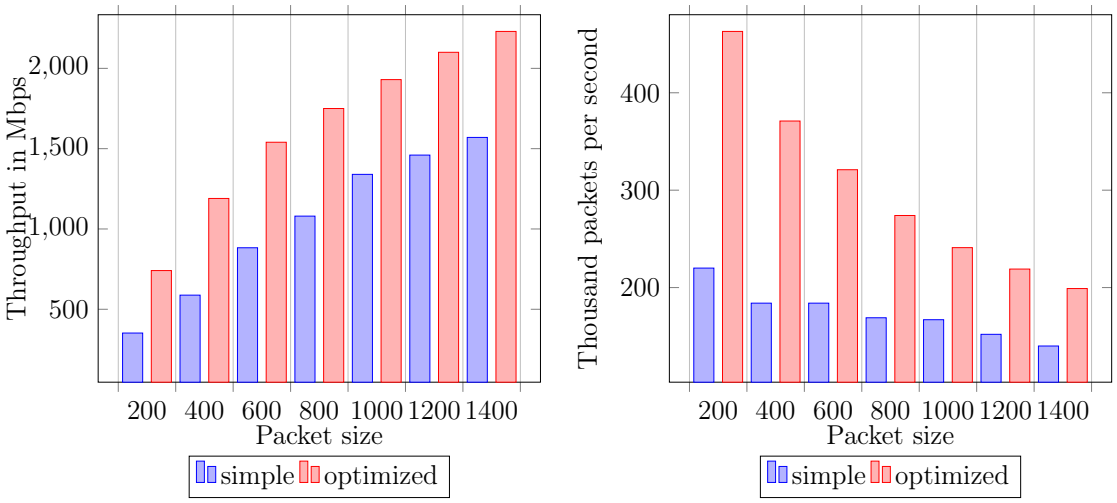

Figure 15: Throughput performance of the simple socket implementation and the optimized implementation with utilizing one thread

CPU #	Task	CPU #	Task
0	IRQ RX-queue 1	1	application thread 6
2	IRQ RX-queue 2	3	application thread 5
4	IRQ RX-queue 3	5	application thread 4
6	IRQ RX-queue 4	7	application thread 3
8	application thread 8	9	application thread 2
10	application thread 7	11	application thread 1

Table 4: CPU task allocation in the multi threaded socket implementation

With a single core performance of up to 2.23 Gbit/s the now optimized implementation is changed into a multi thread application with multiple threads working as the function invoker. To bind the same port and address combination to multiple threads the SO_REUSEPORT option is used. Every thread creates its own socket and binds it to the same port and address. The number of threads is configurable and each thread performs the performance optimized implementation.

For the multi threaded application it makes sense to enable multiple queues if performance benefits. As the numbers in table 3 show an application workload might benefit from fewer queues. This may also be true for multi threading configurations. There will be multiple thread and RX-queue combinations tested to evaluate which of the combinations performs best. To further optimize the application, the threads are bound to CPU cores which doesn't do the interrupt handling for RX queues. To pin the process to specific CPU cores the taskset command is used. Table 4 shows

which of the 12 logical CPU cores are allocated by which task. The allocation of the interrupt request handler (IRQ) CPUs is chosen because simultaneous multithreading is enabled on the test system. The optimal RSS performance is achieved if only one logical CPU per physical CPU is used.[HB18]

Figure 16 contains the results of the different threads (T) and queues (Q) combinations. It is noticable that the throughput doubles from single thread and singe queue to the two thread and two queues run. However, this is not true while doubling the queues and threads again. With 4 threads and either two or 4 queues the throughput is only three times as much as the single core performance. 8 threads and 4 queues are able to provide 4 times the single thread performance.

A comparison of the 4 threads with 2 queues with the 4 threads and 4 queues values shows, that more RSS queues while receiving packets are a benefit for workloads with many small packets and fewer queues provide better performance if the packets get larger and the function consumes more CPU cycles.

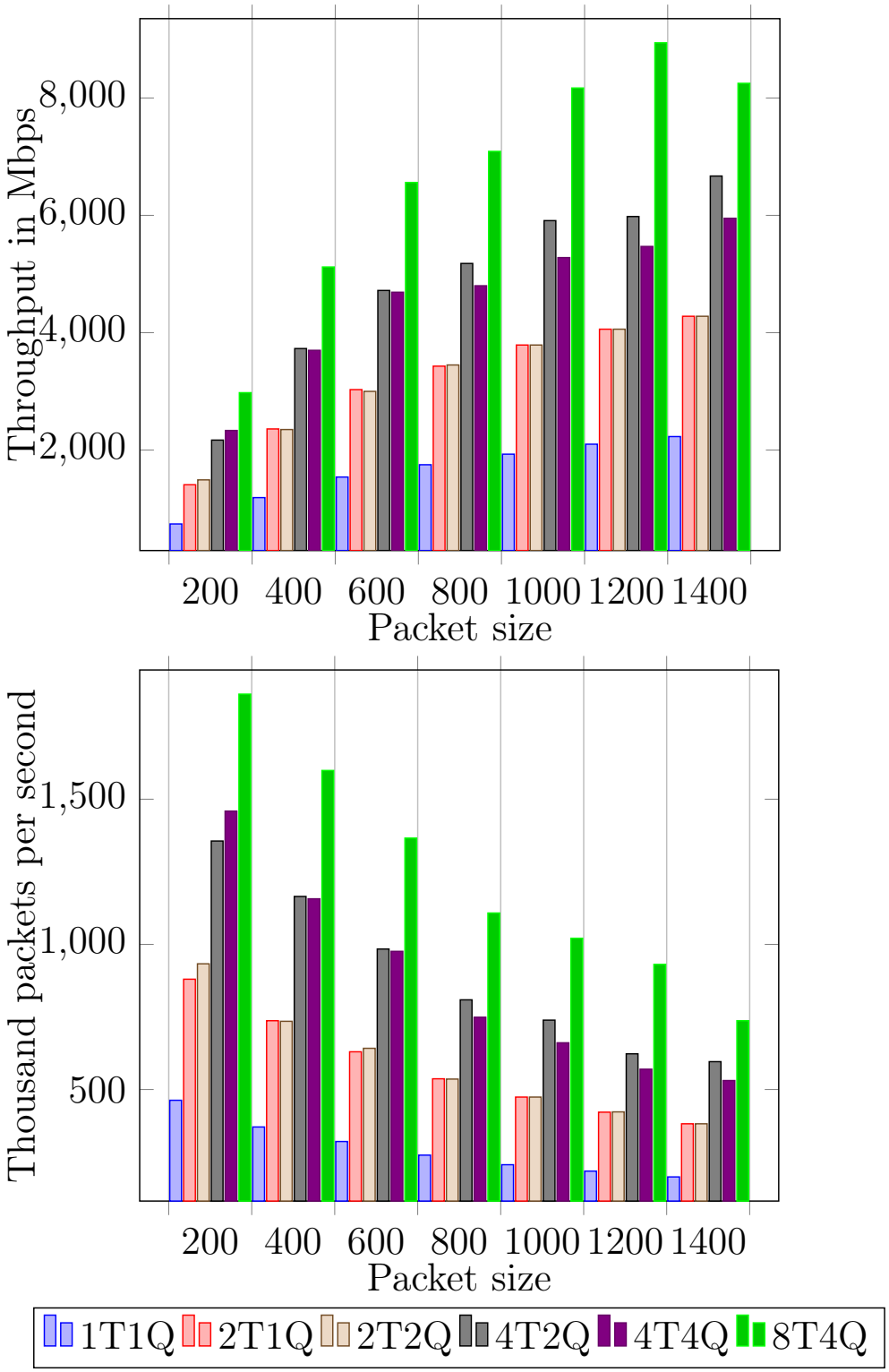

Figure 16: Throughput performance of the optimized socket implementation with several thread and queue combinations

Listing 10 contains the source code for the optimized application. The wrapper around this implementation is not shown. The wrapper starts as many threads as requested whereby every thread executes the function in the listing.

Listing 10: Optimized Linux socket implementation

```
#define MMSGLEN 1024;
#define BUFLEN 2048;
static void* recvpkg (void* arg)
{
    int sock, recv_len, i;
    struct sockaddr_in server, clients[MMSGLEN];
    struct mmsghdr msgs[MMSGLEN];
    struct iovec iovecs[MMSGLEN];
    char buffer[MMSGLEN][BUFLEN];

    // Initialize data structures
    memset(buffer, 0, sizeof(buffer));
    memset(msgs, 0, sizeof(msgs));
    for (i = 0; i < MMSGLEN; i++) {
        iovecs[i].iov_base = buffer[i];
        iovecs[i].iov_len = BUFLEN;
        msgs[i].msg_hdr.msg_iov = &iovecs[i];
        msgs[i].msg_hdr.msg_iovlen = 1;
        msgs[i].msg_hdr.msg_name = &clients[i];
        msgs[i].msg_hdr.msg_namelen = sizeof(clients[i]);
    }

    // Get the function
    function func = get_function(opt_port);
    if (func == NULL){
        fprintf(stderr, "No Function defined...\n");
        return NULL;
    }

    // Create Socket
    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock < 0)
        error_exit ("Socket creation failed\n");
    fprintf(stdout, "Socket created\n");
```

```
// Set Server Connection
memset(&server, 0, sizeof(server));
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port = htons(opt_port);
server.sin_family = AF_INET;
// Set SO_REUSEPORT to allow multithreading on the same socket
int opt_val = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &opt_val,
                sizeof(opt_val));
if (bind(sock, (struct sockaddr*)&server, sizeof(server)) < 0)
    error_exit("Socket bind failed\n");
fprintf(stdout, "Socket bound\n");

for (;;) {
    // Receive up to MMSGLEN messages from the socket
    recv_len = recvmmsg(sock, msgs, MMSGLEN, MSG_WAITFORONE, 0);
    if (recv_len < 0) { // Length negative if recvmmsg failed
        error_exit("recvmmsg() failed\n");
    } else if (recv_len > 0) {
        // Read every packet
        for (i = 0; i < recv_len; i++) {
            // Execute function
            if (!(func)(msgs[i].msg_hdr.msg_iov[0].iov_base,
                        &msgs[i].msg_len, 0)) {
                error_exit("Function failed\n");
            }
            // Set the return message size
            msgs[i].msg_hdr.msg_iov[0].iov_len = msgs[i].msg_len;
        }
        // Respond to the requests (Up to MMSGLEN)
        if (sendmmsg(sock, msgs, recv_len, 0) != recv_len) {
            error_exit("sendmmsg() failed\n");
        }
        // Reset the iov_len to BUFLEN. Was changed to sendmmsg
        for (i = 0; i < recv_len; i++) {
            msgs[i].msg_hdr.msg_iov[0].iov_len = BUFLEN;
        }
    }
}
```

```
    close(sock);
    return NULL;
}
```

5. Concept

This work introduces a function invoker for serverless computing platforms based on AF_XDP. The architecture of the application is chosen with the goal, that load can be distributed across multiple CPU cores and if needed a function can utilize multiple CPU cores to provide scalability. In this section the concept of the application is described and the benefits of using AF_XDP are explained.

AF_XDP relies on two programs. First there is the eBPF program and secondly there is the user space application. There is a connection between both the parts available. Data can be shared between the eBPF program and the user space by eBPF-maps. In case of the application for this work, those maps are used to configure the eBPF program. A new type of maps called XSKMAP has been introduced with AF_XDP. This map contains file descriptors to *XDP sockets* and is shared with the eBPF program. An eBPF program can use multiple XDP sockets to forward packets into. The eBPF program is executed for every incoming packet and forwards the current packet to an XDP socket by calling *bpf_redirect_map()* with an entry of an XSKMAP. Beforehand the user space application needs to allocate a chunk of memory and builds a XDP socket around this memory. The socket file descriptor is shared with the eBPF program through the XSKMAP. The eBPF program can use it and forward packets into it. The user space application can read and write to the socket by reading or writing references to the memory chunk in the RX or TX rings.

Figure 17 shows the message and reference flow between the NIC and the application. This work makes use of multiple AF_XDP sockets. Every AF_XDP socket is only used by requests to a single function. It is possible that multiple AF_XDP sockets are used for the same function. The eBPF and the user space program will be configurable by a few settings during compile time and some configuration can be changed dynamically. With a sound configuration the application should be able to scale worker threads up and down without changing with the eBPF part. Different functions are addressed by different destination ports of the UDP packet.

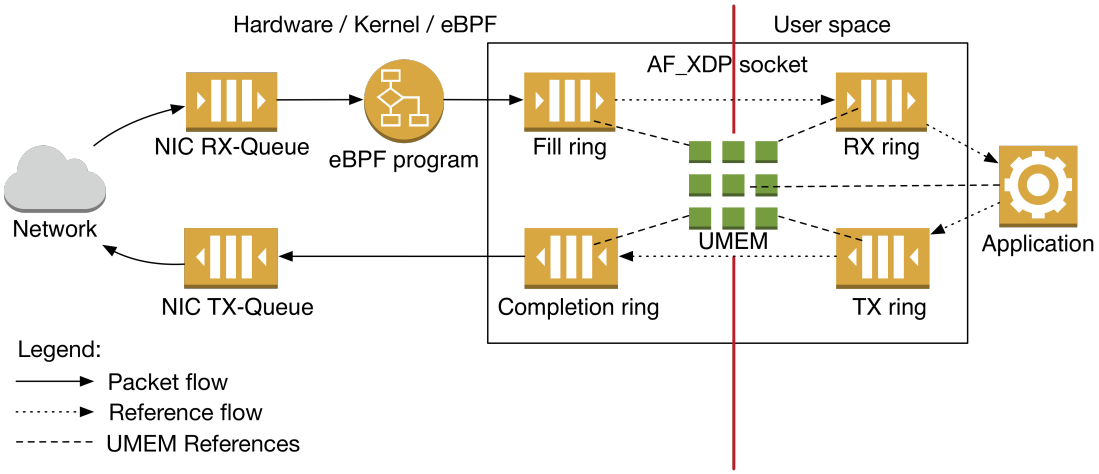

Figure 17: AF_XDP message and reference flow

5.1. eBPF program

The goal of the eBPF program used in this application is to forward the packet to an AF_XDP socket which will later be consumed by a worker thread responsible for the addressed function.

eBPF programs are executed on the same CPU core that is responsible for the interrupt request handling of the RX-queue of the NIC. Using only one RX-queue for the application may result in a bottleneck and therefore non optimal utilization of the system. The eBPF program setup by this work is able to use multiple RX-queues with interrupt request handlers on different CPU cores. An AF_XDP socket can only be bound to a single RX-queue, this leads in multiple sockets for the same function in multi-queue environments. The NIC used by this work does not provide the feature ntuple-filters[65], so every RX-queue might get requests with the same destination port. Therefore every RX-queue needs its own AF_XDP socket for every port. Figure 18 shows an AF_XDP setup with two RX-queues and an application working on the requests to port 1232.

Another possible bottleneck can be the number of AF_XDP sockets available per destination port. If there are CPU intense workloads executed by the function one RX-queue might be sufficient to handle all incoming packets. AF_XDP sockets are

[65]see 3.1.1 Receive Side Scaling, multi-queue NICs and hardware offload

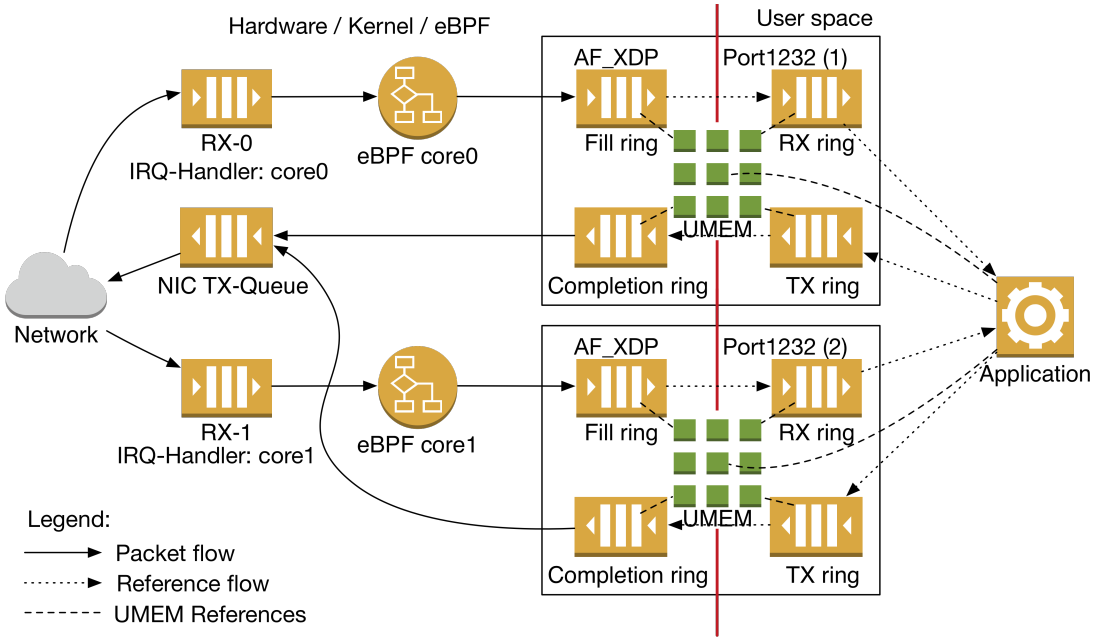

Figure 18: AF_XDP message and reference flow with multiple NIC RX-queues and multiple AF_XDP sockets

restricted to a single consumer. The user space application will work with worker threads responsible for one or more AF_XDP sockets. To maximize the system utilization it should be possible to use multple AF_XDP sockets by a single eBPF instance bound to a RX-queue. The diagram in figure 19 shows this setup.

Both scenarios are combinable resulting in multiple RX-queues working on all the incoming packets of the NIC and each of the cores is able to forward the packet into multiple AF_XDP sockets. Forwarding to multiple AF_XDP sockets will be done by in a round-robin manner and guarantees that the load is balanced between all AF_XDP sockets of that RX-queue. The RSS hash algorithm for selecting the RX-queue might not be balanced for some scenarios. In this case some AF_XDP sockets receive more packets then other. Forwarding requests of the same connection to the same AF_XDP socket is not needed in the serverless computing use case because every request is independent from other requests.

If the server is set to four RX-queues and each eBPF instance has two AF_XDP sockets available for a function there are eight AF_XDP sockets defined for the function. A server can provide multiple functions. The number of RX-queues is

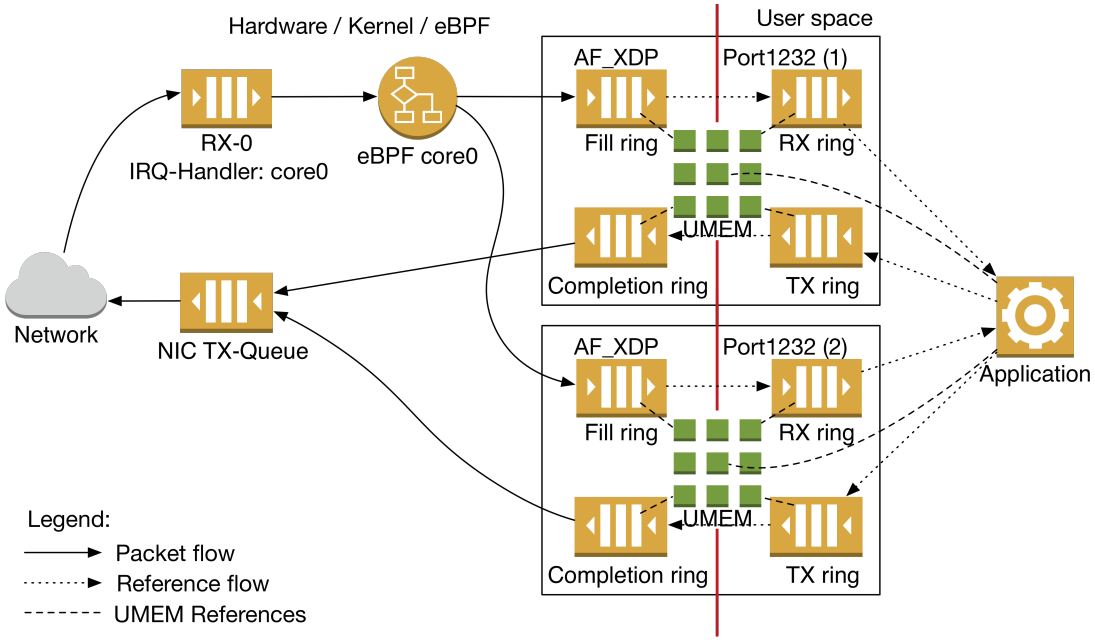

Figure 19: AF_XDP setup with multiple sockets used by a single RX-queue

the same for all functions but the number of AF_XDP sockets per RX-queue is configurable by the user space application via a map per function and queue. This settings result a large variety of different setups. On the other side of the AF_XDP socket the user space application can influence this setup by adding more AF_XDP sockets to the eBPF instances and is able to react on higher or lower request rates to scale up or down.

5.2. User space application

The user space application is responsible for creating and configuring AF_XDP sockets, reading requests from those AF_XDP sockets, invoking a function on each request and sending the result back to the origin. This is a very basic implementation of a function invoker and is used as a prototype in this work.

As described in the concept of the eBPF program in the previous subsection the user space application has the ability to interact with the eBPF program and react on utilization changes by scaling the number of AF_XDP sockets up or down. The components of the user space application are shown in figure 20 on page 51. In

this scenario there are two instances of AF_XDP sockets, whereby each is attached to port 1232. These AF_XDP sockets have been created and configured by the management thread. The management thread also started two worker threads which are each working on one of the AF_XDP sockets. When a worker thread is started, it requests a function pointer of the function to execute from the function repository. In this case the function repository is just a header file containing functions and a method *get_function(port)* which returns a function pointer to the given port. The function repository has already been defined in listing 8 on page 36. The same function was used by the Linux socket implementation and will be used again for the AF_XDP implementation. A worker thread can be configured to consume multiple AF_XDP sockets. The sockets may also vary in the function defined for them.

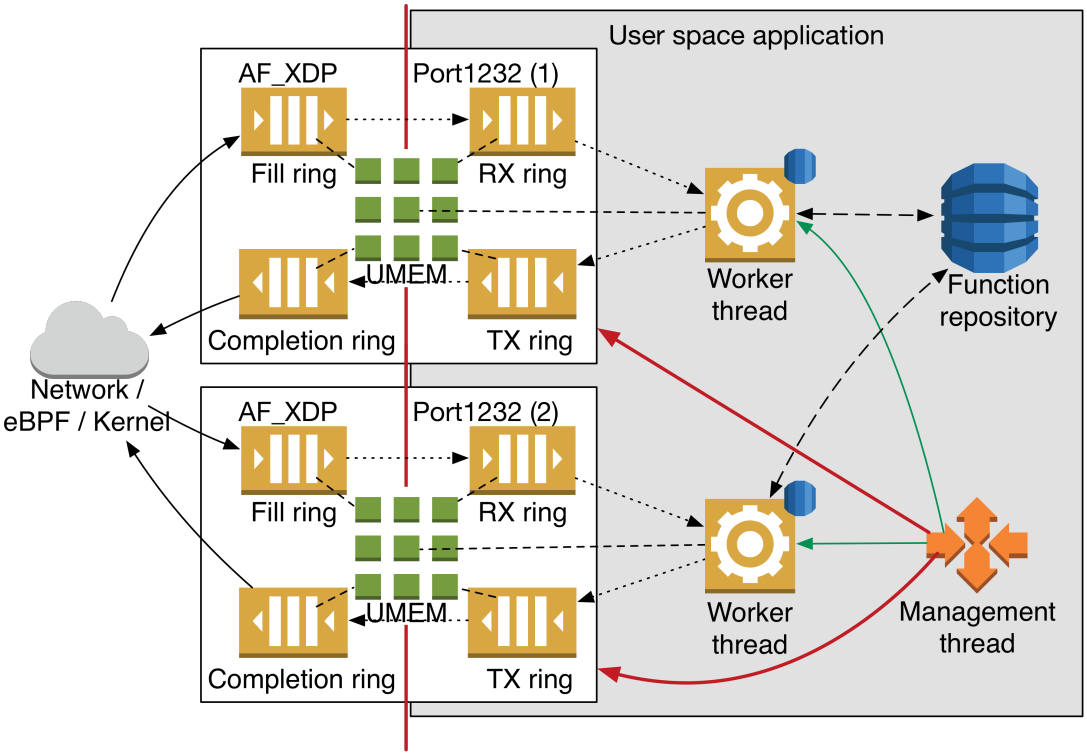

Figure 20: User space application with two worker threads working on two AF_XDP sockets for port 1232

The user space application needs to know how many RX-queues are currently configured on the NIC. The NIC used by this work cannot steer packets based on the destination port[66], packets to every function may arrive at every RX-queue and thus

[66]Feature ntuple-filter is not available on the NIC. See section 3.1.1 (p. 20)

at every eBPF instance. AF_XDP cannot share sockets over multiple RX-queues. Thus, the user space application has to configure one AF_XDP socket per RX-queue for every function. If one AF_XDP socket per RX-queue is not sufficient for the high request rate for this function it is also possible to configure multiple AF_XDP sockets per RX-queue and function.

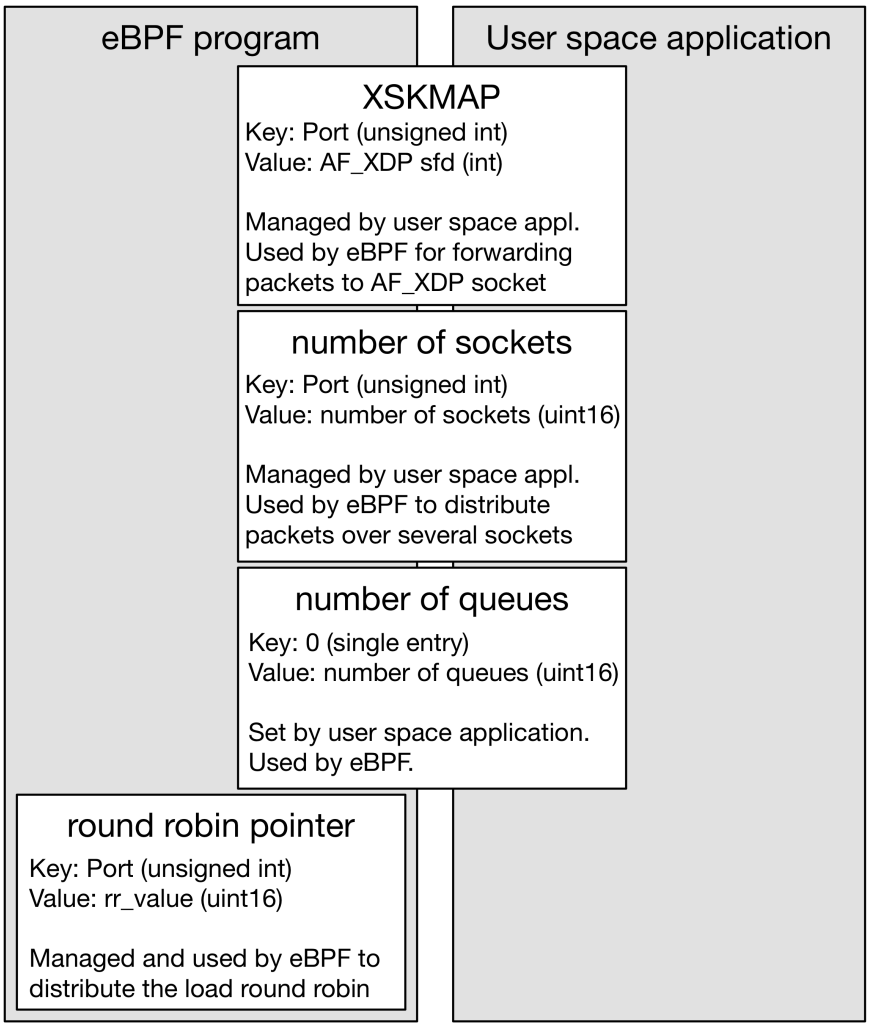

Figure 21: State of the eBPF program and the interaction with the user space application

eBPF-maps are used to handle state and allow the user space application to configure the behaviour of the eBPF program. There are four maps defined. These maps are shown and described in figure 21. The XSKMAP contains all references to AF_XDP socket file descriptors, the number of sockets map is used to tell the eBPF program

how many sockets for a single destination port are available on a queue. The number of queues map is used to inform the eBPF program about the number of queues configured on the NIC and the round robin pointer is used by the eBPF program to iterate over the different AF_XDP sockets available to this queue. The round robin pointer map is needed to store the current state between invocations of the eBPF program.

This chapter described the concept of the application designed for this work. The next chapter contains the implementation of this concept.

6. Implementation and Validation

This chapter describes the implementation of the concept defined in the last chapter. The implementation is based on the AF_XDP sample application which can be found in the kernel source tree at samples/bpf/[67] and consists of the three files *xdpsock.h*, *xdpsock_kern.c* and *xdpsock_user.c*. The header file contains general configuration, the file xdpsock_kern.c contains the eBPF program and the file xdpsock_user.c contains the user space application.

The naming convention will stay the same but for every file the xdpsock is replaced by reqrouter for this work. The header file is included by the eBPF program and the user space application and contains configuration valid for both of them.

6.1. Common parts

The header file contains the general configuration needed by both programs. First there is a configuration option named MAX_SOCKS which defines how many instances of AF_XDP sockets are able to be used by the same function over all. This sums up all instances over all RX-queue and if it should be possible to use multiple sockets per RX-queue this needs to be multiplied with the number of RX-queues.

Since every instance has its own AF_XDP socket file descriptor entered into the XSKMAP there are several entries used for the same function. If the MAX_SOCKS configuration is set to 16, only every sixteenth port can be used by the application. The next 15 entries in the XSKMAP are used for up to 15 more instances of AF_XDP socket file descriptors.

The eBPF program is invoked for every incoming packet on the NIC. A port range is defined in the header by the definitions of PORT_RANGE_LOWER and PORT_RANGE_UPPER. This port range is used by the eBPF program to determine if it is responsible for this packet or if the packet should be forwarded to the TCP/IP stack. If every packet would be handled by the program and user space application, the application would need to respond to Address Resolution Protocol[68] to allow connected devices to lookup the MAC-Address via an Address Resolution

[67] `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/samples/bpf?h=v4.18.6` (downloaded on 2018-10-11)
[68] `https://tools.ietf.org/html/rfc826` (downloaded on 2018-10-11)

Protocol requests.

Listing 11: Settings for the implementation

```
/* Power-of-2 number of sockets per function */
#define MAX_SOCKS 16

/* Port Range for the requestrouter */
#define PORT_RANGE_LOWER 1200
#define PORT_RANGE_UPPER 16000
```

6.2. eBPF program

The eBPF program contains the map definitions. The maps are used to handle state between invocations of the eBPF program. The user space application is able to write or read from the map and configure the eBPF program this way. An example configuration for 5 active sockets over 2 queues on port 1232 is shown in figure 22 on the next page.

Listing 12: Maps configured in the eBPF program

```
struct bpf_map_def SEC("maps") xsks_map = {
    .type = BPF_MAP_TYPE_XSKMAP,
    .key_size = sizeof(unsigned int),    // port
    .value_size = sizeof(int),           // AF_XDP sfd
    .max_entries = PORT_RANGE_UPPER,
};

struct bpf_map_def SEC("maps") num_socks_map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(unsigned int),    // port
    .value_size = sizeof(uint16_t),      // num_socks per queue
    .max_entries = PORT_RANGE_UPPER,
};

struct bpf_map_def SEC("maps") num_queues_map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(unsigned int),    // 0 (single entry)
    .value_size = sizeof(uint16_t),      // num_queues of NIC
    .max_entries = 1,
```

```
};

struct bpf_map_def SEC("maps") rr_map = {
    .type = BPF_MAP_TYPE_PERCPU_ARRAY,
    .key_size = sizeof(unsigned int),    // port
    .value_size = sizeof(uint16_t),      // current rr-pointer
    .max_entries = PORT_RANGE_UPPER,
};
```

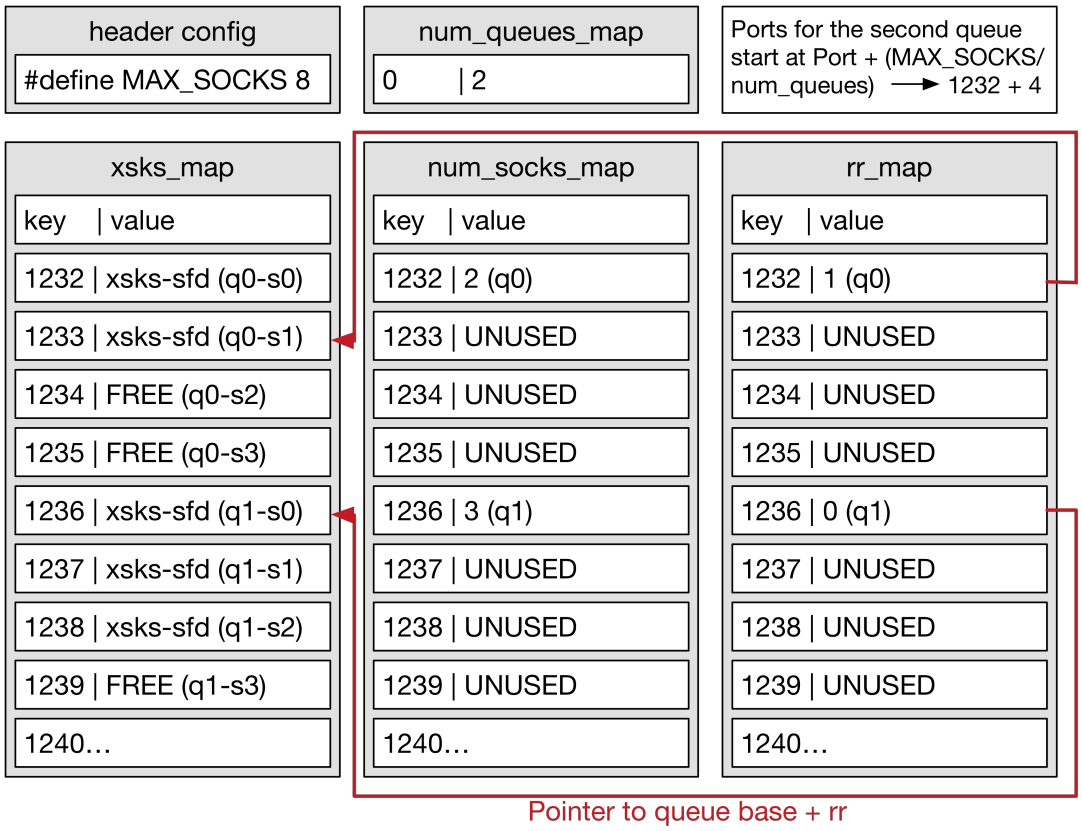

Figure 22: eBPF map data example for 5 configured sockets on a port 1232

The eBPF program shall forward packets based on the destination port of UDP packets. Since the eBPF program is invoked before the packet has been processed by the kernel, there is no other way to get the destination port than parsing the packet headers one by one and have a look into the payload. This implementation is a prototype and cannot handle other packets than UDP over IPv4 over Ethernet. The function used to parse the UDP port from the packet is defined in listing 13.

Listing 13: Function in the eBPF program used to get the destination port of an UDP packet

```
/* Parse the ETH, IP and UDP header and match if the packet should
 * be processed by the RequestRouter
 *
 * returns 0 if the Packet could not be processed, the Port
 * number otherwise
 */
static __always_inline
uint16_t parse_header(struct xdp_md *ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    struct ethhdr *ethh = data;

    u64 offset = sizeof(*ethh);
    /* Check if the Packet contains a whole ethhdr */
    if ((void *)ethh + offset > data_end)
        return 0;

    u16 eth_type = ntohs(ethh->h_proto);
    /* Skip Ethernet II Packets. Focus is on 802.3 packets */
    if (eth_type < ETH_P_802_3_MIN)
        return 0;

    /* Todo: 802.1q, 802.1ad, QinQ etc */

    /* Chedk if it is a IPv4 Packet */
    if (eth_type != ETH_P_IP)
        return 0;

    struct iphdr *iph = data + offset;
    /* Check if the Packet contains a whole iphdr */
    if ((void *)iph + sizeof(*iph) > data_end)
        return 0;

    /* Focus only on UDP Packets */
    if (iph->protocol != IPPROTO_UDP)
        return 0;
```

```
    offset += sizeof(*iph);
    struct udphdr *udph = data + offset;
    /* Check if the packet contains a whole udphdr */
    if ((void *)udph + sizeof(*udph) > data_end)
        return 0;

    u16 dport = ntohs(udph->dest);
    /* Check if the port is in the designated Port range */
    if (dport < PORT_RANGE_LOWER || dport > PORT_RANGE_UPPER)
        return 0;
    return dport;
}
```

At last the entry point of the eBPF program is defined in the following listing. The function is executed for every incoming packet. At first the packet is parsed by the already defined function which returns the destination port of the UDP packet. If the port is zero, the packet is passed to the the Linux TCP/IP stack for further processing. This is done by returning XDP_PASS. Afterwards the port is standardized to a multiple of MAX_SOCKS, since it is only possible to use every MAX_SOCKS port by the application. The next MAX_SOCKS - 1 ports are used for more AF_XDP sockets which are also responsible for the same port.

If there are multiple RX-queues configured, each eBPF program can only forward packets to a subset of AF_XDP sockets because an AF_XDP socket can only be accessed by a single queue. Therefore an offset is set to the current RX-queue index times the maximum number of sockets per queue. If there are multiple sockets defined for this port and queue, the forwarding target is selected round robin based. This allows the load to distribute across the AF_XDP sockets. The offset is changed accordingly. Finally the packet is forwarded to the AF_XDP socket placed at port + queue offset + round robin offset in the XSKSMAP.

Listing 14: eBPF entry point function. Executed for every packet and forwards packet to an AF_XDP socket if possible

```
SEC("xdp_requestrouter")
int xdp_sock_prog(struct xdp_md *ctx)
{
    unsigned int port = parse_header(ctx), offset = 0;
    uint16_t *rr, *num_socks, *num_queues;
```

```
    /* Skip Packet if Port is 0 */
    if (port == 0)
        return XDP_PASS;

    /* Reduce on MAX_SOCKS */
    port = port - (port & (MAX_SOCKS - 1));

    /* Check how many queues exist */
    num_queues = bpf_map_lookup_elem(&num_queues_map, &offset);
    if (!num_queues)
        return XDP_ABORTED;
    if (*num_queues > 1) {
        offset = MAX_SOCKS / *num_queues;
        offset = offset * ctx->rx_queue_index;
        port = port + offset;
    }

    /* Check if multiple sockets for the port exist */
    num_socks = bpf_map_lookup_elem(&num_socks_map, &port);
    if (!num_socks)
        return XDP_ABORTED;
    if (*num_socks > 1) {
        /* Add Round-Robin Value to port */
        rr = bpf_map_lookup_elem(&rr_map, &port);
        if (!rr)
            return XDP_ABORTED;
        *rr = (*rr + 1) % *num_socks;
        port = port + *rr;
    }

    /* Forward Packet to xsks Socket */
    return bpf_redirect_map(&xsks_map, port, 0);
}
```

6.3. User space application

The counterpart of the AF_XDP socket filled with packets by the eBPF program is the user space application. This application sets up the AF_XDP sockets, loads the eBPF program into the kernel and configures the eBPF program via shared maps. Additionally the user space application consumes incoming data from the AF_XDP socket, invokes the function on the payload and transmits the modified packet via the AF_XDP socket back to the origin.

6.3.1. AF_XDP socket creation, RX and TX functions

The user space application is based on the AF_XDP sample application from the Linux kernel source code repository.[69] This sample application already provided some wrapper functions to setup the AF_XDP socket and to send or receive packets. This functions are also used in the implementation of the user space application.

Some properties of the AF_XDP socket and the wrapper functions are configurable via definitions at the head of the source file. For the AF_XDP socket the number of frames allocated for the UMEM and the frame size itself is configurable. Furthermore the number of descriptors used for the Fill- and Completition queue as well as the RX- and TX queues are configurable.

The creation and configuration of an AF_XDP socket is as follows. First a socket with the protocol familiy PF_XDP and socket type SOCK_RAW is created. Then the UMEM is configured by allocating memory for the frames and registered to the socket by a setsockopt call of type XDP_UMEM_REG. The Fill- and Completition queue size is set by using setsockopt system calls of type XDP_UMEM_FILL_RING and XDP_UMEM_COMPLETION_RING. Afterwards the offsets for the memory mapping of the Fill- and Completition queue is retrieved by a getsockopt XDP_MMAP_OFFSETS call and the application maps those queues into its address space by a mmap system call. A similar procedure is executed to configure the RX and TX rings of the socket.

An AF_XDP socket binds to an address structure of type sockaddr_xdp. This structure contains the interface and queue configuration as well as a flag field which contains the XDP_COPY value for the implementation in this work. The copy

[69] Further information on page 6

mode is used because the NIC does not provide the XDP-DRV mode and thus the eBPF program is invoked when the packet arrives in the kernel data path. With the XDP mode enabled, the eBPF program is executed on the driver level and the packet does not need to be copied. Finally the socket is bound to the XDP sockaddr via a bind system call.

To send or receive data via an AF_XDP socket, the application has to use the RX and TX rings which are set up during the UMEM configuration. Incoming packets are available to the application if the RX ring producer is larger than the consumer. The application can consume a descriptor from the RX ring and read the data from the UMEM with the offset provided by the descriptor. If the application wants to send data, it has to store a descriptor at the position of the producer pointer in the TX ring. The descriptor itself points to the frame to send in the umem. The producer pointer is increased and sendto on the socket file descriptor is called to inform the kernel about packets to send.

All this is done by some wrapper functions which are already defined in the sample application and were only modified slightly to remove unused functionality. The definitions of these wrapper functions and the settings can be found in the appendix in listing 21 on page 88. Listing 15 contains the declarations of the functions used in the code of this section. One can refer to them to have a better understanding of the following code.

Listing 15: AF_XDP socket creation and RX / TX function declarations

```
// Create an AF_XDP socket (incl. umem, rings, bind)
// umem: if provided use the same umem and don't create a new one
// queue: RX queue id the socket is bound to
static struct xdpsock *xsk_configure(struct xdp_umem *umem,
            int queue);

// Dequeue descriptors from the RX queue
// descs: descs to store the pointer into
// ndescs: number of descriptors.
// Returns the number of descriptors dequeued
static inline int xq_deq(struct xdp_uqueue *uq,
             struct xdp_desc *descs,
             int ndescs);
```

```
// Enqueue descriptors to the TX queue
// returns the number of descriptors enqueued
static inline int xq_enq(struct xdp_uqueue *uq,
             const struct xdp_desc *descs,
             unsigned int ndescs);

// Get a pointer to a umem frame (packet data)
static inline void *xq_get_data(struct xdpsock *xsk, u64 addr);

// Send the data of the TX queue. Uses send() system call
static inline void complete_tx(struct xdpsock *xsk);
```

6.3.2. Deploy eBPF program and access maps

The eBPF program defined in this section is responsible for forwarding the data to the AF_XDP sockets. Therefore it has to be loaded into the eBPF pseudo-machine and attached to the XDP handler. Loading is done with a single bpf_prog_load_xattr call to the bpf library with the attributes proc_type and file set accordingly. Now the eBPF program can be attached to the interface with by calling bpf_set_link_xdp_fd.

The bpf_prog_load_xattr call creates an bpf object which can be used to access the maps of the eBPF program. To access a map from the user space application, the map object is identified by the name of the map. A map entry can be updated by a call to bpf_map_update_elem with the map object, the key and the value. Listing 16 shows how the eBPF program is loaded and a map entry is set afterwards. The application sets multiple queue entries to get the socket running, but all the entries are changed in this way.

Listing 16: eBPF program load and map access

```
// Initiate the load atrributes
struct bpf_prog_load_attr prog_load_attr = {
    .prog_type  = BPF_PROG_TYPE_XDP,
    .file = "reqrouter_kern.o",
};

// load the eBPF program
int prog_fd;
struct bpf_object *obj;
```

```
if (bpf_prog_load_xattr(&prog_load_attr, &obj, &prog_fd))
    exit(EXIT_FAILURE);
if (prog_fd < 0) {
    fprintf(stderr, "ERROR: no program found: %s\n",
        strerror(prog_fd));
    exit(EXIT_FAILURE);
}

// attach the eBPF program to the interface
if (bpf_set_link_xdp_fd(opt_ifindex, prog_fd, opt_xdp_flags) < 0) {
    fprintf(stderr, "ERROR: link set xdp fd failed\n");
    exit(EXIT_FAILURE);
}

// access the map
struct bpf_map *map;
int num_queues_map;
map = bpf_object__find_map_by_name(obj, "num_queues_map");
num_queues_map = bpf_map__fd(map);
if (rr_map < 0) {
    fprintf(stderr, "ERROR: no num_queues_map found: %s\n",
        strerror(num_queues_map));
    exit(EXIT_FAILURE);
}

// update a map entry
int key = 0, value = 2;
ret = bpf_map_update_elem(num_queues_map, &key, &opt_queues, 0);
if (ret) {
    fprintf(stderr, "Error: bpf_map_update_elem\n");
    exit(EXIT_FAILURE);
}
```

6.3.3. Function invoker

The next application part is the function invoker thread, which is used as a worker thread. Each thread consumes data from one or more AF_XDP sockets, invokes the function on the payload and sends the modified packet back to the origin.

The application has to work with raw packet data. This includes the Ethernet, IP and UDP header in the environment of this work but may contain more or other header in other environments. To send a response to the origin of the packet the source and destination entries on each header have to be swapped. Additionally the UDP checksum has to be recalculated or at least cleared if the data is changed. The UDP checksum is calculated over the UDP header fields, a pseudo IP header field and the data.[70] For this implementation the checksum is cleared. If the packet length changes, the length fields in the IP and UDP header have to be changed. The checksum of the IP header includes the length field in the calculation and has to be recalculated when the length changes. This implementation doesn't change the length of the packet. The checksum has not to be recalculated. The functionality is integrated in the function *swap_header* shown in listing 17.

Listing 17: swap_header function definition

```
static bool swap_header(void *data, u64 l)
{
    // header length = sizeof eth + ip + udp hdr
    if (l < header_length) {
        return false;
    }
    //Eth-Header (MAC Adresses)
    struct ether_header *eth = (struct ether_header *)data;
    struct ether_addr *src_addr = (struct ether_addr *)&eth->ether_shost;
    struct ether_addr *dst_addr = (struct ether_addr *)&eth->ether_dhost;
    struct ether_addr eth_tmp;

    eth_tmp = *src_addr;
    *src_addr = *dst_addr;
    *dst_addr = eth_tmp;

    u64 offset = sizeof(*eth);

    //IP-Header (IP-Adresses)
    struct iphdr *iph = (struct iphdr *)(data + offset);
    u32 ip_tmp = iph->saddr;
    iph->saddr = iph->daddr;
    iph->daddr = ip_tmp;
```

[70] `https://tools.ietf.org/html/rfc768` (downloaded on 2018-10-11)

```
    //Checksum stays the same. Change of length requires recalculation
    offset += sizeof(*iph);

    //UDP-Header (Ports)
    struct udphdr *udph = (struct udphdr *)(data + offset);
    u16 udp_tmp = udph->source;
    udph->source = udph->dest;
    udph->dest = udp_tmp;
    //Clear the checksum
    udph->check = 0;

    return true;
}
```

Each worker thread reads from one or more sockets and invokes the function for each request. This allows the management part of the application to make decisions of putting sockets with less requests together on one thread.

For every socket it is tried to read data. If no data was read by the xq_deq function, the next socket is handled. The dequeuing is done in batch mode. Similar to the multi message send and receive calls it is possible to dequeue up to BATCH_SIZE of descriptors from the RX queue. For every packet a data pointer to the associated frame in the UMEM is calculated and the headers are swapped. Then the function is executed on this packet data. Since the packet contains the header information we have to tell the function how much data belongs to the headers.

Afterwards the descriptors are enqueued to the TX queue and the send mechanism is called.

Listing 18: Worker thread function

```
struct sock_port{
    struct xdpsock **xsks;
    int *ports;
    int length;
    int id;
};
```

```
static void * requestHandler(void *arg)
{
    int ret = 0;
    struct sock_port *sp = (struct sock_port*) arg;
    struct xdp_desc descs[BATCH_SIZE];
    unsigned int rcvd, i, l;
    char *pkt = NULL;
    function func[sp->length];
    for (l = 0; l < sp->length; l++) {
        func[l] = get_function(sp->ports[l]);
        if (func == NULL) {
            fprintf(stderr, "No Function defined...\n");
            return NULL;
        }
    }

    for (;;) {
        for (l = 0; l < sp->length; l++) {
            rcvd = xq_deq(&sp->xsks[l]->rx, descs, BATCH_SIZE);
            if (rcvd == 0)
                continue;

            // Execute the function for every packet
            for (i = 0; i < rcvd; i++) {
                pkt = xq_get_data(sp->xsks[l], descs[i].addr);

                // Swap ETH, IP and UDP header
                if (!swap_header(pkt, descs[i].len)) {
                    fprintf(stderr, "Port %d: Header to short\n",
                        sp->ports[l]);
                    continue;
                }

                if (!(*func[l])(pkt, &descs[i].len, header_length)) {
                    fprintf(stderr, "Port %d: Function failed\n",
                        sp->ports[l]);
                    continue;
                } // Todo: Calculate checksum if length changed
            }
```

```
                // Back to the kernel by TX
                ret = xq_enq(&sp->xsks[l]->tx, descs, rcvd);
                lassert(ret == 0);
                sp->xsks[l]->outstanding_tx += rcvd;
                // Complete the TX
                complete_tx(sp->xsks[l]);
            }
        }
        free(sp->xsks);
        free(sp->ports);
        free(sp);
}
```

6.3.4. Thread and socket management

Finally the last part of the user space application is implemented. The management part of the application is able to increase or reduce the number of AF_XDP sockets per queue and to increase or reduce the number of worker threads according to the utilization of the different functions requests. Due to the flexible design the application should be able to react to utilization changes. This work focuses on the packet processing performance and doesn't need this scalability. Therefore up- and downscaling implementation is left for future work.

For this work it is sufficient to be able to create sockets over multiple queues and to be able to create multiple sockets on one queue. There are two application parameters available. First the opt_queues parameter is set by -q with the program execution and there is tie opt_threads parameter (-t) which defines how many sockets and threads should be started per queue. Listing 19 contains the create_socket function which sets up an AF_XDP socket at the correct place to insert it into the xsks_map. The listing also contains an extract of the main function which sets up the queues based on the parameters.

Listing 19: Thread and socket management

```
static int create_socket(int port, int queue, int thread)
{
    int offset = MAX_SOCKS / opt_queues;
    offset = offset * queue;
    offset = offset + thread;
    offset = offset + port;
    // Check if offset is valid
    if (offset > PORT_RANGE_UPPER || offset < PORT_RANGE_LOWER ||
        offset >= port + MAX_SOCKS)
        return -1;
    // Create socket at queue
    xsks[offset] = xsk_configure(NULL, queue);
    return offset;
}

int main(int argc, char **argv)
{
    // Program load, maps linked, etc omitted
    /* Create sockets... */
    for (p = 0; p < len_ports; p++) { // ports[p] -> port
        for (q = 0; q < opt_queues; q++) { // q -> queue
            for (t = 0; t < opt_threads; t++) { // t -> thread
                pqt = create_socket(ports[p], q, t);
                if ( pqt < 0 )
                    exit(EXIT_FAILURE); // Socket creation failed
                ret = bpf_map_update_elem(xsks_map, &pqt,
                    &xsks[pqt]->sfd, 0);
                if (ret)
                    exit(EXIT_FAILURE); // BPF map update failed

                // Configure and start the consumer thread
                struct sock_port *sp = malloc(sizeof(struct sock_port));
                (*sp).length = 1;
                (*sp).xsks = malloc(sizeof(struct xdpsock *) *
                    (*sp).length);
                (*sp).xsks[0] = xsks[pqt];
                (*sp).ports = malloc(sizeof(int) * (*sp).length);
                (*sp).ports[0] = ports[p];
```

```
                (*sp).id = pqt;
                pthread_create(&pt[pqt], NULL, requestHandler, sp);

                if (t == 0) {
                    // Set the number of threads per queue
                    ret = bpf_map_update_elem(num_socks_map, &pqt,
                        &opt_threads, 0);
                    if (ret)
                        exit(EXIT_FAILURE); // BPF map update failed
                }
            }
        }
        fprintf(stdout, "Started %d Threads for Port %d\n",
            opt_threads * opt_queues, ports[p]);
    }
    // Set the number of queues
    ret = bpf_map_update_elem(num_queues_map, &key, &opt_queues, 0);
    if (ret)
        exit(EXIT_FAILURE); // BPF map update failed
}
```

6.4. Issues during implementation

During the implementation of the eBPF program and the user space application some issues occurred. These issues are described here.

When the work on the topic was started, Linux kernel 4.18 was still under development. It was revealed that under some conditions the polling interface, which AF_XDP provides, had a bug. If there was a packet received by the the AF_XDP socket once, the poll would always return a successful POLLIN for that socket. The bug was resolved and the fixed version was released with Linux kernel v4.18.[71]

Another issue was detected with multiple AF_XDP sockets using the same UMEM and therefore the same fill and completion queues. Under high load with many packets per second the throughput would cut down to a quarter or less of the throughput

[71] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/net/xdp/xsk_queue.h?id=d24458e43b103c7eb7b2fd57bcac392fd7750438` (downloaded on 2018-10-11)

prior to this. If the UMEM space was increased, the throughput cut could be postponed by a few minutes but would eventually occur again. This issue was omitted by having all AF_XDP sockets use their own umem. Further tests regarding this issue will be executed after this work is finished.

While using the eBPF maps, a not well documented function took some time to figure out. The map with type BPF_MAP_TYPE_PERCPU_ARRAY is a map type which has an entry for every CPU core available. The implementation first used this map type and showed strange behaviour. The behaviour was caused by map entries not available to all CPU cores. By calling bpf_map_update_elem to update such map, only the map for the first CPU is updated. Thus other queues which are handled on other CPU cores were misconfigured. To mitigate this, the maps which are configured by the user space application and only read by the eBPF program were changed to map type BPF_MAP_TYPE_ARRAY. This map type only provides one instance for all CPU cores.

Last but not least the implementation runs into problems with polling enabled under high load. Eventually one or more AF_XDP sockets run into timeouts while polling even if packets arrive on the line. This problem did not affect the results of this work because the results were measured without polling active. I contacted the AF_XDP maintainers on this issue and will look into it after the work is finished. A similar problem was detected when using multiple AF_XDP sockets per queue. After some time one or more threads stop receiving data. Tests with multiple sockets per queue were not able to be evaluated for this reason.

7. Evaluation

Three different implementations have been developed in this work. All three are going to be evaluated in regards of I/O performance. First the tests are defined in the next subsection. Afterwards the throughput evaluation is executed and the chapter finishes with the latency evaluation.

7.1. Test execution

The applications under evaluation have to execute a throughput test, a functional test and a latency test while a monitoring system collects the measurements of the network input and output and the test application metrics. The two test systems[72] are directly connected with via 10 Gbit/s network cards. The setup is shown in figure 2 on page 5. One system runs the implementation under test and the other system executes the tests defined as follows.

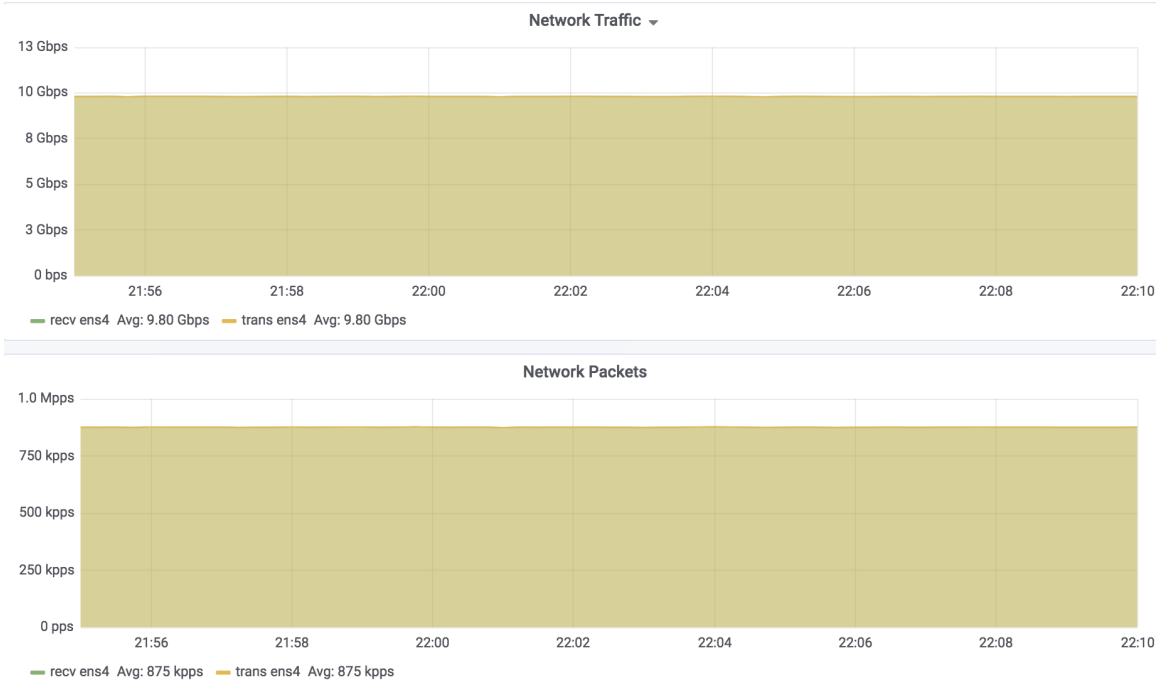

Figure 23: Throughput test results for a test with packet size 1400

The throughput test is executed with different packet sizes of 200, 400, 600, 800,

[72] As described in table 1 on page 6

1000, 1200 and 1400 bytes.[73] Each throughput test run uses one size of packets and generates the load over a 20 minute timespan. The throughput results are measured over a 15 minute timespan which starts at least 60 seconds after the test run was started. This allows the application to settle down if needed. The evaluation uses the average throughput measured in this timespan. Figure 23 shows the throughput of a test run over a 15 minute period.

Every application implementation described in this work had to perform the throughput tests with all packet sizes. The optimized Linux socket implementation and the AF_XDP implementation also executed the throughput tests with multiple thread and queue combinations. The throughput results of this tests are used to demonstrate how well the system scales by adding more CPU cores and / or RX queues.

The other test is a function and latency test. The test application for this test uses 4 threads to send a packet of 800 bytes to the application under test. Then the application waits for the response of the serverless function invoker. The timespan between sending the request and receiving the response is measured and exported to the monitoring system. The monitoring and graphing system allows to generate graphs with multiple measured response times based on the request percentiles. A 50% percentile value is the response time over a timespan of 15 seconds in which 50% of the answers were received. The percentile values for 50, 90, 99 and 99.9 % of the requests are used in the evaluation.

The test application also checks if the function was executed correctly by the application under test. However, the function never failed and every packet contained the correct payload. The latency test is also executed for 20 minutes and the measurement take place for 15 minutes within the timespan. The function and latency test application source code is attached to this work in the appendix on page 104.

7.2. I/O throughput

Evaluation of the I/O throughput is based on the method described earlier in the previous subsection. Throughput is measured while the application invokes as many function executions as possible and sends the results back. The executed function is reversing the payload of the request. The larger the packet length in the test,

[73] The packet generator achieved a throughput of 5 to 6 Gbit/s for 200 byte sized packets. Larger packets lead to packet generation at line speed.

the more CPU time is consumed by the function execution and not by the network stack. Function source code is listed in listing 8 on page 36.

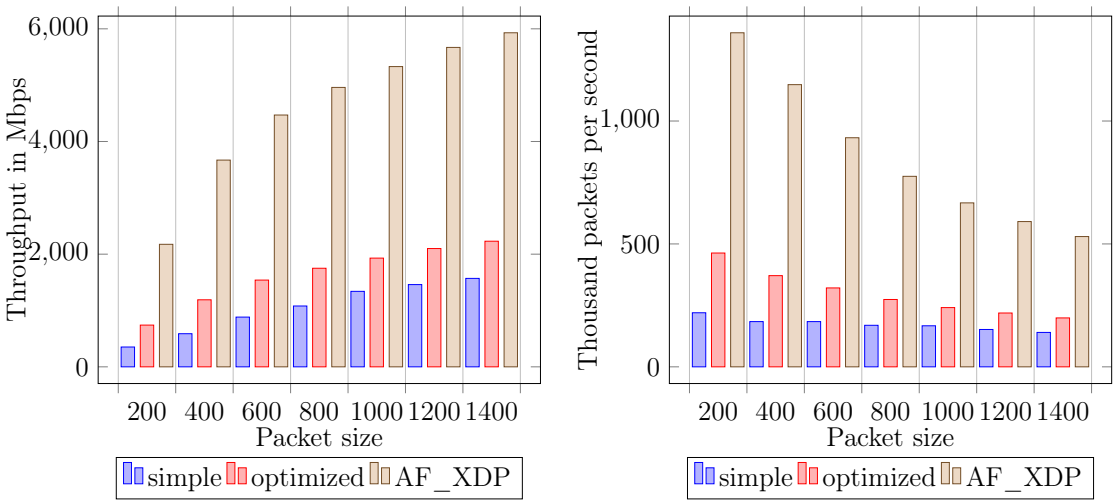

Figure 24: Throughput performance of the simple socket implementation, the optimized socket implementation and the AF_XDP implementation with one thread

At first the single thread performance of the AF_XDP function invoker is compared with the single thread performance of the optimized Linux network stack implementation. The optimized version uses multiple message system calls and optimized RSS settings. The third implementation is the unoptimized Linux socket implementation. Figure 24 shows the throughput of different sized payloads for the three implementations. The throughput of the AF_XDP implementation is 2.9 times the throughput of the optimized Linux socket implementation or 6.1 times the throughput of the unoptimized implementation for small packets with a size of 200 bytes. For packet sized 1400 bytes the throughput is 2.7 times better than the optimized version and 3.8 times better than the unoptimized implementation.

Performance improvements from 2.7 to 2.9 times the throughput for this use case show that the optimized Linux socket implementation does not perform very well in comparison with the AF_XDP implementation. The Linux socket implementation consumes a lot of CPU cycles by copying the data into the sk_buff structure, processing it in the kernel and copying it into a user space buffer.

It has been shown, that the single core performance of the AF_XDP implementation provides better throughput by a factor of 2.7 to 2.9. However, the server was still not able to use the 10 Gbit/s NIC at line speed.

CPU #	Task	CPU #	Task
0	IRQ RX-queue 1	1	application thread 6 + IRQ RX-queue-7
2	IRQ RX-queue 2	3	application thread 5 + IRQ RX-queue-8
4	IRQ RX-queue 3	5	application thread 4
6	IRQ RX-queue 4	7	application thread 3
8	application thread 8 + IRQ RX-queue-5	9	application thread 2
10	application thread 7 + IRQ RX-queue-6	11	application thread 1

Table 5: CPU task allocation in the multi threaded AF_XDP implementation

Next the throughput evaluation of the AF_XDP implementation utilizing multiple cores and queues is performed. The setting is the same as it was already used in the state analysis of the current Linux socket implementation. There are 6 physical CPU cores available in this machine. Thanks to simultaneous multi-threading there are 12 logical CPU usable. The thread placement was also optimized for these tests. Table 5 shows which CPU core does which task.

The results of the scaling test is shown in figure 25 on the next page. A comparison of the throughput results of the one thread and one queue (1T1Q) test run with the results of two threads and two queue (2T2Q) test run shows, that the throughput increased by 83 (packet size 1200) to 94 % (packet size 200). The 2T2Q test run achieves line rate packet processing with packets sized 1400 Bytes.

The scaling from 2T2Q to 4T4Q doesn't increase the throughput that much. This relates to the usage of simultaneous multi-threading. 4 threads and 4 queues are distributed on 8 virtual CPU cores. With only 6 physical cores available some of the physical cores are executing two workloads. The 4T4Q implementation reaches line speed with packet sized 600 bytes and larger. The 8T8Q test run achieves similar throughput rates to the 4T4Q test run.

A throughput test result of a test run with 2 threads working on 1 queue shows the issue described in section 6.4 on page 69. When multiple threads per queue are used eventually one thread does not receive any more data. One can see that the throughput decreases with the 1200 byte test and is exactly half the throughput at the 1400 byte test. By looking at the packet rate per second, one can see, that the

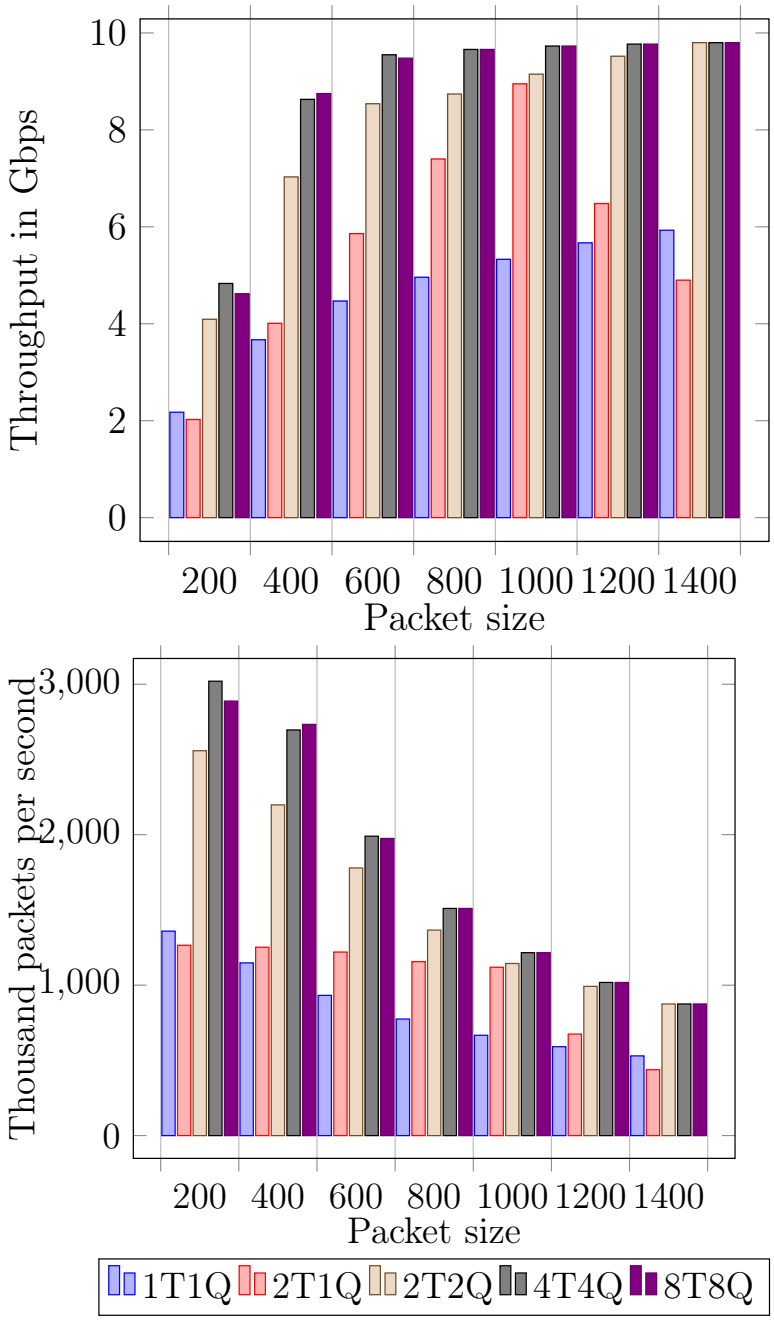

Figure 25: Scaling performance of AF_XDP by utilizing multiple cores

eBPF application can handles larger packet sizes well and only reduces the packets per second throughput marginally. The results stay between 1.26 million packets per second (200 bytes) and 1.14 million packets per second(1000 bytes). Workloads with larger packets benefit from fewer RX-queues and more worker threads.

Table 6 and 7 contain the measurement of the throughput evaluation.

Test & packet size	200	400	600	800	1000	1200	1400
unoptimized	352	588	883	1080	1340	1460	1570
opt. 1T1Q	741	1190	1540	1750	1930	2100	2230
opt. 2T1Q	1409	2360	3030	3430	3790	4060	4280
opt. 2T2Q	1492	2350	3000	3430	3790	4060	4280
opt. 4T2Q	2170	3730	4720	5180	5910	5980	6670
opt. 4T4Q	2334	3700	4690	4800	5280	5470	5950
opt. 8T4Q	2334	3700	4690	4800	5280	5470	5950
XDP 1T1Q	2175	3670	4470	4960	5330	5670	5930
XDP 2T1Q	2025	4010	5860	7400	8950	6480	4900
XDP 2T2Q	4093	7030	8540	8740	9150	9520	9800
XDP 4T4Q	4832	8630	9550	9660	9730	9770	9800
XDP 8T8Q	4620	9060	9480	9660	9730	9770	9800

Table 6: Measured throughput in Mb/s

Test & packet size	200	400	600	800	1000	1200	1400
unoptimized	220	184	184	169	167	152	140
opt. 1T1Q	463	371	321	274	241	219	199
opt. 2T1Q	880	737	630	537	474	422	382
opt. 2T2Q	933	135	630	536	474	423	382
opt. 4T2Q	1356	1164	984	809	739	623	596
opt. 4T4Q	1459	1157	976	749	661	570	531
opt. 8T4Q	1862	1599	1366	1108	1021	931	746
XDP 1T1Q	1359	1148	932	775	667	591	530
XDP 2T1Q	1265	1252	1220	1156	1119	675	438
XDP 2T2Q	2558	2198	1779	1366	1144	992	875
XDP 4T4Q	3020	2696	1990	1510	1216	1018	875
XDP 8T8Q	2888	2830	1975	1510	1216	1018	875

Table 7: Measured throughput in thousand packets per second

7.3. I/O latency

Another important performance value for network services is the latency. In general a client awaits responses for the requests as fast as possible. The latency was tested by using a test application which measures the time between sending the data and receiving the answer. The test application also verifies if the serverless function invoker executed the function correctly by comparing the expected response with the actual response.

Each application had to execute this test for 15 minutes. The graphs in figure 26 on the next page show the latency of the three implementations. Each graph contains the latency for 50%, 90%, 99% and 99.9% of the answers to any given time during the test. The values are calculated over a 15 seconds sliding window.

The results in figure 26 on the next page show that the AF_XDP implementation adds the lowest latency to the round trip time between the hosts. Interestingly, the optimized socket implementation has worse latency then the simple socket implementation. The reason for this is, that the optimizes socket implementation works on bulks of messages. Working on multiple messages improves the throughput of the application but adds more latency to the message flow since the messages are queued at the socket. Especially noticeable is the reduction of the latency in for 50% of the messages with the AF_XDP implementation. The AF_XDP implementation averages with 22 μs, the simple socket takes 30 μs on average the the optimized version averages at 34 μs.

The AF_XDP implementation reduce the latency of the 50% percentile by 26% in comparison to the simple socket implementation and by 35% in comparison to the throughput optimized socket implementation. For the 90% percentile the latency is reduced by 20 and 25%.

The functional tests didn't show any errors in either of the three implementations. All the functions have been invoked and executed correctly.

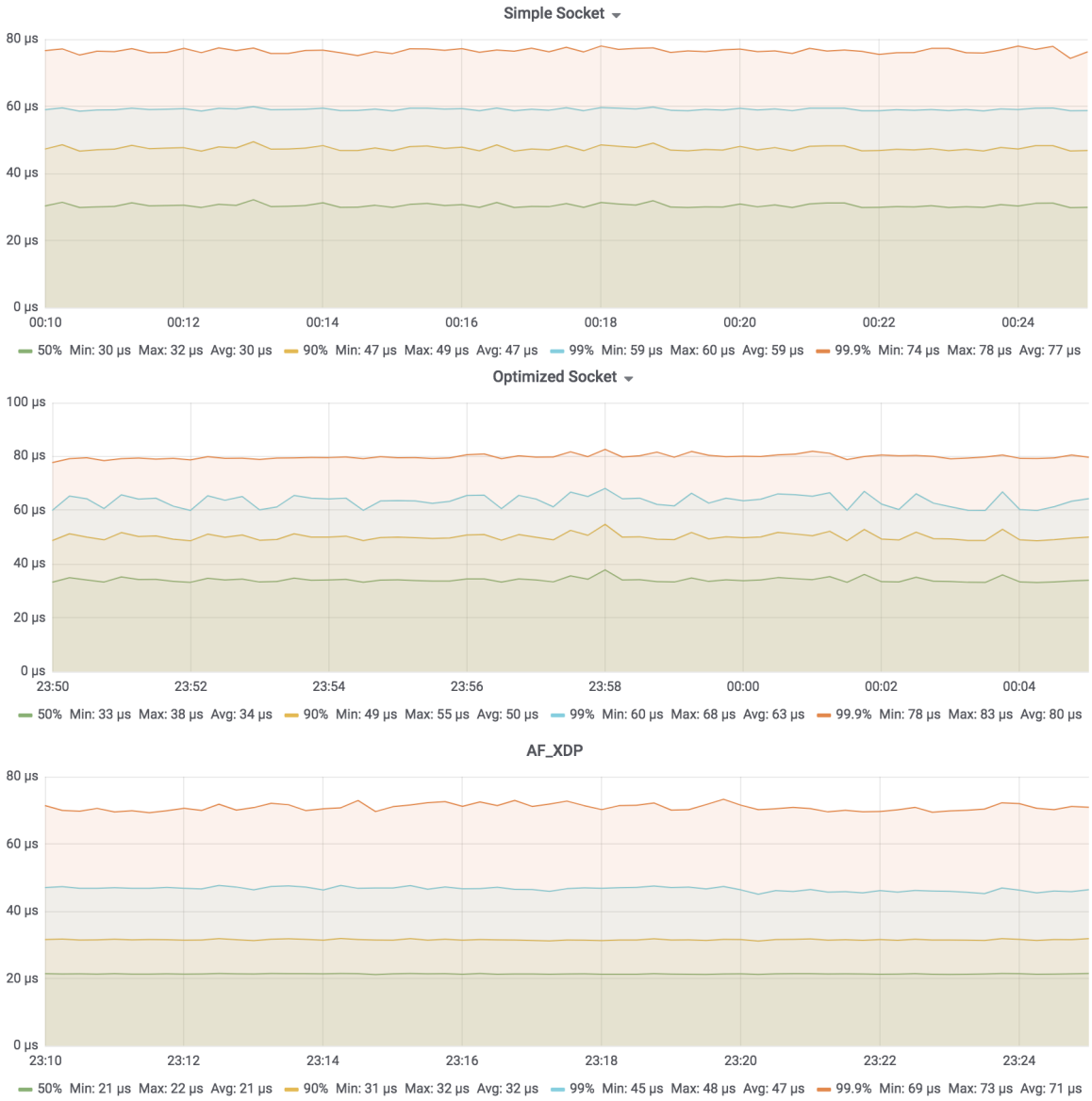

Figure 26: Results of the latency test for the three implementations

8. Conclusion and future work

8.1. Conclusion

Networking and thus packet processing is done by a higher number of devices every year. With the higher number of devices, faster connection speeds and more payload service providers are challenged to provide fast services while remaining profitable. This work was done with the goal of performance improvements in the packet processing of Linux servers, which may lead to fewer servers required for a workload and thus also works more efficient.

First the work started with an introduction to the technologies used by this work. This includes the Linux TCP/IP stack, the eBPF pseudo-machine and the AF_XDP sockets which just arrived in the Linux kernel with version 4.18. AF_XDP claims to be faster than the Linux TCP/IP stack and thus also more efficient in terms of this work.

A Linux socket application has been implemented and state of the art technologies like multi-queue NICs with optimized RSS configuration have been implemented into the Linux socket application. The improved Linux socket implementation doubled the throughput of the unoptimized implementation for some workloads generated. It has been shown that AF_XDP further increases the throughput by a factor of 2.7 to 2.9 times depending on the workload. Additionally by scaling the AF_XDP socket application over multiple cores and RX queues it was possible to execute the workload at line speed of 10 Gbit/s. This was slightly not possible with the Linux socket implementation. Beside the throughput increases the network latency was also reduced by up to 35%. This work has shown that AF_XDP has a high potential for use cases needing higher network performance. A fast overview of the results is presented in figure 27 on the next page.

As eBPF and especially AF_XDP are new technologies there are still some bugs to work on and documentation work to be done. The AF_XDP maintainers were welcoming the feedback which resulted from this work. However the technologies worked for most setups and tests and the results look promising. Especially promising are the the results because the test system didn't provide a network card which is capable of performing XDP in driver mode, which would increase the throughput performance further. Additionally there will be a XDP driver and zero-copy mode

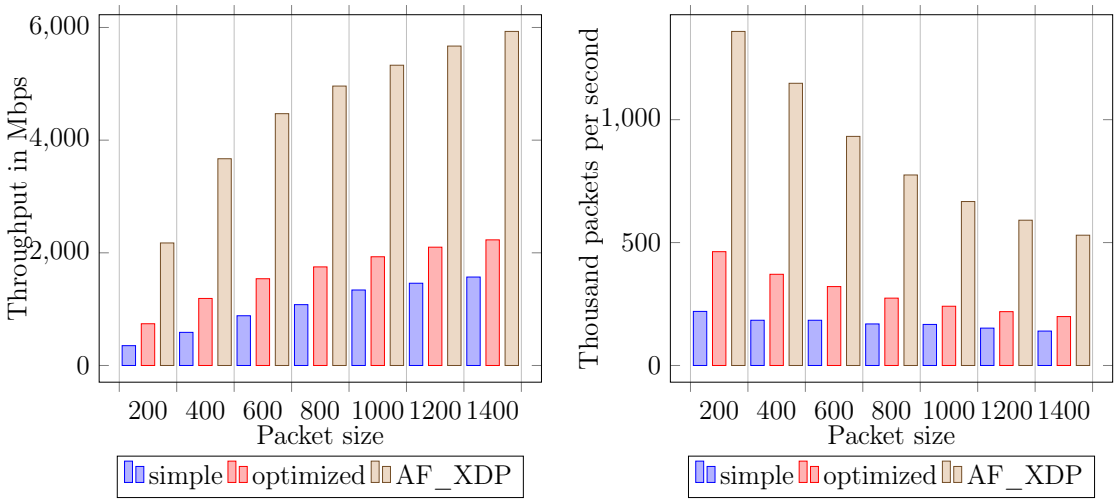

Figure 27: Throughput performance of the three implementations while utilizing a single thread

available for AF_XDP in the future which again increases the performance. Other improvements are expected by utilizing an eBPF enabled SmartNIC which is able to execute the eBPF program on the network card itself.

An eBPF program has access to the whole packet and make forwarding decisions on application layer data before Linux even has a socket buffer initiated for the packet. The programmability of the data layer provides new opportunities to the application integrating it.

AF_XDP bypasses the Linux TCP/IP stack completely and therefore every network function usually provided by the network stack has now to be integrated into the application. Unfortunately it bypasses NIC offload features like UDP or IP checksumming. This application may need to calculate those checksums on CPU time. The development of AF_XDP and eBPF/XDP in general is fairly active, thus new features may introduce UDP and IP checksum offloading in the future.

Controlled environments which doesn't need many features of a general-purpose network stack currently benefit the most from employing AF_XDP. For the network intense workload of a serverless function invoker, which was simulated in this work, the throughput increase by almost three times. If one can cut the number of servers used for a workload down to one third of the servers, the operational costs are reduced. Service providers using such technology may lower their prices and outperform the competitors at the same time.

8.2. Future work

This work focussed on the packet processing in the Linux kernel by employing AF_XDP. In the future new application designs may utilize the eBPF program and a user space application to get the best of both worlds. Applications like this are able to steer the packet on arrival directly into the correct part of the application.

In the meantime it might be useful to develop an API which doesn't differ to much from the POSIX send() and recv() calls. Providing such an API might accelerate the adoption of AF_XDP and improve the performance of some applications with only a few changes. Integrations of AF_XDP to other programming languages is currently an open topic. Currently there are libraries available for C / C++, Rust[74] and Go[75]. Further programming languages like Java or Python might also benefit from a library, which provides the ability to use AF_XDP sockets.

As already stated, features like UDP checksum offloading to the NIC are available when using the Linux TCP/IP stack. It was not possible to use these features from the AF_XDP path. currently. If it would be possible to enable such features via socket options, AF_XDP might benefit a lot from it because the user space applications don't have to calculate those checksums.

Specific use cases which require only a small amount of work may be encapsulated completely in the eBPF program. In terms of a serverless platform as used in this work, this could mean that simple serverless functions are implemented directly into the eBPF program. This will perform even better than an AF_XDP implementation in the user space.

User space network stacks have been a thing for a few years already and are capable of high performance packet processing. This work did not compare the AF_XDP integration to a DPDK integration. A comparison with a user space network stack for such specific use cases is useful. It would be interesting to see in which cases a DPDK integration outperforms AF_XDP in zero-copy mode and vice versa. An application developed for AF_XDP or DPDK depends on these. Portability cannot be guaranteed at the moment. In the future AF_XDP might be available on every Linux server, which improves the portability. Some features like multi-queue pro-

[74] `https://crates.io/crates/ebpf` (downloaded on 2018-10-11)
[75] `https://go.googlesource.com/sys/+/d47a0f3392421c5624713c9a19fe781f651f8a50` (downloaded on 2018-10-11)

cessing or executing eBPF programs in XDP driver mode might never be available at all servers.

The test systems used did not allow the usage of XDP driver mode for the AF_XDP packet processor. With new features like AF_XDP zero-copy on the way it would be interesting to have a performance comparison of those three modes. The eBPF program of the packet processor was built to work with multiple queues where every queue may receive packets for every port. A NIC which support ntuple-filters can be used to reduce the amount of computing work in the eBPF program. A filter on the destination port for UDP packets would omit the need of parsing the UDP header in the eBPF program.

As one can see there is still a lot of tasks to look into in the field of AF_XDP, eBPF, DPDK, NICs and SmartNICs. The source code of this work is openly available and can be used by others for future work. `https://github.com/gerl1ng/afxdp-packet-processor`

A. List of abbreviations

API Applicatin Programming Interface
BaaS Backend as a Service
BPF Berkeley Packet Filter
cBPF classical Berkeley Packet Filter
DMA Direct Memory Access
DPDK Data Plane Development Kit
eBPF Enhanced Berkeley Packet Filter
FaaS Functions as a Service
IaaS Infrastructure as a Service
IPv4 Internet Protocol Version 4
IPv6 Internet Protocol Version 6
ISA Intelligent Server Adapter
JIT Just-in-Time
MTU Maximum Transmission Unit
NIC Network Interface Card
PaaS Platform as a Service
POSIX Portable Operating System Interface
QoS Quality of Service
RDMA Remote Direct Memory Access
RFS Receive Flow Steering
RPS Receive Packet Steering
RSS Receive Side Scaling
RX Receiver
SaaS Software as a Service
TCP Transmission Control Protocol
TPS Transactions per second
TX Transmitter
UDP User Datagram Protocol
VM Virtual Machine
VPN Virtual Private Network
XDP eXpress Data Path
XPS Transmit Packet Steering

References

[AFLV08] Al-Fares, Mohammad ; Loukissas, Alexander ; Vahdat, Amin: A Scalable, Commodity Data Center Network Architecture. In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. New York, NY, USA : ACM, 2008 (SIGCOMM '08). – ISBN 978–1–60558–175–0, 63–74

[BAAZ09] Benson, Theophilus ; Anand, Ashok ; Akella, Aditya ; Zhang, Ming: Understanding Data Center Traffic Characteristics. In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*. New York, NY, USA : ACM, 2009 (WREN '09). – ISBN 978–1–60558–443–0, 65–72

[BAM10] Benson, Theophilus ; Akella, Aditya ; Maltz, David A.: Network Traffic Characteristics of Data Centers in the Wild. In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. New York, NY, USA : ACM, 2010 (IMC '10). – ISBN 978–1–4503–0483–2, 267–280

[BH07] Barroso, L. A. ; Hölzle, U.: The Case for Energy-Proportional Computing. In: *Computer* 40 (2007), Dec, Nr. 12, S. 33–37. `http://dx.doi.org/10.1109/MC.2007.443`. – DOI 10.1109/MC.2007.443. – ISSN 0018–9162

[Chi05] Chimata, Ashwin K.: Pach of a packet in the Linux kernel stack. In: *Dartmouth.edu* (2005), July. `https://www.cs.dartmouth.edu/~sergey/netreads/path-of-packet/Network_stack.pdf`

[Cil18] Cilium ; cilium.io (Hrsg.): *BPF and XDP Reference Guide*. 1.2. https://cilium.readthedocs.io/en/v1.2/bpf/: cilium.io, 2018

[Cor18] Corbet, Jonathan ; LWN.net (Hrsg.): *Accelerating networking with AF_XDP*. https://lwn.net/Articles/750845: LWN.net, April 2018

[DS18] Dasineni, Ranjeeth ; Shirokov, Nikita: *Open-sourcing Katran, a scalable network load balancer*. `https://code.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/`. Version: May 2018

[Gra18] Graf, Thomas: *Why is the kernel community replacing iptables with BPF?* `https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables/`. Version: April 2018

[HB18] Herbert, Tom ; Bruijn, Wellem de ; Google (Hrsg.): *Scaling in the Linux Networking Stack.* https://www.kernel.org/doc/Documentation/networking/scaling.txt: Google, 2018

[Her10] Herbert, Tom ; Google (Hrsg.): *rfs: Receive Flow Steering.* https://lwn.net/Articles/381955/: Google, 2010

[HS16] Herbert, Tom ; Starovoitov, Alexei ; Facebook (Hrsg.): *eXpress Data Path (XDP).* https://github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf: Facebook, March 2016

[IBM18] IBM ; IBM (Hrsg.): *IBM Cloud Functions Platform architecture.* IBM Cloud Docs. https://console.bluemix.net/docs/openwhisk/openwhisk_about.html#openwhisk_how: IBM, July, 2018

[IN10] Inoue, H. ; Nakatani, T.: Performance of multi-process and multi-thread processing on multi-core SMT processors. In: *IEEE International Symposium on Workload Characterization (IISWC'10)*, 2010, S. 1–10

[IOV16] IOVISOR ; iovisor.org (Hrsg.): *XDP - eXpress Data Path.* https://www.iovisor.org/technology/xdp: iovisor.org, 2016

[Ker10] Kerrisk, Michael: *The Linux Programming Interface: A Linux and UNIX System Programming Handbook.* 1st. San Francisco, CA, USA : No Starch Press, 2010. – ISBN 1593272200, 9781593272203

[Ker13] Kerrisk, Michael ; LWN.net (Hrsg.): *The SO_REUSEPORT socket option.* https://lwn.net/Articles/542629/: LWN.net, 2013

[Ker18] Kerrisk, Michael ; Linux man pages (Hrsg.): *BPF(2) Linux Programmer's Manual.* 4.16. http://man7.org/linux/man-pages/man2/bpf.2.html: Linux man pages, February 2018

[KPS+16] Kaufmann, Antoine ; Peter, SImon ; Sharma, Naveen K. ; An-

DERSON, Thomas ; KRISHNAMURTHY, Arvind: High Performance Packet Processing with FlexNIC. In: *SIGPLAN Not.* 51 (2016), März, Nr. 4, 67–81. http://dx.doi.org/10.1145/2954679.2872367. – DOI 10.1145/2954679.2872367. – ISSN 0362–1340

[LWP04] LIU, Jiuxing ; WU, Jiesheng ; PANDA, Dhabaleswar K.: High Performance RDMA-Based MPI Implementation over InfiniBand. In: *International Journal of Parallel Programming* 32 (2004), Jun, Nr. 3, 167–198. http://dx.doi.org/10.1023/B:IJPP.0000029272.69895.c1. – DOI 10.1023/B:IJPP.0000029272.69895.c1. – ISSN 1573–7640

[MJ93] MCCANNE, Steven ; JACOBSON, Van: The BSD Packet Filter: A New Architecture for User-level Packet Capture. In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. Berkeley, CA, USA : USENIX Association, 1993 (USENIX'93), 2–2

[Mon18] MONNET, Quentin ; NETRONOME (Hrsg.): *The challenges of XDP hardware offload.* FOSDEM 2018. https://archive.fosdem.org/2018/schedule/event/xdp/: NETRONOME, February 2018

[PLZ+15] PETER, Simon ; LI, Jialin ; ZHANG, Irene ; PORTS, Dan R. K. ; WOOS, Doug ; KRISHNAMURTHY, Arvind ; ANDERSON, Thomas ; ROSCOE, Timothy: Arrakis: The Operating System Is the Control Plane. In: *ACM Trans. Comput. Syst.* 33 (2015), November, Nr. 4, 11:1–11:30. http://dx.doi.org/10.1145/2812806. – DOI 10.1145/2812806. – ISSN 0734–2071

[Rob18] ROBERTS, Mike: *Serverless Architectures.* https://martinfowler.com/articles/serverless.html. Version: May 2018

[Sch14] SCHOLZ, Dominik: A Look at Intel's Dataplane Development Kit. In: *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)* (2014), August, 115-122. http://dx.doi.org/10.2313/NET-2014-08-1_15. – DOI 10.2313/NET–2014–08–1_15. – ISSN 1868–2642

[Tau16] TAUSANOVITCH, Nick ; NETRONOME (Hrsg.): *What*

Makes a NIC a SmartNIC, and Why is it Needed? https://www.netronome.com/blog/what-makes-a-nic-a-smartnic-and-why-is-it-needed/: Netronome, 2016

[TK18a] Töpel, Björn ; Karlsson, Magnus ; Intel (Hrsg.): *AF_XDP*. Kernel Documentation v4.18. https://www.kernel.org/doc/html/v4.18/networking/af_xdp.html: Intel, August 2018

[TK18b] Töpel, Björn ; Karlsson, Magnus ; Intel (Hrsg.): *Fast Packet Processing in Linux with AF_XDP*. FOSDEM 2018. https://archive.fosdem.org/2018/schedule/event/af_xdp/: Intel, February 2018

[TSP17] Tu, Cheng-Chun ; Stringer, Joe ; Pettit, Justin: Building an Extensible Open vSwitch Datapath. In: *SIGOPS Oper. Syst. Rev.* 51 (2017), September, Nr. 1, 72–77. `http://dx.doi.org/10.1145/3139645.3139657`. – DOI 10.1145/3139645.3139657. – ISSN 0163–5980

B. Appendix

B.i. Server Code

AF_XDP implementation

Listing 20: reqrouter.h

```
/* SPDX-License-Identifier: GPL-2.0 */
#ifndef XDPSOCK_H_
#define XDPSOCK_H_

/* Power-of-2 number of sockets per function */
#define MAX_SOCKS 16

/* Port Range for the requestrouter */
#define PORT_RANGE_LOWER 1200
#define PORT_RANGE_UPPER 16000

#endif /* XDPSOCK_H_ */
```

Listing 21: reqrouter_user.c

```
    // SPDX-License-Identifier: GPL-2.0
    /* Copyright(c) 2017 - 2018 Intel Corporation. */
    /* Extended by Marius Gerling 2018 */

#include <assert.h>
#include <errno.h>
#include <getopt.h>
#include <libgen.h>
#include <linux/bpf.h>
#include <linux/if_link.h>
#include <linux/if_xdp.h>
#include <linux/if_ether.h>
#include <net/if.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <net/ethernet.h>
#include <sys/resource.h>
#include <sys/socket.h>
#include <sys/mman.h>
#include <sys/sysinfo.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>
#include <locale.h>
#include <sys/types.h>
#include <poll.h>

#include <bpf/libbpf.h>
#include "bpf_util.h"
#include <bpf/bpf.h>

#include <linux/ip.h>
#include <linux/udp.h>

#include "reqrouter.h"

#include "../common/functions.h"

#ifndef SOL_XDP
#define SOL_XDP 283
#endif
```

```
#ifndef AF_XDP
#define AF_XDP 44
#endif

#ifndef PF_XDP
#define PF_XDP AF_XDP
#endif

#define NUM_FRAMES 131072
#define FRAME_HEADROOM 0
#define FRAME_SIZE 2048
#define NUM_DESCS 1024
#define BATCH_SIZE 16

#define FQ_NUM_DESCS 1024
#define CQ_NUM_DESCS 1024

#define DEBUG_HEXDUMP 0

typedef __u64 u64;
typedef __u32 u32;
typedef __u16 u16;

static unsigned long prev_time;

static unsigned int header_length = sizeof(struct ether_header) + sizeof(struct iphdr) + sizeof(struct udphdr);

static u32 opt_xdp_flags;
static const char *opt_if = "";
static int opt_ifindex;
static int opt_queues = 1;
static int opt_poll;
static int opt_threads = 1;
static u32 opt_xdp_bind_flags;

struct xdp_umem_uqueue {
    u32 cached_prod;
    u32 cached_cons;
    u32 mask;
    u32 size;
    u32 *producer;
    u32 *consumer;
    u64 *ring;
    void *map;
};

struct xdp_umem {
    char *frames;
    struct xdp_umem_uqueue fq;
    struct xdp_umem_uqueue cq;
    int fd;
};

struct xdp_uqueue {
    u32 cached_prod;
    u32 cached_cons;
    u32 mask;
    u32 size;
    u32 *producer;
    u32 *consumer;
    struct xdp_desc *ring;
    void *map;
};

struct xdpsock {
    struct xdp_uqueue rx;
    struct xdp_uqueue tx;
    int sfd;
    struct xdp_umem *umem;
    u32 outstanding_tx;
};

struct xdpsock *xsks[PORT_RANGE_UPPER];
```

```
struct sock_port{
    struct xdpsock **xsks;
    int *ports;
    int length;
    int id;
};

static unsigned long get_nsecs(void)
{
    struct timespec ts;

    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec * 1000000000UL + ts.tv_nsec;
}

#define lassert(expr)                                   \
    do {                                                \
        if (!(expr)) {                                  \
            fprintf(stderr, "%s:%s:%i: Assertion failed: "  \
                #expr ": errno: %d/\"%s\"\n",           \
                __FILE__, __func__, __LINE__,           \
                errno, strerror(errno));                \
            exit(EXIT_FAILURE);                         \
        }                                               \
    } while (0)

#define barrier() __asm__ __volatile__("": : :"memory")
#ifdef __aarch64__
#define u_smp_rmb() __asm__ __volatile__("dmb ishld": : :"memory")
#define u_smp_wmb() __asm__ __volatile__("dmb ishst": : :"memory")
#else
#define u_smp_rmb() barrier()
#define u_smp_wmb() barrier()
#endif
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)

static inline u32 umem_nb_free(struct xdp_umem_uqueue *q, u32 nb)
{
    u32 free_entries = q->cached_cons - q->cached_prod;

    if (free_entries >= nb)
        return free_entries;

    /* Refresh the local tail pointer */
    q->cached_cons = *q->consumer + q->size;

    return q->cached_cons - q->cached_prod;
}

static inline u32 xq_nb_free(struct xdp_uqueue *q, u32 ndescs)
{
    u32 free_entries = q->cached_cons - q->cached_prod;

    if (free_entries >= ndescs)
        return free_entries;

    /* Refresh the local tail pointer */
    q->cached_cons = *q->consumer + q->size;
    return q->cached_cons - q->cached_prod;
}

static inline u32 umem_nb_avail(struct xdp_umem_uqueue *q, u32 nb)
{
    u32 entries = q->cached_prod - q->cached_cons;

    if (entries == 0) {
        q->cached_prod = *q->producer;
        entries = q->cached_prod - q->cached_cons;
    }

    return (entries > nb) ? nb : entries;
}

static inline u32 xq_nb_avail(struct xdp_uqueue *q, u32 ndescs)
```

```
{
    u32 entries = q->cached_prod - q->cached_cons;

    if (entries == 0) {
        q->cached_prod = *q->producer;
        entries = q->cached_prod - q->cached_cons;
    }

    return (entries > ndescs) ? ndescs : entries;
}

static inline int umem_fill_to_kernel(struct xdp_umem_uqueue *fq, u64 *d,
                      size_t nb)
{
    u32 i;

    if (umem_nb_free(fq, nb) < nb)
        return -ENOSPC;

    for (i = 0; i < nb; i++) {
        u32 idx = fq->cached_prod++ & fq->mask;

        fq->ring[idx] = d[i];
    }

    u_smp_wmb();

    *fq->producer = fq->cached_prod;

    return 0;
}

static inline size_t umem_complete_from_kernel(struct xdp_umem_uqueue *cq,
                           u64 *d, size_t nb)
{
    u32 idx, i, entries = umem_nb_avail(cq, nb);

    u_smp_rmb();

    for (i = 0; i < entries; i++) {
        idx = cq->cached_cons++ & cq->mask;
        d[i] = cq->ring[idx];
    }

    if (entries > 0) {
        u_smp_wmb();

        *cq->consumer = cq->cached_cons;
    }

    return entries;
}

static inline void *xq_get_data(struct xdpsock *xsk, u64 addr)
{
    return &xsk->umem->frames[addr];
}

static inline int xq_enq(struct xdp_uqueue *uq,
             const struct xdp_desc *descs,
             unsigned int ndescs)
{
    struct xdp_desc *r = uq->ring;
    unsigned int i;

    if (xq_nb_free(uq, ndescs) < ndescs)
        return -ENOSPC;

    for (i = 0; i < ndescs; i++) {
        u32 idx = uq->cached_prod++ & uq->mask;

        r[idx].addr = descs[i].addr;
        r[idx].len = descs[i].len;
    }
```

```
    u_smp_wmb();

    *uq->producer = uq->cached_prod;
    return 0;
}

static inline int xq_deq(struct xdp_uqueue *uq,
             struct xdp_desc *descs,
             int ndescs)
{
    struct xdp_desc *r = uq->ring;
    unsigned int idx;
    int i, entries;

    entries = xq_nb_avail(uq, ndescs);

    u_smp_rmb();

    for (i = 0; i < entries; i++) {
        idx = uq->cached_cons++ & uq->mask;
        descs[i] = r[idx];
    }

    if (entries > 0) {
        u_smp_wmb();

        *uq->consumer = uq->cached_cons;
    }

    return entries;
}

static bool swap_header(void *data, u64 l)
{
    if (l < header_length) {
        return false;
    }
    //Eth-Header (MAC Adresses)
    struct ether_header *eth = (struct ether_header *)data;
    struct ether_addr *src_addr = (struct ether_addr *)&eth->ether_shost;
    struct ether_addr *dst_addr = (struct ether_addr *)&eth->ether_dhost;
    struct ether_addr eth_tmp;

    eth_tmp = *src_addr;
    *src_addr = *dst_addr;
    *dst_addr = eth_tmp;

    u64 offset = sizeof(*eth);

    //IP-Header (IP-Adresses)
    struct iphdr *iph = (struct iphdr *)(data + offset);
    u32 ip_tmp = iph->saddr;
    iph->saddr = iph->daddr;
    iph->daddr = ip_tmp;
    //Checksum stays the same. Change of length requires recalculation
    offset += sizeof(*iph);

    //UDP-Header (Ports)
    struct udphdr *udph = (struct udphdr *)(data + offset);
    u16 udp_tmp = udph->source;
    udph->source = udph->dest;
    udph->dest = udp_tmp;
    //Clear the checksum
    udph->check = 0;

    return true;
}

static struct xdp_umem *xdp_umem_configure(int sfd)
{
    int fq_size = FQ_NUM_DESCS, cq_size = CQ_NUM_DESCS;
    struct xdp_mmap_offsets off;
    struct xdp_umem_reg mr;
    struct xdp_umem *umem;
    socklen_t optlen;
```

```
    void *bufs;

    umem = calloc(1, sizeof(*umem));
    lassert(umem);

    lassert(posix_memalign(&bufs, getpagesize(), /* PAGE_SIZE aligned */
                NUM_FRAMES * FRAME_SIZE) == 0);

    mr.addr = (__u64)bufs;
    mr.len = NUM_FRAMES * FRAME_SIZE;
    mr.chunk_size = FRAME_SIZE;
    mr.headroom = FRAME_HEADROOM;

    lassert(setsockopt(sfd, SOL_XDP, XDP_UMEM_REG, &mr, sizeof(mr)) == 0);
    lassert(setsockopt(sfd, SOL_XDP, XDP_UMEM_FILL_RING, &fq_size,
            sizeof(int)) == 0);
    lassert(setsockopt(sfd, SOL_XDP, XDP_UMEM_COMPLETION_RING, &cq_size,
            sizeof(int)) == 0);

    optlen = sizeof(off);
    lassert(getsockopt(sfd, SOL_XDP, XDP_MMAP_OFFSETS, &off,
            &optlen) == 0);

    umem->fq.map = mmap(0, off.fr.desc +
                FQ_NUM_DESCS * sizeof(u64),
                PROT_READ | PROT_WRITE,
                MAP_SHARED | MAP_POPULATE, sfd,
                XDP_UMEM_PGOFF_FILL_RING);
    lassert(umem->fq.map != MAP_FAILED);

    umem->fq.mask = FQ_NUM_DESCS - 1;
    umem->fq.size = FQ_NUM_DESCS;
    umem->fq.producer = umem->fq.map + off.fr.producer;
    umem->fq.consumer = umem->fq.map + off.fr.consumer;
    umem->fq.ring = umem->fq.map + off.fr.desc;
    umem->fq.cached_cons = FQ_NUM_DESCS;

    umem->cq.map = mmap(0, off.cr.desc +
                CQ_NUM_DESCS * sizeof(u64),
                PROT_READ | PROT_WRITE,
                MAP_SHARED | MAP_POPULATE, sfd,
                XDP_UMEM_PGOFF_COMPLETION_RING);
    lassert(umem->cq.map != MAP_FAILED);

    umem->cq.mask = CQ_NUM_DESCS - 1;
    umem->cq.size = CQ_NUM_DESCS;
    umem->cq.producer = umem->cq.map + off.cr.producer;
    umem->cq.consumer = umem->cq.map + off.cr.consumer;
    umem->cq.ring = umem->cq.map + off.cr.desc;

    umem->frames = bufs;
    umem->fd = sfd;

    return umem;
}

static struct xdpsock *xsk_configure(struct xdp_umem *umem, int queue)
{
    struct sockaddr_xdp sxdp = {};
    struct xdp_mmap_offsets off;
    int sfd, ndescs = NUM_DESCS;
    struct xdpsock *xsk;
    bool shared = true;
    socklen_t optlen;
    u64 i;

    sfd = socket(PF_XDP, SOCK_RAW, 0);
    lassert(sfd >= 0);

    xsk = calloc(1, sizeof(*xsk));
    lassert(xsk);

    xsk->sfd = sfd;
    xsk->outstanding_tx = 0;
```

```
    if (!umem) {
        shared = false;
        xsk->umem = xdp_umem_configure(sfd);
    } else {
        xsk->umem = umem;
    }

    lassert(setsockopt(sfd, SOL_XDP, XDP_RX_RING,
               &ndescs, sizeof(int)) == 0);
    lassert(setsockopt(sfd, SOL_XDP, XDP_TX_RING,
               &ndescs, sizeof(int)) == 0);
    optlen = sizeof(off);
    lassert(getsockopt(sfd, SOL_XDP, XDP_MMAP_OFFSETS, &off,
               &optlen) == 0);

    /* Rx */
    xsk->rx.map = mmap(NULL,
                off.rx.desc +
                NUM_DESCS * sizeof(struct xdp_desc),
                PROT_READ | PROT_WRITE,
                MAP_SHARED | MAP_POPULATE, sfd,
                XDP_PGOFF_RX_RING);
    lassert(xsk->rx.map != MAP_FAILED);

    if (!shared) {
        for (i = 0; i < NUM_DESCS * FRAME_SIZE; i += FRAME_SIZE)
            lassert(umem_fill_to_kernel(&xsk->umem->fq, &i, 1)
                == 0);
    }

    /* Tx */
    xsk->tx.map = mmap(NULL,
                off.tx.desc +
                NUM_DESCS * sizeof(struct xdp_desc),
                PROT_READ | PROT_WRITE,
                MAP_SHARED | MAP_POPULATE, sfd,
                XDP_PGOFF_TX_RING);
    lassert(xsk->tx.map != MAP_FAILED);

    xsk->rx.mask = NUM_DESCS - 1;
    xsk->rx.size = NUM_DESCS;
    xsk->rx.producer = xsk->rx.map + off.rx.producer;
    xsk->rx.consumer = xsk->rx.map + off.rx.consumer;
    xsk->rx.ring = xsk->rx.map + off.rx.desc;

    xsk->tx.mask = NUM_DESCS - 1;
    xsk->tx.size = NUM_DESCS;
    xsk->tx.producer = xsk->tx.map + off.tx.producer;
    xsk->tx.consumer = xsk->tx.map + off.tx.consumer;
    xsk->tx.ring = xsk->tx.map + off.tx.desc;
    xsk->tx.cached_cons = NUM_DESCS;

    sxdp.sxdp_family = PF_XDP;
    sxdp.sxdp_ifindex = opt_ifindex;
    sxdp.sxdp_queue_id = queue;

    if (shared) {
        sxdp.sxdp_flags = XDP_SHARED_UMEM;
        sxdp.sxdp_shared_umem_fd = umem->fd;
    } else {
        sxdp.sxdp_flags = opt_xdp_bind_flags;
    }

    lassert(bind(sfd, (struct sockaddr *)&sxdp, sizeof(sxdp)) == 0);

    return xsk;
}

static void int_exit(int sig)
{
    (void)sig;
    bpf_set_link_xdp_fd(opt_ifindex, -1, opt_xdp_flags);
    exit(EXIT_SUCCESS);
}
```

```
static struct option long_options[] = {
    {"interface", required_argument, 0, 'i'},
    {"queues", required_argument, 0, 'q'},
    {"poll", no_argument, 0, 'p'},
    {"xdp-skb", no_argument, 0, 'S'},
    {"xdp-native", no_argument, 0, 'N'},
    {"threads", required_argument, 0, 't'},
    {0, 0, 0, 0}
};

static void usage(const char *prog)
{
    const char *str =
        "  Usage: %s [OPTIONS]\n"
        "  Options:\n"
        "  -i, --interface=n    Run on interface n\n"
        "  -q, --queues=n    Number of queues (defaults to 1)\n"
        "  -p, --poll        Use poll syscall\n"
        "  -S, --xdp-skb=n   Use XDP skb-mod\n"
        "  -N, --xdp-native=n    Enfore XDP native mode\n"
        "  -t, --threads=n   Specify worker threads (default to 1).\n"
        "\n";
    fprintf(stderr, str, prog);
    exit(EXIT_FAILURE);
}

static void parse_command_line(int argc, char **argv)
{
    int option_index, c;

    opterr = 0;

    for (;;) {
        c = getopt_long(argc, argv, "i:q:pSNt:", long_options,
                &option_index);
        if (c == -1)
            break;

        switch (c) {
        case 'i':
            opt_if = optarg;
            break;
        case 'q':
            opt_queues = atoi(optarg);
            break;
        case 'p':
            opt_poll = 1;
            break;
        case 'S':
            opt_xdp_flags |= XDP_FLAGS_SKB_MODE;
            opt_xdp_bind_flags |= XDP_COPY;
            break;
        case 'N':
            opt_xdp_flags |= XDP_FLAGS_DRV_MODE;
            break;
        case 't':
            opt_threads = atoi(optarg);
            break;
        default:
            usage(basename(argv[0]));
        }
    }

    opt_ifindex = if_nametoindex(opt_if);
    if (!opt_ifindex) {
        fprintf(stderr, "ERROR: interface \"%s\" does not exist\n",
            opt_if);
        usage(basename(argv[0]));
    }
}

static void kick_tx(int fd)
{
    int ret;
```

```
    ret = sendto(fd, NULL, 0, MSG_DONTWAIT, NULL, 0);
    if (ret >= 0 || errno == ENOBUFS || errno == EAGAIN || errno == EBUSY)
        return;
    lassert(0);
}

static inline void complete_tx(struct xdpsock *xsk)
{
    u64 descs[BATCH_SIZE];
    unsigned int rcvd;
    size_t ndescs;

    if (!xsk->outstanding_tx)
        return;

    kick_tx(xsk->sfd);
    ndescs = (xsk->outstanding_tx > BATCH_SIZE) ? BATCH_SIZE :
         xsk->outstanding_tx;

    // re-add completed Tx buffers
    rcvd = umem_complete_from_kernel(&xsk->umem->cq, descs, ndescs);
    if (rcvd > 0) {
        umem_fill_to_kernel(&xsk->umem->fq, descs, rcvd);
        xsk->outstanding_tx -= rcvd;
    }
}

static int create_socket(int port, int queue, int thread)
{
    int offset = MAX_SOCKS / opt_queues;
    offset = offset * queue;
    offset = offset + thread;
    offset = offset + port;
    // Check if offset is valid
    if (offset > PORT_RANGE_UPPER || offset < PORT_RANGE_LOWER || offset >= port + MAX_SOCKS)
        return -1;
    // Create socket at queue
    xsks[offset] = xsk_configure(NULL, queue);
    return offset;
}

static void * requestHandler(void *arg)
{
    int timeout = 1000, ret = 0;
    struct sock_port *sp = (struct sock_port*) arg;
    struct xdp_desc descs[BATCH_SIZE];
    unsigned int rcvd, i, l;
    char *pkt = NULL;
    function func[sp->length];
    struct pollfd pfd[sp->length];
    memset(&pfd, 0, sizeof(pfd));
    for (l = 0; l < sp->length; l++) {
        func[l] = get_function(sp->ports[l]);
        if (func == NULL) {
            fprintf(stderr, "No Function defined...\n");
            return NULL;
        }
        pfd[l].fd = sp->xsks[l]->sfd;
        pfd[l].events = POLLIN;
    }

    for (;;) {
        if (opt_poll) { // Poll new data
            ret = poll(pfd, sp->length, timeout);
            if (ret <= 0) {
                fprintf(stdout, "Timeout(%d)\n", sp->id);
                fflush(stdout);
                continue;
            }
        }

        for (l = 0; l < sp->length; l++) {
            if (opt_poll && pfd[l].revents == 0)
                continue;
            rcvd = xq_deq(&sp->xsks[l]->rx, descs, BATCH_SIZE);
```

```
            if (rcvd == 0)
                continue;

            // Execute the function for every packet
            for (i = 0; i < rcvd; i++) {
                pkt = xq_get_data(sp->xsks[l], descs[i].addr);

#if DEBUG_HEXDUMP
                fprintf(stdout, "Port %d: ", sp->ports[l]);
                hex_dump(pkt, descs[i].len, descs[i].addr);
                fflush(stdout);
#endif
                // Swap ETH, IP and UDP header
                if (!swap_header(pkt, descs[i].len)) {
                    fprintf(stderr, "Port %d: Header to short\n", sp->ports[l]);
                    continue;
                }

                if (!(*func[l])(pkt, &descs[i].len, header_length)) {
                    fprintf(stderr, "Port %d: Function failed\n", sp->ports[l]);
                    continue;
                } // Todo: Calculate checksum if length changed
            }

            // Back to the Kernel by TX
            ret = xq_enq(&sp->xsks[l]->tx, descs, rcvd);
            lassert(ret == 0);
            sp->xsks[l]->outstanding_tx += rcvd;
            // Complete the TX
            complete_tx(sp->xsks[l]);
        }
    }
    free(sp->xsks);
    free(sp->ports);
    free(sp);
}

int main(int argc, char **argv)
{
    struct rlimit r = {RLIM_INFINITY, RLIM_INFINITY};
    struct bpf_prog_load_attr prog_load_attr = {
        .prog_type  = BPF_PROG_TYPE_XDP,
    };
    int prog_fd, xsks_map, rr_map, num_socks_map, num_queues_map;
    struct bpf_object *obj;
    char xdp_filename[256];
    struct bpf_map *map;
    int t, q, p, pqt, key = 0, ret;
    int ports[] = {1232};
    int len_ports = (sizeof(ports) / sizeof(ports[0]));
    struct sock_port *sp = NULL;

    pthread_t pt[PORT_RANGE_UPPER];

    parse_command_line(argc, argv);

    if (setrlimit(RLIMIT_MEMLOCK, &r)) {
        fprintf(stderr, "ERROR: setrlimit(RLIMIT_MEMLOCK) \"%s\"\n",
            strerror(errno));
        exit(EXIT_FAILURE);
    }

    snprintf(xdp_filename, sizeof(xdp_filename), "%s_kern.o", argv[0]);
    prog_load_attr.file = xdp_filename;

    if (bpf_prog_load_xattr(&prog_load_attr, &obj, &prog_fd))
        exit(EXIT_FAILURE);
    if (prog_fd < 0) {
        fprintf(stderr, "ERROR: no program found: %s\n",
            strerror(prog_fd));
        exit(EXIT_FAILURE);
    }

    map = bpf_object__find_map_by_name(obj, "xsks_map");
    xsks_map = bpf_map__fd(map);
```

```
if (xsks_map < 0) {
    fprintf(stderr, "ERROR: no xsks map found: %s\n",
        strerror(xsks_map));
    exit(EXIT_FAILURE);
}

map = bpf_object__find_map_by_name(obj, "num_socks_map");
num_socks_map = bpf_map__fd(map);
if (num_socks_map < 0) {
    fprintf(stderr, "ERROR: no num_socks map found: %s\n",
        strerror(num_socks_map));
    exit(EXIT_FAILURE);
}

map = bpf_object__find_map_by_name(obj, "rr_map");
rr_map = bpf_map__fd(map);
if (rr_map < 0) {
    fprintf(stderr, "ERROR: no rr map found: %s\n",
        strerror(rr_map));
    exit(EXIT_FAILURE);
}

map = bpf_object__find_map_by_name(obj, "num_queues_map");
num_queues_map = bpf_map__fd(map);
if (rr_map < 0) {
    fprintf(stderr, "ERROR: no rr map found: %s\n",
        strerror(num_queues_map));
    exit(EXIT_FAILURE);
}

if (bpf_set_link_xdp_fd(opt_ifindex, prog_fd, opt_xdp_flags) < 0) {
    fprintf(stderr, "ERROR: link set xdp fd failed\n");
    exit(EXIT_FAILURE);
}
fprintf(stdout, "Let's create Sockets!\n");

/* Create sockets... */
for (p = 0; p < len_ports; p++) { // ports[p] -> port
    for (q = 0; q < opt_queues; q++) { // q -> queue
        for (t = 0; t < opt_threads; t++) { // t -> thread
            pqt = create_socket(ports[p], q, t);
            if ( pqt < 0 ) {
                fprintf(stderr,
                    "ERROR: Socket creation failed\n");
                exit(EXIT_FAILURE);
            }
            ret = bpf_map_update_elem(xsks_map, &pqt, &xsks[pqt]->sfd, 0);
            if (ret) {
                fprintf(stderr, "Error: bpf_map_update_elem %d\n", pqt);
                fprintf(stderr, "ERRNO: %d\n", errno);
                fprintf(stderr, strerror(errno));
                exit(EXIT_FAILURE);
            }

            // Configure and start the consumer thread
            sp = malloc(sizeof(struct sock_port));
            (*sp).length = 1;
            (*sp).xsks = malloc(sizeof(struct xdpsock *) * (*sp).length);
            (*sp).xsks[0] = xsks[pqt];
            (*sp).ports = malloc(sizeof(int) * (*sp).length);
            (*sp).ports[0] = ports[p];
            (*sp).id = pqt;
            pthread_create(&pt[pqt], NULL, requestHandler, sp);
            fprintf(stdout, "Socket %d created\n", pqt);

            if (t == 0) {
                // Set the number of threads per queue
                ret = bpf_map_update_elem(num_socks_map, &pqt, &opt_threads, 0);
                if (ret) {
                    fprintf(stderr, "Error: bpf_map_update_elem %d\n", pqt);
                    fprintf(stderr, "ERRNO: %d\n", errno);
                    fprintf(stderr, strerror(errno));
                    exit(EXIT_FAILURE);
                }
            }
```

```
            }
        }
        fprintf(stdout, "Started %d Threads for Port %d\n", opt_threads * opt_queues, ports[p]);
    }
    // Set the number of queues
    ret = bpf_map_update_elem(num_queues_map, &key, &opt_queues, 0);
    if (ret) {
        fprintf(stderr, "Error: bpf_map_update_elem\n");
        fprintf(stderr, "ERRNO: %d\n", errno);
        fprintf(stderr, strerror(errno));
        exit(EXIT_FAILURE);
    }

    signal(SIGINT, int_exit);
    signal(SIGTERM, int_exit);
    signal(SIGABRT, int_exit);

    setlocale(LC_ALL, "");

    prev_time = get_nsecs();

    sleep(72000); //Sleep for 20 hours

    return 0;
}
```

Listing 22: reqrouter_kern.c

```
// SPDX-License-Identifier: GPL-2.0
#define KBUILD_MODNAME "foo"
#include <linux/bpf.h>
#include <uapi/linux/if_ether.h>
#include <uapi/linux/if_packet.h>
#include <uapi/linux/if_vlan.h>
#include <uapi/linux/ip.h>
#include <uapi/linux/in.h>
#include <uapi/linux/udp.h>
#include <asm/byteorder.h>

#include "bpf_helpers.h"
#include "reqrouter.h"

struct bpf_map_def SEC("maps") xsks_map = {
    .type = BPF_MAP_TYPE_XSKMAP,
    .key_size = sizeof(unsigned int),
    .value_size = sizeof(int),
    .max_entries = PORT_RANGE_UPPER,
};

struct bpf_map_def SEC("maps") num_socks_map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(unsigned int),
    .value_size = sizeof(uint16_t),
    .max_entries = PORT_RANGE_UPPER,
};

struct bpf_map_def SEC("maps") num_queues_map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(unsigned int),
    .value_size = sizeof(uint16_t),
    .max_entries = 1,
};

struct bpf_map_def SEC("maps") rr_map = {
    .type = BPF_MAP_TYPE_PERCPU_ARRAY,
    .key_size = sizeof(unsigned int),
    .value_size = sizeof(uint16_t),
    .max_entries = PORT_RANGE_UPPER,
};

/* Parse the ETH, IP and UDP header and match if the packet should
 * be processed by the RequestRouter
 *
 * returns 0 if the Packet could not be processed, the Port
```

```
 * number otherwise
 */
static __always_inline
uint16_t parse_header(struct xdp_md *ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    struct ethhdr *ethh = data;

    u64 offset = sizeof(*ethh);
    /* Check if the Packet contains a whole ethhdr */
    if ((void *)ethh + offset > data_end)
        return 0;

    u16 eth_type = ntohs(ethh->h_proto);
    /* Skip Ethernet II Packets. Focus is on 802.3 packets */
    if (eth_type < ETH_P_802_3_MIN)
        return 0;

    /* Todo: 802.1q, 802.1ad, QinQ etc */

    /* Chedk if it is a IPv4 Packet */
    if (eth_type != ETH_P_IP)
        return 0;

    struct iphdr *iph = data + offset;
    /* Check if the Packet contains a whole iphdr */
    if ((void *)iph + sizeof(*iph) > data_end)
        return 0;

    /* Focus only on UDP Packets */
    if (iph->protocol != IPPROTO_UDP)
        return 0;

    offset += sizeof(*iph);
    struct udphdr *udph = data + offset;
    /* Check if the packet contains a whole udphdr */
    if ((void *)udph + sizeof(*udph) > data_end)
        return 0;

    u16 dport = ntohs(udph->dest);
    /* Check if the port is in the designated Port range */
    if (dport < PORT_RANGE_LOWER || dport > PORT_RANGE_UPPER)
        return 0;
    return dport;
}


SEC("xdp_requestrouter")
int xdp_sock_prog(struct xdp_md *ctx)
{
    unsigned int port = parse_header(ctx), offset = 0;
    uint16_t *rr, *num_socks, *num_queues;

    /* Skip Packet if Port is 0 */
    if (port == 0)
        return XDP_PASS;

    /* Reduce on MAX_SOCKS */
    port = port - (port & (MAX_SOCKS - 1));

    /* Check how many queues exist */
    num_queues = bpf_map_lookup_elem(&num_queues_map, &offset);
    if (!num_queues)
        return XDP_ABORTED;
    if (*num_queues > 1) {
        offset = MAX_SOCKS / *num_queues;
        offset = offset * ctx->rx_queue_index;
        port = port + offset;
    }

    /* Check if multiple sockets for the port exist */
    num_socks = bpf_map_lookup_elem(&num_socks_map, &port);
    if (!num_socks)
        return XDP_ABORTED;
```

```
    if (*num_socks > 1) {
        /* Add Round-Robin Value to port */
        rr = bpf_map_lookup_elem(&rr_map, &port);
        if (!rr)
            return XDP_ABORTED;
        *rr = (*rr + 1) % *num_socks;
        port = port + *rr;
    }

    /* Forward Packet to xsks Socket */
    return bpf_redirect_map(&xsks_map, port, 0);
}

char _license[] SEC("license") = "GPL";
```

Linux socket implementation

Listing 23: reqrouter.c

```
/* reqrouter.c */
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <signal.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <getopt.h>
#include <libgen.h>
#include <pthread.h>
#include <poll.h>

#include "../common/functions.h"

static unsigned int opt_port = 1232;
static unsigned int opt_threads = 1;
static bool opt_unoptimized = false;

#define BUFLEN 2048
#define MMSGLEN 1024
#define HEADER_SIZE 42 //ETH-HEADER + IPv4 HEADER + UDP-HEADER

static void error_exit(char *errormessage)
{
    fprintf(stderr, "%s: %s\n", errormessage, strerror(errno));
    exit(EXIT_FAILURE);
}

static struct option long_options[] = {
    {"port", optional_argument, 0, 'p'},
    {"threads", optional_argument, 0, 't'},
    {"help", no_argument, 0, 'h'},
    {"unoptimized", no_argument, 0, 'u'},
    {0, 0, 0, 0}
};
static void usage(const char *prog)
{
    const char *str =
        " Usage %s [OPTIONS]\n"
        " Options: \n"
        " -p    PORT (Defaults to 1234)\n"
        " -t    THREADS (Defaults to 1)\n"
            " -h    prints this Message\n"
            " -u    turn off the optimizations\n";
    fprintf(stderr, str, prog);
    exit(EXIT_FAILURE);
```

```
}

static void parse_command_line(int argc, char **argv)
{
    int option_index, c;
    opterr = 0;
    for (;;) {
        c = getopt_long(argc, argv, "p:t:uh", long_options,
                &option_index);
        if (c == -1)
            break;
        switch(c) {
        case 'p':
            opt_port = atoi(optarg);
            break;
        case 't':
            opt_threads = atoi(optarg);
            if (opt_threads < 1) {
                fprintf(stdout, "To few threads, defaulting to 1 Thread\n");
                opt_threads = 1;
            }
            break;
        case 'u':
            opt_unoptimized = true;
            break;
        case 'h':
        default:
            usage(basename(argv[0]));
            break;
        }
    }
}

static void* recvpkg_unoptimized (void* arg)
{
    struct sockaddr_in server, client;
    int sock, recv_len;
    unsigned int sockaddr_len = sizeof(client);
    char buffer[BUFLEN];
    memset(buffer, 0, sizeof(buffer));

    // Get the function
    function func = get_function(opt_port);
    if (func == NULL){
        fprintf(stderr, "No Function defined...\n");
        return NULL;
    }

    // Create Socket
    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock < 0)
        error_exit ("Socket creation failed\n");
    fprintf(stdout, "Socket created\n");

    // Set Server Connection
    memset(&server, 0, sizeof(server));
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    server.sin_port = htons(opt_port);
    server.sin_family = AF_INET;

    if (bind(sock, (struct sockaddr*)&server, sizeof(server)) < 0)
        error_exit("Socket bind failed\n");
    fprintf(stdout, "Socket bound\n");

    for (;;) {
        recv_len = recvfrom(sock, buffer, BUFLEN, 0, (struct sockaddr*) &client, &sockaddr_len);
        if (recv_len < 0) {
            error_exit("recvfrom() failed\n");
        } else if (recv_len > 0) {
            // Execute function
            if (!(func)(buffer, &recv_len, 0)) {
                error_exit("Function failed\n");
            }

            //Answer
```

```
            if (sendto(sock, buffer, recv_len, 0, (struct sockaddr*)&client, sockaddr_len) != recv_len) {
                error_exit("sendto() failed\n");
            }
        }
    }
    close(sock);
    return NULL;
}

static void* recvpkg (void* arg)
{
    int sock, recv_len, i;
    struct sockaddr_in server, clients[MMSGLEN];
    struct mmsghdr msgs[MMSGLEN];
    struct iovec iovecs[MMSGLEN];
    char buffer[MMSGLEN][BUFLEN];

    // Enabling the timeout has a performance impact
//  struct timespec timeout;
//  timeout.tv_sec = 1;
//  timeout.tv_nsec = 0;

    // Initialize data structures
    memset(buffer, 0, sizeof(buffer));
    memset(msgs, 0, sizeof(msgs));
    for (i = 0; i < MMSGLEN; i++) {
        iovecs[i].iov_base = buffer[i];
        iovecs[i].iov_len = BUFLEN;
        msgs[i].msg_hdr.msg_iov = &iovecs[i];
        msgs[i].msg_hdr.msg_iovlen = 1;
        msgs[i].msg_hdr.msg_name = &clients[i];
        msgs[i].msg_hdr.msg_namelen = sizeof(clients[i]);
    }

    // Get the function
    function func = get_function(opt_port);
    if (func == NULL){
        fprintf(stderr, "No Function defined...\n");
        return NULL;
    }

    // Create Socket
    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock < 0)
        error_exit ("Socket creation failed\n");
    fprintf(stdout, "Socket created\n");

    // Set Server Connection
    memset(&server, 0, sizeof(server));
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    server.sin_port = htons(opt_port);
    server.sin_family = AF_INET;

    // Set SO_REUSEPORT to allow multithreading on the same socket
    int opt_val = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &opt_val, sizeof(opt_val));

    if (bind(sock, (struct sockaddr*)&server, sizeof(server)) < 0)
        error_exit("Socket bind failed\n");
    fprintf(stdout, "Socket bound\n");

    for (;;) {
        // Receive up to MMSGLEN messages from the socket
        // Timeout disabled because of performance impact
//      recv_len = recvmmsg(sock, msgs, MMSGLEN, MSG_WAITFORONE, &timeout);
        recv_len = recvmmsg(sock, msgs, MMSGLEN, MSG_WAITFORONE, 0);

        if (recv_len < 0) { // Length negative if recvmmsg failed
            error_exit("recvmmsg() failed\n");
        } else if (recv_len > 0) {
            // Read every packet
            for (i = 0; i < recv_len; i++) {
                // Execute function
                if (!(func)(msgs[i].msg_hdr.msg_iov[0].iov_base, &msgs[i].msg_len, 0)) {
                    error_exit("Function failed\n");
```

```
                }
                // Set the return message size
                msgs[i].msg_hdr.msg_iov[0].iov_len = msgs[i].msg_len;
            }
            // Respond to the requests (Up to MMSGLEN)
            if (sendmmsg(sock, msgs, recv_len, 0) != recv_len) {
                error_exit("sendmmsg() failed\n");
            }
            // Reset the iov_len to BUFLEN. Was changed to sendmmsg
            for (i = 0; i < recv_len; i++) {
                msgs[i].msg_hdr.msg_iov[0].iov_len = BUFLEN;
            }
        }
    }

    close(sock);
    return NULL;
}

static void prog_exit(int sig)
{
    fprintf(stdout, "Signal %d catched. Exiting...\n", sig);
    fflush(stdout);
    // Todo: Stop all threads and close the sockets
    exit(EXIT_SUCCESS);
}

int main (int argc, char **argv)
{
    parse_command_line(argc, argv);

    //Start x threads
    pthread_t pt[opt_threads];
    if (opt_unoptimized) {
        fprintf(stdout, "Unoptimized version. Single thread will be started");
        pthread_create(&pt[0], NULL, recvpkg_unoptimized, NULL);
    } else {
        fprintf(stdout, "Optimized version. %d threads to start\n", opt_threads);
        for (int i = 0; i < opt_threads; i++){
            pthread_create(&pt[i], NULL, recvpkg, NULL);
            fprintf(stdout, "Thread started");
        }
    }

    // Set signal Handlers
    signal(SIGINT, prog_exit);
    signal(SIGTERM, prog_exit);
    signal(SIGABRT, prog_exit);

    sleep(72000); // Wait up to 20 Hours

    exit(EXIT_SUCCESS);
}
```

B.ii. Client Code

Listing 24: client.go

```
package main

import (
    "flag"
    "fmt"
    "log"
    "net"
    "net/http"
    "strconv"
    "strings"
    "time"

    "github.com/prometheus/client_golang/prometheus"
```

```
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

// Flags
var (
    addr       = flag.String("listen-address", ":8080", "The address to listen on for HTTP requests.")
    server     = flag.String("server", "10.0.0.10", "The IP of the Server")
    ports      = flag.String("ports", "1232", "Ports for the applications , seperated by comma")
    packetSize = flag.Int("packet-size", 800, "Size of the packets sent")
    numConns   = flag.Int("num-conns", 4, "Number of simultaneous connection to an endpoint")
    verbose    = flag.Bool("verbose", false, "Show more output")
)

// prometheus metrics
var (
    answerTimesHistogram = prometheus.NewHistogramVec(prometheus.HistogramOpts{
        Name:    "answer_time_histogram",
        Help:    "Answer Time in Milliseconds",
        Buckets: []float64{0.0001, 0.0002, 0.0003, 0.0004, 0.0005, 0.001, 0.0015, 0.002, 0.0025, 0.003, \
            0.0035, 0.004, 0.0045, 0.005, 0.006, 0.007, 0.008, 0.009, 0.01, 0.011, 0.012, 0.013, 0.014, \
            0.015, 0.016, 0.017, 0.018, 0.019, 0.020, 0.025, 0.03, 0.035, 0.04, 0.045, 0.05, 0.06, 0.07,\
            0.08, 0.09, 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.1, 1.2, 1.3, 1.4, \
            1.5, 1.6, 1.7, 1.8, 1.9, 2, 2.5, 3, 3.5, 4, 4.5, 5, 6, 7, 8, 9, 10},
    }, []string{"port"})
    requests = prometheus.NewCounterVec(prometheus.CounterOpts{
        Name: "requests",
        Help: "Counter for sent requests",
    }, []string{"port"})
    failedRequests = prometheus.NewCounterVec(prometheus.CounterOpts{
        Name: "failedRequests",
        Help: "Counter for failed requests",
    }, []string{"port"})
    responses = prometheus.NewCounterVec(prometheus.CounterOpts{
        Name: "responses",
        Help: "Counter for received responses",
    }, []string{"port"})
    failedResponses = prometheus.NewCounterVec(prometheus.CounterOpts{
        Name: "failedResponses",
        Help: "Counter for received responses",
    }, []string{"port"})
    timeouts = prometheus.NewCounterVec(prometheus.CounterOpts{
        Name: "timeouts",
        Help: "Counter for requests without an answer within the timeout timerange",
    }, []string{"port"})
    falseResponses = prometheus.NewCounterVec(prometheus.CounterOpts{
        Name: "falseResponses",
        Help: "Counter for Responses with the wrong content",
    }, []string{"port"})
)

func serverPrometheus() {
    // Expose the registered metrics via HTTP.
    http.Handle("/metrics", promhttp.Handler())
    log.Fatal(http.ListenAndServe(*addr, nil))
}

func sendRequest(conn net.Conn, payload string, expectedResult string) {
    var err error
    var n int
    var timeSend, timeRecv time.Time
    buffer := make([]byte, 2048)
    for {
        n, err = conn.Write([]byte(payload))
        timeSend = time.Now()
        requests.With(prometheus.Labels{"port": conn.RemoteAddr().String()}).Inc()
        if err != nil {
            fmt.Println(err)
            failedRequests.With(prometheus.Labels{"port": conn.RemoteAddr().String()}).Inc()
        } else if n != len(payload) {
            fmt.Println("Sent to few bytes...")
            failedRequests.With(prometheus.Labels{"port": conn.RemoteAddr().String()}).Inc()
        }
        conn.SetReadDeadline(time.Now().Add(time.Second))
        n, err = conn.Read(buffer)
        timeRecv = time.Now()
```

```
        if err != nil {
            fmt.Println(err)
            failedResponses.With(prometheus.Labels{"port": conn.RemoteAddr().String()}).Inc()
            continue
        } else if n != len(expectedResult) {
            fmt.Println("Received to few bytes...")
            falseResponses.With(prometheus.Labels{"port": conn.RemoteAddr().String()}).Inc()
            continue
        } else if string(buffer[:n]) != expectedResult {
            fmt.Printf("Result is other than expected... \"%v\" != \"%v\"\n", string(buffer), expectedResult)
            falseResponses.With(prometheus.Labels{"port": conn.RemoteAddr().String()}).Inc()
            continue
        }
        responses.With(prometheus.Labels{"port": conn.RemoteAddr().String()}).Inc()
        answerTimesHistogram.With(prometheus.Labels{"port": conn.RemoteAddr().String()})
                                    .Observe(float64(timeRecv.Sub(timeSend)) / 1000000)
    }
}

func reverseString(s string) (result string) {
    for _, v := range s {
        result = string(v) + result
    }
    return
}

func init() {
    //Register the prometheus metrics
    prometheus.MustRegister(answerTimesHistogram)
    prometheus.MustRegister(requests)
    prometheus.MustRegister(failedRequests)
    prometheus.MustRegister(responses)
    prometheus.MustRegister(failedResponses)
    prometheus.MustRegister(falseResponses)
}

func main() {
    flag.Parse()
    go serverPrometheus()

    headerSize := 42

    var data string
    for i := headerSize; i < *packetSize; i++ {
        data += strconv.Itoa(i % 10)
    }

    for _, port := range strings.Split(*ports, ",") {
        for i := 0; i < *numConns; i++ {
            connStr := *server + ":" + port
            conn, err := net.Dial("udp", connStr)
            if err != nil {
                fmt.Println(err)
            }
            defer conn.Close()
            go sendRequest(conn, data, reverseString(data))

            if *verbose {
                fmt.Printf("Connection to %v setup\n", connStr)
            }
        }
    }
    time.Sleep(time.Duration(20) * time.Minute)
}
```