BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

ZERO COPY PACKET PROCESSING
ZPRACOVÁNÍ PAKETŮ POMOCÍ ZERO COPY

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. ONDŘEJ PLOTĚNÝ
SUPERVISOR VEDOUCÍ PRÁCE	Ing. MATĚJ GRÉGR, Ph.D.

BRNO 2019

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2018/2019

Master's Thesis Specification

22089

Student:	**Plotěný Ondřej, Bc.**
Programme:	Information Technology Field of study: Computer Networks and Communication
Title:	**Zero Copy Packet Processing**
Category:	Networking

Assignment:

1. Get familiar with packet processing techniques used in Linux kernel, e.g., af_packetv4, packet_zerocopy, XDP etc.
2. Design and implement a high speed packet processing method between a basic application and a network interface card.
3. Extend the application with a function for generation of flow records.
4. Test the application performance, discuss application limits and further optimisation.

Recommended literature:

- Bharadwaj, R. (2017). Mastering Linux Kernel development: A kernel developer's reference manual. ISBN: 978-1-78588-613-3.
- Robert Love. 2010. Linux Kernel Development (3rd ed.). Addison-Wesley Professional. ISBN: 978-0-672-32946-3

Requirements for the semestral defence:

- Items 1 and 2.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Grégr Matěj, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2018
Submission deadline:	May 22, 2019
Approval date:	October 31, 2018

Abstract

The aim of this thesis is a design and implementation of a net flow probe for 10GbE traffic. This thesis provides an overview of GNU/Linux utilities used for capture packets at high speeds and its fundamental mechanism. Next chapters introduce design and implementation of zero – copy probe capable to capture 10GbE traffic. The application uses the Express data path and its AF_XDP socket to capturing traffic on interface. The test platform is used FIT VUT NETX platform.

Abstrakt

Cílem této magisterské práce je návrh a implementace síťové sondy pro sledování toků na 10GbE rozhraní. Text se zabývá přehledem GNU/Linux nástrojů využívaných ve vysokorychlostních sítích a principů jejich fungování. Dále pak je uveden návrh a implementace sondy využívající mechanismu zero – copy pro sledování provozu na 10GbE rozhraní. Aplikace využívá Expresní datové cesty (XDP) a jeho AF_XDP soketu pro zachycení provozu na rozhraní. Jako testovací platforma byla vybrána platforma NETX používaná na FIT VUT.

Keywords

zero – copy, AF_XDP, XDP, XSK, eBPF, probe, netflow, packet processing, 10GE, NETX, xProbe

Klíčová slova

zero – copy, AF_XDP, XDP, XSK, eBPF, síťová sonda, síťový tok, zpracování paketů, 10GbE, NETX, xProbe

Reference

PLOTĚNÝ, Ondřej. *Zero Copy Packet Processing.* Brno, 2019. Master’s thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Matěj Grégr, Ph.D.

Rozšířený abstrakt

Sledování toků v počítačové síti je jednou ze základních úkonů pro zajištění její bezpečnosti a správného fungování. Nicméně objem provozu se neustále zvyšuje a úměrně tomu roste i množství přenesených dat přes rozhraní. V dnešní době již nejsou 10Gb, 40Gb dokonce ani 100Gb rozhraní výjimkou, což zvyšuje nároky na monitorovací systémy.

Monitorovací aplikace, běžící na běžném systému Linux, však nemohou plně zachytit provoz na 10Gb rozhraní. Důvodem je neefektivní architektura síťového subsystému uvnitř Linuxového jádra. Během mnoha let vývoje jádra Linuxu se vývojáři zaměřovali především na flexibilitu systému. Důraz byl kladem na modularitu systému tak, aby obsáhl potřeby všech síťových aplikací. Nové moduly zpracovávající nové protokoly mohou být přidávány do jádra, avšak tento koncept není vhodný pro zpracování paketů ve vyšších rychlostech. Datová cesta paketu mezi síťovou kartou a aplikací je příliš komplikovaná a zdlouhavá. Paket musí projít skrze několik softwarových vrstev, jako je například firewall nebo kontrola zahlcení, než dorazí do aplikace. Tyto vrstvy zvyšují časovou náročnost zpracování pro jednotlivé pakety, což se projeví snížením celkové výkonnosti aplikace.

Například propustost na 10Gb rozhraní může dosáhnout až 14.8 milionu paketů za vteřinu. To znamená, že procesor má pouze 67.2 ns na zpracování jediného paketu. 67.2 ns odpovídá zhruba 200 cyklům procesoru, a to není mnoho, bereme-li v potaz režii operačního systému [22].

Mimo to, architektura operačního systém Linux je založena na monolitickém jádru. To znamená, že celý systém je rozdělen na dvě oddělené části: oblast jádra a uživatelskou oblast. Jak název napovídá, v oblasti jádra je umístěno samotné jádro systému a ovladače hardwaru. V uživatelské části běží uživatelské aplikace. Obě oblasti jsou od sebe odděleny a komunikují mezi sebou pouze pomocí systémových volání. Tento koncept umožňuje uživateli oprostit se od komunikace s hardwarem. Nicméně, během zpracování paketů ve vyšších rychlostech, je přepínání kontextu mezi jádrem a aplikací velmi nežádoucí.

Aplikace běžící uvnitř jádra je podstatně rychlejší, avšak její vývoj je náročnější, jelikož aplikace nesmí ovlivnit chování jádra. Pokud aplikace běží uvnitř uživatelské oblasti, její výkonnost bude značně ovlivněna množstvím přepínání kontextu a obsluhou systémových volání, a navíc nebude mít kontrolu nad datovou cestou paketu uvnitř jádra. Několik softwarových utilit nabízí řešení tohoto problému, jako například Data Plane Development Kit (DPDK), PF_RING nebo eXpress Data Path (XDP).

Tato práce se zaměřuje na návrh a implementaci síťové sondy pro monitorování provozu na 10Gb rozhraní, která poběží na standardním Linuxovém operačním systému. Navržený systém odchytávání paketů pokryje limity standardního síťového subsystému tak, aby bylo možné plně zachytávat provoz na vysokorychlostním rozhraní. Navržený systém je založen na systému XDP, konkrétně na použití AF_XDP soketu. Takto je systém schopen zachytit a zpracovat rychlosti 10Gb.

Tato práce je rozdělena do několika kapitol. V první kapitole je představen koncept zachytávání toků v síti a jsou uvedeny základní části, ze kterých se skládá monitorovací systém. Zachytávání paketů je stěžejním úkonem takového systému a má velký dopad na celkovou výkonnost systému. Z tohoto důvodu jsou následující kapitoly zaměřeny na efektivní zpracování paketů.

V kapitole 3 je uveden proces zpracování paketů ve standardním Linuxu a ukazuje datovou cestu paketu mezi síťovou kartou a aplikací. Jsou představeny hlavní limitující prvky, což je důležité pro pochopení mechanismů zlepšujících zpracování paketu.

V kapitole 4 je uveden přehled mechanismů zlepšujících zpracování paketu a frameworků, které jsou na nich postaveny. Detailněji je popsán mechanismus zero-copy a jeho

implmentace v systému Linux. Zero-copy dovoluje vytvořit sdílenou pamět mezi jádrem a aplikací, čímž umožňuje efektivnější předávání dat.

Kapitola 5 se zaměřuje na popis systému XDP, který je použit při implementaci monitorovací sondy. Tento systém je založen na Berkeley Packet Filter (BPF), se kterým je úzce propojen. Systém XDP umožňuje uživateli nahrát jedoduchý kód přímo do ovladače síťové karty. Tento kód je spuštěn pokaždé, když síťová karta přijme nový paket. Uživatel tak může nadefinovat pravidla pro filtraci paketů v nejnižším bodě, což zefektivňuje celý proces zpracování. Novinkou tohoto systému je AF_XDP soket, který umožňuje obejít celý síťový subsystém a předat data ze síťové karty přímo do aplikace nejkratší možnou cestou. Nespornou výhodou tohoto systému je fakt, že funkcionalita jádra zůstává zachována, což neplatí u jiných podobných řešení (například DPDK).

Kapitola 6 představuje návrh síťové sondy postavené na systému XDP. Jedná se o vícevláknovou aplikaci, která využívá několika oddělených AF_XDP soketů pro paralelní zpracování paketu při vyšších rychlostech. Aplikace uchovává agregované informace o zachycených tocích ve vnitřních strukturách.

Kapitola 7 se zabývá implementací sondy. Jako programovací jazyk byl zvolen jazyk C. Implementace využívá knihovu *Libbpf* pro práci s AF_XDP sokety a knihovnu *pthread* pro práci s více vlákny.

V kapitole 8 je uvedeno laboratorní testování sondy na zařízení NETX. Sonda úspěšně zachytila a zpracovala provoz 14.8 milionu paketů za vteřinu.

Jako vylepšení síťové sondy se nabízí úprava XDP subsystému tak, aby bylo možné odchytávat i pakety jdoucí standardní cestou k jiným aplikacím. Současné řešení umožňuje pouze přeposlání paketu do monitorovací sondy. Další možnou prací je zdokonalení exportu sondy na plné pokrytí IPFIX standardu. Nevýhodou použitého systému je omezený počet ovladačů podporujících XDP.

Zero Copy Packet Processing

Declaration

Hereby I declare that this term project was prepared as an original author's work under the supervision of Mr. Ing. Matěj Grégr, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Ondřej Ploteňý
May 22, 2019

Acknowledgements

I would like to express my thanks to my supervisor Ing.Matěj Grégr Ph.D. for his numerous advice, the guidance, and pointing me in the right direction. Also, I wish to thank the Red Hat developer Jesper D. Brouer for his advices on XDP during NetDev0x13. Finally, I must express my very profound gratitude to my parents and to my friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Contents

Chapter 1

Introduction

Flow monitoring is an essential task to control and secure modern networks. However, the network traffic is increasing, and the traffic flowing through interfaces is proportionately growing . Due to this trend, the 10, 40, or even 100Gb interfaces are used more and more and put higher demands on monitoring systems.

For regularly used applications running on the Linux operating systems, it is not possible to fully capture the traffic of 10Gbps connection on common hardware. The reason is an architecture of Linux operation system and design of in-kernel network stack subsystem and its cooperation with device drivers.

During the evolution of the Linux kernel, the open-source community emphasis on flexibility of the system to be used in many different applications. The network stack within the Linux kernel is designed as flexible as possible to accommodate all the various networking needs. The stack is generic and allows new protocols to be added utilizing kernel modules, but it is suboptimal for high-speed performance. So in effect, the datapath for a newly arrived packet from the network card to an application is relatively long and complicated. The packet has to pass through several software layers, including a firewall or traffic shaper that increase per-packet processing overhead, and the overall performance decreases considerably.

For example, the raw throughput on a 10Gb link is up to 14.88 million Packets Per Second (pps) (64B packets), which means that every 67.2 ns a new packet can occur. Thus, the system has only 67 ns for single packet processing, which corresponds to 200 CPU cycles (depending on CPU frequency) between packets. This is not a lot of time considering the per-packet overheads generated by standard operating systems [22].

Moreover, a standard used Linux system is based on the monolithic-kernel architecture, where the system is divided into two parts: a kernel-space and user-space. In the kernel-space resides a core system functions and drivers controlling hardware and the user-space occupies regular user's programs. Both spaces are separated to each other, and cooperation is enabled only through secure system-calls. This concept allows a user to ignore the underlying communication with hardware. However, the context switching between kernel-space and user-space during packet propagation is a costly operation, and it is not suitable for packet processing in user-space at higher rates.

When a network program is running inside kernel's realm, it will be faster but much more complex to design to be safe for the kernel. When the network program is running inside user-space, the numerous system-calls and context switching will decrease the performance and program will not have any control over the datapath through the network stack. Several software solutions solve this schism such as the Data Plane Development Kit (DPDK), PF_RING or an eXpress Data Path (XDP).

This thesis aims to design and implement a packet capturing probe that meets today's packet processing requirements while running on a standard operating system. The capturing mechanism will overcome certain restrictive factors and overheads in the Linux network stack so that we can fully exploit the potential of the high-speed interfaces. The designed system introduced in this thesis is based on the XDP system, especially with AF_XDP socket usage, and it is able to capture and process flows at 10Gb speeds.

The thesis is organized as follows. The first chapter describes the basis of flow monitoring, the architecture of flow monitoring system and necessary processes from which the system is composed. The packet capturing has a significant impact on the system's performance so the following chapters are devoted to obtaining a data from network card more effectively.

The second chapter describes the packet processing in a standard GNU/Linux network stack and shows a datapath from a network card (NIC) to the user application without any improvements. Some limits of network stack architecture are introduced and their understanding is important for further improvements.

Chapter 4 dedicate to overview of commonly used mechanisms and frameworks for fast packet processing. These techniques are trying to eliminate disadvantage and bottlenecks of standard network stack. The Zero-copy mechanism and its implementation in Linux kernel are introduced.

The chapter 5 focuses on the XDP system and its usage in the fast packet processing. The XDP system is based on Extended packet filter so its describes as well.

Chapter 6 introduces a design of a network probe, which is able to capture traffic on 10GE interface. It is multithread application based on XDP system.

The following chapters 7 and 8 deal with implementation and testing the designed flow probe with usage of AF_XDP sockets.

Chapter 2

Flow Monitoring

Flow monitoring is an important part of a network administrator's tool-chain. It is useful in many activities e.g. billing, traffic analysis, network visibility, congestion control, and intrusion detection [25]. Flow monitoring embraces the complete chain of packet observation, flow export using protocols such as NetFlow and IPFIX, data collection, and data analysis.

The essence of flow monitoring lies in the creation of aggregated information about traffic passing through given point in the network infrastructure. Nowdays, a modern monitoring system focusing on flow monitoring, rather than individual packets analysis. Deep packet inspection over all packet data is too computationally expensive to be performed on high-speed networks, while flow monitoring provides only packet header processing, without traffic payload inspection, which is faster and more scalable [26, 9].

2.1 Flow definition

A flow is defined as „*a sequence of packets passing an observation point in the network during a certain time interval. All packets that belong to a particular flow have a set of common properties derived from the data contained in the packet, previous packets of the same flow, and from the packet treatment at the observation point*" [26]. The set of common properties is not strictly defined. Generally, as set is being used a 5-tuple: Source IP address, destination IP address, source port number, destination port number and IP protocol number, but addition information can be included, depending on vendor such as VLAN ID, IP ToS or interface number [9]. This set of common properties is also called a *Flow key* and expresses abstract indentification of a communication between two points, which passes through the observation point. The observation point can be line cards or interfaces of packet forwarding devices.

2.2 Flow Monitoring Architecture

A flow monitoring system is designed to record and make an aggregated information about flows available to the user. The observed information (such as number of packets, IP or TCP flags, payload size, etc) are stored in generic data structures called *flows records* and each record is uniquely identified by a particular flow key[26]. Thus, a flow monitoring system must be able to convert raw packets to corresponding flow records, collect them and proccess them to user-readable form (charts, graphs) or as input to another system [25]. The flow monitoring system requires several steps:

1. Capturing packets at one or more observation points
2. Assigning packets to flows
3. Creating and exporting flow records for the flows
4. Collecting, storing, and processing of the exported flow records

These steps can be divided into two separate subprocesses: a Flow monitoring process and Flow data processing. These processes can run on separated dedicated devices, then a device where the flow monitoring process running is called a *flow probe* or *flow exporter* and a device on which the flow data processing is working, is called a *flow collector*. The communication between probe and collector is ensured via Flow Export protocol. There are several standards for the export protocol. In this thesis, I will focus on IPFIX standard by IETF. Figure 2.1 shows a high-level overview of the generic monitoring system [26].

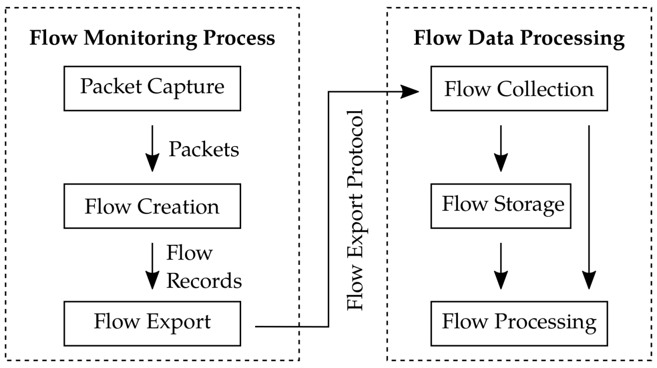


Figure 2.1: Flow monitoring system overview [26]

2.2.1 Flow monitoring process

Flow monitoring process has the task of transforming the raw packet data to a flow record and exports them to a collector. The process contains a packet capturing, flow creation, and export of flow records subprocesses.

Packet capture

First step in the monitoring system is a packet capturing from the line and it is typically carried out by a standard Network Interface Card (NIC). The device checks checksum and store packet in memory. Then a NIC driver passes data to operating system for further processing. The packet is marked with a timestamp, pointing to time at which packet was received. The timestamp can be registred by the NIC, NIC's driver or later by user-space application. Additionally, some extra metadata can by attached to the packet such as next hop, AS number and so on.

This phase can be very crutial to the overall system performance so some specialised hardware – accelerated cards can be used, which provides better performance of system at higher rates [26].

Flow creation

After packet capturing, it is necessary to extract values to determine a flow key. The flow key coresponds to communication which the packet belongs to. The packet headers has to be parsed and set of used flow keys has to be extracted. As mentioned previously, the key attributes are not strictly defined, so the attributes are picked up depending on the flow selection function. Commonly used key attributes are IP addresses, transport protocol, and ports. Some extra information can be extracted from the packet such as number of bytes or TCP flags. These metadata are used to update flow record for further analysis.

In this phase, the packet sampling and packet filtering can by done. The paket sampling reduces the amount of processed packets in order to maintain performance. It can use two different patterns to pick up packet samples, the random sampling or deterministic sampling (every N-th packet). The packet filtering separate the packets based on packet's certain property, such as IP address, port number or packet hash.

The extracted packet metadata are used to create a flow record or update existing one. All records are stored in a *flow cache* and the flow key identifies particular entries in the cache. A flow creating process calls for each captured packet a flow selection function which compares the current key with keys in flow cache. If no match is found a new record is pushed to the flow cache, if the key has been matched, the process update attributes in coresponding record. Algorithm 1 illustrates the flow creation process [26].

Algorithm 1 Construction of Flow Records (taken from [26])

```
1: Loop
2:     Get new packet $\mathcal{P}$
3:     Extract packet metadata $\mathcal{M}$
4:     Set found = false
5:     for all flow record $\mathcal{F}$ in flow cache do
6:         Apply flow selection function $\phi$ to $\mathcal{F}$ and $\mathcal{M}$
7:         if $\phi(\mathcal{F}, \mathcal{M}) = true$ then
8:             Aggregate $\mathcal{M}$ to $\mathcal{F}$
9:             Set found = true;
10:            break
11:        end if
12:    end for
13:    if not found then
14:        Create new flow record $\mathcal{F}$ from $\mathcal{M}$
15:        Insert $\mathcal{F}$ into flow cache
16:    end if
17: End Loop
```

The flow records are stored in flow cache until a flow is considered to have terminated and the record is expired. This occurs for several reasons: timeout expired, connection is closed normally (FIN flag), lack of resources or exporter shutdown.

Flow Export

The flow export maintains the process of delivering flow records to flow collectors, when the record has been expired. This task is consist of data serialization and message transmission. There is also possibility to sample or filter flow record which will be export in the same purpose as sampling and filtering after packet capturing. A crucial part of flow export is ensuring the security of the exported flow records. The information must be delivered only to the authorised destination. Therefore, an authorisation, confidentiality, preferably, also integrity should be provided.

The communication between probes and collector is described by protocols such as NetFlow or IPFIX. They define how to serialize and encode flow record and how to use diferrent transport protocol to deliver data to collector. In the following section 2.3, I will focus on IPFIX protocol, because it is a IETF standardized protocol, which is supported by broad range of vendors and suppports variable length of exported elements.

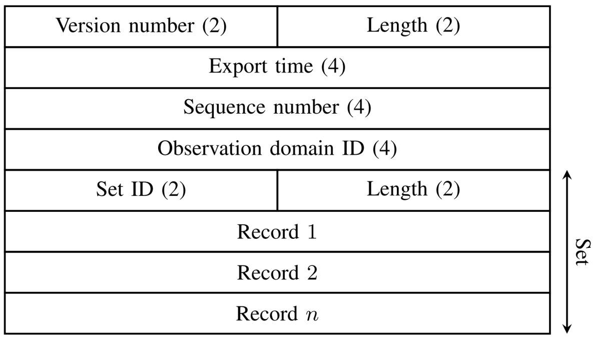


Figure 2.2: IPFIX message example [26]

2.2.2 Flow data processing

The aim of a flow data processing is to store a flow data, after they are delivered to a flow collector from multiple probes in the network, and allows their further analysis.

Flow Collection and processing

The flow collection subproccess providies recieving messages, which contains a flow records captured on probes. The probes and collector has to negotiate the same transport protocol, its security and flow export protocol, and collector validates each received message from probes, if it is in expected format. After validation, the messages are parsed to particular records and aggregation, data anonymization, filtering and summary generation can be done.

The information about flows can be stored for later processing or can be process in real-time, depending on an use – case. A flat files, a row-oriented databases (mySQL) or column-oriented (FastBit) databases can be used for permanent storage. Flow information can consume a big amount of space, so some kind of compression can be used as well [26].

The flow processing can be used to achive several goals: long-term statistics can be computed to capacity planning, live statistics can be used for tracking down network problems, anomaly detection techniques can be applied to flow data to detect suspicious behaviour which indicates a problem or attack on the network or modern machine learning techniques are utilised in flow processing, such as user identification, an IDS or traffic classification.

The flow analysis is performed usually in time batches. It might generate delays to data analysis so in time-critical application, a stram processing can be used.

2.3 IPFIX

As mentioned above, IP Flow Information Export protocol (IPFIX) is an IETF standard for exporting network flow based on NetFlow version 9, and is defined in several RFCs (5102, 5103). The IPFIX is designed less restrictive and allows dynamically reconfigure observed information and unlike Netflow, IPFIX contains specific fields which can be used by vendors to store proprietary information. It is possible due to defining a metadata called *Templates*. The Template is exported with flow information in IPFIX message and describes a layout of flow information which includes attribute id and its lenght. The IPFIX message is shown in Fig. 2.2. Each message is consist of message header, and one or more *Sets*. The message header holding information version, overall length of message, export time sequence number and domain ID. The *Set* has ID and variable length of set and its records can be filled with templates or a flow information data [26].

Chapter 3

Packet processing

This chapter describes a packet processing in a GNU/Linux kernel network stack. In general, the network stacks in operating systems are typically designed for flexibility. The high throughput of networking traffic can be problematic and commonly used Linux kernel stack does not offer effective operations. Within Linux network stack, packets need to be copied twice after being received to delivery to an application: first, from a network interface to the kernel buffer and then from kernel to the user−space (shown in Fig. 3.1). The data transmitting between these buffers is managed by NAPI (New API) interrupts. The understanding of packet processing in the standard stack is essential for its further increasing the performance and for describing mechanisms that do that.

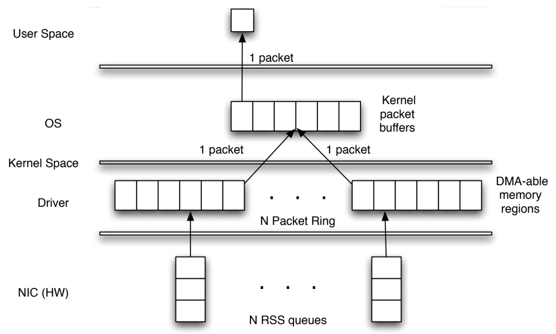


Figure 3.1: Linux network stack [18]

3.1 Linux network stack

When a new packet arrives, the NIC attaches the packet to a descriptor in the NIC's circular receiving queues (RX ring). Direct Memory Access (DMA) transfers the packet data to the DMA−able memory region of RAM without a CPU intervention. The packet descriptor in RX contains a DMA memory region address. At this point, the NIC needs to inform the system that the packet has been received, so the NIC raises an interrupt signal. Each time a packet RX interrupt is raised, the corresponding interrupt software handler is executed and copies the packet from the DMA memory region, in which the DMA transfer left the packet, into a local packet buffer in the kernel. Implementation of this kernel buffer is a `sk_buff` structure, which is the primary data structure for packet handling in Linux. However, an interrupt handling for each packet is a CPU intensive, so the NAPI mechanism is used instead. The NAPI starts a poll loop and interrupts are only enabled for the first packet of a batch. The poll loop gets periodically enabled and inspect the devices for received packets needed to be forwarded to the network stack.

At this moment the driver harvests and unmap the network ring buffer data so additional packets may be received. If the NIC supports multiple queues, the packets are distributed among CPUs, and the data in kernel buffer are passed up to the networking layer for further processing. The kernel has to check the socket's allocated memory. If the memory has exceeded, the kernel drops the packet. If the socket grants an unoccupied memory, the data are attached to the socket memory. The kernel checks any BPF filters as well.

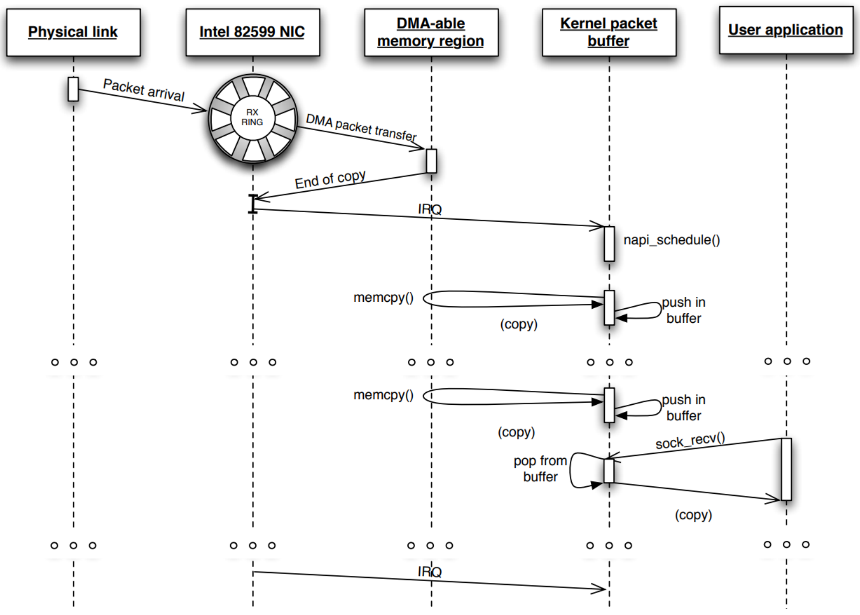


Figure 3.2: Linux network stack, RX scheme [18]

Packet capturing is the first step of the flow monitoring process and has a significant impact on the overall performance. As follows [18], the main causes of performance degradation during this phase are:

1. Per-packet allocation and deallocation of resources – memory management of the `sk_buff` in high-speed rates increases CPU overhead. Moreover, the `sk_buff` may contain unnecessary information, depending on use-case

2. Serialized access to traffic – it is advisable to parallelize the packet processing to multiple CPU and distribute traffic to multiple queues

3. Multiple data copies from driver to user-space – the packet data path includes several buffer copies. One of this copy can consume hundreds of cycles. Above that, the per-packet copy is inefficient.

4. Kernel-to-userspace context switching – the packet data path includes some system calls too. Each system call requires to switch context to kernel mode and vice versa.

5. No exploitation of memory locality – the significant number of cache misses are causing a performance degradation

Chapter 4

Fast packet processing

This chapter presents some mechanisms and frameworks, which are commonly used to speed-up packet processing. To achieve high packet processing performance on commodity hardware, it is necessary to remove any software bottlenecks between NIC and the program providing the packet processing.

4.1 Zero-copy

As mentioned in chapter 3, the standard NIC's receiving scheme is to store a packet data in the kernel-space buffer after Direct Memory Access (DMA) transfer. The application has to issue a read/write system calls to copy data from kernel-space to user-space buffers to packet processing and vice versa. The zero-copy aims to avoid this memory-to-memory copy and reduce unnecessary memory access. The *Zero-copy* is a common name for various techniques and design improvements. In this section, I will focus on the technique that is directly supported in the GNU/Linux kernel, and that is a *page remapping*.

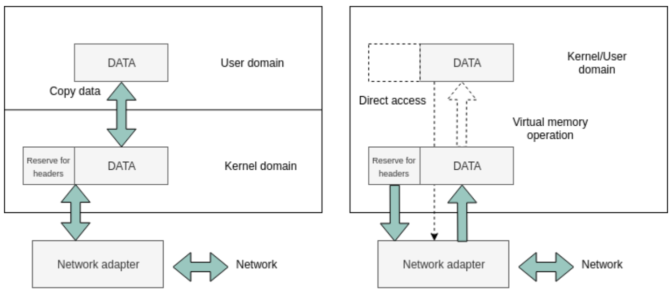


Figure 4.1: Original Data path vs. Zero-copy data path [16]

The basic idea behind the page remapping is to create a cache-like image of some file (or memory) in the virtual address area within the user process. In other words, a file on disk became a chunk of RAM area that the process can access and from a user's point of view, it looks like the OS allocates a block of memory, in which the file has been copied. The Linux supports the `mmap()` (or `do_mmap()`) system call, which provides the page remapping and allows part of a file or the memory stored on a block device to be mapped into a user's address space.

Thus, the page remapping creates a shared buffer between the kernel and a user-space. Moreover, the buffer can be shared between a device's driver and user-space. Then there is no need to copy the data between both of buffers and a received packet from NIC can be directly accessible in the user-space. Data from the user-space application can be directly sent to the NIC. It highly reduces the number of system calls, CPU processing time, memory usage as well as power consumption for data transmission.

4.1.1 Page remapping

Linux separates the address spaces of the kernel and other processes from each other. These address spaces consist of virtual memory addresses, which are only abstracted from physical addresses. To translate between virtual addresses and physical addresses, Linux uses special hardware – a memory management unit (MMU). The MMU translates addresses in larger batches called *pages*, which are the smallest units of memory that can have different permissions and behavior. The translation of virtual memory to physical memory inside the MMU is done through *page tables*, which holds information about page ownership. The hardware itself provides the mapping, but the kernel can manage these tables and their configuration.

If page remapping is used, the kernel modifies page tables to create a new virtual memory area, in which a mapped file resides. The backed file is divided into page-sized pieces and attached to new virtual memory area. Actually, the file pages are not directly loaded to physical memory; it provides lazy loading - a memory within a particular page is loaded, only when the first reference to this page will occur.

There is only one copy of pages in physical and virtual memory. If another process calls the same memory mapping no other virtual pages are copied, only file's reference count has to be increment, so the usage of this function increase time and space efficiency. Therefore, closing the file descriptor after mapping the file, will not cause loss of access to data [17].

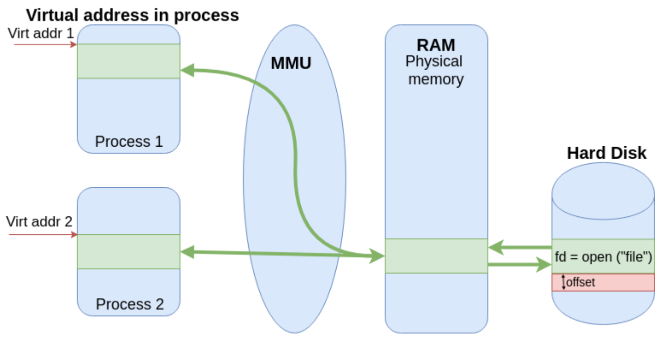


Figure 4.2: MMAP with `MAP_SHARED` flag, Fig. based on [6]

4.1.2 Mmap function

The `mmap` function maps some files or devices into the calling process virtual memory. The call is defined in sys/mman.h as:

```
void *mmap(void *start, size_t len, int prot, int flags, int fd, off_t offset)
```

The call will map an *len* bytes of an object represented by the file descriptor *fd*. The mapping begin can be moved within the object by *offset*. If *start* pointer is included, new mapped memory will start at this address. The access permissions are restricted by *prot*

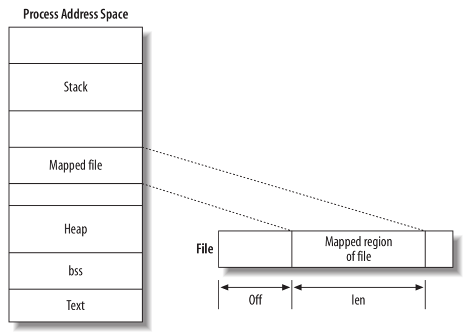


Figure 4.3: MMAP geometry [6]

and *flags* can add some special behavior. The function returns the address at which the new mapping will be placed [15].

Due to page-sized granularity described in 4.1.1, all mapping operations have to be done in multiple of page size, so mapped length must be rounded up and addresses for mapping must be page-aligned. To determine the default size of a page on a current machine, Linux user should use:

`size_t page_size = (size_t) sysconf(_SC_PAGESIZE)` or
`int page_size = getpagesize()`.

Mmap can be performed in two ways:

- **Private mapping** - defined by `MAP_PRIVATE`, this map is private to the process. The file is mapped as copy – on – write, and any changes are not reflected in the actual file, or the other processes mapping. The page is copied, and modifications are performed on the new page.
- **Shared mapping** - defined by `MAP_SHARED`, this map shares the mapping with all other processes that map this same file. Any modification performed in the file is written back to the disk and is available for other processes to read. There is no guarantee, that data writes to disk are immediately processed, due to Copy-in-write technique.

There is a desired memory protection of the mapping, which must agree with the open mode of the file:

- `PROT_EXEC` - pages may be executed
- `PROT_READ` - pages may be read
- `PROT_WRITE` - pages may be written
- `PROT_NONE` - pages may not be accessed

If a file descriptor and offset are given, the mapping is called a *file – backed mapping*. There is also an *Anonymous mapping* (flag `MAP_ANONYMOUS`), which is not file – backed and the file descriptor is NULL. Anonymous mapping contents are initialized to zero.

The complement function of `mmap()` is a `munmap()` - unmapping virtual memory. User has to call `munmap()` for each mmaped file descriptor to free memory. Both functions are thread safe.

4.2 Multithreading

An excellent way to improve performance of monitoring system is to capture packet using multiple threads. Nowadays, the multi-core CPUs are commonly used and parallel computing is widely support in Linux OS [22]. However, passing data between thread consumes systems's resources and the performance can be limited. When the application using multiple threads, the thread affinity should be used to reduce communication between cores.

4.2.1 Thread Affinity

The process, which provides a packet processing, should allocate a memory assigned to the executing CPU. The reading from a local cache of CPU is more effective because it decreases a cache miss probability.

The affinity is a technique fixes the thread execution and its resources localization to a particular processor or core. There are several affinity domains, that is thread affinity, process affinity, interrupt affinity or memory affinity. It aims to avoid expensive message passing between processes, thread scheduling, and polling [5].

Thread and process affinity being able to assign specific thread or process to a particular processor/core. In Linux can be used a `pthread_ setaffinity_np` from the POSIX *pthread* library.

The Interrupt affinity handles software and hardware interrupts by specific cores or processors. In Linux, the Interrupt affinity may be accomplished by writing a binary mask of CPU to `/proc/irq/IRQ#/smp_affinity` to assigned an interrupt handler IRQ# to CPU.

The memory affinity is used mainly in NUMA systems. There is a tendency to avoid cache-misses and hold particular data closer to executing CPU in a memory hierarchy.

4.2.2 RSS

The Receive-Side Scaling (RSS) is a NIC's feature, which allows distributing network traffic across several queues within NIC. Due to this approach, the traffic can be processed by multiple CPUs in multiprocessor systems. RSS is enabled by default but can be configured by a user and a spread rules can be modified by use-case. For example, traffic can be scattered by IP addresses or port number in packet headers. Many NIC drivers use an `ethtool` command to defined spread rules [22].

4.3 Frameworks

There are several frameworks which alternate standard network stack packet processing in Linux to overcome some limitations.

XDP

EXpress Data Path is specialized in-kernel facility allowing a access to kernel network data path without modifying the kernel. Appropriate use – cases are a load balancing, the DDoS protection or fast forwarding. The XDP using eBPF subsystem and it does not provide a kernel bypass mechanism [10].

PacketShader

PacketShader is a GPU – accelerated software router. PacketShader provides a huge packet buffer with batching processing and minimizes packet movement between local and remote memory in a NUMA system. The used strategy enables a kernel stack bypassing for easier and faster GPU operations.

Netmap

Netmap is a kernel module supporting multi – queue fast packet processing and pipes between applications. It uses zero – copy, kernel bypass and batched processing techniques. However, the circular ring buffer is fixed size, which may not be appropriate for the application with lots of packets in buffer [5].

PF_RING ZC

PF_RING ZC is a kernel module using a DNA/LibZero drivers allowing direct access to packets on the network interface by simultaneously bypassing both the Linux kernel and the PF_RING module in a zero – copy method. The framework adds support for virtualization and inter – process communication and it is possible to use zero – copy with non – PF_RING – aware drivers. The main disadvantage of this framework is the non – free licence.

DPDK

The Intel Data Plane Development Kit (DPDK) is a framework optimized for Intel hardware (NICs, CPU, chipset) with enhanced NUMA – awareness, and libraries for packet manipulation across cores. DPDK is most widely used framework. However, DPDK requires maintaining full separate drivers and its integration into solutions is hard, due to taking over entire NIC and the need to reimplement a TCP/IP stack.

OpenOnload

OpenOnload is a proprietary SolarFlare solution for fast packet processing. OpenLoad provides a user-level network stack, allowing to accelerate existing applications quickly.

Framework	XDP	PacketShader I/O	Netmap	PF_RING ZC	DPDK	OpenOnload
Zero-copy	Y	N	Y	Y	Y	Y
Kernel bypass	N	Y	Y	Y	Y	Y
I/O Batching	Y	Y	Y	Y	Y	Y
Hardware multi-queue support	N	Y	Y	Y	Y	Y
Devices family supported	ALL	1	8 ZC / ALL (non-ZC)	4 ZC / ALL (non-ZC)	11	All SolarFlare
Pcap library	Y	N	Y	Y	Y	Y
License	GPLv2	GPLv2	BSD	Proprietary	BSD	Proprietary
IXGBE version	Last	2.6.28	Last	Last	Last	N/A

Table 4.1: Summarize the features of the I/O frameworks [4]

4.3.1 Summary

Table 4.1 summarizes framework overview. A most popular technique used in high-performance packet processing is the kernel bypassing, which overcome the limitation of the Linux kernel networking layers by skipping them. Packet processing is done from user-space including the NIC driver handling. By giving full control of the NIC to user-space program, the kernel overhead (context switching, networking layer processing, interruptions, etc.) can be significantly reduced, especially when 10Gb or higher speeds are used. However, using the kernel bypass has several disadvantages 5:

- Does not used OS's abstraction for hardware resources. Custom user-space driver might be less tested, verified and reusable than an OS's one
- The program works as sand-box, with integration and interaction limits
- Kernel functionality is skipped, User need to reimplement them
- Security layer of OS is skipped

The better way is not to move packet-processing out of the kernel's realm into user-space, but to move user-space networking programs (filters, mappers, routing, etc.) into the lowest point of the kernel's domain. Such opportunity offers an XDP framework [10].

Chapter 5

The eXpress Data Path

This chapter presents in detail one of the fast packet processing framework – the eXpress Data Path (XDP). The XDP is a system that allows user programmability directly in the operating system network stack in a cooperative way while ensuring the safety and integrity of the rest of the system. The approach of XDP is to keep hardware control inside the kernel, but move packet processing operations into driver level. It is an alternative methodology to the kernel bypassing design and represents a good tradeoff between performance, integration into the system and general flexibility [1, 10].

In addition, XDP can completely bypass the network stack and provides the zero-copy socket (AF_XDP), which offers higher performance than common kernel modules hooking the stack. This feature is an ideal candidate for use in fast traffic monitoring. Moreover, the XDP is build-in mainline Linux kernel since its version 4.8 (AF_XDP since 4.18) and no specific HW requirements are needed 5.

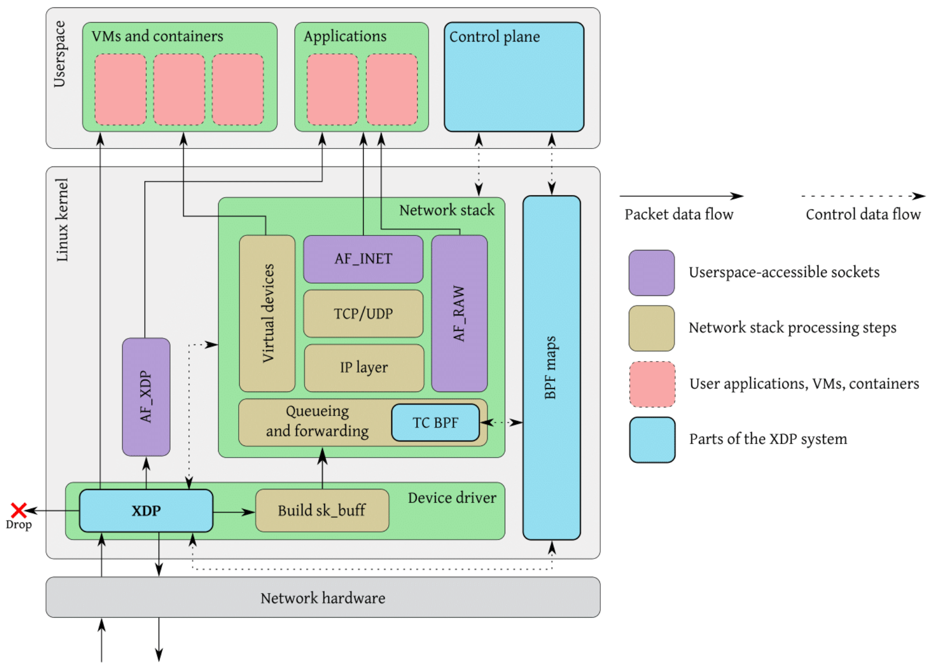


Figure 5.1: XDP integration with Linux network stack [10]

The XDP is a layer inside NIC driver that allows validated code execution for every single packet. The code can be loaded, controlled and inspected from user-space and some action on each packet can be taken before the operating system networking stack allocates the SKB structure and process the packet [10]. Figure 5.1 shows XDP integration into the Linux kernel. The data-plane is split between the kernel and in-kernel injected program while the control-plane is completely ensured by user-space. Thus, the user's validated code can be control from user-space, but it will run in the kernel space.

5.1 XDP actions

As was mentioned previously, XDP executes a simple code for every single packet to packet classification. The validated code decides datapath through the system based on the current packet context. Program can manipulate packet in arbitrary ways (encapsulate header or change some bytes), but finally, the program must return a verdict to the driver, describing how to handle the packet. There are several verdict actions that can be used:

1. **XDP_PASS** - allows the packet to pass into the standard network stack
2. **XDP_DROP** - drops the packet
3. **XDP_TX** - bounce the packet back on the same interface
4. **XDP_REDIRECT** - redirects the packets to another interface
   (a) **BPF_MAP_TYPE_DEVMAP** - redirecting raw frames to the user-defined device
   (b) **BPF_MAP_TYPE_CPUMAP** - redirecting raw frames to remote CPU
   (c) **BPF_MAP_TYPE_XSKMAP** - redirecting raw frames into userspace
5. **XDP_ABORTED** - drops the packet with an error

5.1.1 XDP and eBPF

The XDP system is closely linked to eBPF (extended Berkeley Packet Filtering) ecosystem. XDP does not have its own programming language, so it uses eBPF programming language (it uses C-like syntax) code. The eBPF code is compiled into custom bytecode, analyzed and translated into native instructions by the kernel and injected directly into the driver level as a sandbox. Due to this, the action decision can be done very quickly after the packet has been received. Nevertheless, the eBPF program has some restrictions and does not support completely arbitrary code. The restrictions are as follows [21]:

- contains no loop, Not Turing complete
- accesses only valid memory
- uses a limited number of eBPF instructions (no more than 4000 instructions)
- bounded program size

5.2 The design of XDP

The XDP system consists of four major components that cooperate with each other. The components are illustrated in Figure 5.2, which explains, how they fit together:

1. **The XDP driver hook** - entry point of XDP, attached to a driver event
2. **The eBPF virtual machine** - just – in – time compilation and execution of eBPF program
3. **BPF Maps** - communication channel with the rest of the system
4. **The BPF verifier** - static analysis to protect running kernel memory

Due to the using of the eBPF programming language for injected code, the eBPF compiler toolchain occupies a more substantial part of XDP system. XDP itself is just one driver level hook using and invoking eBPF's features, full toolchain is illustrated in Fig. 5.5. Nevertheless, the use of eBPF has a significant advantage over tradition loadable kernel modules, namely eBPF does not lead to a kernel-space memory corruption or kernel instability. It means that eBPF subsystem will only run code that has been deemed entirely safe to run.

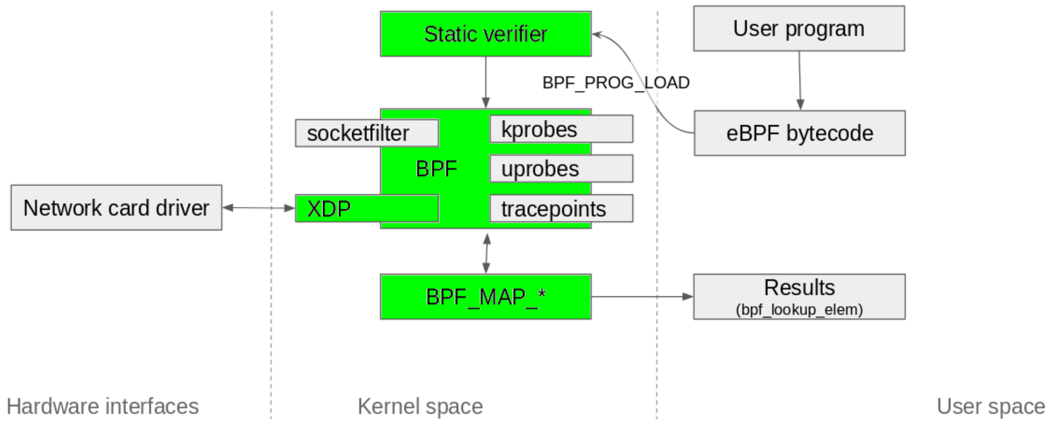


Figure 5.2: Four components of XDP design highlighted in green [8]

5.2.1 The XDP driver hook

In general, any generic kernel event can be potentially intercepted, and eBPF can react to it: message (socket-layer) received, data written to disk, page fault in memory and much more. So nowadays, the eBPF system is being used not only for networking purposes but also it is a tracepoint tool for kernel developers and production engineers to run user-space code inside the kernel [10].

Several hook points (event callbacks) for networking purposes exist in the network stack, where a user-defined eBPF program can be attached to, for example, kprobes and uprobes, socket (original tcpdump use case) and tc filters. The metadata associated with a packet (and dispatched to the eBPF program) and allowed kernel helpers are changed according to the hook point that has been used.

However, all hook points take place in higher levels of the network stack, so the XDP defined a new hook point at the lowest level of network stack datapath – in driver space, and the program execution can be triggered by the arrival of a packet to NIC. But not all network device drivers implement the XDP hook. In such a case, it is used the generic XDP hook (also called SKB_mode), which take place after SKB allocation [1, 10].

The hook execution

The XDP hook simply attaches the eBPF file-descriptor handle to netdev and the eBPF program is executed directly in the device driver whenever a packet is received from a hardware. Typically, the execution flow is divided into three steps, shown in Fig.5.3: reading, metadata processing, and writing packet data. In addition, some communication with the rest of the system can be made. These steps can be alternate or repeated in arbitrary ways, and whole packet processing can be split into multiple eBPF programs through *tail call*, which passes control between them [10].

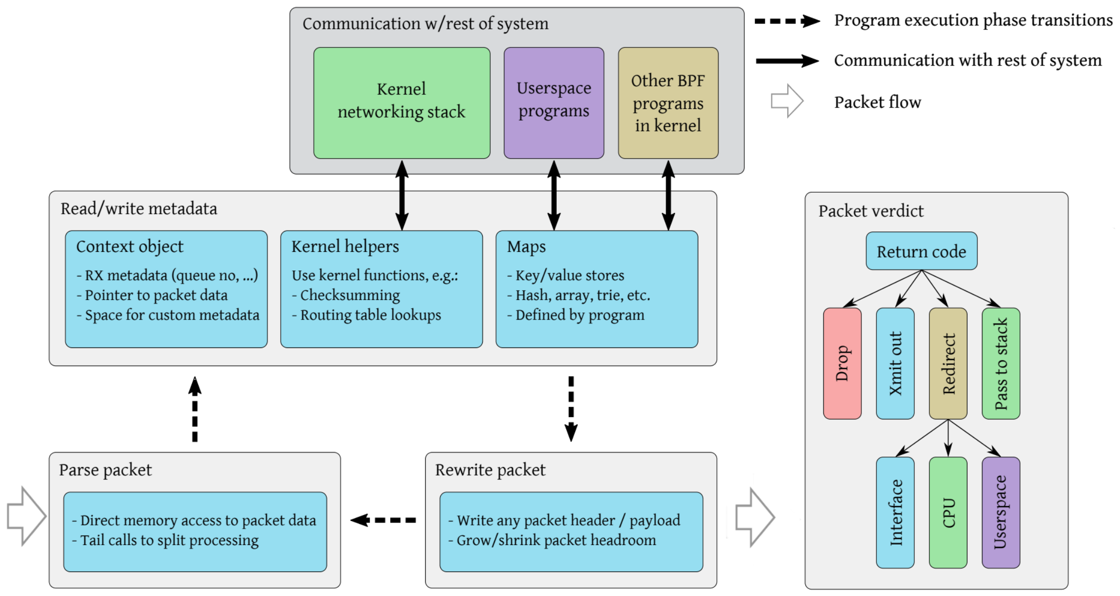

Communication w/rest of system
Kernel networking stack
Userspace programs
Other BPF programs in kernel
Program execution phase transitions
Communication with rest of system
Packet flow
Read/write metadata
Context object
- RX metadata (queue no, ...)
- Pointer to packet data
- Space for custom metadata
Kernel helpers
Use kernel functions, e.g.:
- Checksumming
- Routing table lookups
Maps
- Key/value stores
- Hash, array, trie, etc.
- Defined by program
Packet verdict
Return code
Drop
Xmit out
Redirect
Pass to stack
Interface
CPU
Userspace
Parse packet
- Direct memory access to packet data
- Tail calls to split processing
Rewrite packet
- Write any packet header / payload
- Grow/shrink packet headroom

Figure 5.3: Execution flow of typical XDP program [10]

A context of the received packet in the XDP hook includes a pointer to raw data, its length, and metadata describing which interface and queue was the packet received on, and the program typically begin by parsing these data. The context structure also gives access to a contiguous buffer resides in memory next to the packet data, where the program can attach its own metadata to the packet.

Maps in eBPF programs allow to communicate with the rest of the system (see more in section 5.2.3) and a persistent data can be changed depending on the current packet data. The packet and its metadata can be modified. The program can remove or rewrite any part of the packet, such as shrinking headers or rewrite address fields for forwarding and recalculate the checksum. To ease packet modification, the helper functions add existing kernel functionality without the need to go through the full kernel stack.

Finally, the packet verdict has to be made, and the program has to return one of the four codes that say how to deal with the packet [10]. No other parameters are returned, except the redirect verdict which requires an additional parameter that specifies a redirection target. For example, it is possible to specify a userspace socket (AF_XDP) as the target. The implementation of redirect function is very flexible (as map lookup), which means that a redirect target can be changed dynamically without program modification.

5.2.2 The eBPF virtual machine

The eBPF program is executed inside a specialized virtual machine resides in the kernel. The virtual machine performs just – in – time compilation of the eBPF instruction into native machine code and its execution. The VM exposes to the user a virtual processor, a set of eleven virtual CPU registers, a program counter, accumulator and a 512-byte memory stack and its RISC instruction set includes arithmetic and logic instructions and call instruction as well.

The virtual machine completely separates inside running bytecode from kernel space. The isolated environment of the virtual machine causes that bytecode cannot arbitrarily call other kernel functions or access into memory outside its own environment. To interact with rest of the system some helper functions can be called, depending on the type of the BPF program (see more in 5.2.4).

The main benefit of VM is that a user can dynamically load and inject eBPF programs without kernel reboot. All communication between user-space and VM is through a `bpf()` system call, which provides all control operations like loading programs, attaching them to specific events, creating eBPF maps and access the map contents from tools.

Another benefit is that the VM provides a stable ABI towards user space and guarantees that existing eBPF programs can be portable across different architectures and keeps them running with newer kernel versions. Moreover, the VM is build-in part of mainline Linux kernel distribution and there is no need for any third party kernel modules.

Registers

The set of VM's registers is listed in the table 5.1. The registers are always 64-bit wide (even if running inside a 32-bit ARM processor kernel), but they support 32-bit subregister addressing if the most significant 32 bits are reset [10]. Because registers R0-5 are reserved for function calls, the maximum number of function arguments is 5, and the first register always holds a return value.

Register	Function	x86_64 equiv
R0	return value from in-kernel function and exit value for eBPF prog	rax
R1	first arg to in-kernel function/scratch variable	rdi
R2	second arg to in-kernel function/scratch variable	rsi
R3	third arg to in-kernel function/scratch variable	rdx
R4	fourth arg to in-kernel function/scratch variable	rcx
R5	fifth arg to in-kernel function/scratch variable	r8
R6	callee saved registers that in-kernel function preserves	rbx
R7	callee saved registers that in-kernel function preserves	r13
R8	callee saved registers that in-kernel function preserves	r14
R9	callee saved registers that in-kernel function preserves	r15
R10	read-only frame pointer to access stack	rbp

Table 5.1: VM's registers and usage within eBPF program [21]

The operation set

The operation set of eBPF is around 100 instructions, and this number is continually increasing with the expansion of eBPF functionality in kernel releases. The operation of BPF is 64-bit wide to correspond to the 64-bit host architecture to perform pointer arithmetics and pass return values. There are three types of instruction: ALU instructions, memory instruction and branch instruction.

The instruction format is designed as two operand instructions, which helps to map BPF instructions to native instructions during the compilation phase. Due to this, the eBPF program can support helper functions that cooperate effectively with the kernel.

5.2.3 BPF Maps

The eBPF programs do not have access to persistent memory and every invocation starts in the initial state, so maps are the only way to communicate with the other parts of the system. In terms of XDP, the map is a data structure shared between the user – space and the eBPF program. Basically, the map is key/value store which exists in several different types: hashmap, array, queue, radix tree and so on and exists in two different variants also: global and per-CPU private.

A single BPF program can currently access up to 64 different maps directly, and they serve several purposes: coordination tools for change behavior; persistent data storage or communication mechanism, because data can be accessed on the user, kernel or eBPF sides.

Map implementations are provided by the core kernel and from the kernel point of view, BPF maps and programs are behaving as regular resources so that they can be only handled through file descriptors, backed by anonymous inodes in the kernel. To overcome limitation associated with descriptor sharing between processes, a lightway BPF filesystem in kernel space has to be used, and then multiple eBPF programs can be pinned to one map object as shown Fig.5.4.

Creating a BPF map is done by defining a global struct `bpf_map_def`, which includes type (hash, radix tree and so on), size of the key, size of the values and maximum allowed entries. Creating and loading maps into the program is the responsibility of user-space, but the kernel natively defined set of functions (helpers) which are available from bytecode and perform some complex interaction with maps as key lookup, update or delete items. The helpers also arbitrate access to maps and provide mutual exclusion, if it is needed.

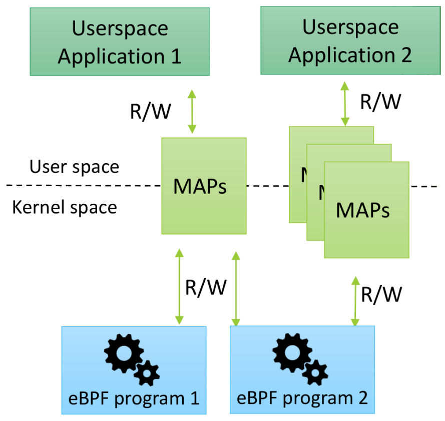


Figure 5.4: BPF maps and their interaction [21]

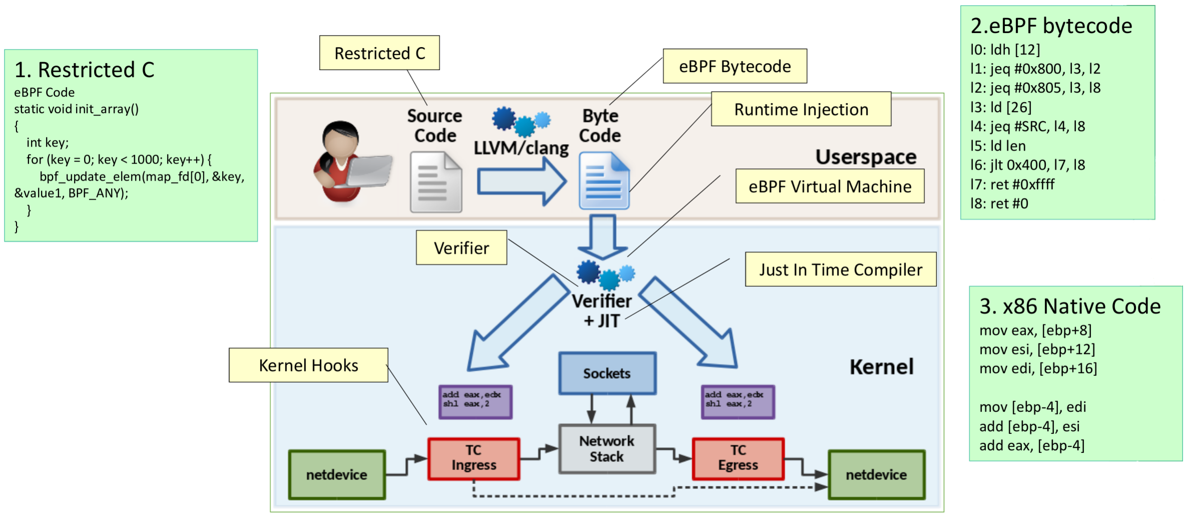
5.2.4 The BPF verifier

The eBPF program runs in kernel address space and arbitrary kernel memory might be corrupted, if malicious or buggy program will be loaded. There are many possible risks, which are unsafe: infinite loops could crash the kernel, buffer overflows, uninitialized variables, out of bounds jumps and so on. To avoid these, kernel build-in verifier performs a static analysis of the program byte code, when program is loading via `bpf()` syscall.

The verifier builds a directed acyclic graph of the control flow and ensures that the graph is truly acyclic, no unsafe memory access has been occurring and the code contains only supported and reachable instructions. This is checked by doing a depth-first search of the graph. Any program that contains unreachable instructions will fail to load. Then the verifier simulates the execution of the eBPF program one instruction by one. The VM's registers and stack have to be valid before and after execution. For example, a uninitialized register that has never been written to causes the program load to fail.

Finally, the verifier uses the eBPF program type to restrict which kernel functions can be called from eBPF programs and which data structures can be accessed. There are many types of eBPF programs that differ by where the program can be attached, which in-kernel helper functions will allow to being called, whether network packet data can be accessed directly, and the type of object passed as the first argument to the program. In the case of XDP, when the program is loaded via syscall, the type of program has to be set to BPF_PROG_TYPE_XDP.

If verifier doesn't prove that byte code is safe, then it will terminate the program loading. Also, the total program size is limited, and verifier has to ensure that it is not exceeded.



```
eBPF Code
static void init_array()
{
   int key;
   for (key = 0; key < 1000; key++) {
      bpf_update_elem(map_fd[0], &key,
&value1, BPF_ANY);
   }
}
```

```
l0: ldh [12]
l1: jeq #0x800, l3, l2
l2: jeq #0x805, l3, l8
l3: ld [26]
l4: jeq #SRC, l4, l8
l5: ld len
l6: jlt 0x400, l7, l8
l7: ret #0xffff
l8: ret #0
```

```
mov eax, [ebp+8]
mov esi, [ebp+12]
mov edi, [ebp+16]

mov [ebp-4], edi
add [ebp-4], esi
add eax, [ebp-4]
```

Figure 5.5: eBPF: overview of the runtime architecture [21]

5.3 Creating eBPF program

There are many ways to create a BPF program. One method is that the user writes his eBPF program directly using the eBPF assembler in Intel-like assembly syntax [19, 7]. The instruction set is available also as macros defined in *bpf/libbpf.h* in the kernel source tree, and C syntax can be used.

Another method is to implement a program in high-level languages such as C and let the compiler translate the code to eBPF and hide assembly instruction for user. The eBPF community selected the LLVM Clang compiler for such a task. The compiler generates the ELF object file which can be loaded using `bpf()` syscall to an eBPF virtual machine. Due to eBPF program restriction mentioned above, only the restricted C can be used.

To make compilation easier, there is a BPF Compiler Collection. It is a toolkit for writing, compiling and loading eBPF programs in C, Python and Lua. Even more, bcc provides nice object-oriented bindings when working with maps and includes many tools useful for tracing [7]. The eBPF program is possible to load via iproute2 or perf [7, 8].

Libbpf

Another helper for writing eBPF program is the Libbpf library. It is a generic library inside kernel source tree which performs wrapping function for loading (`bpf_load_program()`), reading and manipulation with eBPF objects from user-space to ease writing eBPF programs in C [7].

Very useful functions inside libbpf are wrappers for working with AF_XDP socket (libbpf/xsk.c and libbpf/xsk.h). They offer APIs for low-level access to the packets rings and its data and high-level control plane for creating and setting up UMEMs and AF_XDP sockets themselves. A simple eBPF program for socket utilization is included, so the adoption of AF_XDP to new or existing programs is very comfortable.

The kernel tree also provides some neat examples (located in samples/bpf/) which show how to use the libbpf. User can link the library statically or as a DSO. The library is used by other kernel projects such as perf or bpftool, and it is dual-licensed under the LGPL 2.1 and BSD 2-Clause [7].

For debugging and introspecting BPF programs and BPF maps, a user can use a *bpftool*. It is a tool developed by the Linux kernel community like libbpf. The bpftool allows dumping all active eBPF object in the system or disassembling JITed BPF instructions [7].

5.4 AF_XDP

AF_XDP (previously known as PACKET_V4) is a new address family based on XDP layer benefits, designed to pass network traffic from the driver up to user-space as fast and efficiently as possible. Fig. 5.6 illustrates a comparison between AF_XDP socket and standard AF_INET socket: the tradition network stack is bypassed before SKB allocation. The core idea behind is to use the `XDP_REDIRECT` action and `bpf_redirect_map` function when packets arrive on NIC to redirect them to the user-space socket (AF_XDP socket, also called XSK). Redirect can be done without any copy, so this socket is able to deliver a raw packet from NIC to the user-space very fast [14, 23].

5.4.1 Driver support and zero-copy mode

AF_XDP socket (also called XSK) can operate in three different modes depending on NIC capabilities:

1. Generic **SKB** mode
2. Native **XDP_DRV** mode
3. **XDP_DRV+ZC** mode

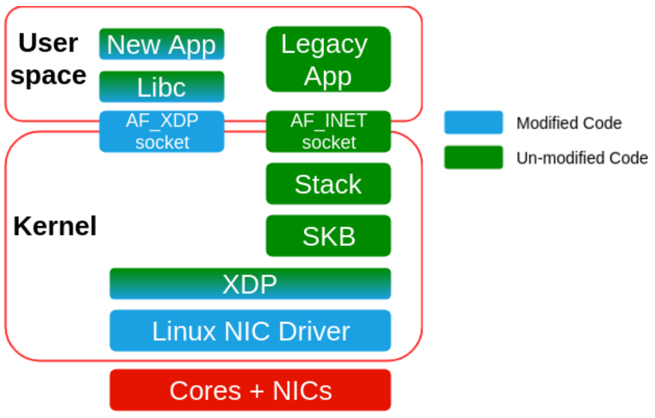


Figure 5.6: AF_XDP socket vs. standard socket cite

The Generic mode works on any NIC, and for this mode, the hook point is not at the lowest level of datapath, but only after an skb allocation. The Native mode works on all devices with XDP hook support; the XDP_DRV+ZC is a native zero−copy mode requiring special driver support.

The zero-copy mode has a higher performance because the DMA stores data in a user allocated frames. Otherwise, the kernel has to allocate the memory and copy the frames to the application. The ZC driver needs to implement and expose the API for using the memory area directly in the NIC RX-ring structure for DMA delivery and nowadays, only ixgbe and i40e drivers[1] by Intel do that (Mellanox coming soon). Table 5.2 shows the current driver's XDP support; zero-copy support is highlighted with an asterisk.

Vendor	Driver				
Broadcom	bnxt				
Cavium	thunderx				
Intel	ixgbe*	ixgbevf	i40e*		
Mellanox	mlx4	mlx5			
Netronome	nfp				
Qlogic	qede				
Solarflare	sfc				
Marvell	qede				
Others	veth	virtio_net	net	tun	dpaa2

Table 5.2: XDP support, zero-copy support with asterisk [7, 13]

[1]Using a ZC requires a NIC driver from vanilla kernel tree, the out of tree drivers do not contain the XDP-ZC support

5.4.2 A socket redirect

A new type of map makes it possible to perform redirect packets to user-space. It is called XSKMAP (or BPF_MAP_TYPE_XSKMAP in full), and it is a simple array containing a file descriptor corresponding to one AF_XDP socket. As mentioned earlier in section 5.2.3, the map is key/value store, and in case of an array, the key is array index of increasing integers and the value is an array item.

A process can create a socket with an attached memory buffer and push socket's file descriptor in the XSKMAP via `bpf()` call. Actually, an internal kernel descriptor is stored in the map, but from an application point of view, it is not visible.

A BPF program loaded into the driver can redirect a packet to an arbitrary descriptor in this map, and XDP has to validate if the descriptor is indeed bounded to the device and some queue. If the chosen index has not passed the validation, then the packet is dropped. The packet is also dropped when an item on the chosen index is empty. In the opposite case, the packet will be directed to receive queue corresponding to the AF_XDP socket in the selected map entry. Thus, it is mandatory to have an eBPF program loaded and have at least one entry pushed in the map, while an AF_XDP socket is used. Otherwise, the application will not be able to get any traffic through AF_XDP socket.

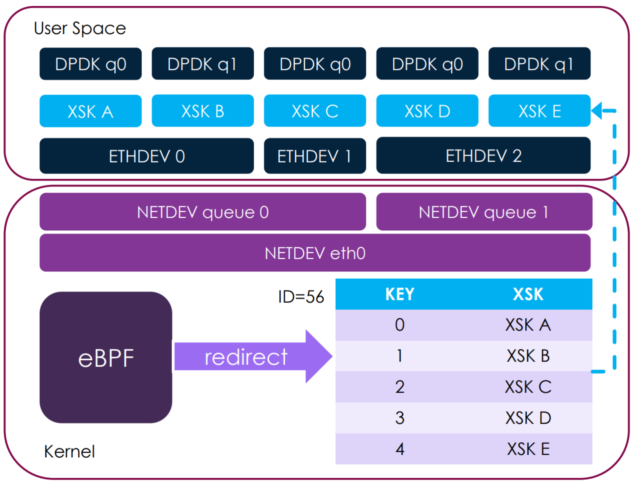


Figure 5.7: Example of BPF_MAP_TYPE_XSKMAP redirecting in DPDK [20]

5.5 Memory model

The memory model used in AF_XDP has a great credit for the performance boost because there is no memory allocation per packet. All packets within XSK are held in pre-allocated memory called UMEM. Moreover, RX and TX queues can share the same UMEM and packet descriptors are separated from packet buffer.

The UMEM buffer is contiguous memory area divided into several equally sized chunks called *packet buffers* in which a single packet and its metadata can be stored. Every chunk is identified by an integer index (also called *descriptor*), which is a relative offset from the UMEM begin, masked to the power of two. E.g., for a chunk size of 2k, the $log_2(2048)$ LSB of the address will be masked off; it means that 2048, 2050 and 3000 refer to the same chunk. Indexes are used in communication between kernel and application to tell each other where a particular packet resides. For example, when a new packet is stored in packet buffer and kernel wants to give this data to the application, there is no need to data copy or any manipulation, the only thing the kernel has to do is pass the proper chunk index to the application.

For this reason, four buffer rings are used together with AF_XDP socket and into which indexes can be written. One pair of rings is associated with the socket and it is responsible for sent and received packets, one pair of rings is associated with UMEM buffer, and its purpose is handling chunk ownership. All rings behave as single-producer/single-consumer, who is the producer depending on particular ring vocation.

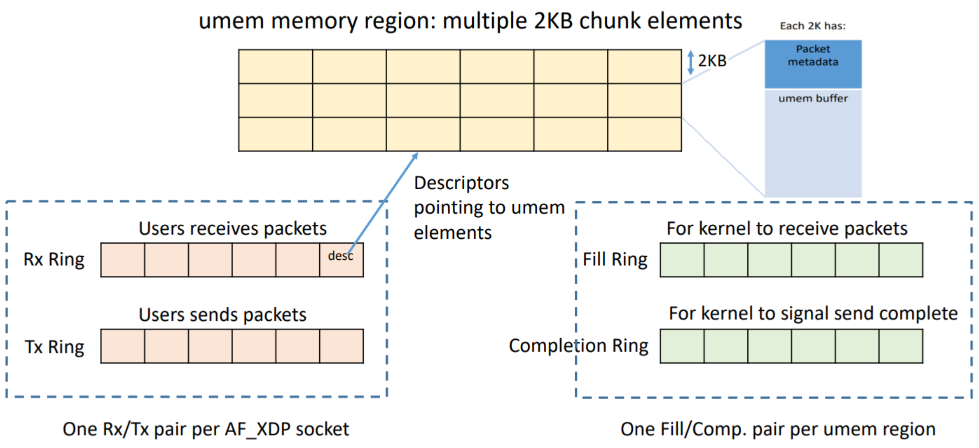

umem memory region: multiple 2KB chunk elements
2KB
Each 2K has:
Packet metadata
umem buffer
Descriptors pointing to umem elements
Users receives packets
Rx Ring
desc
Users sends packets
Tx Ring
For kernel to receive packets
Fill Ring
For kernel to signal send complete
Completion Ring
One Rx/Tx pair per AF_XDP socket
One Fill/Comp. pair per umem region

Figure 5.8: UMEM with four rings: RX and Fill rings are used for recieving, TX and Comp. rings are used for sending packets [24]

The signalization about a chunks ownership between the kernel and the user-space application is provided by buffers called Fill and Completion rings. Within Fill ring, the application is the producer who wants to report the kernel-consumer which chunks are available for new incoming packets. On the other side, within Completion ring, the kernel is the producer who tells the application which chunks can be used for outgoing packets.

Another pair (associated with socket) handles incoming and outcoming packets. The indexes of received packets are stored in a ring called *RX ring*. By checking this buffer, an application can indicate if it has received a new packet. The indexes of outgoing packets are stored in a ring called *TX ring* and an application fill in indexes of packets ready to be sent.

Each socket is bound to one umem which can have single Fill and single Completion ring, but one socket may include multiple Rx/Tx rings. Even if zero−copy mode has been used, the RX and TX descriptor queues are not shared to user-space. Only the kernel can manipulate them, and it is the kernel driver's responsibility to translate hardware specific descriptors to descriptor rings that user-space sees. This way, a malicious user-space program cannot mess with the NIC.

The use of rings

When a packet is received by NIC and kernel driver pick it up, then the XDP program is executed and decided if the packet should be pass to a socket. If zero − copy mode has been used, the DMA has already put the packet in the UMEM area in user-space, so kernel only fill in the packet descriptor to RX ring. The application checks the Rx ring for new items, and if a new index appears in RX ring, it processes the data behind the pointer. When the application has finished processing, it has to return a packet buffer ownership to the kernel to reuse the memory. It will be done by adding the particular index to the fill ring so that the kernel can see which chunks are available for a newly arrived packets and where a new packet data can reside. Thus, the RX ring and the Fill ring must be involved in the receiving side.

The TX path works similar, but TX and Completion rings have to be used. When the application has a packet ready to send, it fills out the next available descriptor in the TX ring to notify a kernel, which packet buffer wants to send. Then the kernel sends the packet to hardware, and after the packet has been successfully sent, the kernel writes a used memory index back to the Completion ring.

In summary, the RX and Fill rings are used for the RX path and the TX and Completion rings are used for the TX path. A schema of using is illustrated in Fig. 5.9.

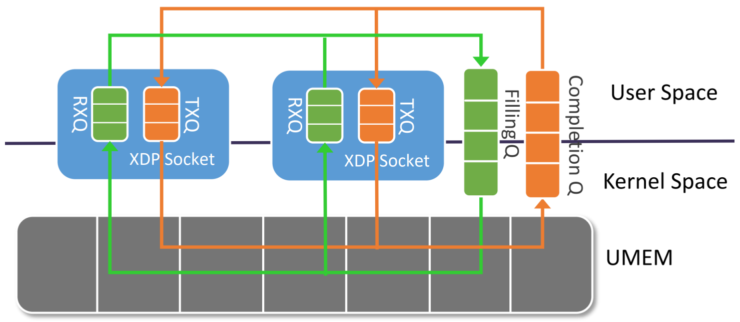


Figure 5.9: The ring cooperation, green arrow for recieve side, red for send side

5.6 Socket creation

The XSK can be created via standard syscalls as `socket()`, `setsockopt()` and `bind()`, but all memory management is directed by the user, so some memory allocation has to be done before binding. The followings list is a sequence of necessary operations and their function equivalents in C, if a fully functional XSK should be created:

1. Socket creation - `socket()`
2. UMEM allocation - `malloc()`
3. Registration UMEM socket - `setsockopt(XDP_UMEM_REG)`
4. Creation of rings - `setsockopt(XDP_UMEM_FILL_RING)`
5. memory map to user-space - `mmap()`
6. binding socket to interface and particular queue - `bind()`

The socket is created by usual `socket()` syscall with `AF_XDP` as domain parameter. After that, the memory area, where packets will be stored has to be allocated. It depends on the user what function this will be done (malloc, mmap, huge pages, etc.), but it is mandatory to register allocated memory to the kernel through the `setsockopt` function with `XDP_UMEM_REG`.

As mentioned above, the RX and Fill rings are used for the RX path and the TX and Completion rings are used for the TX path so at least one couple (or both) has to be created with UMEM. The creation of these rings is possible via setsockopt calling with one of the `XDP_UMEM_FILL_RING`, `XDP_UMEM_COMPLETION_RING`, `XDP_RX_RING` or `XDP_TX_RING` parameters for a particular ring and its size (the level parameter is set to `SOL_XDP`). The size of the rings needs to be of size power of two. The setsockopt than allocates and set up the particular ring.

Next step is map memory buffers to user-space. This mapping is done by using the `mmap()` function described in 4.1.2. Before that, the application has to request kernel about socket structure to figure out the actual addresses where all used rings begin because the ring's structure is highly optimized to reduce cache coherency and looks different depending on architecture. However, the structure is returned to `setsockopt()` call with a `XDP_MAP_OFFSET` parameter and then the `mmap()` can be called. The map to user-space is required as shared read and write for all of the used queues and specified by `XDP_PGOFF_TX_RING`, `XDP_PGOFF_RX_RING`, `XDP_PGOFF_FILL_RING` or `XDP_PGOFF_COM_RING` parameters.

Finally, the socket has to be attached to a particular interface and queue number, from which the packets will be received or transmitted, by the `bind()` function call.

Chapter 6

Design

The first section of this chapter analyses a specification of the packet capturing probe that will be implemented and in the following section is suggested a possible solution and the main logical steps that need to be done by the application.

6.1 Specification analysis

The flow probe should be able to capture traffic the highest possible speed, at best 14.8 Mpps. The probe has to support IPv4 traffic as well as IPv6 on the Ethernet layer. The application should be able to maintain a flow cache of unidirectional flowd; it means to store aggregated information about connections, which captured packets belong to. It requires a network and transport layer information, so raw packet data has to be captured and socket settings have to be adapted to that. The flow subprocesses must be able to make a fast data lookup within the flow cache. The exporter may not have adopted a full flow exporter protocol support and full template support is not required. The probe should be portable, non – blocking, and compatible with other applications running on the Linux operating system.

6.2 Architecture

The packet capturing at higher rates would not be possible using a standard network stack. It is appropriate to use one of the fast packet processing frameworks described in section 4.3. XDP framework meets requirements from previous section 6.1. XDP is supported in the new kernel source tree and does not need to install another library or kernel module. Therefore, it can be used in arbitrary Linux – based system with kernel version 4.19 or higher. The XDP supports AF_XDP socket which bypassing network stack to avoid its limitations, such as per-packet allocation or long datapath through the stack. AF_XDP offers a direct path from the network card to the application. Moreover, the kernel itself is not bypassed so kernel functions are still available as well as in-kernel NIC driver and interface are still accessible to other applications.

Multithreading

As discussed in chapter 4, the most efficient performance improving idea is to distribute traffic among multiple CPUs. Modern network cards support RSS so that the traffic can be split into multiple queues and every queue can be processed by a separated CPU. This

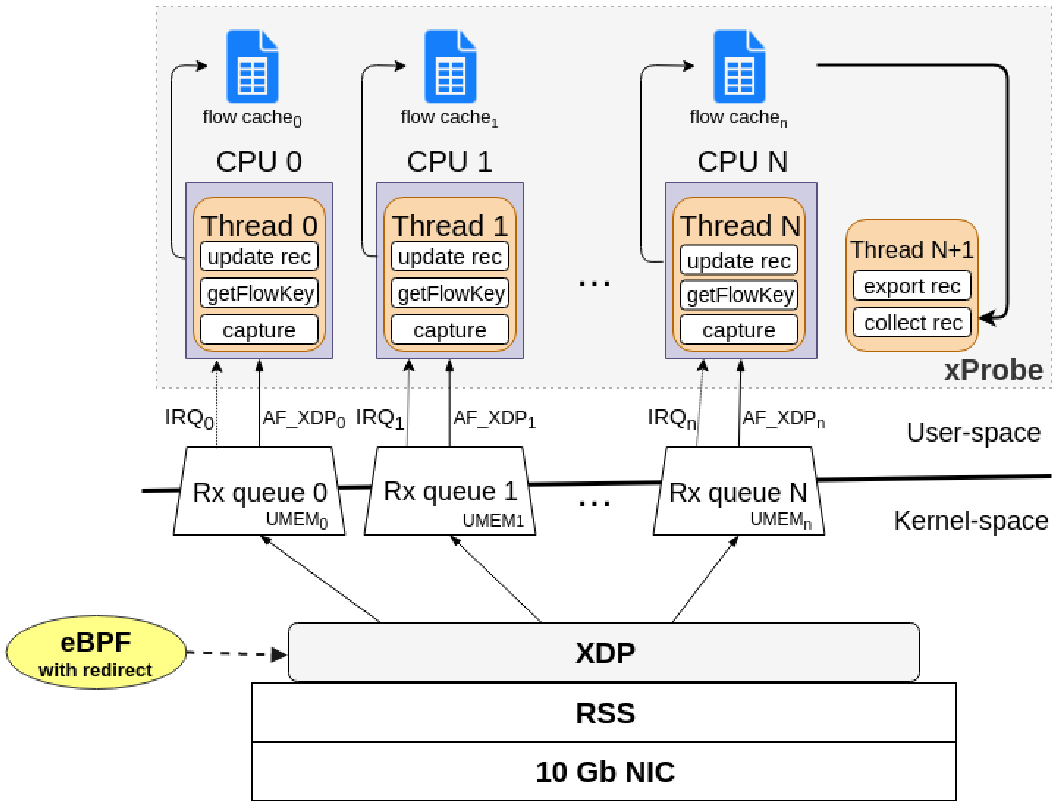


Figure 6.1: xProbe architecture

concept is scalable, and the traffic load can be suitably spread across processors, depending on the number of CPUs and NUMA cores.

The suggested probe invokes as many threads capturing traffic as RSS queues and each thread resides on one CPU using thread affinity. Each thread opens its AF_XDP socket, which binds to one of the queues, so that each thread can handle one of the queues, and traffic processing can be spread out between CPU cores.

Shared buffer

For direct access to the ring memory (RX and Fill buffers), the thread uses the `mmap` function to memory. Then, the probe, which is running in user-space, can share the buffer with the kernel. This approach highly reduces system calls generated by application and context switching between address spaces.

Flow maintaining

Each capturing process maintains its flow cache to store information about communication. Because of splitting traffic to separated queues, each process will have only one part of traffic belonging to the particular queue. The RSS distribution among queue is based on IP addresses and port numbers, thus packets belonging to the same communication will always be in the same queue. It is more auspicious than to maintain one big cache for all processes. It can reduce a flow cache size and access time to cache item.

The IPFIX was chosen as a flow exporting protocol. It offers variable length of elements and native support to IPv6 addresses. Moreover, it is standardized by IETF, and it is compatible with proprietary NetFlow9.

Architecture scheme

Figure 6.1 shows the architecture scheme of the suggested probe. At the bottom (and the first in a packet datapath) is a network card and its built-in RSS, which spreads traffic between multiple queues. Inside the NIC driver is added an XDP layer, which allows execution of validated eBPF code. The eBPF is executed for every packet that has been received and has to make a decision to which socket will be the packet redirect to, depending on receiving queue. If the AF_XDP socket in user-space is opened, the packet is redirected to UMEM of the particular socket (illustrated as a trapezoid) and certain buffer descriptor is added to the RX ring, and `POLLIN` event is raised (the IRQx arrow). The thread that owns the UMEM buffer can read and parse the packet data (AF_XDPx arrow), and the flow key and metadata from packet header can be extracted (the „capture“, „getFlowKey“ rectangles). After packet processing, the metadata are used to update a flow record within the flow cache and each thread has its flow cache (blue tabular). There is an control thread, which ensures aggregating of flow cache and export to the collector via IPFIX.

6.3 Packet capturing

Each thread reading data from one of the queues, so the thread needs to create and set up an AF_XDP socket and binds it to a particular queue. As was mentioned in chapter 5.6, the setting up AF_XDP socket includes the creation of a socket, allocation UMEM buffer, creation the proper rings and bind. The sequence diagram is enclosed in Appendix A.

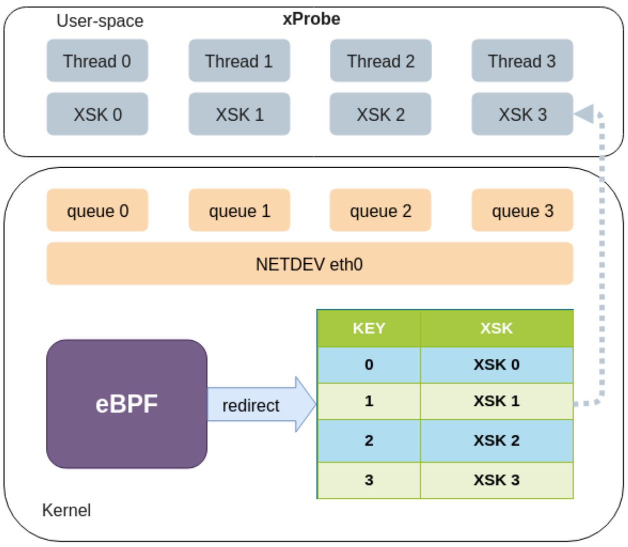


Figure 6.2: XSKMAP redirection on xProbe sockets

The socket creation is similar to AF_INET socket creation and can be done via a syscall. The process has to create a UMEM buffer, where the packet data will be stored. The UMEM buffer allocation must be ensured by the process in user-space and must be mapped to be shared between user-space and kernel-space. Within the monitoring probe, the packets are only received so there is no need to create Completion and TX rings. The application has to set up only RX and Fill rings to be able to receive incoming packets. After that, the socket has to be bound to the interface and the particular queue. The application will use a poll function to detect an event on a socket. The poll function blocks the process until the kernel will raise the `POLLIN` (data ready to read) event or timeout expired, and then the data can be processed.

It is necessary to load the eBPF program before the socket creation has begun so that the socket can properly receive data. The eBPF code has to maintain an XSKMAP with open socket indexes and redirect packets to a certain socket in user-space, depending on which queue the packet belongs to, as is shown in Fig. 6.2. The socket indexes are pushed into the XSKMAP, at the moment of socket creation, so the map has to be done earlier.

The AF_XDP socket can be used in zero-copy mode, if the probe runs on supported NIC. Then the DMA stores packet data directly in the UMEM area.

6.4 Flow creation

The probe regards the following 5-tuple as a flow key, which uniquely identifies a communication between two points:

1. destination IP address
2. source IP address
3. destination protocol number
4. source protocol number
5. protocol in IP header

The flow key and metadata is stored in the flow record. The observed metadata is as follows:

1. number of packets
2. number of bytes (heads and payload)
3. number of payload bytes
4. byte histogram
5. the lowest TTL
6. IP layer flags
7. IP layer options
8. transport layer flags
9. communication start timestamp
10. communication end timestamp

The packet headers have to be parsed after the packet data has been captured and stored in UMEM. The subprocess providing parsing has to select flow key from IP header and transport header, but data link headers need to be parsed as well because VLAN tags may occur.

Parsing

The first header in raw packet data is Ethernet header (without preamble). The only element that needs to be checked is the ethernet type field. The ethernet type specifies the type of the following header. There are two main values of the field: `0x0800` value indicates an IPv4 header, and the `0x86DD` value indicates an IPv6. But, there is also a VLAN tagging which is indicated by `0x8100` or `0x9100` value. Ethernet header has a fixed length of 14 bytes, but in the case of VLAN tagging it can be 4 bytes longer or even 8 bytes longer, in the case of the Double tagging (IEEE 802.1ad) and the first byte of the IP header is given by this size.

The IPv4 header is not fixed in size; it consists of 14 fields, of which 13 are required, and the last one is optional and flexible. The length of IPv4 header (this also coincides with the offset to the data) is encoded to 4 bits in IHL field. To get the exact size of the header, the parser has to multiply the IHL value by 5. The IP header contains three of the five flow key attributes: destination IP address, source IP address and protocol number, which has to be exctract. As metadata attributes are stored TTL, Flags, total length fields.

In the case of `0x86DD` value in the Ethernet type field, the IPv6 header has to be parsed. The IPv6 header is fixed in size but can be concatenated with extension headers. The main header includes source and destination IP addresses which has to be selected as flow key attributes, but protocol number attribute has to be taken from last extension header so the parser must iterate to the last header by checking *Next header* field. Flow record attributes can be taken from the main header also (*Hop Limit* like *TTL*).

The next header in the order is the transport header. The parsing subprocess can recognize UDP, TCP, ICMP, ICMPv6, IGMP headers. A destination port number and a source port number has to be extracted from the transport header. Some transport headers such as ICMP, ICMPv or IGMP do not contain a port number so a header type and message code can be used instead. In the case of TCP, the flags have to be extracted to detect connection termination, once the FIN flag has occurred.

Flow cache

Demanding part of monitoring probe is the way how to store flow records. The data structure holding flow records has to be able to access elements efficiently and quickly. For this purpose, I choose a hash table, which search complexity may be at worst case $O(n)$, depending on a hash function. Very fast hash functions suitable for hash-based lookup offers MurmurHash family [11, 2]. I suggest using the current version MurmurHash3 in 32-bit variant, which is faster than previous versions. The function uses a multiply (MU), rotate (R), XOR and several magic constants on 128-bit blocks of data in its inner loop. The input of hash function will be a flow key data structure, and the hash function produces a 32-bit hash value that will index the flow cache table. The table will contain pointers to flow records in memory (as shown in Fig. 6.3).

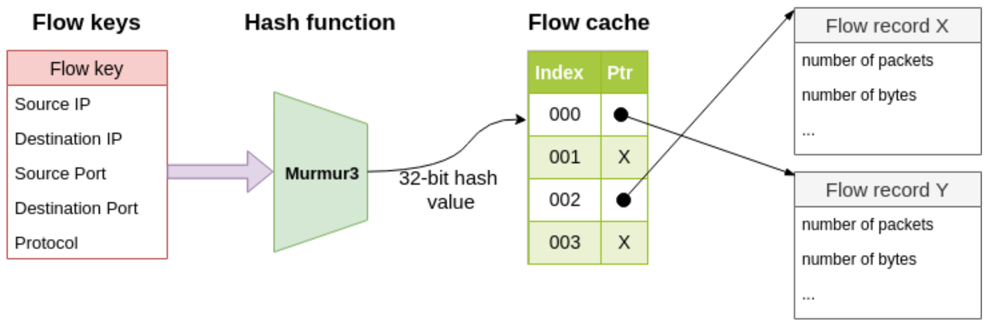


Figure 6.3: Flow cache and hash function scheme

6.5 Flow record export

The probe creates an extra thread to provide a record exporting. The thread periodically serializes records stored in the flow caches and exports data to output. The user can specify a time interval between exporting. Additionally, the thread periodically prints socket statistics to determine sockets utilization and performance.

Chapter 7

Implementation

This chapter is devoted to the actual implementation of the traffic capturing probe designed in the previous chapter 6. The implementation language was chosen as GNU C/C++. The program periodically captures the traffic from multiple NIC queues of a given interface and maintains the flow cache up-to-date. For implementation was used a *Libbpf* and *pthread* library. As a starting point for implementation was used a `/samples/xdpsock_user.c` example from vanilla kernel tree, which demonstrates XSK socket handling.

7.1 Multithreading

A *pthread* C/C++ library is used for creating multiple processes which provide a packet processing. The application holds identifiers of all created threads in an array inside `xProbe_info` structure. The thread array is allocated to `MAX_SOCKS` size so it can hold only `MAX_SOCKS` threads. The number of threads is given by `-p` argument, (see below on section 7.8), but the argument is limited by the `MAX_SOCKS` constant, and the application does not create more than `MAX_SOCKS` threads and sockets.

The main thread is creating subprocesses in `xProbe_configure_and_run` function in a loop and then waiting to their terminations via `pthread_join`, which is placed in `xProbe_wa-it_to_end` function. The thread executes an `XDProbe_thread` function, whichin the thread performs a socket creation, packets capturing and processing and resource release. When the application creates a new thread, the `thread_input_t` data structure is put to the new thread as a parameter. The structure containing basic information about the new thread: identification within the probe (`tid`), the descriptor of a queue (`if_queue`), from which the thread will be capturing packets and pointer to socket descriptor array (`xsk`), where all socket descriptors are stored.

There is created one extra thread, which periodically prints statistics and its code is defined in `poller` function. This thread opens a file, where periodically prints content of flow caches, and at the same time it prints socket statistics on standard output. The flow cache is reset after exporting by this thread.

The packet capturing processes has to be explicitly assigned to one CPU. It is done via `pthread_setaffinity_np` call after the thread has been created. For distribution packets among CPU is used RSS on NIC. The RSS rule sets by an `ethtool` command in `src/optiNIC.sh` script.

The socket creation inside threads has to be synchronized. The eBPF program has to be hooked to the interface before the first socket creation and only one eBPF program can be hooked on interface. Otherwise, the creation will fail. Due to this, I used a `xsk_creation_mtx` mutex that serializes a socket creation, and the first locked thread attaches an eBPF program to interface while the other threads are waiting. When eBPF is successfully attached by the first thread, the other threads do not try to hook it again. Without this mutex, it often happened that two thread tried to attach an eBPF at the same time, and it caused a hook error.

The application also using two barriers to synchronize socket initialization. The `xdp_ready` barrier is used to wait for each other after successful socket creation. Second barrier `xsks_ready` synchronized threads with the main process and indicates the moment when all threads are ready to packet processing.

7.2 eBPF

The cornerstone of the XDP system is an eBPF program, which is executed inside the driver. When an application wants to use an AF_XDP socket in user-space, the eBPF program that performing an `XDP_REDIRECT` action must be running inside the driver. Such eBPF program maintaining an open sockets map and redirecting packet to socket based on queue, on which the packet has been received.

The assembler instructions can be used to write the redirecting eBPF program, and the application can attach the program to interface via `bpf_load_program` and `bpf_set_link-_xdp_fd` calls. But the Libbpf library offers a wrapper which does that. The `xsk_load-_xdp_prog` function from Libbpf defines a simple eBPF program ensuring proper redirecting action and attached the program to the interface. It is a simple way to use an AF_XDP socket without prior knowledge of eBPF programming. The implemented probe uses this wrapper because the application does not need any other eBPF features and basic redirecting, which offers this build-in program, is sufficient.

The redirecting eBPF program has to be detached when a signal interrupt is sent to the application. An identification of hooked program can be recognize with `bpf_get_link-_xdp_id` function. The unattached is done by calling a `bpf_set_link_xdp_fd` with a -1 value as the second parameter. The maps has to be destroyed as well. Due to this, the `xsk_clear_bpf_maps` and `xsk_delete_bpf_maps` functions are called.

The eBPF maps use lock memory, which is very low by default and application needs to increase resource limit `RLIMIT_MEMCLOCK` with `setrlimit()` call. The limit is set on application start to `RLIM_INFINITY`, it means no limit for lock memory.

7.3 Memory model

The AF_XDP uses a different way of storing packets than a standard socket. It uses shared UMEM area, which is managed by user-space (described in section 5.5). Within the probe application, the `xsk_umem_info` data structure represents an UMEM memory model. It contains a pointer to a memory area, its size, and pointers to receiving rings which manage the incoming packets.

The application allocates UMEM memory via `posix_memalign` because allocated memory needs to be aligned to page size. The size of allocated memory is set to `XSK_UMEM_-_DEFAULT_FRAME_SIZE * NUM_FRAMES`, the first constant represent packet buffer size and the second constant gives a total number of packets that can be stored in one UMEM. The packet buffer size is set to 2048B, because one packet buffer contains a headroom along with frame data buffer. The maximal frame size can be 1500B and the headroom length is 320B so theirs summary is rounded up to power of two to 2048.

The UMEM buffer is created in user-space, and the application has to register memory buffer with the socket. For this purpose, the application using the `XDP_UMEM_REG` command of the `setsockopt()` system call and then the UMEM serving as socket buffer. The creation and register memory are done in the `xsk_configure_umem` function.

The application using an RX and Fill rings to manage receiving packets. The rings are created in conjunction with socket creation, which is described in the next section. The rings reside in kernel-space and the kernel has to share them with the probe. For this purpose, the page remapping is used, and `mmap` function with ring pointers as parameters are used to remap a memory.

The `munmap` function is called to all mmaped regions when the probe is interrupted, as well as the UMEM memory is freed.

7.4 Socket creation

Each packet capturing thread creates a separate socket with an independent packet buffer. As was mentioned in section 5.6, the socket creation includes several parts and the whole procedure is implemented in the `xsk_configure_socket` and `xsk_configure_umem` functions, where Libbpf wrappers are called to socket creation.

First, the socket structure has to be created with `socket` syscall. The socket type is defined as `RAW_SOCK` with default protocol number 0, and the `AF_XDP` value (44 in real) is given as a domain parameter. Then the UMEM is registered with the socket as was described in section 7.3.

After that, the Fill and RX ring are created with `setsockopt` syscall. The `SOL_XDP` level (value 283) has to be set when the syscall manipulating with any XDP options. The setsockopt option name creating a Fill ring is an `XDP_UMEM_FILL_RING` and an `XDP_UMEM_RX_RING` for creating the RX ring. The size of rings are defined in `FILL_RING_SIZE` and `RX_RING_SIZE` constants, and both are sets to 8192 descriptors. This number corresponds to the maximum number of network card descriptors (`ethtool -g` command).

As mentioned earlier, the application using the page remapping to share rings between address spaces. After the rings creating, the mmap function with queues descriptors is called to propagate memory to user-space. But the application does not know the actual pointer to rings data, so the `setsocktopt` with `XDP_MMAP_OFFSETS` command has been called before. It returns a data structure with memory offsets, on which the rings actually reside. These offsets are used as parameters to mmap function.

Finally, the bind syscall is issued, which assigns the socket to interface and particular queue. The interface name is given by the user as an `-i` argument, and the interface is the same for all packet capturing threads. On the other hand, the queue id is unique to all threads and it is specified by thread's `tid` variable. It means, that the first packet capturing thread with a $tid = 0$, will be bound to the queue with a $id = 0$, the thread with $tid = 1$ will bind the second queue ($id = 1$) and so on.

When the socket is ready to receive packets, the packet buffer indexes must be written into the fill ring to mark available chunks. It is done via Libbpf wrapper `xsk_ring_prod_-_fill_addr`, which is called in a loop to fill up Fill ring.

7.5 Packet capturing

The thread is using a `poll` function to detect when a new batch of packets is ready to process. The thread registers the poll to socket's `POLLIN` event and waits until the event occurs or a timeout expires. When the new packet is ready to process, the thread reads its location and its length from RX ring by an `xsk_ring_cons__rx_desc` call. The chunk descriptor is used to index a UMEM buffer, but it is not real address in memory. To get real data pointer is used a `xsk_umem__get_data` function, which returns a pointer. Then the data pointer is given as parameter to a `frame_parser` function, which provides packet processing. After the packet processing, the chunk ownership has to be returned back to the kernel, so the chunk index is written to the Fill ring via `xsk_ring_prod__fill_addr`.

7.6 Packet processing

The packet processing includes a flow key and metadata extraction from a raw data; flow cache lookup and updating flow record. The extraction provides a `frame_parser` function, which gets a pointer to a raw packet and its length as the parameter. Firstly, the Ethernet header is decapsulated by checking 12th and 13th bytes of data, where the frame type field resides. These two bytes are used to switch condition to determine the type and length of the data link header. The Ethernet header length coincides with the data offset, where the next header begins.

The next header to decapsulate is an IP layer header, which contains a source and destination IP addresses that are used as the flow key attributes. The application can recognize only two types of IP headers: IPv4 header or IPv6 header. A data pointer is type casting to a header data structure, which represents IP headers, and particular fields are copied. The data structures are taken from *linux/ip.h* and *linux/ipv6.h* header files.

If the IPv4 header contains any options, the option type is recorded to bit field. In the case of IPv6 header, the header extensions can be placed after the main header, and parser needs to proccess them as well. The parser iterates through extensions until the `NEXTHDR_NONE` (59) constant occurs in the header type field.

The port numbers are extracted from transport layer headers similarly as IP addresses. The pointer to data is type casted to a header data structure. The data structures are taken from *linux/*.h* header files as well. The TCP option flags are stored in flow record as bit field and extracted flags are aggregated by OR function to the bit filed. After parsing L3 header are metadata stored in flow cache.Figure 7.1 shows a data structure used for the flow key and flow record.

7.6.1 Flow cache

The flow cache is implemented as hash table. The `hash_table_t` data structure represents a flow cache and holds information about cache within thread.

The index of hash table is given by `flow_key_hash` function, which implemented a Murmur3 hash function. The hash seed is random generated when the application starts and it is shared in every flow cache. The input of hash function is `flow_key_t` data structure, which represents a flow key (Fig. 7.1). A value returned from hash function is divided modulo size of flow cache.

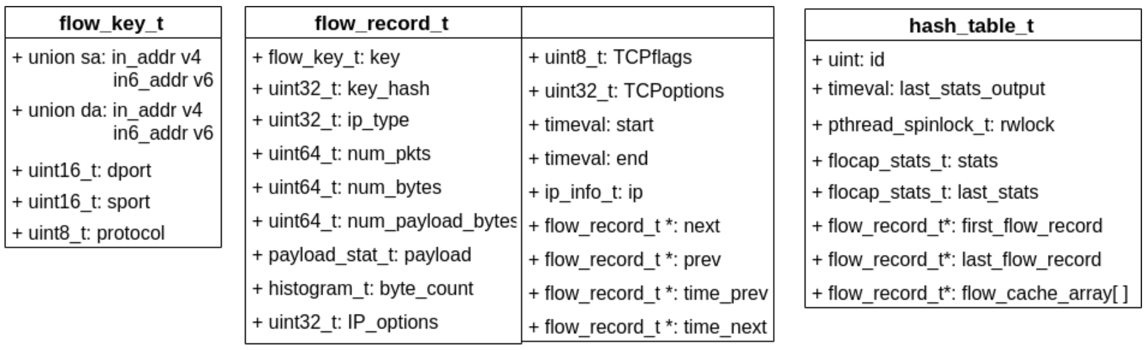


Figure 7.1: Flow key, flow record and flow cache data structures

The hash table contains lists of flow records in order to store records with same hash value. The cache lookup returns a pointer to the first flow record of record list and linear iteration through the list has to be done to find out specific record.

The flow cache maintains a chronological order of stored records. Each record has a `next` and `prev` pointers to the chronological predecessor and successor. The head of chronological list is stored in `hash_table_t.first_flow_record`.

The flow cache is protected by spinlock `rwlock`. When the packet capturing process updating a flow cache, the process locks this spinlock for case of cache exporting by the control thread.

7.7 Record exporting

The Probe does not provides a full support of IPFIX protocol. The control thread serializes the data in the `export_all_tables` function and print statistics to the file. The control thread interates throught the flow caches and takes its flow records by chronological order to print metadata. The record is destroyed after printing.

The control thread prints sockets statistics to stadard output. The packet counter of the particular socket is holded in `xsk_socket_info` data structure and it is incremented when a packet batch is ready to process.

7.8 Application exection

It is mandatory to execute application on Linux system which supports a XDP, eBPF and AF_XDP. The probe was tested on vanilla kernel version 5.0.6. The manual for installation can be found in Attachement B.

The probe is consists of several source files, which are included on attached DVD. The standard `Makefile` file is enclosed. The `make` command should be called in `/src` folder to source code compilation. The command creates an `xProbe` binary file, which can be executed with following parameters:

```
./xProbe -i [INTERFACE] [OPTION]
  -i, --interface=n   Run on interface n
  -q, --queue=n       Use queue n (default 0)
  -p, --poll          Use poll syscall
  -S, --xdp-skb=n     Use XDP skb-mod
```

```
-N, --xdp-native=n  Enfore XDP native mode
-n, --interval=n    Specify statistics update interval (default 1 sec)
-z, --zero-copy     Force zero-copy mode
-c, --copy          Force copy mode
-t, --thread        Number of open queues. default 1
-o, --output=n      Output file name n
```

The application is terminated ater sending one of `SIGINT`, `SIGTERM` or `SIGABRT` signals. Before running the xProbe application, the `src/optiNic.sh` and `src/disableHt.sh` should be executed to get better performance and to set RSS queues.

7.9 Optimalization

All describes optimalization used on NETX−F router are based on the manuals [3, 12]. The optimalization included resizing NIC's descriptor buffer to the maximum value with `ethtool -G ens2f1 rx 4096` (to show current settings and maximum value can be used `ethtool -g ens2f1`). A number of queues should be equal to the number of used cores: `ethotool -L ens2f1 combined 8`.

Next usefull optimalization is turn off the adaptive interrupt moderation with `ethtool -C ens2f1 adaptive-rx off adaptive-tx off` and sets a fixed interrupt rate to 8 μs with `ethtool -C ens2f1 rx-usecs 8 tx-usecs 8`. All optimalizations are applied in `src/scripts/optimalization.sh` script.

The better results are achieved if the CPUs hyperthreading is turned off. It is possible via `/sys/devices/system/cpu/cpu#/online` variable in kernel. The source code contains a `scripts/disableHT.sh` script that diable hyperthreading and `scripts/enableHT.sh` script that turn hyperthreading on. Additionally, it is recommended to boot the kernel with `iommu=pt` parameter.

Chapter 8

Testing

8.1 Test topology

A test topology was deployed to verify basic functionality of implemented probe with zero − copy mechanism. The topology is depicted in Fig. 8.1 and consists of 3 machines: the test PC generates a 10Gbps traffic via PF_RING. The NETX-F router, where the tested application running and switch in the middle. The NETX-F router was connected through a 10 Gb/s INTEL X552 NIC with an ixgbe driver.

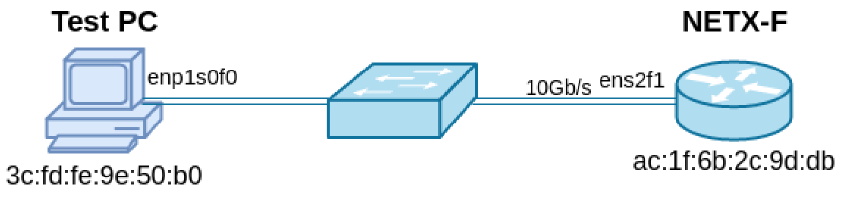


Figure 8.1: Test topology

8.1.1 NETX − F

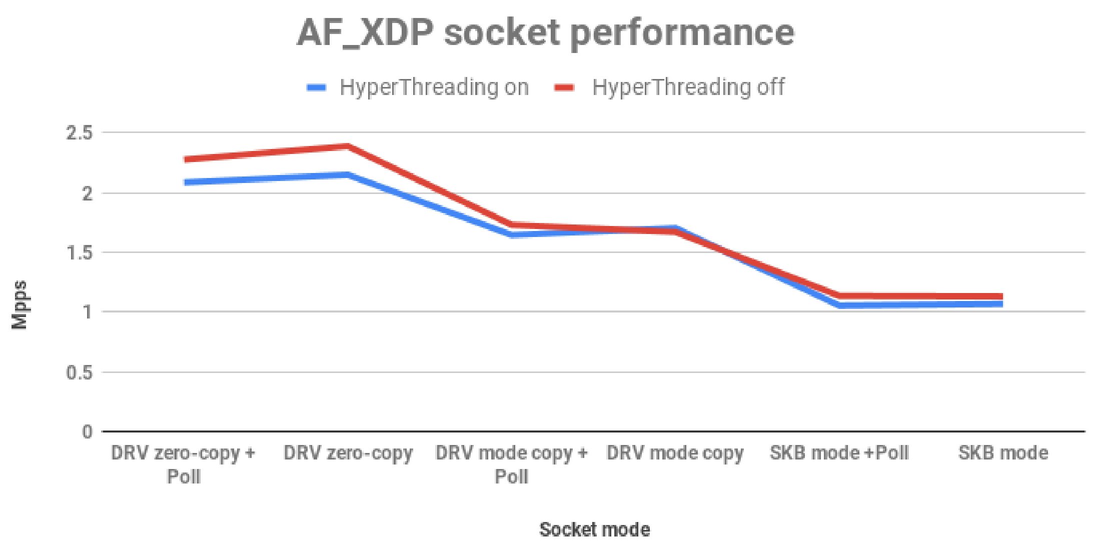
The NETX-F device was used for testing. The latest version of vanilla kernel(5.0.6) with BPF and XDP support was installed and boot on them.

```
Platform:              NetX-X1120
NetX OS version:       NetXOS release 7.5.1804 (Core)
Netc version:          1.13
Kernel version:        5.0.6
Serial number:         VM175S024749
Motherboard:           Supermicro X10SDV-16C-TLN4F+
Memory:                1GB DIMM DDR4 2133 MHz
CPU:                   Intel(R) Xeon(R) CPU D-1587 @ 1.70GHz 16 core
NIC:                   INTEL X552 SPF+
Driver:                ixgbe
Version:               5.1.0-k
Firmware-version:      0x800005b9
```

8.2 Socket performance

This test compares the performance of different socket modes. Figure 8.2 confirms the zero-copy socket as the most effective mode. The Test PC transmitted a 100 M set of 64 B packets using PF_RING and pfsend utility on enp1s0f0. The application was running on 4 NETX cores and each core capturing one of the queue with an independent AF_XDP socket. The value is average performance per second on single socket. The packet processing is turn off and packets are dropped after capturing.

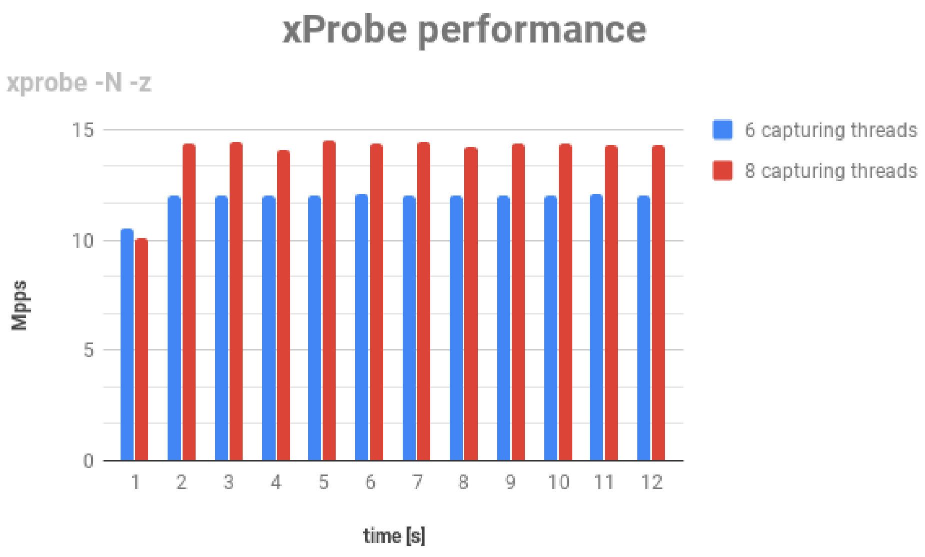
Figure 8.2: AF_XDP socket performance



8.3 Probe performance

This test shows a core utilization. The value is summary of captured packets by all 8 or 6 threads. The performance of one socket without flow creation is about 2.4 Mpps as shows Fig. 8.2. So 6 cores can capture traffic of 10GbE interface. With flow maintaining turned on, the 8 cores are needed as show 8.3.

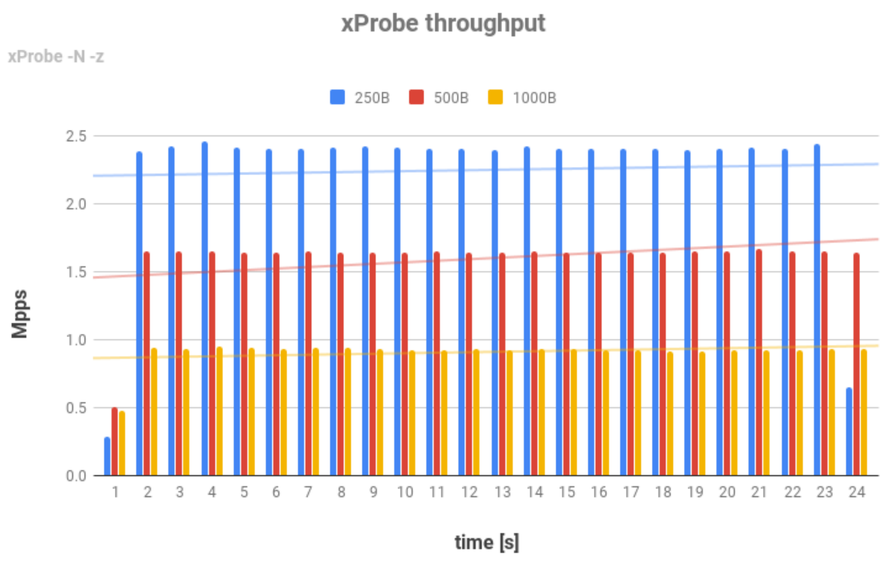
Figure 8.3: xProbe performance



8.4 Probe throughput

In this test, the Test PC generates a packet sets of different sizes. The packets are 250 B, 500 B or 1000 B long. The xProbe running on 8 cores.

Figure 8.4: xProbe throughput



Chapter 9

Conclusion

The aim of this thesis was to design and implement a flow monitoring application, capable of capturing traffic on 10Gb interface. To observe packets at high speeds, for the application was necessary to adopt some methods of fast packet processing. One of used methods was the zero-copy mechanism, which makes data transfer between network card and application more efficient with using shared memory. Another applied method was a network stack bypassing with using a new AF_XDP socket. These methods aims to eliminate disadvantages of standard packet procesing within network stack in Linux kernel and offer significant performance improvements during packet capturing.

In the first chapter of this thesis, the issue of the traffic monitoring on the high-speed interfaces was shown.

In the second chaper was defined a concept of *flows* and the flow monitoring system was introduced. The monitoring system is consist of an exporting part and a collecting part. Both parts have been described in more detail, but this thesis aimed to implement only the export segment.

The standard packet processing in Linux network stack and its limitation were described in the chapter 3. It is used to understand the mechanisms, which have been developed to make packet processing more efficient.

The fast pacekt processing mechanism was introduced in the chapter 4. Special emphasis was placed on the use of zero-copy technique in the Linux kernel. The *Page remaping* was introduced as one of the possible ways to apply zero-copy within the Linux. The next section provides an overview of the commonly used frameworks improving packet processing.

The chapter 5 was dedicated to XDP toolkit. It is one of the fast packet processing frameworks, which does not make the kernel bypassing, but adds programmability directly in the kernel network stack. It allows an execution of validated code on kernel's driver and makes possible to create a socket (also called *AF_XDP* socket or *XSK*) from driver directly to the application. This system has a huge impact to performance improvement. In following sections are described, how to create the AF_XDP socket and its memory model.

In the chapter 6 was given a design of suggested monitoring probe. It is a multithread application, which is using a NIC's capabilities to distributing traffic among multiple CPU to process packets in higher rates. Packet data are transmitted from network card to application through AF_XDP sockets and theirs buffers are shared between network card and application threads. Each probes's thread processing packets, captured in buffer, and maintaing table of useful information about communications.

The chapter 7 explained the implementation on suggested application and its optimalization. The application was implemented in C/C++ and is portable across Linux realm. The last chapter was showed application testing on model situations.

The goals of this work have been achieved. The probe is capable of capturing 10GbE traffic with using 8 CPUs. All XDP driver modes was tested and zero-copy mode showed the best result. A single AF_XDP socket can capture about 2.3 Mpps without flow creating. With flow creating it is about 1.8 Mpps. The concept has some limitations resulting from the using of XDP system:

- There is only a limited number of network interfaces that support XDP (ixgbe, i40e, mlx5, veth, tap, tun, virtio_net and others) and only ixgbe and i40e drivers has implemented a true zero-copy mode support, which offer the best performance result. In other drivers, the XDP hook is attached at a higher point in the network stack
- the eBPF program does not offer a multiple packet actions, it means that packet can not be redirect to user-space and at the same time pushed to network stack. It limits the use of the probe to not running on forwarding interfaces, but it is suitable to mirrored port
- the eBPF drops undelivered packets. When an open socket for particular queue is missing in XSKMAP, then the packet is dropped, so the sockets have to be bounded one-to-one with queues
- The maintaing one flow cache per core is addicted to RSS capability. It does not allow maintaing a bidirectional flows

However, this application successfully demonstrates the use of new AF_XDP socket in practice. The AF_XDP is suitable technology for use-cases, where the fast packet processing are required.

Bibliography

[1] Ahern, D.: *Leveraging Kernel Tables with XDP*. 2018. cumulus Networks, In Linux Plumbers Conference 2018.

[2] Appleby, A.: *aappleby/smhasher SMHasher on Github*. 2012. [Online; Accessed 17.4.2019].
Retrieved from: https://github.com/aappleby/smhasher/wiki

[3] Bainbridge, J.; Maxwell, J.: *Red Hat Enterprise Linux Network Performance Tuning Guide*. Red Hat. first edition. 3 2015.

[4] Barbette, T.; Soldani, C.; Mathy, L.: *Fast userspace packet processing. 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2015: pp. 5–16.

[5] Betreuer, K. T.; Wohlfart, F.; Raumer, D.: *A Survey of Trends in Fast Packet Processing*. 2014.

[6] Chang, T.: *Ioremap and mmap in Linux*. [Online; Accessed 3.1.2019].
Retrieved from:
https://www.slideshare.net/gene7299/linux-mmap-ioremap-introduction

[7] Cilium Authors: *BPF and XDP Reference Guide*. Revision 88016938, [Online; Accessed 21.4.2019].
Retrieved from: https://docs.cilium.io/en/v1.4/bpf/

[8] Fontana, L.: *Load XDP programs using the ip (iproute2) command*. [Online; Accessed 19.4.2019].
Retrieved from: https://medium.com/@fntlnz/load-xdp-programs-using-the-ip-iproute2-command-502043898263

[9] Hofstede, R.; Čeleda, P.; Trammell, B.; et al.: *Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. IEEE Communications Surveys Tutorials*. vol. 16, no. 4. Fourthquarter 2014: pp. 2037–2064. ISSN 1553-877X. doi:10.1109/COMST.2014.2321898.

[10] Høiland-Jørgensen, T.; Brouer, J. D.; Borkmann, D.; et al.: *The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel*. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. New York, NY, USA: ACM. 2018. ISBN 978-1-4503-6080-7. pp. 54–66. doi:10.1145/3281411.3281443.
Retrieved from: http://doi.acm.org/10.1145/3281411.3281443

[11] Horvath, A.: *MurMurHash3, an ultra fast hash algorithm for C# .NET*. 2012. [Online; Accessed 17.4.2019].
Retrieved from: http://blog.teamleadnet.com/2012/08/murmurhash3-ultra-fast-hash-algorithm.html

[12] Intel Corporation: *Intel® Ethernet Controller X710/XL710 and Intel® Ethernet Converged Network Adapter X710/XL710 Family, Linux Performance Tuning Guide*. Intel Corporation. first edition. 3 2016.

[13] IO Visor Project contributors: *BPF Features by Linux Kernel Version*. [Online; Accessed 19.5.2019].
Retrieved from:
https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#xdp

[14] Karlsson, M.; Töpel, B.: *The Path to DPDK Speeds for AF_XDP*. In *Linux Plumbers Conference*. Nov 2018.

[15] Kerrisk, M.; other: *MMAP(2) Linux Programmer's Manual*. [Online; Accessed 12.12.2018].
Retrieved from: http://man7.org/linux/man-pages/man2/mmap.2.html

[16] Li, Y.-C.; Chiang, M.-L.: *LyraNET: a zero-copy TCP/IP protocol stack for embedded operating systems*. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*. Aug 2005. ISSN 2325-1271. pp. 123–128. doi:10.1109/RTCSA.2005.57.

[17] Love, R.: *Linux Kernel Development*. Addison-Wesley Professional. third edition. 2010. ISBN 0672329468, 9780672329463.

[18] Luis García-Dorado, J.; Mata, F.; Ramos, J.; et al.: *High-Performance Network Traffic Processing Systems Using Commodity Hardware*. 01 2013. ISBN 9783642367830. pp. 3–27. doi:10.1007/978-3-642-36784-7_1.

[19] Miano, S.; Bertrone, M.; Risso, F.; et al.: *Creating Complex Network Services with eBPF: Experience and Lessons Learned*. 06 2018.

[20] Qi, Z.; Xiaoyun, L.: *DPDK PMD for AF_XDP*. 2018. intel, DPDK UserSpace; [Online; Accessed 15.4.2019].
Retrieved from: https://www.dpdk.org/wp-content/uploads/sites/35/2018/10/pm-06-DPDK-PMD-for-AF_XDP.pdf

[21] Risso, F.: *Toward Flexible and Efficient In-Kernel Network Function Chaining with IOVisor*. [Online; Accessed 9.3.2019].
Retrieved from:
http://hpsr2018.ieee-hpsr.org/files/2018/06/18-06-18-IOVisor-HPSR.pdf

[22] Rizzo, L.; Deri, L.; Cardigliano, A.: *10 Gbit / s Line Rate Packet Processing Using Commodity Hardware : Survey and new Proposals*. 2011.

[23] Tu, W.; Stringer, J.; Sun, Y.; et al.: *Bringing the Power of eBPF to Open vSwitch*. 2018. vMware Inc. and Cilium.io; [Online; Accessed 12.4.2019].
Retrieved from:
http://vger.kernel.org/lpc_net2018_talks/ovs-ebpf-afxdp.pdf

[24] Tu, W.; Stringer, J.; Sun, Y.; et al.: *Bringing the Power of eBPF to Open vSwitch*. 2018. vMware Inc. and Cilium.io; In Linux Plumbers Conference [Online; Accessed 12.5.2019].
Retrieved from:
http://vger.kernel.org/lpc_net2018_talks/ovs-ebpf-afxdp-presentation.pdf

[25] Umer, M. F.; Sher, M.; Bi, Y.: *Flow-based intrusion detection: Techniques and challenges. Computers and Security*. vol. 70. 2017: pp. 238 – 254. ISSN 0167-4048.

[26] Velan, P.: *Application-Aware Flow Monitoring*. Doctoral theses, dissertations. Masaryk University, Faculty of Informatics, Brno. 2018.

Appendix A

Sequence diagram

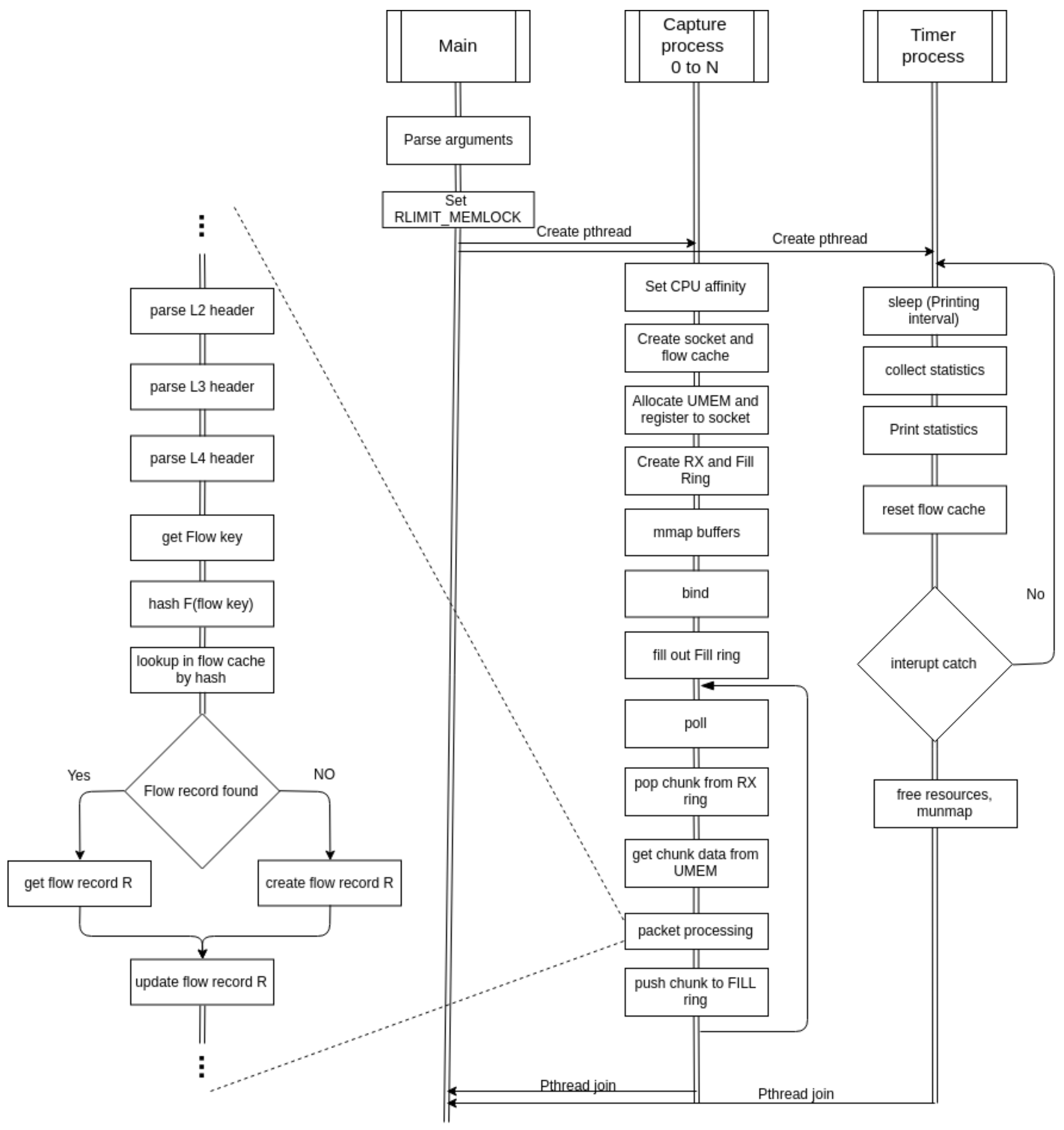


Figure A.1: Sequence diagram of packet capturing probe

Appendix B

XDP installation

AF_XDP required a kernel 4.19 or higher and it is mandatory to allow eBPF and XDP support when kernel is compiling. To do that is needed to rewrite some flags to `.config` file after kernel source tree has been unpack:

```
CONFIG_CGROUP_BPF=y
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
CONFIG_NET_SCH_INGRESS=m
CONFIG_NET_CLS_BPF=m
CONFIG_NET_CLS_ACT=y
CONFIG_BPF_JIT=y
CONFIG_LWTUNNEL_BPF=y
CONFIG_HAVE_EBPF_JIT=y
CONFIG_BPF_EVENTS=y
CONFIG_TEST_BPF=m
CONFIG_XDP_SOCKETS=y
CONFIG_XDP_SOCKETS_DIAG=y
```

After new kernel is running, it is recommended to turn a BPF JIT on to code optimalization with command:

```
sysctl net/core/bpf_jit_enable=1
```

Then the libbpf library has to be installed:

```
git clone https://github.com/libbpf/libbpf.git
cd libbpf/src
mkdir build root
OBJDIR=build DESTDIR=/ make install
```

For compiling eBPF is useful to install a LLVM version 3.8 or higher.

Appendix C

Content of the attached DVD

- Source code of the implemented application in directory `/src`.
- Readme file in `/src/README.md`
- This technical report including LaTeXsource code in `/tz`