Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Hochschule
für Technik, Wirtschaft
und Kultur Leipzig

Lightweight Autonomic Network Architecture

Daniel Borkmann

Master's Thesis MA-2011-01
December 2011

Advisors, ETH Zurich: Ariane Keller, Dr. Wolfgang Mühlbauer
Supervisor, ETH Zurich: Prof. Dr. Bernhard Plattner
Supervisor, HTWK Leipzig: Prof. Dr.-Ing. Dietmar Reimann

ETH Zurich
Computer Engineering and Networks Laboratory
Communication Systems Group

Abstract

During the last decades the Internet architecture matured and gained more and more popularity. Networking devices of all kind ranging from high-end servers to personal computers, tablet devices, mobile phones or even embedded sensor networks make use of it. The Internet became ubiquitous for us. Even though we witness such a diversity of devices, the progress of having an *efficient* interconnection between all of them only advances in small steps. No matter if we consider a server, a mobile phone or a resource-constrained temperature sensor on an embedded device from this large spectrum, with high probability, all of them have the same, one-size-fits-all TCP/IP protocol stack, although there are more appropriate communication mechanisms than that.

Recent research tries to challenge this stagnation by introducing dynamic adaption into computing systems. For instance, the EPiCS project aims at making devices more self-aware, so that they can dynamically adapt to their current needs. Imagine such a temperature sensor that only has a very minimal protocol stack, hence it has only protocols active, it is actually using in order to save resources. If such a sensor detects that it is loosing too much battery power, it could automatically switch to a less reliable but less power-intense protocol. Therefore it needs to dynamically adapt its communication stack during runtime. For the EPiCS networking part, the Autonomic Network Architecture (ANA) lays the foundation of such an architecture.

This master's thesis is situated in the ANA context. The first ANA prototype implementation has shown that such a dynamic adaption of the protocol stack is possible, but it heavily suffers from performance issues. While complying with the most basic ANA design principles, we have redesigned the architecture to a great extend and implemented it from scratch in the Linux kernel by utilizing design principles for efficient networking architectures. The outcome that we have called Lightweight ANA (LANA) has a competitive packet per second performance with the Linux networking subsystem and is about 21 times faster than the original ANA prototype.

In this work, we have shown that it is possible to fulfill three partially conflicting goals with LANA: i) high flexibility for network programmers, ii) re-configuration of the network stack at runtime, and iii) high packet processing rates. With a voice-over-Ethernet example application, we have demonstrated that it is possible to adapt the underlying protocol stack during runtime without notable voice interruption. Hence, we claim that with LANA we have developed the base for a successful future Internet architecture.

Contents

Chapter 1

Introduction

1.1 Motivation

Today's Internet architecture has clearly matured out of its infancy and its success has found its way into our everyday life. Yet, most people are not aware of its internals. However, in professional circles issues arise such as the transition from IPv4 to IPv6 which targets to prevent the exhaustion of the IPv4 address space. IPv6 already found its way into the Linux kernel 2.1.8 in 1996 [1]. But, international events such as the IPv6 day reveal that IPv6 usage has only a median rate of 1.96 Mbps for the entire Swiss national research network, for instance, an IPv6 traffic proportion of less than one percent [2]. This example demonstrates the difficulty of deploying changes within the current Internet architecture. Hence, this inflexibility shows a major weakness that needs to be challenged.

But instead of challenging this, most research and development rather concentrate on improving performance on the operating system level through the introduction of zero-copy mechanisms [3] [4] [5], networking device driver multiqueues [6], CPU locality of flows [7], TCP segmentation and checksum offloading [8], or device driver polling (NAPI [9]) to reduce interrupt load. Especially after the introduction of 10 Gigabit Ethernet or even 100 Gigabit Ethernet products, such optimizations became more interesting for carrier-grade platforms based on Linux. Furthermore, most research in protocols or networking applications shifted towards upper protocol layers such as peer-to-peer overlay networks and applications like Tor [10], Bitcoin [11] or peer-to-peer DNS [12]. Also, there are tremendous efforts to fix security design flaws in today's Internet architecture such as the introduction of TCP SYN Cookies [13], DNSSEC [14] or DNScurve [15].

Concluding, there are no *radical* changes in the underlying Internet architecture that address its inflexibility. Only a few research projects like the x-kernel [16], FreeBSD's Netgraph [17] or MIT's Click modular router [18] have tried to introduce more flexibility. However, we argue that these projects only concen-

trate on specific parts of the Internet architecture, but not its full framework. What if we could have a future Internet architecture that is *designed for change on all levels*?

Innovations on lower network protocol layers could be made attractive to researchers and to the networking industry. More appropriate communication mechanisms could be introduced that reside next to IP. Some of these mechanisms could have its application in resource-constraint devices like in sensor networks or even in the high-performance computing area where protocol overhead is avoided or minimised in favour to increase processing capabilities. Also, communication mechanisms we do not yet know of could be easily included.

A first step towards this flexibility has been taken with the Autonomic Network Architecture (ANA) [19]. The concepts of this dynamic architecture have been adopted into the ongoing EPiCS research project. There, one aspect is to develop a dynamic network architecture with high-performance capabilities. Such capabilities will be introduced by dynamically offloading calculation intensive functions to hardware threads during runtime. A further step on top of this is to introduce self-awareness into these systems, so that based on sensor values, an algorithm will autonomously decide which processing functions will be mapped to hardware for improving performance. To achieve that, the software part must provide a framework for this first. Therefore, ANA divides network functionality into small, independent building blocks, also referred to as functional blocks. Each functional block processes portions of incoming or outgoing network packets. Moreover, functional blocks can be assembled into a graph-like construct that represents the host's protocol stack (figure 1.1). Such binding of functional blocks is dynamic and can even change during runtime without restarting the networking component of the operating system. It even allows for an automatic reconfiguration during runtime and builds the foundation for offloading blocks to hardware. Compared with the traditional Internet architecture (figure 1.1), performance through a static binding or configuration of protocol functions is being traded for more flexibility in the protocol stack.

This thesis focuses on the networking aspect in software of such systems by using the idea of ANA as a basis for further research. The ANA prototype heavily suffers from performance issues. We demonstrate a networking benchmark with 64 Byte packets per second (pps) on Gigabit Ethernet where Linux achieves a maximum of 1.385 million pps and ANA only a maximum of 64,000 pps in its minimal configuration. Hence, the scope of this thesis will focus on a rework of the architecture to tackle the challenge of fulfilling three partially conflicting goals: i) high flexibility for network programmers, ii) re-configuration of the network stack at runtime, and iii) high packet processing rates.

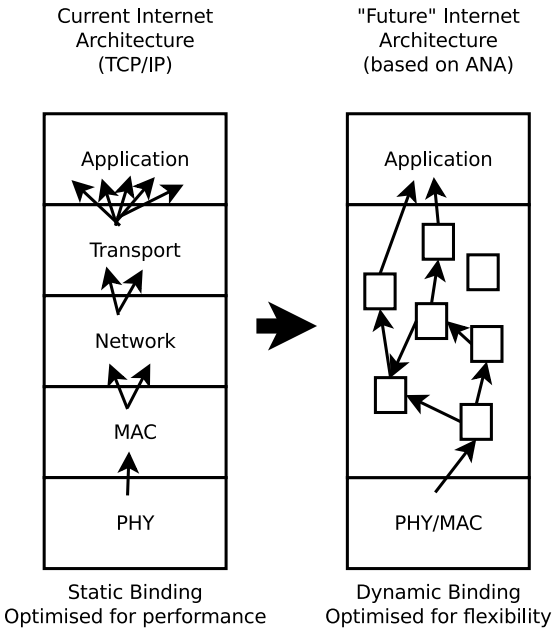

Figure 1.1: Basic idea of the traditional, static TCP/IP architecture compared to a dynamic (L)ANA protocol stack

1.2 Aims of this Thesis

The aim of this thesis is to design a lightweight ANA architecture (LANA) and implementation. The architecture has to follow basic ANA principles while the implementation has to significantly outperform the previous prototype implementation.

The first step will be to identify the critical parts of the ANA architecture and to combine them in a new architecture. In the second step, this architecture will be implemented where main parts of this implementation should reside in the Linux kernel space. In the third step, the performance will be evaluated and compared to the original ANA prototype.

1.3 Outline

This thesis is structured in the following way: chapter 2 provides theoretical aspects about possible networking bottlenecks on intermediate and end nodes in networks and describes possible solutions on how to eliminate them. Chapter 3 gives insights into the path of Linux network packets in ingress and egress direction as a basis for further discussion and points to related work that has been conducted within this research area. In chapter 4, the architecture of LANA is explained in detail including a discussion about the major differences to the original ANA prototype, what possible bottlenecks have been removed and about

ingress and egress entrance points to the packet path of the Linux kernel. Next, chapter 5 describes implementation details about the LANA core framework as well as a set of implemented functional blocks. An evaluation of LANA's performance is done in chapter 6 where LANA is compared to the Linux kernel's networking subsystem, to the Click Modular Router and the original ANA prototype. We come to a final conclusion and point to future work within the last chapter, chapter 7.

Chapter 2

Design Concepts of Network Architectures

With main reference to Varghese [20], we debate aspects on how efficient network architectures should be designed. Later, design principles of this chapter are also further discussed in design and performance evaluation chapters of LANA. Before talking about optimizations, we discuss typical network bottlenecks for end nodes and intermediate nodes of computer networks first.

2.1 Network Bottlenecks

Bottlenecks are phenomena where the performance or capacity of an entire system is limited due to few single components. In computer networks, the loss of performance is usually measured in packets per second a system can process. According to Varghese, network bottlenecks are subdivided into bottlenecks on end nodes and bottlenecks on intermediate nodes in computer networks. Although one might believe that intermediate node bottlenecks are less likely due to powerful backbones and core routers, recent measurement studies [21] [22] have shown that bottlenecks on ISP or AS levels should not be underestimated[1].

2.1.1 Bottlenecks on End Nodes

Classification

On networking endpoints of the Internet like personal computers, workstations or servers of all different kinds, causes for network bottlenecks can be classified according to Varghese into **structure** and **scale**.

[1] Note that in this chapter we assume that network traffic on intermediate nodes is not being throttled on purpose, so that the discussed bottlenecks refer to possible bottlenecks caused by the system itself.

For the sake of having systems designed in a general purpose manner, *protection mechanisms* and *software layers* are added to keep up with the intention to have a flexible software architecture. The aim is to still have maintainable code where new components can be added without much efforts. Next to this, different usage scenarios play an important role, thus not every user on the end node should have the same privileges, for instance. Therefore, software is usually *structured* into various software architectural layers or design patterns. A little exaggerated example about software layering is demonstrated in figure 2.1. How the introduction of such software layers can cause performance penalties is discussed in the examples section.

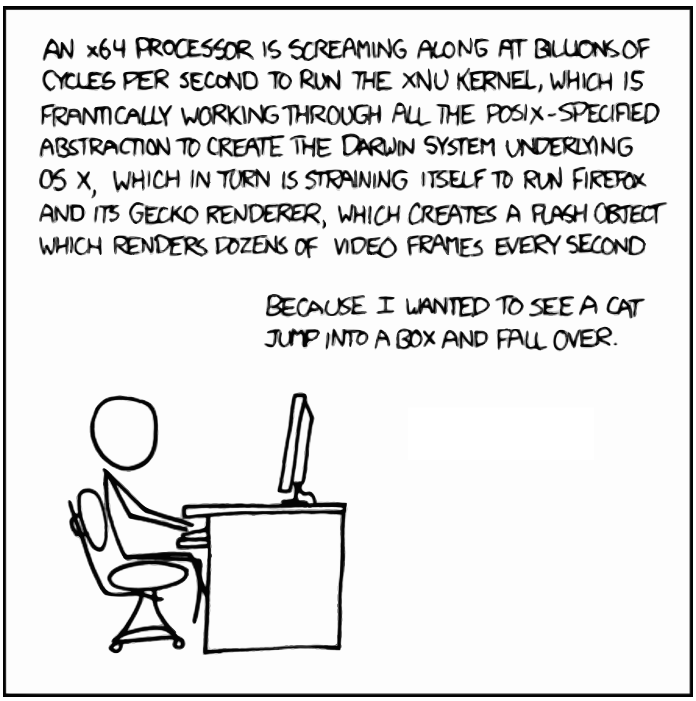

Figure 2.1: Not so serious example of software architectural layers on an end node (from: `XKCD, 676`, slightly modified).

Next to *structural* problems there are *scaling* problems, for instance on web servers. Concurrent client requests to the server can cause high communication latency, if underlying data structures for the web server software or the operating system were not designed for high concurrency.

Examples and their cause

Summary of discussed bottlenecks:

- **Copying**
- **Context switching**

- **System calls**
- **Timers**
- **Demultiplexing**
- **Checksums, CRCs**
- **Protocol code**

A few examples of end node bottlenecks and their causes are presented in this section. One of the most expensive end node bottlenecks is **copying**. This is usually done due to reasons of protection. For instance, the operating system kernel copies data between two address spaces for security reasons [23] [24]: data from the address space where the kernel with a high level of privileges runs is copied to the address space where all user-level applications with low privileges reside, since user-level applications have no direct access to kernel space memory. Copying can also occur within complex software systems, such as a network stack. There, packet data could be copied between layers of the protocol stack due to protection from race conditions by concurrent packet data access. Hence, copying in general is performed because of *protection* or software *structure*. According to Varghese, one solution could be to copy data in hardware without operating system intervention like in remote direct memory access (RDMA) [25], for instance.

Another bottleneck on end nodes can be **context switching**. Context switching is the process of saving all current CPU registers of a running process in RAM and restoring all CPU registers and data of another process for resuming execution from a point where it was previously interrupted. Context switching is done on i) multitasking, thus several processes can share a CPU for their execution[2], ii) on the reception of hardware interrupts, or iii) context switching can occur if the operating system needs to switch between user and kernel mode. Context switching can become a bottleneck on *complex scheduling* situations according to Varghese. For instance, this can be the case on server applications that spawn new processes for each connected client [26]. There, if too many clients are connected at the same time, scheduling between these processes becomes a major issue. Event-driven servers with a fixed thread pool could resolve scheduling overhead, for example. Applications that make use of `select(2)`, `poll(2)` or even better, the more efficient `epoll(2)` [27] for their opened file descriptors can reduce context switching, since the operating systems process scheduler only wakes them whenever new incoming data is present.

Performing **system calls** can yet result in another bottleneck on end nodes. Usually, they are performed by the user space to communicate with hardware through the operating system kernel, for instance, for sending or receiving network packets. Due to security reasons, user space must not have control over the

[2]On an Intel Core 2 Quad Q6600 Linux 3.0 workstation with approx. 190 tasks, and an rather idle CPU, about 850 context switches occur per second (measured with `ifpps`, D.2).

hardware directly like the privileged operating system kernel. The only interface the kernel therefore offers to user space applications are system calls. The main reasons for this are to *protect* from malicious user space applications and also to add a layer for access control to applications. Hence, *structure* can be seen as another reason for system calls. An example for bypassing system calls is the traffic generator that we have developed (appendix chapter D.1). Here, instead of performing system calls for each network packet that needs to be transmitted, a shared ring buffer between user space and kernel space is set up for handing over packet data. A kernel thread polls status flags of each ring buffer slot, thus it knows which packet can be processed for transmission [28]. This approach saves system calls and CPU resources for copying data and therefore speeds up the implemented traffic generator significantly to more than three times of the processing capabilities of usual system call methods [29]. So, direct channels to underlying infrastructure can reduce networking bottlenecks significantly.

Another possible system bottleneck can be the **maintenance of timers**. Timers are used in the operating system for several reasons. One is to recover from failure, for instance, with the help of watchdog timers. There, some piece of hardware needs to notify the timer controller periodically that it is working properly. However, when the hardware has a failure that prevents it from working correctly, the timer expires and the operating system forces the hardware to restart, for example. Timers are also present in algorithms like the TCP state machine. Data packets sent to a remote host need to be acknowledged by the remote host to signal its successful data reception to the sender. However, if the acknowledgement packet is not being received by the sender's host, a timer on the sender's host will expire and trigger retransmission of specific data packets. Next to networking algorithms, timers are also applied in several other subsystems, like scheduling. Application of timers can become a performance bottleneck, if they are increasingly used due to *scaling issues* with the number of timers in the system. According to Varghese, timing wheel data structures [30] which are circular buffers for start, stop or maintenance of timers, can be used for a faster lookup to reduce this bottleneck.

The next possible network bottleneck refers to **demultiplexing**. Demultiplexing is basically the process of delivering a received network packet to the appropriate client application. In case of a server application, it traverses the protocol stack layers of IP, TCP or others, and queues data into the right application socket receive queue. Which of the network protocols is being applied is determined from the packet headers, starting with the Ethernet type field. Traditional demultiplexing is fairly straightforward, since the lookup of the corresponding protocol handler can be done with an exact match of the type field, thus hashing can be applied. In general, demultiplexing *scales* with the number of network nodes. However, *early* demultiplexing is much more challenging, especially on high speed. It determines the path of the packet in one operation during arrival of the packet. This means that early demultiplexing would determine that

the path of a packet is e.g., Ethernet → IP → TCP → Web. Its original idea comes from user-level protocol implementations and network monitoring applications, where the decision whether to keep or to drop a packet should not just be made in user space but as early as possible. Hence, features like Berkeley Packet Filter [31] [32], a minimal assembler-like filter language that operates on network packets (appendix section D.3), have been introduced to shift the decision to an early point into the kernel with having less processing overhead.

Furthermore, another possible network bottleneck is **calculating checksums**, since this *scales* with link speed, no matter if it is for validation purpose of incoming packets or if it is for outgoing packets. Checksums in packets usually are variants of CRC [33]. In traditional networks like the Internet, there are commonly checksum calculations performed from the network layer down to the link layer. However, in TCP/IP both have different algorithms applied which are usually calculated and checked by the operating systems network stack. Since this can lead to a significant amount of computation time on high packet load or on packet fragmentation, today's hardware vendors implement checksum offloading [34] where checksums are pre-calculated by the network card's chipset instead of by the kernel.

Last but not least, **protocol code** can cause network bottlenecks on end hosts according to Varghese. Especially TCP plays an important role in this case, because it is used by over 90 percent of the end hosts of today's Internet traffic. Since processing of incoming TCP packets is not complicated but long and detailed, TCP would have been almost replaced with a faster variant namely XTP [35], if Van Jacobson didn't optimize TCP processing path by a feature called header prediction [36]. Much of the TCP processing complexity is caused by rather rare cases, so header prediction provides a fast-path by predicting TCP header values that are very unlikely to change (or that contain only few important information) based on previously incoming packets.

2.1.2 Bottlenecks on Intermediate Nodes

Classification

Unlike end nodes, network bottlenecks on intermediate nodes are of different nature. Intermediate nodes in computer networks are, for instance, routers, gateways, security appliances like firewalls, bridges or network monitors. Structure is usually not a bottleneck on such specialized devices. Mostly, they are deployed with a lightweight operating system that has short packet forwarding paths. Also, in many cases, parts of that forwarding path are implemented in hardware to achieve a better throughput. However, Varghese states two classifications for causes of network bottlenecks on intermediate nodes, namely **scale** and **service**.

Scaling issues on intermediate nodes can be subdivided into *bandwidth scaling* and *population scaling*. Bandwidth scaling is characterized by the steady growth

of today's Internet traffic and by innovation on networking links that keep getting faster and faster from 1 Gbps up to 100 Gbps for Internet backbones [37]. On the other hand, population scaling addresses the challenge in which intermediate nodes have to cope with the increasing number of end nodes that get connected to the Internet, no matter if caused by the increase of company workstations or electronic consumer devices.

In addition, intermediate nodes need to deal with service issues. The term service is especially addressed to *network guarantees* regarding performance, security and reliability [38]. Here, intermediate nodes shall protect company networks during attacks. They shall guarantee availability on fail-over or bandwidth guarantees for certain applications, for instance.

Examples and their cause

Summary of discussed bottlenecks:

- **Exact lookups**
- **Prefix lookups**
- **Packet classification**
- **Switching**
- **Fair queueing**
- **Internal Bandwidth**
- **Measurement**
- **Security**

Most of intermediate nodes network bottlenecks are caused by link speed *scaling*. Such an example is the issue of **exact lookups**. In a simple forwarding model, the destination address of an incoming packet is being looked up in order to determine a router's destination port. Once the destination has been found, the packet is scheduled for QoS on the port of the outgoing packet queue. Here, the destination port lookup could be a database query that is realized with (parallel) hashing [39]. However, other scenarios for exact lookups could apply to firewalls, for instance. Here, certain pre-defined rules for specific IP addresses could first be looked up and then applied.

Next to exact lookups, intermediate nodes also have to deal with **prefix lookups** which are used for packet forwarding in routers. According to Varghese, in 2004, core routers stored about 150,000 prefixes instead of billions of IP entries for each possible Internet address. Such prefixes are usually stored in compressed multi-bit tries such as Patricia tries [40], also known as Radix tries. Network bottlenecks cased by prefix lookups are classified as *scaling* issues of link speeds and as *scaling* issues of the prefix database size.

Packet classification [41] can be a network bottleneck on intermediate nodes. Causes for this are that *service differentiation* needs to be done, or that issues

arise due to link speed *scaling* problems or size scaling problems of the decision tree. Usually, classification is done by information that is found in the network packet. This refers to the destination address, source address, TCP or UDP ports and other header fields such as TCP flags. It can become more complex when deep packet inspection (DPI) is performed. Classification is applied within forwarding functions of firewalls, QoS routing or unicast and multicast routing mechanisms.

Further network bottlenecks are **packet switching** and **internal bandwidth** limitations of intermediate nodes. Both refer to the hardware side. Within packet switching, bottlenecks are caused by the optical-electronic speed gap [42] or by head-of-line blocking [43]. The latter phenomenon is basically a blocking of a switch output port. Since the switching fabric is busy forwarding a packet to a specific output port, another packet from an input FIFO to the same output port has to wait until the first one has been processed. Internal bandwidth refers to bottlenecks cased by the limits of internal bus speeds.

Commonly, **measurement** is performed on intermediate nodes. On excessive traffic rates, this can cause a bottleneck induced by link speed *scaling*. Reasons for measurement are mostly in the area of optimization, for instance, in mid- to longterm capacity planning, accounting on ISP-side for billing customers or traffic analysis like DPI. The latter is done to identify certain types of traffic in a network like peer-to-peer traffic, or malicious traffic such as port scans (done in intrusion detection systems), or simply to have a general overview of the traffic distribution.

A further bottleneck can be **fair queueing** [44] for reasons of QoS routing. Causes of this bottleneck depend on the packet scheduling algorithm and mainly refer to *scaling* issues of link speeds and memory.

The last network bottleneck on intermediate nodes is **security**. In security the bottleneck scales with the number and intensity of attacks. Examples are denial-of-service attacks [45].

2.2 Fifteen Implementation Principles

After having discussed possible network bottlenecks, design and implementation principles will be presented within this section. These principles provide hints on how to reduce system bottlenecks. They can be categorized into three kinds of principles, namely i) system principles, ii) principles for modularity with efficiency and iii) principles for speeding up routines. Hence, categorized principles reach from the abstraction of subsystems and modules down to individual routines. Principles of each category are first named and then explained in more detail.

2.2.1 Systems Principles

Systems principles take advantage of the fact that the system is built out of subsystems. By regarding subsystems system-wide rather than black boxes, performance can potentially be improved.

Summary of principles:

- ***P1:*** **Avoid obvious waste**
- ***P2:*** **Shift computation in time**
  - **Pre-compute**
  - **Evaluate lazy**
  - **Share expenses**
- ***P3:*** **Relax system requirements**
  - **Trade certainty for time**
  - **Trade accuracy for time**
  - **Shift computation in space**
- ***P4:*** **Leverage off system components**
  - **Exploit locality**
  - **Trade memory for speed**
  - **Exploit hardware features**
- ***P5:*** **Add hardware**

Principle **P1 'avoid obvious waste'** mainly means to reduce redundancy in code or to quash operations that are unnecessary and degrade performance. Optimizing compilers, for instance, search for repeated subexpressions, like $i := 5.1 * n + 2$, and $j := (5.1 * n + 2) * 4$ and replace the repeated expression in j with $j := i * 4$, thus the value of $5.1 * n + 2$ is only calculated once. Another example for networking would be to make multiple copies of network packets in cases where it can be avoided. Each single operation like computing a statement often seems to have no waste on the first hand, but the combination of operations like computing the same statement twice will make it wasteful. Therefore, detecting and optimizing such routines mostly need an inspection on a higher layer of abstraction.

Principle **P2 'shift computation in time'** divides a system into space and time. The term space refers to subsystems that compose the system as a whole. Time refers to different points in time of the system's lifetime like fabrication time, initialization time, compile time or runtime. Three sub-principles apply in P2, namely **pre-computation**, **lazy evaluation** and **sharing of expenses**.

The principle of pre-computation means that quantities are computed or allocated before they are used in order to save time at the actual point of use. One

such method is to use lookup tables. Here, instead of having to execute an expensive function during runtime, a mapping with pre-computed values for every element of the function's domain is being created at initialization time. Also, as another example, memory could be allocated beforehand, thus during runtime, expensive memory operations such as de-fragmentation can be avoided. In networking, TCP/IP headers could be precomputed under the assumption that only few header fields change for each packet.

Lazy evaluation is a technique which is often known in combination with functional programming languages like Haskell [46]. It is the process of postponing expensive operations at critical times in hopes that the operation will not be needed later, or a less busy time will occur for the execution of this operation. In operating systems, lazy evaluation is used in techniques like *copy-on-write* [47]. Assume that processes are migrated between two virtual address spaces, hence they must be copied from virtual address space A to virtual address space B. A general solution would be to simply copy all pages from A to B. However, a better solution would be to point all page table entries in B's space to pages of A. If a process using B's space writes to certain entries, a copy of only these specific pages from A are made and B's page table is updated to point at the newly copied pages. By this, unnecessary copies can be avoided.

The last principle belonging to P2 discusses sharing of expenses. Sharing of expenses means taking advantage of expensive operations that are performed by other parts of the system. According to Varghese, sharing of expenses means also trading latency for throughput. For instance, this is done in batching where several expensive operations can be done more efficiently than doing them separately. Besides others, this is the case in Linux NAPI [48] (section 3.1) where network packets are first collected and then processed in the network stack instead of processing each packet directly on arrival time.

P3 'relax system requirements' proposes that implementation difficulties can sometimes be solved by relaxing or loosening specification requirements. This means that i.e., weakening the specification of subsystem A from S_{A_1} to S_{A_2} at the cost of making subsystem B obey to a stronger property S_{B_2} compared to the previous S_{B_1}. This can be arranged into **trade certainty for time**, **trade accuracy for time** and **shift computation in space**.

The first one, trade certainty for time, suggests using randomized strategies in cases where deterministic ones are too slow. In networking, this principle can be applied to measurement, in particular Netflow [49], where sampling can be performed randomized instead of collecting all traffic data. By collecting such samples certainty degrades, but at the same time, the process of measurement consumes less processing time or resources.

Trade accuracy for time can, for instance, be applied to packet scheduling algorithms on routers that require sorting packets by their departure deadline. Here, the sorting overhead could be reduced during high traffic loads by approximate sorting. This would slightly reduce the quality of the QoS algorithm, but increase

processing capabilities at the same time.

Last but not least, shift computation in time means that computations are moved from one subsystem to another. For example, this is applied in packet fragmentation. There, fragmentation of packets on intermediate nodes like routers is avoided by having end systems doing fragmentation after calculating the maximum packet size a router can process.

P4 'leverage off system components' states that used algorithms should be aware of their underlying hardware. Thus, hardware features can better be exploited. Systems are usually designed top-down, but for performance-critical components, a bottom-up approach can help to improve systems performance. This can be done by **exploiting locality**, **trading memory for speed** or **exploiting new features**, for instance.

In operating systems, locality is exploited in file systems. There, B-tree algorithms [50] are mostly used since they are optimized for systems that read and write large blocks of data.

Trading memory for speed, on the other hand, can be seen from two perspectives. To retrieve the example with lookup tables, more memory is used since all values of the functions domain need to be calculated, but with the benefit of not needing to calculate an expensive function. On the other hand, less memory could be used to make data structures fit into faster memory such as CPU caches. This is done in the Lulea IP lookup algorithm [51] for storing and searching Internet routing tables efficiently.

The last part of P4 is about exploiting new hardware features. In modern operating systems, this is done by offloading checksum calculation to the hardware, for instance. Another lower-level example would be the exploitation of CPU-specific features such as MMX (Multimedia Extension) registers or SSE (Streaming SIMD Extensions) registers. Thus, copying of larger memory regions can be accelerated by using wider CPU registers combined with hardware pre-fetching. However, as a disadvantage, systems optimized by this principle could be less portable or less performant on different hardware.

If everything else fails, then the brute-force principle **P5 'add hardware'** can help. This basically means making systems faster by more modern hardware such as faster CPUs, higher amounts of memory, faster bus speeds or faster interfaces such as preferring network cards with PCIe instead of PCI or PCI-X. Also, P5 states to program special purpose chips with the help of design tools such as VHDL synthesis packages in order to offload software routines to hardware to improve performance. Of course, this can only be done with routines that are appropriate for hardware such as encryption, pattern matching or hashing, for instance.

2.2.2 Principles for Modularity with Efficiency

Principles mentioned here suggest performance improvements while still having complex systems built with modularity.

Summary of principles:

- ***P6:* Create efficient specialized routines**
- ***P7:* Avoid unnecessary generality**
- ***P8:* Don't be tied to reference implementation**
- ***P9:* Pass hints in layer interfaces**
- ***P10:* Pass hints in protocol headers**

P6 'create efficient specialized routines' implies that sometimes the one-size-fits-all approach of general purpose functions can lead to inefficiencies as in Stonebraker [52]. In these cases it can pay off to create a specialized and optimized routine which might break the general purpose manner, but with the gain of a better performance. A networking example for this can be found in [53] where UDP processing has been improved by introducing special purpose routines.

Principle **P7 'avoid unnecessary generality'** states that modules can be sped up by removing rarely used features. The tendency to design abstract and general subsystems can lead to the inclusion of unnecessary or rarely used features. Hence, instead of applying P6 and replacing these routines, they could also be removed entirely to gain performance. According to Varghese, this was the case in RISC processors, for instance, where complex instructions have been eliminated such as multiplications that required to be emulated by firmware.

Principle **P8 'don't be tied to reference implementation'** can lead to performance improvements, too. Specifications are mostly written for the sake of clarity and not for describing efficient implementations. The problem with specifications is that they tend to over-specify and their implementations try to strictly copy this. Since implementors are free to change implementation details as long as two implementations still have the same *external* behaviour, the internals are not tied to follow the reference implementation. Hence, faster or more efficient modules could be developed.

P9 'pass hints in layer interfaces' refers to passing information from a client to a server, thus if the information is correct, the server can avoid expensive computations. Here, in contrast to caches it is not guaranteed that the provided information has to be correct and therefore must be checked. For instance, such a hint can be used to supply a direct index for a processing state at the receiver. Thus, a lookup operation can be avoided in case the data at the index is still valid. However, incorrect hints must not corrupt the correctness of a system, but only result in a performance degradation.

P10 'pass hints in protocol headers' implies a logical extension to P9. Here, information is passed within the header of a message that is applied in

distributed systems, for instance. An application example is tag switching [54]. There, packets carry additional information next to the destination address in order to help the destination address being looked up more quickly. Tags can be regarded as hints in this case since this information is not guaranteed to be consistent.

2.2.3 Principles for Speeding up Routines

Next to subsystems or modules, this subsection considers individual functions or routines by themselves and suggests possible performance improvements.

Summary of principles:

- ***P11:*** **Optimize the expected case**
  - **Use caches**
- ***P12:*** **Add or exploit state for speed**
  - **Compute incrementally**
- ***P13:*** **Optimize degrees of freedom**
- ***P14:*** **Use bucket sorting, bitmaps or similar for finite universes**
- ***P15:*** **Create efficient data structures by algorithmic techniques**

Principle **P11 'optimize the expected case'** implies that systems performance can be improved by optimizing the expected system behaviour. Behaviours of systems usually fall into a smaller subset, called the expected case. This means that systems should operate mostly in a fault and exception free way. Often, heuristics can be applied to guess such a case as in a CPU's branch prediction unit, for instance. There, the aim is to improve the flow in the CPU's instruction pipeline by heuristic-based guessing which path of an conditional construct will be taken next. To further support branch prediction, `likely` and `unlikely` macros [55] in source code can provide hints for modern compilers such as GCC [56] about the potential path of the branch. Another example refers to **caches** that hold i.e., pages that are expected to be accessed very often. By using a clever cache eviction strategy, it is aimed to minimize expensive fetches of pages from slower memory like hard-disks.

P12 'add or exploit state for speed' suggests considering additional or redundant state for expensive operations in order to speedup execution. For example, it can sometimes pay off to have an additional index for entities via hash tables or balanced trees for improving lookup times. Costs for this would be the use of more memory and the maintenance of additional data structures. However, additional state can sometimes also be avoided by exploiting existing

states like the **incremental computation** of checksums when only a few fields in the packet change.

Another principle in speeding up routines refers to **P13 'optimize degrees of freedom'**. In this case, it helps to be aware of *variables* that are under one's control and *evaluation criteria* that is used to determine a better performance. Compilers, for instance, are aware of a specific CPU architecture with their instructions, registers and others. Thus, they can apply techniques such as coloring algorithms [57] for register assignment in order to minimize register spills.

If a system deals with small finite universes such as a moderately sized integer space, **P14 'use bucket sorting, bitmaps or similar for finite universes'** can help improving performance. There, often more efficient techniques than general purpose search algorithms can be used. Examples are bucket sorting [58], array lookups or bitmaps [59]. The latter is used in memory management, too, and known under the term bitmap allocator. There, it is used to keep track of unused memory locations for book-keeping purpose [60]. Another example refers to the virtual-to-physical address space translation of pages. In the translation process the CPU's translation look-aside buffer (TLB) is consulted first [61]. It is a small cache with a fixed number of entries that maps virtual addresses to physical ones. However, if the lookup in the TLB fails, the processor then refers to the page table, where a prefix of the address that is looked up is used to index the table directly. With that said, both examples avoid the use of hash tables, binary search or other general purpose algorithms.

The use of **'efficient data structure and algorithmic techniques' as in P15** can also improve performance of systems. However, in many cases, principles P1 to P14 mostly apply before any algorithmic issue becomes a major bottleneck under the assumption that the system was designed carefully. Nevertheless, algorithmic techniques as described in [62], [63] can range from standard structures to techniques as divide-and-conquer or randomized algorithms.

Chapter 3

Related Work

Research projects that are related to this work are shortly presented within this chapter. At first, introductory information about the processing path of the Linux kernel networking subsystem is provided, since LANA is implemented in kernel space. Understanding this packet processing path is essential to find suitable ingress and egress points for the LANA framework. Then, related projects to LANA are discussed. This concerns FreeBSD's Netgraph, MIT's Click modular router project, the x-kernel and the prototype of the ANA project.

3.1 Linux Kernel Networking Subsystem

The journey of a network packet from the device driver layer up to the dispatch of packets to the socket receive handlers is discussed within this section for both the ingress and egress direction of the network stack. Throughout this subsection, our descriptions are referenced partly to Love [64], Benvenuti [65] and the Linux kernel source code [66] itself. However, only lower-level kernel parts are covered without going into detail of Linux protocol implementations or routing, which could be handled differently in LANA. Since recent literature does not cover the packet paths in depth, we have used the kernel's source code and written a small kernel module that is able to generate stack traces during runtime (appendix D.4) to gain a detailed knowledge.

3.1.1 Packet Path in Ingress Direction

If a network packet is received on the hardware layer, for instance on a PCI network interface card, a `PCI MSI-X` interrupt request is sent by the hardware to the CPU to signal the availability of data. The CPU then executes the kernel's `do_IRQ` function and invokes the registered interrupt handler of the network device driver (figure 3.1). The context of the function call to the interrupt handler is the top half context where further incoming interrupts are disabled, thus the

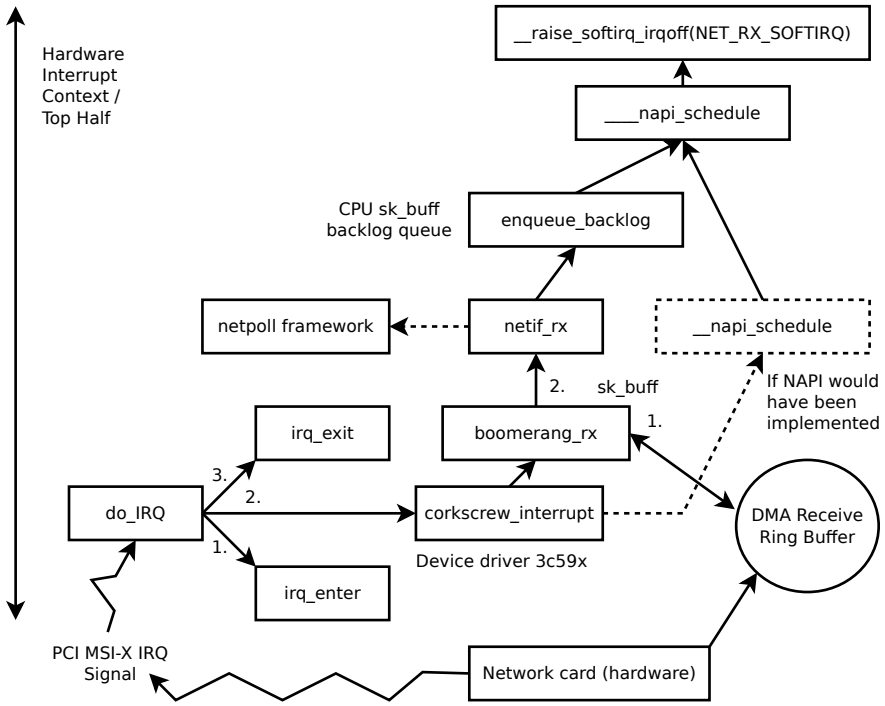

Figure 3.1: Overview of the packet ingress path within hardware interrupt context

handler cannot be interrupted by other sources during execution. Such interrupt sections are marked within the kernel with functions `irq_enter` and `irq_exit`. This, on the other hand, means that the interrupt handler must return as quickly as possible to allow continuing normal operation of the kernel. However, within the NICs interrupt handler, usual operations are performed as the following:

- Copy of the frame into a `sk_buff` data structure (the Linux representation of network buffers, later also referred to as *socket buffer* or `skb`). However, if direct memory access (DMA) is used by the hardware, then only the `sk_buff` data pointer is set (figure 3.2)
- Initialization of `sk_buff` parameters like `skb->protocol` that are used for upper layers in the stack
- Update of private device parameters like per-CPU network statistic counters
- Signalization to the kernel of a new frame by scheduling a `NET_RX_SOFTIRQ` software interrupt for execution (figure 3.1)

Besides packet reception, interrupts from the network card can be signaled for different reasons. Thus, the kernel is given a number along with the interrupt notification for the interrupt service routine of the device driver.

Regarding DMA, most of today's network device drivers usually have at least one internal receive and at least one internal transmit ring buffer for DMA memory. As shown in figure 3.2, a typical network device driver maps the `sk_buff`

data structure into its DMA slot of the receive ring buffer, except for small packets, they are directly copied into non-DMA memory by the driver as in figure 3.2. The `boomerang_rx` function of the `3c59x/3c515` network device driver is only one example that performs such operations on DMA memory. This function is called directly from the interrupt service routine of the `3c59x/3c515` driver, namely `corkscrew_interrupt`.

```
static int boomerang_rx(struct net_device *dev)
{
  ...
    while ((rx_status = le32_to_cpu(vp->rx_ring[entry].status)) &
           RxDComplete) {
  ...
        struct sk_buff *skb;
        dma_addr_t dma = le32_to_cpu(vp->rx_ring[entry].addr);
  ...
        /* Check if the packet is long enough to just accept without
           copying to a properly sized skbuff. */
        if (pkt_len < rx_copybreak &&
            (skb = dev_alloc_skb(pkt_len + 2)) != NULL) {
  ...
            /* Small packets are copied directly into the skb */
  ...
        } else {
            /* Pass up the skbuff already on the Rx ring. */
            skb = vp->rx_skbuff[entry];
            vp->rx_skbuff[entry] = NULL;
            skb_put(skb, pkt_len);
            pci_unmap_single(VORTEX_PCI(vp), dma, PKT_BUF_SZ,
                             PCI_DMA_FROMDEVICE);
            vp->rx_nocopy++;
        }
        skb->protocol = eth_type_trans(skb, dev);
  ...
        netif_rx(skb);
        dev->stats.rx_packets++;
  ...
    }
  ...
}
```

Figure 3.2: Device drivers receive ring with DMA memory (3c59x.c)

However, to notify the kernel about the reception of a new frame, most older network device drivers like the `3c59x/3c515` call directly `netif_rx` within their interrupt service routine, if its receive ring is populated with enough Ethernet frames (figure 3.2). Later versions apply a more performant technique, namely Linux NAPI (New API) that was implemented after the findings of Salim et al.

[48].

The `netif_rx` function posts the socket buffer to the network code and acts as the entrance point into the Linux network stack. When the socket buffer structure has reached this point, then it has left the device driver layer. `netif_rx` queues the buffer for upper level processing, which, in most cases, is protocol level code (figure 3.1). Besides passing the buffer to `netpoll`, a kernel space framework for implementing UDP clients and servers (like the Linux kernel source level debugger over Ethernet `kgdboe`), it is being queued in one of the per-CPU `backlog` queues for later processing in software interrupt mode. Either the socket buffer is being enqueued to the current CPU's `backlog` queue or, if enabled, a technique called receive packet steering (RPS) [7] is applied which is used to distribute the load of ingress packet processing across multiple CPUs[1]. There, the CPU's `backlog` queue is selected by a 4-tuple hash value (i.e. IP + TCP, IP + UDP) that is determined by the network packets header data. Nevertheless, in each case all socket buffers are queued up and the interrupt handler returns.

NAPI, on the other hand, is a modification to the device driver packet processing framework, which is designed to improve the performance of high-speed networking through [9]:

- *Interrupt mitigation:* High-speed networking can create thousands of interrupts per second, all of which tell the system something it already knew: it has lots of packets to process. NAPI allows drivers to run with some interrupts disabled during times of high traffic with a corresponding decrease in system load.
- *Packet throttling:* When the system is overwhelmed and must drop packets, it is better if those packets are disposed of before much effort goes into processing them. NAPI-compliant drivers can often cause packets to be dropped in the network adaptor itself, before the kernel sees them at all.

This technique is usually applied in modern Gigabit-Ethernet or 10 Gigabit Ethernet device drivers like `Intel's e1000/e1000e` network driver that is capable of enabling and disabling chip interrupts from software side. The NAPI framework in Linux consists of kernel threads that periodically poll the device driver for new packets done by `net_rx_action` (figure 3.3). The device driver's interrupt handler, on the other hand, arranges this polling through the `__napi_schedule` function (figure 3.1). In any of these cases, `netif_rx` is not being called anymore. Once the NAPI thread decides to poll the device driver or, in other words, to call the device driver's registered poll routines (figure 3.3), all previously stored incoming frames are processed similarly as the interrupt handler did within non-NAPI device drivers [9]. Also, within the poll handler, `netif_rx` is not being

[1]The Linux kernel in general refers to the term 'CPU' in the meaning of CPU core, thus for instance one `Intel Core 2 Quad Q6600` processor has 4 cores and therefore 4 'CPUs' represented in the kernel. On the other hand, two single core `AMD Opteron 250` processors that are present on a system have 2 'CPUs' in the kernel.

called, instead, `netif_receive_skb` is invoked with the socket buffer as parameter (figure 3.3) whose implementation looks similar to `netif_rx` on the first hand. However, the main difference is that the calling context is the software interrupt context instead of the hardware interrupt context.

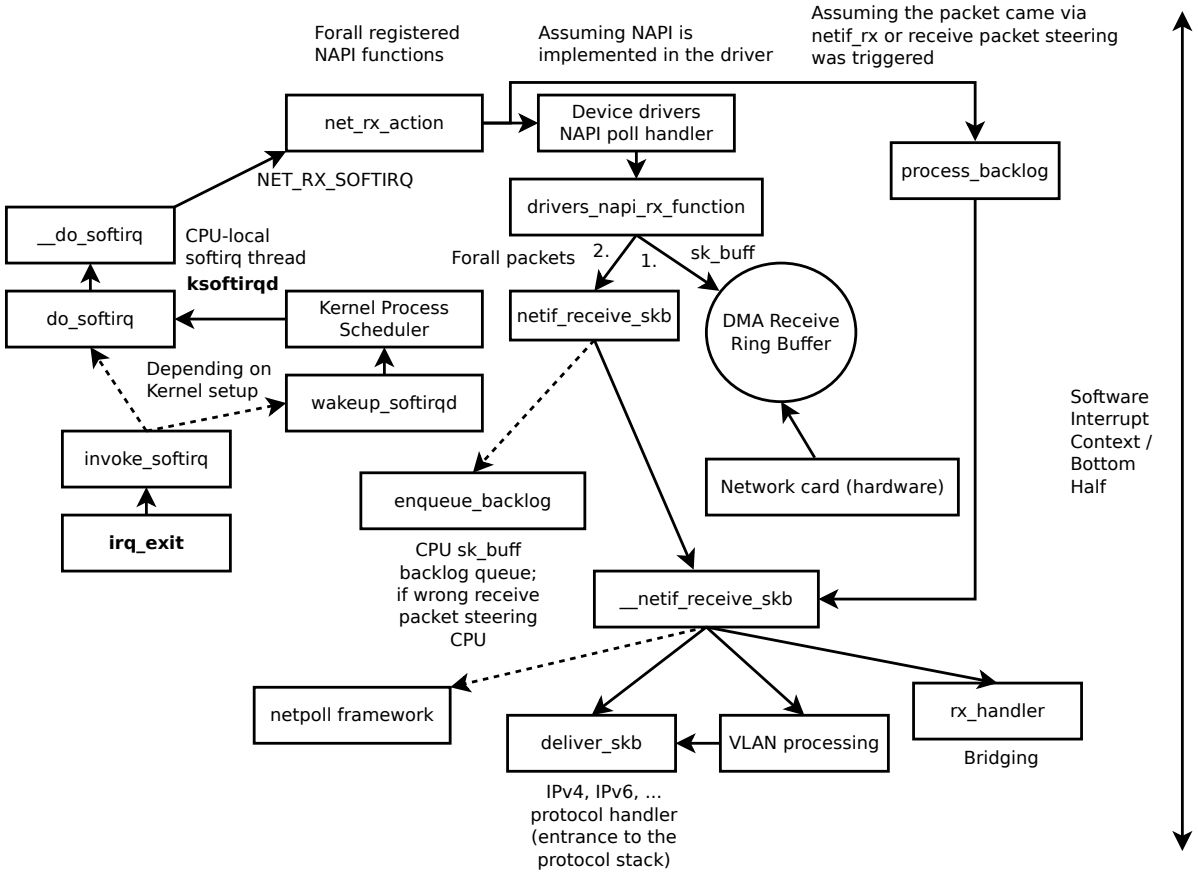

Figure 3.3: Overview of the packet ingress path within software interrupt context

This means that after `irq_exit` was called, the kernel process scheduler schedules the `do_softirq` (figure 3.3) function with the `net_rx_action` function that was registered for `NET_RX_SOFTIRQ`. Within this context, the device driver's bottom half handler (here, the poll routine for NAPI) does all the time-consuming work that was avoided within the top half interrupt handler, because in this case the kernels process scheduler or another interrupt may suspend the bottom half handler during its work. However, the `do_softirq` function is part of the `ksoftirq` daemon which is a per-CPU kernel thread that processes NAPI routines, the CPUs `backlog` queue, timer tasks and other code that can be triggered by software interrupts[2].

[2]Statistics for `ksoftirq` daemons execution reasons like `NET_RX_SOFTIRQ` or `NET_TX_SOFTIRQ` are available in the procfs file `/proc/softirqs`.

Now, within software interrupt context, `netif_receive_skb` can either apply receive packet steering and queue the packet to a different CPU's `backlog` queue than its own like in `netif_rx` or, if unnecessary, the `__netif_receive_skb` function is called (figure 3.3).

In both cases, `netif_rx` and receive packet steering, enqueued packets of the `backlog` queue of a CPU are dequeued via the `process_backlog` function that is called from `net_rx_action`, too.

This function also internally calls `__netif_receive_skb` for further processing. Hence, the CPUs `backlog` queue can also be seen as a transition point from hardware to software interrupt context.

The main focus is now within `__netif_receive_skb`, where both, the new and the old driver approach, join their paths. Within this function, the socket buffer can either be pulled from the `netpoll` mechanism in case the packet came here through NAPI, or otherwise possible VLAN tags are handled and the packet is handed over via `deliver_skb` to upper protocol implementations like IPv4 or IPv6. However, if a special receive handler (`rx_handler`) has been registered by a kernel module to the networking device, the kernel bypasses the delivery to the socket protocols and hands the packet to the `rx_handler` for processing instead. This is the case in Linux network bridges [67], for instance, that connect two Ethernet segments together in a protocol-independent way. There, packets are forwarded based on Ethernet addresses and do not need to walk through the upper protocol stack.

In `deliver_skb` (figure 3.3), on the other hand, a packet handler is called that matches the correct Ethernet type field of the frame which was set within the driver's interrupt handler as in figure 3.2 (`eth_type_trans` function). This can be a value of `ETH_P_IP` or `0x0800` for IPv4 packets, thus the `ip_rcv` protocol handler will be called (figure 3.4) from `deliver_skb`.

```
static struct packet_type ip_packet_type = {
    .type = cpu_to_be16(ETH_P_IP),
    .func = ip_rcv,
 ...
}
```

Figure 3.4: Packet type for IPv4 packets seen from the kernel structure (af_inet.c)

The upper protocol levels will not be in focus of this section, since the description of their complexity would go beyond this work. Basically, protocol functions process their part of the network packet according to the given protocol identifier and call other, higher-level protocol handler such as for UDP or TCP. These protocol handler also have `NF_HOOK`s included, which are callbacks to the `netfilter` framework of Linux. Thus, packets can be dropped according to the given firewall rules that have been passed to the `iptables` front-end from the user space.

Most likely, the packet will end up in a socket receive queue from the UDP or TCP handler for instance. Thus, user space system calls to `read(2)`, `recvfrom(2)`, `recvmsg(2)` and others can then copy the kernel space socket buffer into the provided user space memory buffer. It is also noteworthy, that socket buffers can be queued in one of the protocol implementations before queueing into the socket receive queue. By that, the function can return earlier within the software interrupt context. Processing of such a queued socket buffer will then continue in socket context that is invoked, if an application from the user space performs system calls on that socket. Further details about the socket protocol code can be read in literature [65], [68] and from the Linux source code itself.

3.1.2 Packet Path in Egress Direction

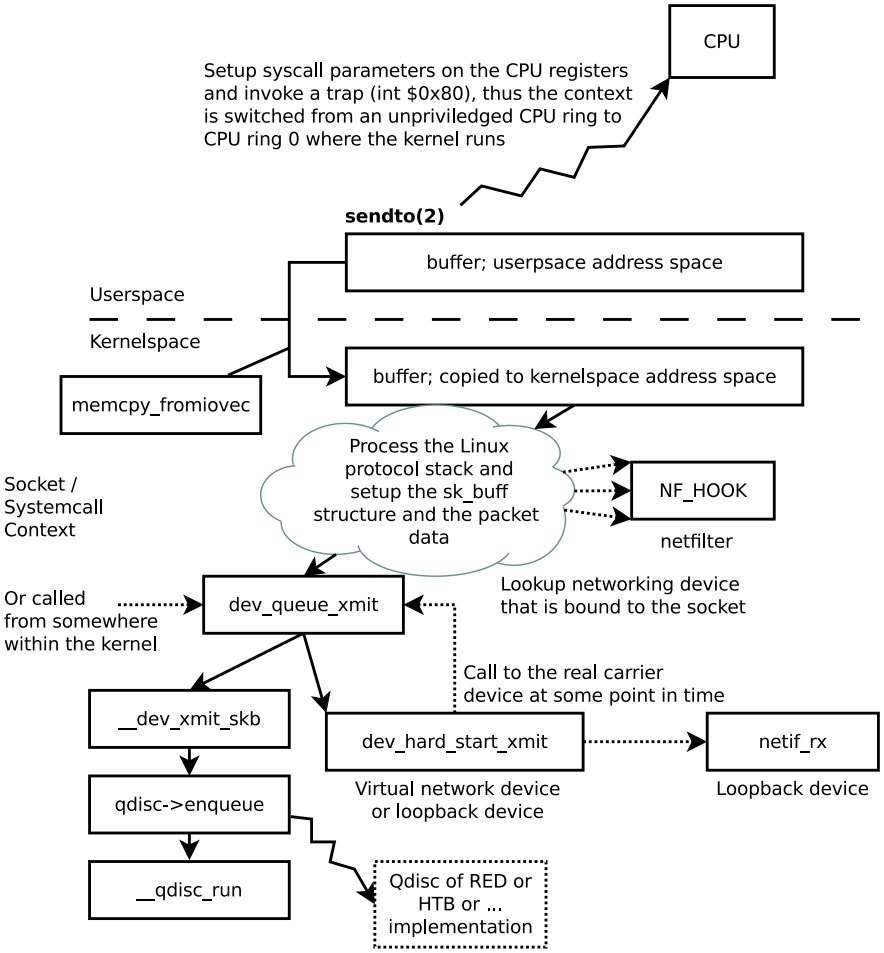

Figure 3.5: Overview of the packet egress path within system call context

The opposite direction, starting from the user space, is quite similar on the first hand. Data that will be transmitted is copied from the user space address

space to the kernel space via `memcpy_fromiovec` into a socket buffer structure, then traverses the protocol layers downwards, again, combined with `NF_HOOK`s from netfilter. At the exit point of the protocol handling functions such as of TCP, UDP or IP, the function `dev_queue_xmit` is being called (figure 3.5).

`dev_queue_xmit` is responsible for queueing a socket buffer for transmission to a networking device. At this point, the packet leaves the protocol layers and enters the traffic management section of the Linux networking stack which, by the way, does not exist on the receive side.

As a side note, `dev_queue_xmit` can also be called within interrupt mode, for instance, if the kernel handles ARP packets and other protocols that do not leave the kernel space, or when bridging is done.

Typically, elements and methods of such traffic management are [69]:

- *Shaping:* shaping is the mechanism by which packets are delayed before transmission in an output queue to meet a desired output rate. This is one of the most common desires of users seeking bandwidth control solutions. The act of delaying a packet as part of a traffic control solution makes every shaping mechanism into a non-work-conserving mechanism, meaning roughly: "Work is required in order to delay packets." However, shapers attempt to limit or ration traffic to meet but not exceed a configured rate.
- *Scheduling:* scheduling is the mechanism by which packets are arranged or rearranged between input and output of a particular queue. The overwhelmingly most common scheduler is the FIFO scheduler. From a larger perspective, any set of traffic control mechanisms on an output queue can be regarded as a scheduler because packets are arranged for output.
- *Classifying:* classifying is the mechanism by which packets are separated for different treatment, possibly different output queues. During the process of accepting, routing and transmitting a packet, a networking device can classify the packet in a number of different ways. Classification can include marking the packet which usually happens on the boundary of a network under a single administrative control or classification can occur on each hop individually.
- *Policing:* policing, as an element of traffic control, is simply a mechanism by which traffic can be limited. Policing is most frequently used on the network border to ensure that a peer is not consuming more than its allocated bandwidth. A policer will accept traffic to a certain rate, and then perform an action on traffic exceeding this rate. A rather harsh solution is to drop the traffic, although the traffic could be reclassified instead of being dropped.
- *Dropping:* dropping a packet is a mechanism by which a packet is simply discarded.
- *Marking:* Marking is a mechanism by which the packet is altered. For instance, traffic control marking mechanisms install a DSCP (Differentiated

Services Codepoint) on the packet itself, which is then used and respected by other routers inside an administrative domain (usually for DiffServ [70]).

Depending on the settings by the user, different traffic management mechanisms can be applied: random early detection queue (RED [71]), hierarchical token bucket (HTB [72]), stochastic fairness queueing (SFQ [73]) and many others. More information about the details of such disciplines can be read in [20] or [69]. The most important kernel data structure where traffic management mechanisms are implemented is called `Qdisc`.

All of these methods are controlled by the user space tool `tc(8)` which is part of Linux `iproute2` suite. The control commands are sent via the Netlink communication mechanism from user space to the kernel's traffic management framework. The usual policy that is applied by default is a simple fifo queue called `pfifo` with a queue length of 1000 slots times the MTU of the device in Byte. The 1000 slots is usually the default length, whose value can be obtained from user space with the `ifconfig` command (see figure 3.6). There, the value is stated in `txqueuelen`.

```
eth10     Link encap:Ethernet  HWaddr f0:bc:ca:38:65:1f
          ...
          TX packets:39805 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:92643642 (92.6 MB)  TX bytes:6941162 (6.9 MB)
          Interrupt:20 Memory:e3200000-e3220000
```

Figure 3.6: `ifconfig` output of the networking device `eth10`

Nevertheless, socket buffer queues that are used in the traffic management layer are represented by double linked lists. Thus in every case, socket buffer structures are passed by reference and not by copy.

Special cases where a networking device has no queue at all, thus traffic management is being bypassed, are loopback devices, VLAN devices and tunnels, for instance. Hence, the packet directly enters the next stage by calling the `dev_hard_start_xmit` function where it leaves the Linux networking subsystem and is being passed to the device driver layer. However, since these are virtual networking devices, at some point in time they need to call the `dev_queue_xmit` function for their non-virtual carrier networking device. Hence, the traffic management procedure is being applied afterwards (figure 3.5).

In any non-virtual device case, the socket buffer is enqueued to the networking device transmit queue via `q->enqueue` in `__dev_xmit_skb`. `__dev_xmit_skb` also triggers the current `Qdisc` of the networking device to run by calling `__qdisc_run` (figure 3.5 and 3.7), which invokes the `qdisc_restart` function. If a certain quota of packets is reached or the CPU is needed by another process, the processing of socket buffers of this `Qdisc` is being postponed by `__netif_schedule` that raises

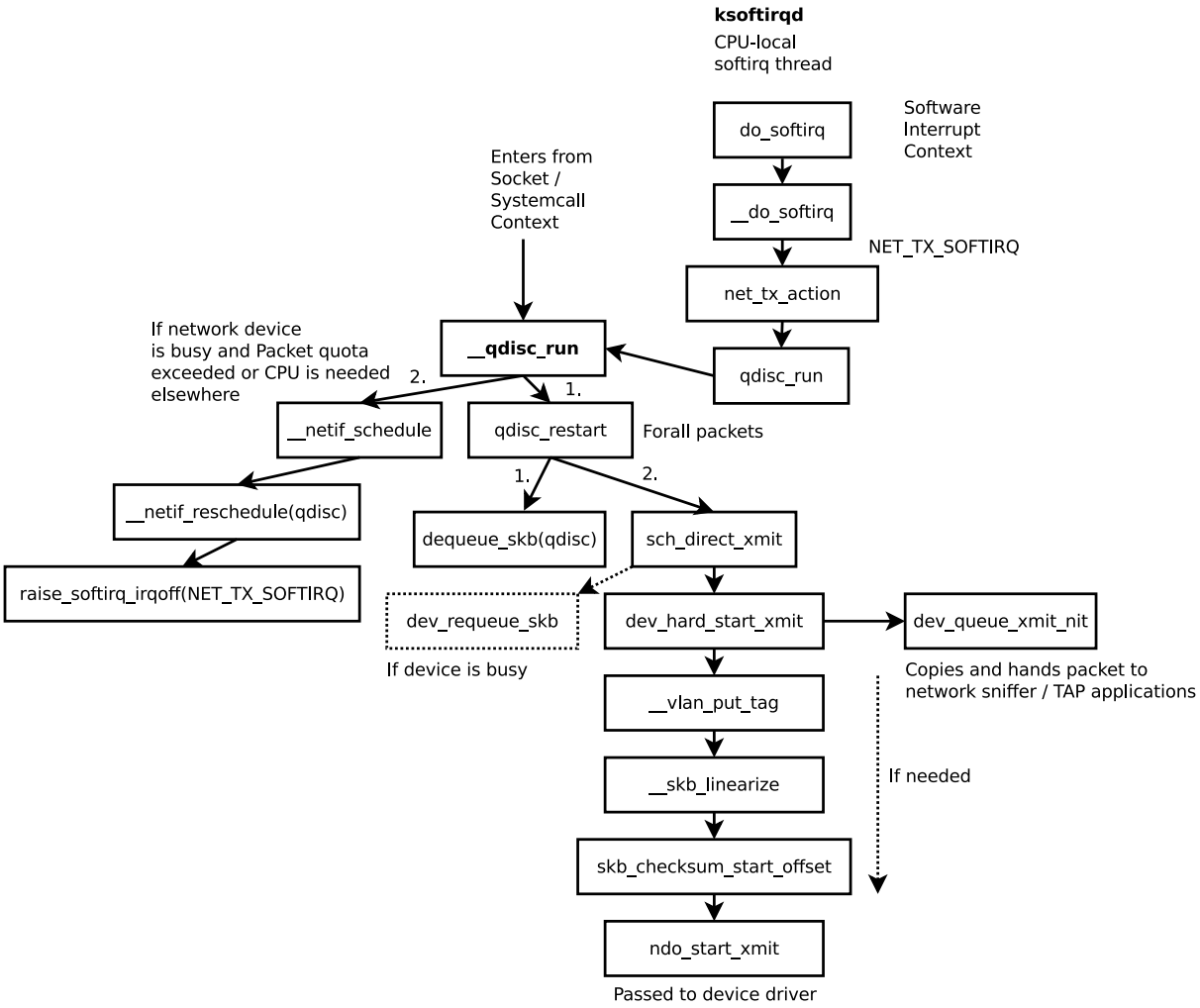

Figure 3.7: Rough summary of the packet egress path within software interrupt context

a software interrupt for `NET_TX_SOFTIRQ`, the opposite of `NET_RX_SOFTIRQ` from the previous section. Hence, all the further transmit processing is transfered to the `ksoftirq` daemon in software interrupt context (figure 3.7).

Within the raised `NET_TX_SOFTIRQ`, the `ksoftirq` daemon runs the `net_tx_action` function which internally calls `__qdisc_run`, too (figure 3.7). Hence, `qdisc_restart` is invoked at this point in time, too, and if the CPU is needed again or a quota has been reached, it registers to the `ksoftirq` for re-scheduling.

This means that both paths continue its actual processing in the `qdisc_restart` function. Within this function, a socket buffer is being dequeued from the `Qdisc` and passed to `sch_direct_xmit`, which locks the transmit queue and passes the socket buffer to the function `dev_hard_start_xmit`. As mentioned, `dev_hard_start_xmit` was called earlier in the fast-path, if the networking de-

vice does not have a transmit queue. However, now it processes the socket buffer for the *physical* carrier device.

The `dev_hard_start_xmit` function prepares the last steps the networking subsystem needs to perform before the socket buffer is finally handed over to the device driver. Hence, `dev_hard_start_xmit` adds VLAN tags and linearizes the socket buffer's underlying data memory (figure 3.7). This is done when the socket buffer's memory is fragmented and the device does not support fragmentation or when the socket buffers memory is fragmented and some fragments are within the high-memory area where the device cannot perform DMA operations. In both cases, the socket buffer is reallocated in non-fragmented and DMA-capable memory. Also, if checksumming of upper layer protocols like UDP or TCP cannot be offloaded to the hardware of the networking device, it is performed within `dev_hard_start_xmit`. Finally, the packet is handed over to the device driver by calling the `ndo_start_xmit` function (figure 3.7). Similar to the ingress path, the `3c59x/3c515` driver implementation is used as an example. This time, relevant parts of the `3c59x/3c515` implementation of `ndo_start_xmit`, namely `boomerang_start_xmit` is discussed.

Within `boomerang_start_xmit` (figure 3.8), the next transmit ring slot is calculated first and the current address of the socket buffer is set to this ring slot. Status flags are determined. Thus, the hardware knows if checksumming still needs to be done, for instance. With such status flags the binary or is calculated on the `skb->len` field, since the upper Bits of the 32 Bit are unused. Furthermore, if the socket buffer memory is linear, the DMA address is being put into the address field of the current transmit ring slot and a status flag is set to mark this single memory fragment as the last one of this socket buffer. CPU interrupts are turned off and the address of the current transmit ring slot is written to the hardware. Thus, it can process initialized fields such as the status register and the address of the current entry. Afterwards, CPU interrupts are enabled and the transmit function returns. The frame is now sent out by the hardware.

```
static netdev_tx_t
boomerang_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
  ...
    /* Calculate the next Tx descriptor entry. */
    int entry = vp->cur_tx % TX_RING_SIZE;
  ...
    vp->tx_skbuff[entry] = skb;
  ...
    /* Set register flags for checksumming, if supported by hardware */
  ...
    if (!skb_shinfo(skb)->nr_frags) {
        vp->tx_ring[entry].frag[0].addr =
                cpu_to_le32(pci_map_single(VORTEX_PCI(vp), skb->data,
                                           skb->len, PCI_DMA_TODEVICE));
                vp->tx_ring[entry].frag[0].length = cpu_to_le32(skb->len |
                                                                LAST_FRAG);
    } else {
  ...
    }
  ...
    spin_lock_irqsave(&vp->lock, flags);
    /* Wait for the stall to complete. */
    issue_and_wait(dev, DownStall);
  ...
        iowrite32(vp->tx_ring_dma + entry * sizeof(struct boom_tx_desc),
                  ioaddr + DownListPtr);
        vp->queued_packet++;
  ...
    iowrite16(DownUnstall, ioaddr + EL3_CMD);
    spin_unlock_irqrestore(&vp->lock, flags);
    return NETDEV_TX_OK;
}
```

Figure 3.8: A networking device drivers `ndo_start_xmit` implementation (3c59x.c)

3.2 FreeBSD's Netgraph Project

A project related to (L)ANA is Netgraph [74], which is part of the FreeBSD operating system kernel. Netgraph is a graph-based kernel networking subsystem and was originally developed with the aim to provide uniform modules for different underlying transmission protocols below the IP layer. The first implementation prototype was targeted for router products that need to deal with bit-synchronous serial WAN connections, i.e. dedicated high speed lines that run up to T1 speeds, where IP packets are transmitted via HDLC (ISO13239), Cisco HDLC[3], frame relay[4], Point-to-Point Protocol (PPP) over HDLC (RFC1662) or PPP over frame relay (RFC1973), for instance [17].

FreeBSD's answer of implementing such scenarios was a system that divides protocol functionality into a single module called *node* with the possibility to combine these nodes in a graph-like manner. Such a combination forms the building blocks of the network stack up to the IP layer. Hence, if a frame has reached the IP layer, traditional, static binded protocol functions are applied.

A connection between two nodes in Netgraph is called an *edge* in the node-graph. Each end-point of the edge is called a *hook*. Thus, nodes can have multiple *hooks* to other nodes (Figure 3.9).

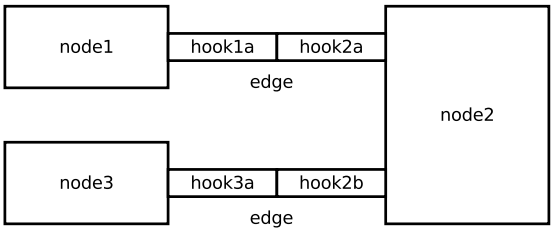

Figure 3.9: Netgraph nodes connected via *hooks* [17].

The data flow on a pair of connected hooks is bidirectionally between nodes. A node may have as many hooks as it needs, and may assign whatever meaning it wants to a hook [74]. Furthermore, there are two data types that can be passed between nodes, in particular *data messages* and *control messages*. However, all exchanged messages are `mbuf`s which are network buffer representations in FreeBSD. There are furthermore two possible data paths: upstream and downstream.

In Netgraph each node has a *type*, which is represented as a unique ASCII name. Compared to object-oriented languages, Netgraph types refer to classes and Netgraph nodes to specific instances of their respective class. The type

[3]Cisco HDLC is an extension to HDLC for multi-protocol support, thus Cisco HDLC frames make use of an alternative framing structure to the standard ISO HDLC.

[4]Industry-standard, switched data link layer protocol that handles multiple virtual circuits using HDLC encapsulation between connected devices, more information: `http://docwiki.cisco.com/wiki/Frame_Relay` (Aug 2011)

implies what the node does and how it may be connected to other nodes [74]. The type name is mainly used for reference from the user space to create instances of a loaded Netgraph module.

Next to *type*, each *hook* and *node* also has a name assignment in ASCII form like in figure 3.9. Both names are used in Netgraph's addressing scheme for control messages. Thus, control messages can send configuration information to arbitrary other Netgraph nodes within the local graph of a host. However, *hook*, *node* and *type* names must not contain the special characters `[:.]`.

In Netgraph, there exists a *relative* and an *absolute addressing scheme* for nodes. If, for instance, `node1` wants to send a control message to `node2`, it can use its absolute address '`node1`' followed by a `:` (colon character) and then the destination. Next to the absolute address, the `.` (dot character) can also be used locally on the source node instead of its name. Relative addresses can be used to describe the destination node via hooks. Hence, if `node1` wants to send a message to `node2`, it uses the address `node1:hook1a` or simply `.:hook1a` [17].

Once a Netgraph node has been created, its lifetime lasts until all hooks have been removed. Thus, the node is not part of the graph anymore. It is automatically removed from the system, except the node is directly bound to a certain hardware. If this is the case, the node stays permanently in the kernel.

Netgraph is configured via user space by the `ngctl` tool. The most important commands of `ngctl` are [17]:

- **connect:** Connects a pair of hooks to join two nodes
- **list:** Lists all nodes in the system
- **mkpeer:** Creates and connect a new node to an existing node
- **msg:** Sends an ASCII formatted message to a node
- **name:** Assigns a name to a node
- **rmhook:** Disconnects two hooks that are joined together
- **show:** Shows information about a node
- **shutdown:** Removes/resets a node, breaking all connections
- **status:** Gets human readable status from a node
- **types:** Shows all currently installed node types

Nowadays, Netgraph has been extended with a larger set of functionality ranging from nodes, which handle ATM, Bluetooth, NAT, Netflow, up to VLAN. However, the source code [75] shows that network packets are not always passed *directly* by reference between nodes. The usual case is that if node A wishes to send an `mbuf` to the neighboring node B, then it calls B's receive handler via the generic Netgraph data delivery method. Nevertheless, each Netgraph node also has an input queue, where incoming data to this node is distinguished to be either a writer or a reader of the internal node state. Therefore, the node implements a reader/writer semantic. Thus, if there is a writer in the node, all other requests

are queued, and while there are readers, a writer, and any following packets are queued [74]. In case there is no reason to queue data, the input method is called directly. If `mbuf`s need to be queued, then they are processed in a later point of time either by a special Netgraph thread that is activated when queued items are present or by the calling thread itself. Next to this, `mbuf`s are automatically queued as soon as a too high stack usage of the current thread is detected with the help of empirical stack usage thresholds. This is necessary to prevent the kernel from running into a stack overflow, since the node receive functions are called successively without returning between two calls.

The main difference from Netgraph to (L)ANA is, that Netgraph was intended to combine various underlying carrier protocols up to IP in an easy manner while still relying on today's Internet architecture. (L)ANA on the other hand follows a clean-slate approach and does not necessarily intend to support or reimplement the traditional Internet architecture. Thus, it rather provides a framework where functional blocks on all protocol layers can be developed and researched.

3.3 Click Modular Router Project

The Click Modular Router is a software developed by the MIT for creating configurable and flexible software routers [18]. Their main motivation for Click is that most routers have inflexible designs and only few configuration possibilities. Thus, network administrators might be able to turn router functions on or off, but they cannot easily specify or identify the interactions of different functions [18]. Also, router extensions require access to software interfaces in the router's forwarding path, but these often do not exist at all, do not exist at the right point, or are not published [18].

Click's router functionality is implemented into so-called *elements* (figure 3.10) which perform simple operations on network packets like decrementing the packets TTL field, doing packet classification or queueing. Their main purpose is that non-complex elements can be connected together to a graph in order to build rather complex routing functionality. Incoming network packets will then traverse a path in the graph and each element processes specific parts of the packet. Such a single element of Click is implemented in the C++ programming language and consists of an *element class*, *input* and *output ports* and optionally a *configuration string*. Element classes can be seen similarly to classes from object-oriented programming languages. The input and output ports are used to connect with other elements in the graph and, in general, an element can have any number of input or output ports [18]. Last but not least, a configuration string is an optional feature where some element classes can support additional arguments for its element initialization.

In Click, there are two types of connections between elements: *push* and *pull* connections (figure 3.10). In a push connection, an upstream element hands a

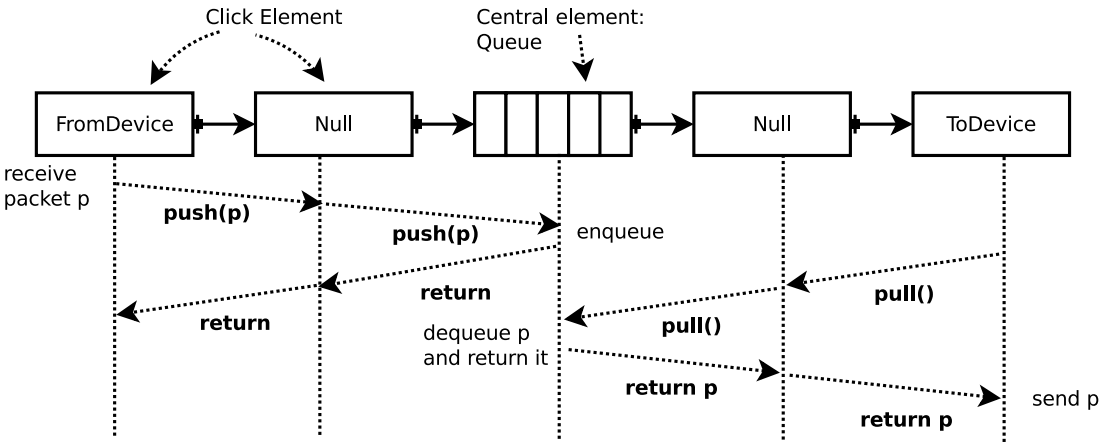

Figure 3.10: Click processing path example with push and pull control flow [18]

packet into a downstream element and in a pull connection a downstream element asks an upstream element to return a packet [18]. Push and pull mechanisms of each element in Click are realized by function calls. As shown in figure 3.10, packets are enqueued into a central queue that is also implemented as an element, so that all push and pull handler can return.

Click elements can be run within kernel space or within user space. If Click is run from user space, it will communicate with the network via Berkeley Packet Filter capable sockets. In contrast to Linux socket buffers, Click has its own packet abstraction. Click also carried out small changes to the Linux kernel code in order to pass control to Click during the receive path of the packet. However, Click code within kernel space is executed in a bottom-half handler and bypasses the traditional Linux network stack.

Summarizing, Click's core system components are [18]:

- **Elements:** atomic entities that process parts of the packet and form the building-blocks of complex routing functionality. Elements can be connected with other elements to build a graph-like structure, where packets traverse a certain path of the graph.
- **Router:** Click's router object mainly collects relevant routing configuration information at initialization time, configures elements and does basic element connection checks for validity.
- **Packets:** Packets are Click's abstractions of Linux socket buffers. The packet data is copy-on-write. Thus, when copying a packet, the system copies the packet header but not the data.
- **Timers:** Some elements use timers to keep track of periodic events. In Click, timers have a 10 ms resolution.
- **Work list:** Work lists are used in Click to schedule Click elements for a later processing. It is a simple single CPU scheduler that is invoked by

Click on either every eighth processed packet or if no packets are queued.

In contrast to (L)ANA, Click is only suited for building software routers, thus it is not applicable for end nodes. Hence, higher-level protocols like UDP or TCP are not of further interest. The main focus in Click is rather on routing and traffic management, whereby (L)ANA focuses on both, end nodes *and* intermediate nodes. Furthermore, Click loads its router configuration via `procfs` on initialization time and completely replaces new router configurations with old ones, thus not allowing to replace only single elements during runtime instead of whole configurations. Regarding implementation, Click made slight changes within Linux kernel source itself which (L)ANA explicitly avoids. Only outdated Linux kernel patches for Click are available.

3.4 x-kernel Project

The x-kernel [76] is an architecture for implementing network protocols. Compared to other related projects within this chapter, the x-kernel is the earliest one that tackled the challenge of introducing more flexibility into the networking stack. The x-kernel architecture describes itself as an operating system kernel that is designed to facilitate the implementation of efficient communication protocols [76].

Therefore, besides protocol implementations, x-kernel has integrated memory management, supports multiple address spaces, offers lightweight process management facilities and kernel-managed interprocess communication mechanisms. For implementing network protocols, three primitive communication objects are available and classified either as static, dynamic or passive in the x-kernel: *protocols*, *sessions*[5] and *messages*.

Protocol objects are statically loaded, passive and correspond to a conventional network protocol like IPv4, UDP or TCP. The relationships between such protocols are defined at initialization time [76], and protocol objects available in a particular network subsystem together with their relationships are defined by a protocol graph at kernel configuration time [16]. A session, on the other hand, is an instance of a running protocol and contains a protocol interpreter, also passive but dynamically created [76]. It is usually bound to a network connection and maintains its state, for instance, similar to the TCP state machine. Moreover, a session can be regarded as a local end-point of a communication channel and is created as channels are opened or closed. Loosely speaking, protocol objects export operations for opening channels resulting in the creation of a session object, and session objects export operations for sending and receiving messages [16] (figure 3.11). x-kernel messages are active objects containing data that is

[5]Protocols and sessions are also referred to as *protocol objects* or *session objects*.

passed between sessions and protocols. Such messages can be seen as x-kernel's representation of network packets that traverse the protocol stack.

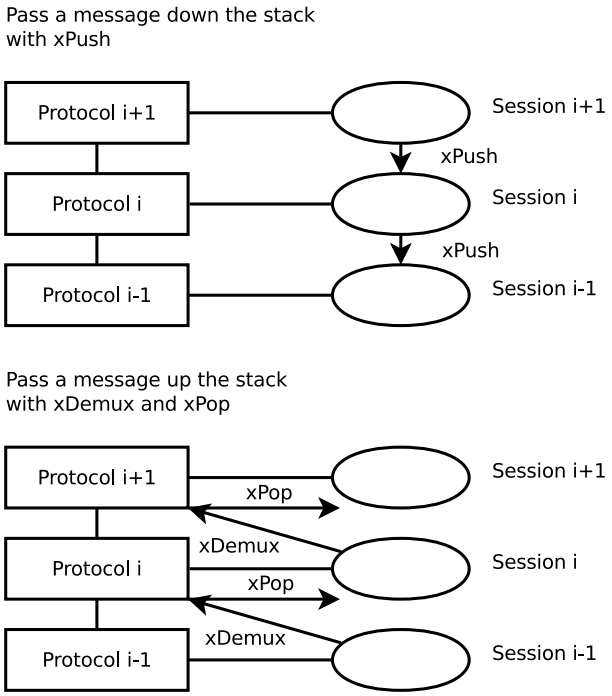

Figure 3.11: x-kernel scheme of passing messages up and down the protocol stack [16]

As shown in figure 3.11, operations as `xPush` and `xPop` are used in each layer to pass a message to the next session object down or up the stack, respectively. However, `xDemux` is used to select the right protocol object depending on the packet's protocol header information. Therefore, `xDemux` also needs to know of all possible protocol objects that are on top of the current one within the protocol stack.

x-kernel's main difference to (L)ANA is that each protocol object must implement a specific protocol in the stack, whereby in (L)ANA this requirement is not tied to a complete protocol. The (L)ANA equivalent of x-kernels protocol objects are functional blocks, which are also able to perform smaller tasks like computing a checksum or more complex tasks than a single protocol, for instance. Moreover, x-kernel rather focuses on end nodes and not on intermediate networking nodes like routers. Next to this, the x-kernel is a full operating system kernel and not just a framework for building a network stack. Since the development of x-kernel has stopped, it is impossible to compile and run it on today's machines. Furthermore, unlike (L)ANA, x-kernel's protocol graph changes can not be performed during runtime and usually requires recompilation.

3.5 Autonomic Network Architecture

Since conceptual parts of this work are derived from the original Autonomic Network Architecture, its idea and architecture is covered within this section with reference mainly to literature [77].

The basic idea of ANA orientates on the concept of UNIX file descriptors, where applications are able to send data to a file descriptor without having to worry whether the data is actually sent to a file or to the console, for instance. Therefore, ANA has introduced four basic abstractions, namely functional block (FB), information dispatch point (IDP), information channel (IC) and compartment.

A functional block is the building block entity of ANA. ANA's whole protocol stack is built out of functional blocks, whereas a functional block represents a protocol entity that generates, consumes, processes or forwards information. This is not only restricted to abstractions of network protocols. Thus, a functional block can just increment a network packet's TTL field, calculate a packet's checksum or even represent a whole monolithic legacy protocol stack. Functional blocks are the smallest atomic entities of ANA.

Information dispatch points in ANA can be seen as connection pivots between functional blocks. Hence, more generally, they draw an analogy to file descriptors in UNIX. Functional blocks are always accessed via IDPs. Thus, instead of passing data to the functional block directly (through a function call), the data is first sent to the corresponding IDP which then internally passes the data to its destination functional block. Therefore, IDPs can be seen as a level of indirection to functional blocks.

Similar to file descriptors, IDPs are usually represented as integer numbers in ANA, but unlike file descriptors, the binding is of dynamic nature and can change over time. Thus, an underlying functional block of an IDP can be exchanged dynamically during runtime. By this, a functional block sending data to another functional block is not aware of the other functional block's semantic meaning. With the help of IDPs, ANA can provide flexibility to reorganize the communication paths within the network stack.

Further, an information channel in ANA is an abstraction of a communication service between a distributed set of functional blocks, where the term *distributed* refers to non-local functional blocks like two identical functional blocks but on different hosts in the network. If an IC is set up between two functional blocks, they can be accessed via additional, dynamically created IDPs, too. There are four possible types of ICs:

- **Unicast IC:** one-to-one association of IDPs
- **Multicast IC:** one-to-many association where multiple endpoints usually receive data simultaneously in a single transmission (special case: **Broadcast**)

- **Anycast IC:** one-to-one-of-many association where a packet is transmitted to a single member of a group of potential receivers (used in load-balancing)
- **Concast IC:** many-to-one association where multiple senders transmit data to a single receiver that merges the received data to a single packet

Compartments, the fourth abstraction in ANA, can be seen as a context of a region of a network that is homogeneous in some regard with respect to addresses, packet formats, transport protocols or others. This abstraction permits hiding of the internals and specific underlying technologies like the composition of a set of functional blocks of a compartment. However, many compartments can co-exist either in vertical (stacked) or horizontal layers on one networking node. Interworking of compartments is also possible, but compartment-dependent. Possible ANA compartments are Ethernet segments, the public IPv4 Internet, private IPv4 subnets, peer-to-peer systems or others. Their communication entities are all built out of functional blocks.

Two types of compartments exist in ANA, namely *node compartments* (LC) and *network compartments* (NC).

A node-local compartment or LC is present in every ANA node and forms a primary hub for all node-local interactions with ANA elements like functional blocks. Such a node compartment controls what is visible and what not to each functional block, and thus allows implementing resource isolation or fine-grained access control. Therefore, on startup, each functional block publishes itself to the node compartment with a *service* and *context* identifier string. Thus, the node compartment can retrieve requests for a functional block that offers a certain *service* (e.g. `tcp:80`) within a given *context* (e.g. `127.0.0.1`).

The network compartment on the other hand involves several ANA nodes that communicate across an underlying network infrastructure like Ethernet. Hence, it consists of a set of FBs, IDPs and ICs, where the FBs provide the communication stack on each ANA node. Besides end nodes, a network compartment can also include intermediate nodes like routers. Communication in a network compartment is always performed via ICs.

As an example for network compartments, one could think of a corporate network where hosts and other network nodes host a set of FBs that provide the communication stack and use an underlying infrastructure like Ethernet, VPNs, or other technologies that represent ICs. A similar scenario would be a peer-to-peer overlay network where overlay nodes host FBs that provide the communication stack and virtual links that are connected via the Internet that represents ICs. Hence, network compartments can operate at different layers; they can be limited to a single layer (e.g. to a layer of the OSI model), several layers or a whole communication stack within one compartment. Noteworthy is that such network compartments can be stacked. Thus, multiple network compartments can co-exist on a single ANA node.

Figure 3.12 shows an example of four ANA nodes connected via ICs. ANA

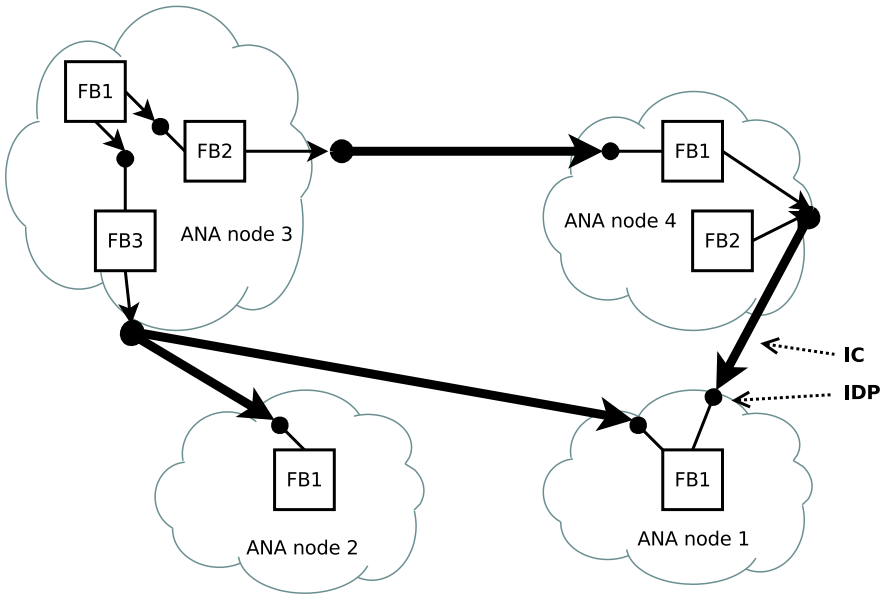

Figure 3.12: ANA network with four nodes connected to each other

node 3 consists of 3 functional blocks namely FB1, FB2, and FB3, where FB1 is able to send data via IDPs to FB2 and FB3. FB3 on the other hand sends its data to an IDP of an Multicast IC that forwards its data to ANA node 2 and ANA node 1. FB2 of ANA node 3 sends its data to an IDP of an Unicast IC. This data is then received by FB1's IDP. Both of ANA node 4's functional blocks send its data to an IDP of an Unicast IC that is connected to ANA node 1.

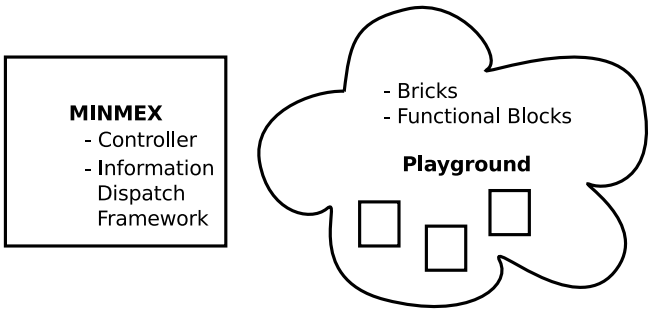

Figure 3.13: Basic components of ANA's implementation

Next to the high-level abstraction (compartment) and to the medium-level abstractions (the internals of the compartment: functional blocks, information dispatch points and information channels), there are also other terms that refer to the implementation level of the ANA architecture.

ANA's implementation consists of two main parts, namely *MINMEX* and the *ANA playground* (figure 3.13). The former stands for 'Minimal Infrastructure for Maximal Extensibility' and is the mandatory software part of ANA that needs

to be present on every ANA node. MINMEX can be considered as ANA's kernel that is needed for bootstrapping a node or for managing functional blocks. Among other things, MINMEX consists of a controller that performs continuous assessment of basic operations of an ANA node with health or sanity checks, for instance. By this, MINMEX aims to protect itself up to a certain extent against misbehaving components. Examples of this include periodical checks of all forwarding paths with loop detection, a garbage collector for unused or expired dynamically assigned IDPs (e.g. from allocated ICs) or a heartbeat check of all active functional blocks.

Next to the controller, MINMEX also contains an information dispatch framework that is responsible for the delivery of data packets to the right IDP of a FB or IC. Within an ANA node, data is forwarded on a chain of IDPs similar to hop-by-hop forwarding of data within the Internet. However, a hop in the meaning of ANA is a functional block that processes the received packet. Depending on the context, the information dispatch framework maintains several IDPs to functional block mapping tables. Interaction with this framework happens through ANA's API with functions like `publish` or `resolve`.

However, the information dispatch framework is not responsible for path setup or path reconfiguration during runtime. This is rather the job of an ANA application. Also noteworthy is that since IDPs are all dynamically assigned, they are encoded into network packets as the next hop field.

The next part of ANA's implementation is the ANA playground. All optional software components that bring actual functionality to ANA are part of the playground. It hosts, for instance, all optional protocols one is free to develop. Although components inside the playground are optional, some standard features are shipped for basic operation. These components are either so-called bricks or functional blocks. In the implementation, a functional block consists of one or more bricks. Thus, bricks represent atomic elements.

Bricks from the ANA playground can be attached in two possible modes: *plugin mode* or *gates mode*. In plugin mode, the brick is usually called a plugin brick and if the brick is loaded within kernel space, it can be called by the MINMEX via function calls. The data reception on that brick is done via message queues. Thus, data is being copied into the bricks receive queue. To protect from concurrency issues, data is copied between every brick. However, if a brick is loaded in gates mode, it is being called a *stand-alone brick*. There, an ANA node's MINMEX communicates either via UDP or Netlink to the brick. Stand-alone means that for each brick a process is being spawned (e.g. a kernel thread or a POSIX thread). In this mode, data that is forwarded between such bricks is copied into the message buffer that is provided for the appropriate IPC mechanism. The MINMEX and the bricks can either be run from user space or from kernel space. The ANA playground has a special brick called *virtual link support brick* (or vlink brick), that is the only brick that has access to the underlying carrier layer, like Ethernet or Bluetooth. The implementation for Ethernet, for instance, has its

own ANA Ethernet type for compatibility with common hardware. Also, a UDP tunnel for ANA has been written to test communication across the Internet.

An example is given where two ANA IP stacks communicate over Ethernet compartments. Node N and node M refer to each of the communication partners:

Inside node M:

FB_{IP} publishes itself to FB_{Eth} with `publish(y,"*","1.2.3.4")`, where y is IDP_y of FB_{Eth}, the 'static' IDP of the compartment that is stored in the MINMEX. The string "1.2.3.4" is the address through which FB_{IP} wants to become reachable from FB_{Eth}. On success FB_{Eth} then stores the mapping between FB_{IP} and its bound IDP_z. Note that this is not stored in the MINMEX, but in FB_{Eth}.

Inside node N:

To instantiate the communication, FB_{IP} calls `resolve(e,"*","1.2.3.4")`, where e is the IDP_e of FB_{Eth}, the 'static' IDP of the compartment that is stored in the MINMEX. The MAC lookup of address "1.2.3.4" is FB_{Eth} specific. However, on success FB_{IP} gets the IDP_s from FB_{Eth}, which can be used by FB_{IP} to send data to FB_{IP} of node M. Again, this IDP is stored in the FB_{Eth}, but not in the MINMEX.

How the resolve works:

The resolving of an address or more specific a service in a certain context can be seen as ANA's equivalent of today's ARP. Since in this example the context is `"*"`, it gets translated into `"FF:FF:FF:FF:FF:FF"` by the $compartment_{Eth}$. FB_{Eth} in M then receives the resolve message, looks into its own translation table, finds out that service "1.2.3.4" is registered and hence replies with a token t used for demultiplexing incoming messages to IDP_z, so that $t \rightarrow z$. FB_{Eth} in N creates a local mapping of the new $IDP_s \rightarrow$ (MAC address, t), where t is the 16 Bit next header field. Afterwards, sending of data from N to M can be done with `send(s,data)`.

Chapter 4

Architecture

This chapter describes the architecture of LANA. At first, requirements for the architecture are defined and basic changes of the core machinery in comparison to the current prototype implementation [78] are discussed. Next, we provide a rough overview of interactions between components and each core component is explained in more detail afterwards. Throughout this chapter, we also discuss major architectural differences to the ANA architecture.

4.1 The Big Picture

The main goal of this work is to design and implement a lightweight ANA architecture for Linux. While we follow the basic ANA principles, the implementation has to significantly outperform the previous prototype implementation. To achieve this, the main part of LANA should run in Linux kernel space. It is not explicitly required that LANA should be implemented from scratch. We discuss major design differences of LANA to the original prototype implementation first and come to a conclusion whether to enhance the current implementation or to build a new one.

In the original ANA architecture, memory that is passed between functional blocks is copied. This is the case of incoming or outgoing network packets, or internal messages that are used to exchange configuration data between functional blocks. We assume that passing memory by creating copies is the most crucial, time-consuming bottleneck of the ANA prototype implementation. Hence, all messages that are exchanged between functional blocks are passed by reference throughout the whole LANA framework. By doing this, we apply principle P1 from section 2.2.1.

ANA's API for developing functional blocks consists of a set of functions that are layered from low-level functions to higher-level functions. There, functional block developers can choose to program their functional block by only using i) the low-level interface, ii) the high-level interface, or iii) a mixture of low-level and

high-level interface functions. High-level interface functions themselves internally call the low-level interface functions. This complexity should be avoided, since it increases the learning curve for novice ANA developers and introduces code bloat as well as a possible performance decrease by multiple, unnecessary layers of abstraction [79]. Ideally, a single-layered, well-defined API exists for LANA that is conform to the Linux kernel coding conventions. This change applies principle P7 from section 2.2.1.

In the implementation of ANA, there is one thread per functional block that communicates with MINMEX. Hence, the more functional blocks are attached to the running system, the more threads are running, which can lead to starvation of other kernel threads, user space processes or functional block threads. Also, on high packet load, the scheduler more often needs to switch contexts between these threads. Instead of having one thread per functional block, we tie this down to only one thread per CPU under the assumption that multiple CPUs are used. This method, further described in the implementation section 5, reduces context switching and exploits locality of data in order to reduce CPUs cache line movements. By optimizing this for multi-core and multiprocessor systems, we also introduce support for NUMA-awareness (non-uniform memory access) for memory that is allocated per CPU. This basic architectural change can be seen as the application of principle P2 from section 2.2.1.

ANA's prototype implementation can be run both, inside the Linux user space and Linux kernel space. If it is run from user space, the modules communicate via UDP or Netlink to the MINMEX. MINMEX receives packets from kernel space via system calls that copy incoming packets into the user space and vice versa. However, the core code of the prototype is system-agnostic, which means that the same core functions are called from kernel space as well as from user space. By this, system agnostic helper routines were implemented in ANA and an own representation of socket buffers was introduced. More specialized and efficient kernel data structures and algorithms could therefore not be used. All in all, we identify bottlenecks that are caused by copying packets into the user space MINMEX which involves system calls and possibly context switches. Also, we identify bottlenecks in code generality, since helper routines were written from scratch and the use of optimized kernel code was avoided. Furthermore, in the user space MINMEX, bottlenecks are assumed in the communication between functional blocks via UDP or Netlink. Next to the time-consuming preparation of network packets for local functional block communication, it is also assumed in the prototype that a native networking stack must be present for UDP or Netlink.

The most major architectural change is that the renewed architecture will be kernel space only. By this, we are able to i) make the architecture more kernel space compliant by using the offered kernel space routines and data structures and to ii) reduce code complexity of all the user space functionality of the original prototype. We also assume that a kernel space only implementation will increase the overall performance of LANA, since critical packet paths are shorter and

address space overhead as well as context switching overhead can be avoided. Multiple principles apply here, such as P1, P3 (shift computation in time), P4, P6, P7 and P8 from section 2.2.

The interception of packets from the Linux core network in the original ANA architecture is done only for packets with a special Ethernet type field by implementing a special ANA protocol handler that is the ingress point of the ANA stack. Instead of implementing a protocol handler, all packets will be pulled from the Linux network stack as early as possible in the receive path without altering the Linux kernel.

A new virtual link layer will be introduced, too. There, for the Ethernet part, virtual Ethernet devices that are bound to a real underlying carrier can be created where incoming and outgoing packets will enter the LANA stack that have special tagged packets similar to VLANs. This has the advantage to manage these devices with standard Linux tools such as `ethtool`, `ifconfig` or `route`.

Further, by implementing everything in kernel space, the BSD socket interface can be exploited for LANA application development in user space which was not the case in ANA. A common set of system calls will be implemented to allow sending and receiving packets for applications.

Unlike ANA's prototype implementation, functional blocks in LANA are seen as object instances similar as in object oriented programming languages. There, an instance of a given class is spawned upon request. Such object oriented code polymorphism, also applied in some Linux kernel subsystems [80] [81], will be similarly used in LANA. Registered functional blocks ('classes') should be able to create instances ('objects') of themselves, whereby one or more instances of the same functional block can be active at a time. To reach this with the current prototype implementation of ANA, one would need to implement multiple copies with slight modifications of the functional block code i.e., a different unique service/context name.

Last but not least, a basic architectural change will be the introduction of a central, single user space tool called `fbctl` that should be able to perform all sorts of functional block configuration tasks like binding or unbinding of functional blocks.

The prototype source distribution of ANA that can be downloaded from [78] consists of 127 C files and 44,142 lines of code including comments and whitespace. Efforts of refactoring these significant changes in the core machinery would be immense, even if some portions of the lines of code refer to existing functional blocks. Therefore, we decided to perform a redesign from scratch that we call *Lightweight* ANA (LANA).

Contrary to TCP/IP stacks as in Linux, the core of LANA is a meta-architecture and does *not* provide any protocol functionality. The core of LANA rather provides management mechanisms for functional blocks. Like in ANA (section 3.5), functional blocks provide the actual functionality of LANA and are connected together. In contrast to ANA's bricks, functional blocks are the smallest possible

entities in LANA. LANA also does not have a MINMEX or Playground as in ANA. The core of LANA on the contrary consists of:

- **Packet processing engine**, that calls one functional block after the other
- **Functional block builder**, that creates new functional block instances
- **Functional block notifier**, that dispatches event messages between functional blocks
- **Functional block registry** and helper data structures, that manage created functional block instances
- **User space configuration interface**, that is able to receive control commands from user space administration utilities

Next to 'normal' functional blocks, functional blocks can be part of the **virtual link layer** (vlink) that can be seen as an interface between device drivers and the LANA system. Functional blocks can also be part of the **BSD socket layer**, where they provide an interface between user space applications and the LANA system.

An example of a running instance of LANA is given in figure 4.1. There, incoming packets are forwarded from the network device driver layer to a functional block (FB 1) that resides within the virtual link layer. This functional block represents the ingress point to the LANA stack for incoming packets. The control is passed to LANA's packet processing engine (PPE) by this functional block, where the PPE manages all further functional block execution. Depending on the interconnection of the functional block instances, the packet processing engine invokes a receive handler of each functional block that processes parts of the network packet. The scenario of figure 4.1 also includes a functional block (FB 7) that is part of the BSD socket layer which leaves the packet processing engine and passes the incoming network buffer to a user space LANA application with the help of system calls. However, the packet processing engine does not necessarily need to pass buffers to functional blocks that are part of the BSD socket layer. It can also pass buffers to kernel space sinks like functional block instance FB 8. What a functional block actually does depends on the implementation and can reach from very simple functions like incrementing the TTL of a packet to more complex functions like routing.

4.2 Components

Having the overall picture in mind, the design of LANA's individual components are discussed within this section. Components include i) the core machinery or back-end of LANA and ii) functional blocks with two special fictitious layers, namely the virtual link layer and the BSD socket layer.

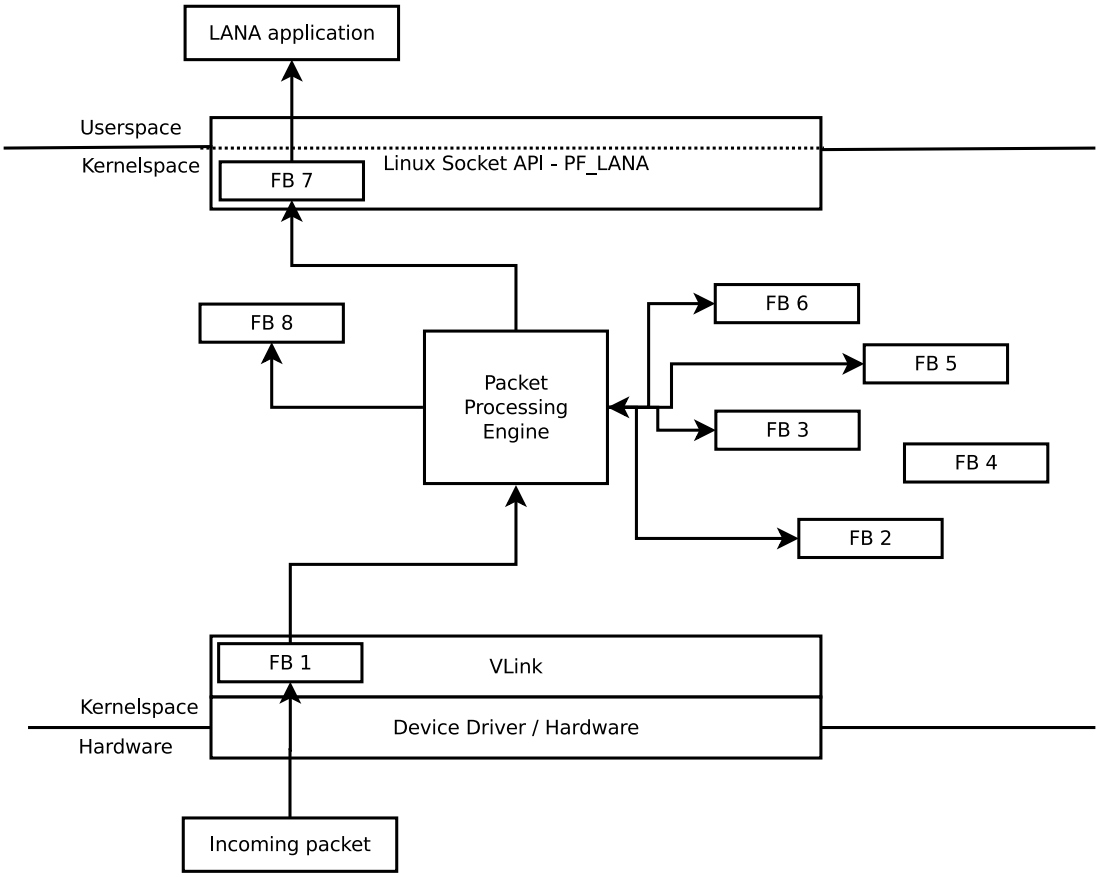

Figure 4.1: Overview of LANA's architecture

4.2.1 Functional Blocks

Functional blocks are LANA's atomic building-blocks of the protocol stack. The core of LANA only provides a meta-architecture without any protocol functionality. Hence, functional blocks fulfill the part of processing packets. LANA can be deployed to a broad range of technologies such as Ethernet, InfiniBand, Bluetooth, or GSM, for instance. A functional block in LANA usually implements a specific protocol in the protocol stack, but functional blocks are not restricted to this. Thus, their functionality can range from performing only small tasks up to containing whole legacy protocol stacks. Example tasks for functional blocks are:

- Checksum calculation
- TTL incrementation
- IP routing
- IP fragmentation
- Packet classification

- Packet duplication
- Payload encryption or compression
- UDP processing or similar
- BSD Socket demultiplexing

Instead of directly accessing functional blocks e.g. via function calls, an indirection layer is introduced which is named information dispatch point. Like in ANA, information dispatch points are represented as Bit-vectors of a fixed size, so that they can be encoded into network packets with respect to their endianness of different hardware platforms. A functional block can only have one IDP at a time. By accessing functional blocks only through their IDP, other functional blocks are not aware of their underlying purpose.

Functional blocks can be bound to other functional blocks, thus a protocol stack in LANA is represented by a graph of functional blocks: vertexes are represented as functional blocks and edges can be seen as a binding of functional blocks.

The process of binding and unbinding is realized by event messages. This means that each functional block can subscribe to an arbitrary number of other functional blocks in order to get notified about new events. If two functional blocks get connected to each other, they are automatically subscribed. The event messaging system can be used to send protocol-specific data to other functional blocks or implementation-specific configuration commands to a running functional block. The dispatch of event messages is independent of the dispatch of packet data and done by the functional block notifier. During runtime, functional blocks can be replaced by others where the IDP is transferred to the new block. By this, a newer version of a functional block can replace the current running version without other functional blocks being aware of this.

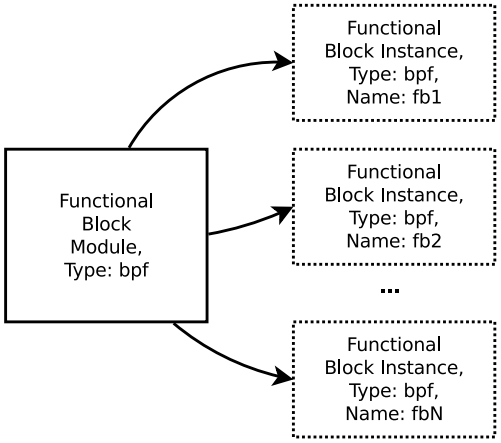

Figure 4.2: LANA's functional block module with spawned instances

When talking about functional blocks in a low-level context, we distinguish between functional block modules and functional block instances. When only

mentioning functional blocks, we usually refer to an instance. The difference between both is the following:

- **Functional block module:** a Linux kernel module that implements a constructor and destructor for its functional block instance as well as commonly used functions; can be seen as a 'class' in object oriented programming languages
- **Functional block instance:** a memory structure with a private data area that was initialized by the constructor of the functional block module; can be seen as 'instances' of a certain 'class' in object oriented programming languages

Unlike ANA, functional blocks are object instances that are created on request by a functional block builder. This means that a functional block module, usually a Linux kernel module, can spawn functional block instances upon demand (figure 4.2).

Each instance has a unique human-readable name for addressing purpose from user space, a unique information dispatch point and its own private data memory containing information about bound functional blocks, for instance. However, all instances from a functional block module have a common type name, and common callback functions for receiving network packet data or event messages. In LANA, all kind of messages exchanged between functional blocks are accessed by reference. Copying of passed network packets as done in ANA is studiously avoided.

Summarized, in LANA each functional block instance consists of the following components:

- A unique information dispatch point
- A unique name that can optionally be a qualified name
- A common type derived from their functional block module
- Private data area containing IDPs to bound functional blocks or other data
- Receive handler for network packets and for event messages from the module
- Subscribed functional blocks
- Own subscriptions to other functional blocks
- A reference counter to determine if the functional block is in use and has dependencies to others
- Common constructor and destructor functions from their module

4.2.2 Functional Block Builder

As already mentioned, LANA uses functional block object instances that are created by a functional block builder as shown in figure 4.3. There, on functional block module insertion, two functions are registered to the functional block builder registry, namely constructor and destructor functions. Both functions need to be implemented for each functional block since they must know what private data parts to allocate, set up and to destruct. If an instance of a functional block is requested, the functional block type and a unique name for the instance must be provided. The functional block type string is then used to look up the constructor and destructor functions of the functional block module. As shown in figure 4.3, the functional block builder then calls the given functional block constructor function which allocates a new instance of a functional block structure and sets up all the necessary functional block internal data. Outside of the functional block builder context, the newly created instance is then registered to LANA's functional block registry and can be used in the LANA stack afterwards. The lifetime of a functional block ends upon deletion request from user space. There, the functional block is unregistered from the core and its destructor, which is responsible for cleaning up internal data and for freeing allocated memory related to this functional block instance, is called.

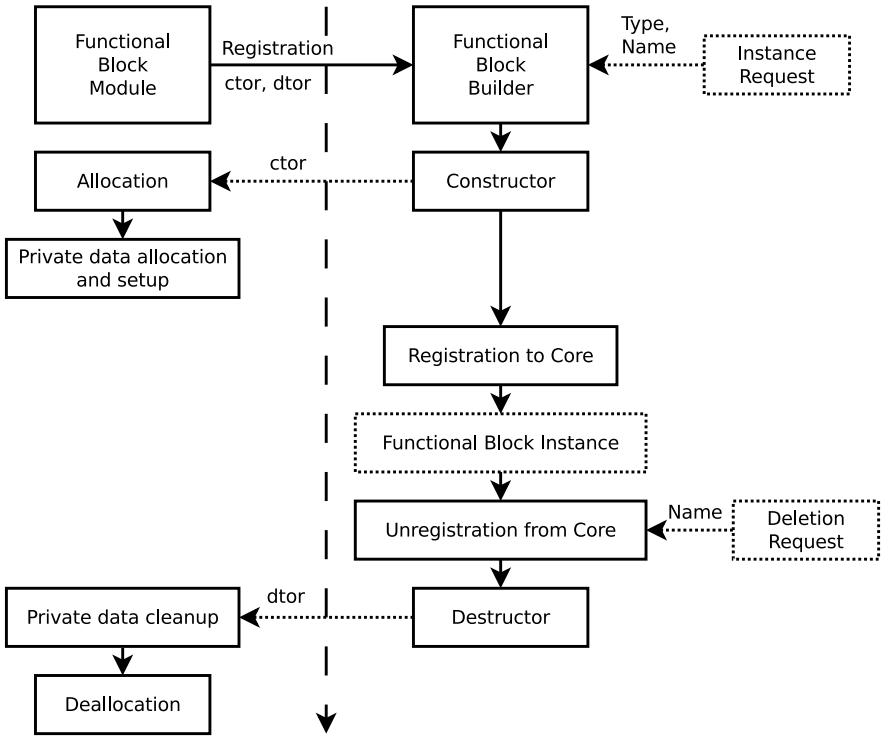

Figure 4.3: LANA's functional block lifetime and interaction with functional block builder

4.2.3 Functional Block Notifier

Another part of the LANA core machinery is the functional block notifier. The functional block notifier is used to notify functional blocks about new events or about configuration instructions. The functional block notifier construct has no central entity and is distributed across all functional blocks. This means that each functional block instance maintains its own list of subscribed functional blocks and of its own subscriptions to other blocks (figure 4.4). The delivery of notifications is done by Linux notification chains.

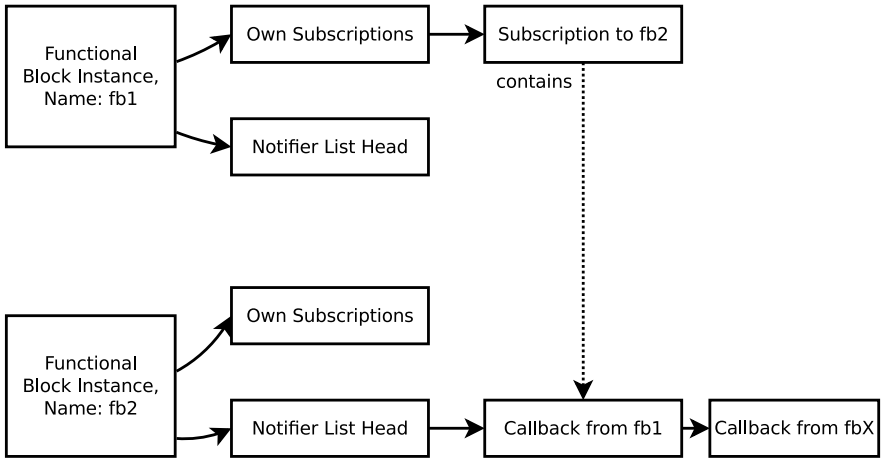

Figure 4.4: LANA's functional block notification chains

If a functional block `fb1` wants to receive event notifications from another functional block `fb2` (figure 4.4), then it needs to subscribe to `fb2`'s notification chain. This is first done by adding a subscription element to its own subscription database for bookkeeping purposes. This subscription element contains a callback function that is provided by functional block `fb1`. This callback function is registered to `fb2`'s notification chain. If functional block `fb2` updates internal information that is needed by some other functional blocks, `fb2` will then execute the registered callback functions of its notification chain with the necessary arguments.

This mechanism is fully asynchronous, so that internal data of remote functional blocks must be protected by locking mechanisms if necessary. The calling context of the provided function from `fb1` will be the execution context of functional block `fb2`. Since this creates also complex functional block dependencies, each functional block is therefore protected from unloading by a reference counter which, besides others, accounts current subscriptions to other functional blocks.

There is one special case of the functional block notifier in which the provided callback function is invoked out of the notification chain context. When performing operations on functional blocks such as binding or unbinding to other functional blocks, the callback function is invoked from the kernel part of the user

space configuration interface (section 4.2.8) in a temporary subscription element that does not belong to a functional block.

In contrast to the original prototype implementation of ANA, this architecture results in a simpler functional block intercommunication model. In ANA, functional block intercommunication is realized by introducing special intercommunication packets, so that communication between functional blocks can be seen similarly to a protocol-like packet communication, where state needs to be maintained.

4.2.4 Functional Block Registry

LANA's functional block registry tracks all active functional block instances and consists of two functions:

- f : name $\rightarrow$ information dispatch point$_{\text{name}}$
- g : information dispatch point$_{\text{name}}$ $\rightarrow$ pointer to structure

Hence, if a functional block instance is addressed by its name i.e., from the user space, the corresponding pointer to the functional block instance is retrieved by applying $g(f(\text{name}))$. However, the usual case in kernel space is an addressing by the information dispatch point of the functional block. Thus, the pointer to the structure of the instance can be retrieved by applying the information dispatch point to the function f. Functional blocks of the LANA stack are not allowed to directly address other functional blocks by their address pointer. Rather, the corresponding information dispatch point has to be used. By this, function g can simply remap the value of a specific information dispatch point (key).

Function g is a performance-critical part of the system since this function is consulted in each IDP to functional block pointer conversion and represents an additional layer of indirection. We assume while function f is only invoked a few times during setup and runtime stack modification, function g is far more often called, especially on a high incoming and outgoing packet rate.

4.2.5 Packet Processing Engine

The packet processing engine is responsible for calling one functional block after the other.

Each functional block instance provides a network packet receive callback function that is conditionally invoked by the packet processing engine (figure 4.5).

Depending on the functional block bindings, a specific network packet receive function of a functional block instance encodes the next information dispatch point into the private data area of a socket buffer. Next, the network packet receive function of that instance returns by telling the packet processing engine

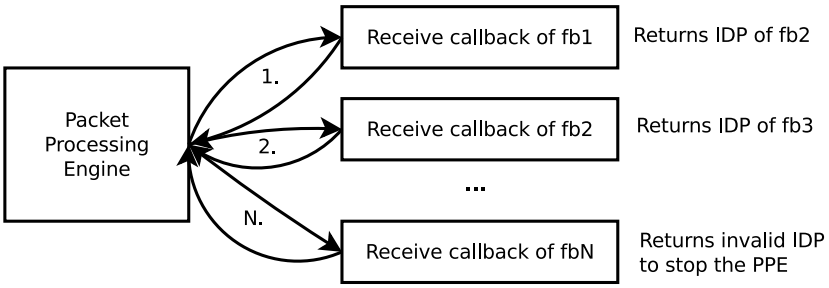

Figure 4.5: LANA's packet processing engine that calls receive handler of functional block instances

whether to continue execution or to stop execution for that specific socket buffer. The execution is stopped in case i) the current functional block is the last one in the processing chain, ii) an information dispatch point was given that is invalid or iii) the functional block drops the socket buffer. If the execution shall continue, the packet processing engine decodes the next information dispatch point from the private data area of the socket buffer. Afterwards, the mapping function g (section 4.2.4) is consulted to translate the information dispatch point into a pointer of the structure of the functional block instance. Next, the receive callback function of that functional block is invoked. The packet processing engine loops through this process until the execution is stopped.

Functional block instances may create new socket buffers that need to be queued for traversing the packet processing engine, too. This is, for instance, the case in functional blocks that clone packets for a network analyzer functional block or in functional blocks that implement parts of the protocol stack and need to send acknowledgement packets to a remote host. Therefore, the packet processing engine contains a backlog socket buffer queue, where new packets can be queued from outside of the context of the packet processing engine. This queue is tested for packets after the processing loop of a socket buffer has been left. If the queue is non-empty, then the head will be dequeued and processed by the packet processing engine.

Next to the socket buffer, information about the path traversal direction is given to the packet processing engine and to the network packet receive callback functions. This direction can either be egress direction or ingress direction.

Functional blocks in LANA therefore have bindings that can be in egress or ingress direction, too. We call one specific binding in egress direction an egress port and respectively one specific binding in ingress direction an ingress port. How many ingress or egress ports (bindings) a functional block has is implementation-specific, but usually, there is one ingress port and one egress port.

Consider the example functional block ingress binding of figure 4.6. A virtual link layer functional block (section 4.2.6) receives an incoming socket buffer and triggers the packet processing engine to run on the current CPU. The packet

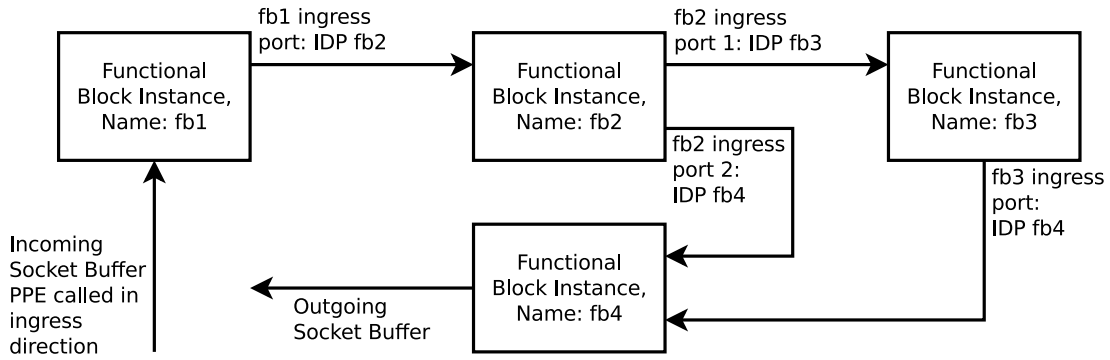

Figure 4.6: LANA's functional block example binding and processing

processing engine calls the receive handler of `fb1` which then forwards the packet to the functional block with the IDP of `fb2`. Being at `fb2`, the packet is conditionally forwarded either to ingress port 1 which is the IDP of `fb3` or to ingress port 2 which will forward the packet to the IDP of `fb4`. If `fb3` is taken into account, then `fb3` forwards the packet after processing to the IDP of `fb4`. Functional block `fb4` could then transmit the packet to an Ethernet device for delivery and exits the packet processing engine. The fact that `fb2` has two egress ports is implementation-specific, so that an internal mechanism exists in `fb2` which is able to decide whether to forward the packet to `fb3` or to `fb4`.

4.2.6 Virtual Link Layer

The functional block in the virtual link layer represents entry and exit points for the packet processing engine on the networking device level. It represents the glue from the operating system to the LANA protocol stack. Functional blocks in this layer are not called directly by the packet processing engine from the ingress path. Rather, they directly receive packets, encode the next IDP into the socket buffer and call the packet processing engine for further stack traversal. Functional blocks in this layer implement the underlying network technology such as Ethernet, Bluetooth, InfiniBand or others. In case of the egress path, they are called by the packet processing engine directly, since they will schedule packets for transmission, they represent the last link of the functional block processing chain.

4.2.7 BSD Socket Layer

Similar to the virtual link layer, there is a BSD socket layer that functional blocks can represent. The main difference to the virtual link layer is that the BSD socket layer is the glue between the LANA stack from kernel space to LANA applications that reside in user space. Therefore, functional blocks in this layer need to implement common system calls such as `socket(2)`, `close(2)`, `recvmsg(2)`,

`sendmsg(2)`, `poll(2)`, `bind(2)` and others. Unlike the virtual link layer, functional blocks in the egress path are not invoked directly by the packet processing engine. Rather, they receive data from user space, encode their bound egress IDP into the allocated socket buffer and invoke the packet processing engine from the socket context. Moreover, in the ingress path, they are called directly by the packet processing engine and represent the last link of the functional block processing chain before the buffer will be copied to the user space. The LANA stack does not necessarily need functional blocks in the BSD socket layer if no LANA user space applications run on that system. This can, for instance, be the case on intermediate networking nodes. There, all processing could reside in kernel space. However, a protocol stack without a virtual link layer would not make much sense, except for networking simulation purposes.

4.2.8 User Space Configuration Interface

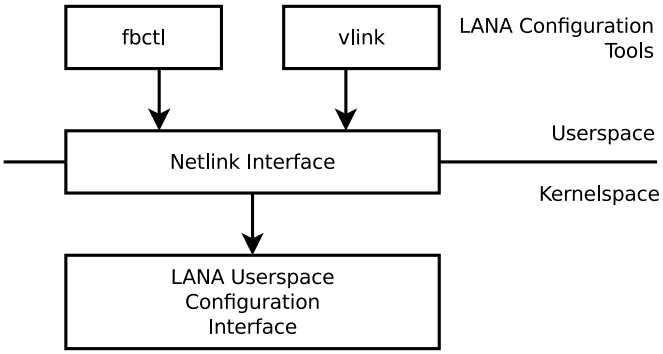

Figure 4.7: LANA's user space configuration interface

The configuration of the LANA machinery is done from user space with the help of two central utilities, namely `fbctl` and `vlink` (figure 4.7). `fbctl` has the role of performing all functional block related configurations such as the creation or deletion of functional block instances, binding and unbinding of functional blocks or subscription to other blocks as well as replacing one functional block instance with another during runtime. The `vlink` utility is responsible for controlling functional blocks of the virtual link layer. If virtual link layer functional blocks support the creation of virtual networking devices, e.g. for Ethernet, then they can be created, managed and removed by the `vlink` tool. Both, `fbctl` and `vlink`, send Netlink messages to the kernel space that are received by LANA's configuration interface.

4.2.9 Controller

Future work on LANA will include a user space controller that communicates through the same Netlink interface as the `fbctl` and `vlink` utilities, but with

the major difference that changes to the LANA protocol stack are performed autonomously by the controller. The controller will have *self-** features such as self-awareness, self-expression and is able to efficiently respond to a multitude of requirements with respect to functionality, flexibility, performance, resource usage, resource costs, reliability, safety or security, for instance. Since LANA is controlled through the Netlink interface, a controller implementation can range from very basic monitoring and reaction tasks up to more complex ones.

Chapter 5

Implementation

Within this chapter, the Linux implementation of the LANA project is described. The focus is first set to the back-end of LANA including its data structures for functional blocks and on the second part, concrete functional block implementations are described. Further, a short overview about the implementation of LANA's user space configuration tools is given as well as a first example application that we have developed.

5.1 Basic Structure and Conventions

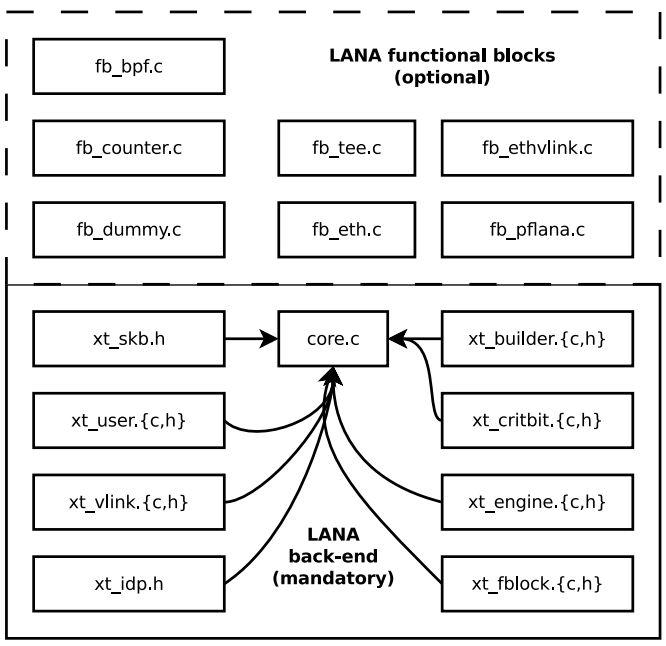

Figure 5.1: Basic code structure of LANA's `src` directory

The source directory of LANA (figure 5.1) contains no further subdirectory, but

files that are defined as the following:

- **core.c file:** The main file of the LANA back-end that implements the kernel module initialization and de-initialization routines and loads or unloads necessary `xt_name` components, which need an initialization respectively de-initialization. Together with all `xt_name` files, it gets compiled to a single kernel module, namley `lana.ko`.
- **xt_*name*.{c,h} files:** LANA core extension files; these files implement library routines or subsystems that are needed for the back-end and offer interfaces via header files for invocation from functional blocks. Each `name` implements a concrete subsystem such as `vlink` or `engine` (packet processing engine), or library functions such as `crit-bit` (crit-bit trees [82]). Names are well-defined in reference to their purpose.
- **fb_*name*.c files:** Each file implements a specific functional block of name `name`. No multiple files for one functional block are allowed. Each functional block gets compiled as its own loadable kernel module.

In contrast to the original prototype, code complexity has been reduced significantly. The whole distribution including functional blocks, consisted of 91 C files and 30,687 lines of code in the original prototype [1]. Our lightweight implementation only consists of 24 C files and 6,597 lines of code (excluding tools from appendix D and the example application):

```
347 ./xt_critbit.c              117 ./xt_vlink.h
 74 ./xt_user.h                 298 ./xt_user.c
869 ./fb_ethvlink.c              93 ./core.c
 21 ./xt_idp.h                  533 ./fb_bpf.c
 80 ./xt_skb.h                  744 ./fb_pflana.c
415 ./fb_eth.c                   64 ./xt_builder.c
 27 ./xt_engine.h               277 ./fb_counter.c
206 ./fb_dummy.c                 21 ./xt_builder.h
189 ./xt_engine.c               251 ./xt_fblock.h
230 ./fb_tee.c                  665 ./xt_fblock.c
345 ./xt_vlink.c                 66 ./xt_critbit.h
239 ../usr/vlink.c              426 ../usr/fbctl.c
            6597 total [lines of code]
```

All files follow the standard Linux kernel coding guidelines [83] and are licensed under the GNU General Public License, version 2 [84]. The implementation was performed under a vanilla Linux 3.0 kernel [85] and without the need for patching the kernel. Everything is implemented as kernel modules. Furthermore, during development, we used Git [86] as a distributed version control system with a public repository that is available at `repo.or.cz` [87].

[1] We have removed the chat application, the IP bricks and routing to have a comparable functionality left over.

5.2 Core Module and Extensions

LANA's core kernel module `lana.ko` includes all basic functionality that is needed for the back-end and for running a protocol stack with functional blocks such as described in section 5.4. On initialization time, the module performs the following steps:

- Setting up a LANA procfs directory under `/proc/net/lana/`
- Setting up a procfs file that shows current functional block instances under `/proc/net/lana/fblock`
- Loading the vlink subsystem
- Loading the functional block subsystem
- Loading the functional block builder
- Loading the Netlink user space interface
- Loading the packet processing engine

On de-initialization time, opposite steps are performed in reverse order, if no other functional block module is present anymore.

5.2.1 Crit-Bit Extension

The crit-bit tree core extension, used by the functional block and the functional block builder extension (subsection 5.2.4), is a library for mapping null-terminated strings to structure pointers, since, to our knowledge, the Linux kernel does not offer a generic data structure for this specific purpose. Compared to a hash table, a crit-bit tree, sometimes also referred to as a radix tree, has comparable speed and two big advantages: the first advantage is that a crit-bit tree supports more fast operations like finding the smallest string, for example, and the second advantage is that a crit-bit tree guarantees good performance, so that it doesn't have any tricky slowdowns for unusual or malicious data [82]. Crit-bit trees are faster than comparison-based structures such as AVL trees (e.g. [88]) and B-trees (e.g. [50]), and they're also simpler, especially for variable-length strings [82].

For this purpose, we have ported Daniel J. Bernstein's public domain crit-bit user space implementation [82] into kernel space, with two major modifications: we have modified the code so that the tree can entirely be used on multi-core systems with efficient RCU protection (read copy update) [89] [90] instead of no locking protection at all, and we have modified the code so that data structures where a given string belongs to can be retrieved with the kernel's `container_of` macro (figure 5.2). The original implementation does not allow for key-value data retrieval. Thus, it is only possible to check whether a crit-bit tree contains a given string or not.

```
#define offsetof(type, member) ((size_t) &((type *)0)->member)
#define container_of(ptr, type, member) ({ \
          const typeof(((type *)0)->member) * __mptr = (ptr); \
          (type *)((char *)__mptr - offsetof(type, member)); })
```

Figure 5.2: `container_of` macro implementation from `include/linux/kernel.h`

However, for the retrieval, there are a few restrictions: the string must be a 'flat' part of the structure (hence, a character array and not a pointer per se) and it must be cache line aligned. The cache line alignment, or at least power-of-two alignment, is needed, since the crit-bit tree internally uses the pointer's least significant bit as a tag to determine if the tree node is an internal node or an external node.

A crit-bit tree has the following basic structure:

```
struct critbit_tree {
        void *root;
        spinlock_t wr_lock;
};
```

The spinlock in RCU context is only used to serialize write operations on the tree. Furthermore, the crit-bit tree has the following API elements that are exported to the kernel:

- `int critbit_insert(struct critbit_tree *tree, char *elem)`: Inserts an element `elem` into the crit-bit tree `tree`
- `char *critbit_get(struct critbit_tree *tree, const char *elem)`: Retrieves an element `elem` from the crit-bit tree `tree`
- `int critbit_delete(struct critbit_tree *tree, const char *elem)`: Deletes an element `elem` from the crit-bit tree `tree`
- `int critbit_contains(struct critbit_tree *tree, const char *elem)`: Checks if the crit-bit tree `tree` contains an element `elem`
- `inline void critbit_init_tree(struct critbit_tree *tree)`: Initializes the Spinlock of the tree root
- `void get_critbit_cache(void)`: Increments the crit-bit tree cache counter that prevents the kernel module from unloading during usage
- `void put_critbit_cache(void)`: Decrements the crit-bit tree cache counter

The above tree access API is an RCU protected variant. The API also contains non-lock holding (only CPU barriers) variants of these functions that are not named here.

Internal elements of the crit-bit tree have the following structure:

```
struct critbit_node {
        void *child[2];
        struct rcu_head rcu;
        u32 byte;
        u8 otherbits;
} ____cacheline_aligned;
```

Concrete functions or meanings of these data structure elements except `rcu` can be taken from literature [91]. The `rcu` element is used for deletion, since such a node will then be enqueued into an RCU deletion queue that gets scheduled after an RCU grace period [89].

Each `critbit_node` tree element is allocated through an slab [92] kernel memory cache that uses hardware alignment for a faster memory access.

```
struct critbit_node_cache {
        struct kmem_cache *cache;
        atomic_t refcnt;
};
```

The atomic `refcnt` element of the `critbit_node_cache` is altered by the two API functions `get_critbit_cache` and `put_critbit_cache`.

A valid LANA kernel structure such as the main structure of functional blocks, that makes use of the crit-bit tree extension, could then look like:

```
struct mystruct {
    char name[IFNAMSIZ];
    int flag;
    ...
} ____cacheline_aligned;
```

Hence, an insert into a valid tree would be performed via `critbit_insert(&mytree, foo->name)`, where we assume that the `name` component of `foo` has a content of `"foobar"` in this example, so that the structure can be retrieved via `critbit_get(&mytree, "foobar")`. Latter returns the pointer of the structure element `name`, that can then be resolved to the structure container via `container_of`, for instance.

5.2.2 Socket Buffer and IDP Extension

Both core extensions, the socket buffer extension and the IDP extension, are header-only extensions that provide storage definitions for LANA-specific control data in Linux socket buffers and for information dispatch points.

The latter defines that an information dispatch point is internally stored as an architecture-independent integer of 32 Bit width. Hence, for all `idp_t` elements the size of 32 Bit applies.

The socket buffer extension provides an API for storing private LANA data within the Linux socket buffer `skb->cb` array. This is used to hand over important data, such as bound information dispatch points, between protocol layers. The following function signatures are therefore exported:

- `inline void write_next_idp_to_skb(struct sk_buff *skb, idp_t from, idp_t to)`: Writes IDPs `from` and `to` into the `skb`'s private data area
- `inline idp_t read_next_idp_from_skb(struct sk_buff *skb)`: Reads the `to` IDP from the `skb`'s private data area
- `inline void write_path_to_skb(struct sk_buff *skb, enum path_type dir)`: Writes the stack traversal direction `dir` (ingress or egress) into the `skb`'s private data area
- `inline enum path_type read_path_from_skb(struct sk_buff *skb)`: Reads the stack traversal direction `dir` from the `skb`'s private data area

The IDP extension is used within the whole source code and the socket buffer extension is mainly used within the packet processing engine extension (subsection 5.2.5) and by functional blocks that spawn new socket buffers or socket buffer clones, for instance.

5.2.3 Virtual Link Extension

The virtual link framework of the LANA core represents the glue between the device drivers of networking hardware and the LANA protocol stack. Functional blocks that are part of this layer need to register themselves to the virtual link framework, so that they can receive and process commands from user space configuration tools like `vlink` (section 5.3).

On initialization of the virtual link framework, a `vlink_subsystem_table` is being created that holds reference to all registered virtual link subsystems. Furthermore, a vlink Netlink group is being created, so that user space tools are able to send Netlink messages to the vlink subsystem. Last, a procfs file is being created under `/proc/net/lana/vlink` that lists all currently available virtual link systems.

A specific virtual link subsystem e.g. for Ethernet, Bluetooth or other technologies, or more concrete, the functional block implementing this subsystem, needs to register the following structure to the virtual link framework:

```
struct vlink_subsys {
        char *name;
        u32 type:16,
              id:16;
        struct module *owner;
        struct rw_semaphore rwsem;
```

```
        struct vlink_callback *head;
};
```

The `name` element represents a unique vlink subsystem name, the `type` element gives information about the underlying technology i.e., `VLINKNLGRP_ETHERNET`, the `id` element is provided by the virtual link framework during subsystem registration, and the `owner` is a reference to the functional block kernel module, which gets locked from unloading from the kernel during packet redirection to the packet processing engine. Furthermore, there is a list of callback functions (`head` element) that is protected by a read-write semaphore `rwsem`. One such list element has the following structure:

```
struct vlink_callback {
        int (*rx)(struct vlinknlmsg *vhdr, struct nlmsghdr *nlh);
        int priority;
        struct vlink_callback *next;
};
```

Such a callback element has a pointer to the implemented function, namely `rx` with a generic vlink message format `vlinknlmsg` and the original Netlink message header `nlmsghdr`. Furthermore, if a callback element is being registered to the list, it can be prioritized by the `priority` element. Last but not least, the element has a `next` pointer for the next list element.

The virtual link framework exports the following API that is used by functional blocks, for example:

- `int init_vlink_system(void)`: Initializes the virtual link framework
- `void cleanup_vlink_system(void)`: Cleans up the virtual link framework
- `int vlink_subsys_register(struct vlink_subsys *n)`: Registers a virtual link subsystem to the `vlink_subsystem_table`
- `void vlink_subsys_unregister(struct vlink_subsys *n)`: Unregisters a virtual link subsystem from the `vlink_subsystem_table`
- `void vlink_subsys_unregister_batch(struct vlink_subsys *n)`: Unregisters a virtual link subsystem from the `vlink_subsystem_table` and removes all callback list elements
- `struct vlink_subsys *vlink_subsys_find(u16 type)`: Returns a virtual link subsystem that has a `type` element of value `type`
- `int vlink_add_callback(struct vlink_subsys *n, struct vlink_callback *cb)`: Adds a single callback element `cb` to the callback list of `n`
- `int vlink_add_callbacks(struct vlink_subsys *n, struct vlink_callback *cb, ...)`: Adds multiple callback elements `cb` to the callback list of `n`

- `int vlink_add_callbacks_va(struct vlink_subsys *n, struct vlink_callback *cb, va_list ap)`: Adds multiple callback elements `cb` to the callback list of `n`
- `int vlink_rm_callback(struct vlink_subsys *n, struct vlink_callback *cb)`: Removes a single callback element `cb` from the callback list of `n`

Currently possible commands that can be accessed from the `vlinknlmsg` within a callback function are: `VLINKNLCMD_ADD_DEVICE`, `VLINKNLCMD_RM_DEVICE`, `VLINKNLCMD_START_HOOK_DEVICE` and `VLINKNLCMD_STOP_HOOK_DEVICE`.

Adding and removing devices refer to virtual LANA networking devices. In section 5.4, for instance, we describe the implementation of an Ethernet functional block that is able to create virtual Ethernet devices that can be managed with standard Linux tools such as `ifconfig`, where networking packets with specific tags are processed. Such devices are created or removed via the vlink `VLINKNLCMD_ADD_DEVICE` or `VLINKNLCMD_RM_DEVICE` commands.

The other two commands, namely `VLINKNLCMD_START_HOOK_DEVICE` and `VLINKNLCMD_STOP_HOOK_DEVICE`, have the purpose to notify the vlink functional block, that it must be switched active or inactive, so that network traffic will bypass the standard Linux network stack and enter LANA's packet processing engines.

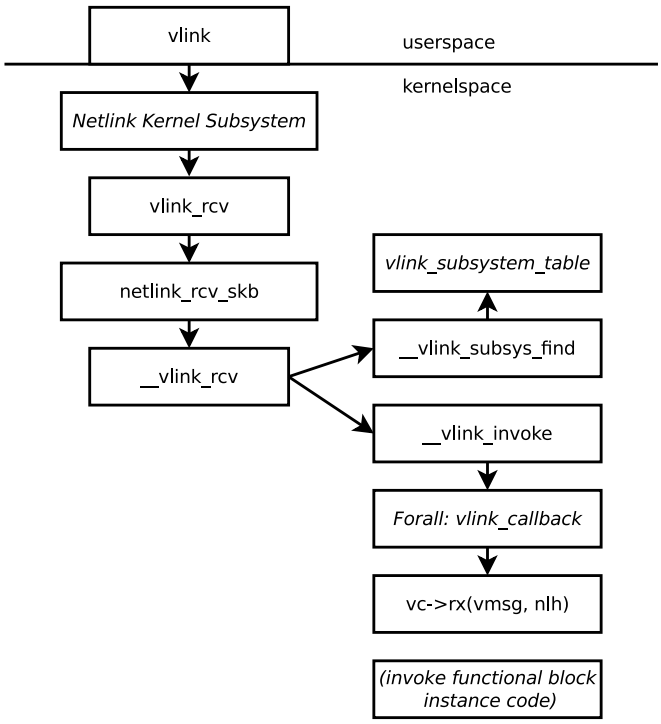

Figure 5.3: Virtual link subsystem invocation from user space

A typical call path for the virtual link framework from user space via Netlink sockets can be seen in figure 5.3. The request first enters the Linux Netlink

messaging subsystem, which triggers `vlink_rcv`, the Netlink handler from the virtual link framework. This handler locks the `vlink_subsystem_table` via `vlink_lock`/`vlink_unlock`, and calls the generic Netlink receive function `netlink_rcv_skb`. This function performs basic Netlink header checks and hands over control to `__vlink_rcv`, which first checks if the request comes from a privileged user, then looks up the corresponding subsystem via `__vlink_subsys_find` and delivers the payload to the callback functions through `__vlink_invoke`. If a hook has been started, the functional block kernel module's reference counter is incremented by the virtual link framework, so that it prevents the user from unloading the kernel module.

In this work, we have implemented two functional blocks (section 5.2.4) that make use of the virtual link framework.

5.2.4 Functional Block, Notifier, Registry and Builder Extension

The functional block and functional block builder extension of LANA provides a common generic framework for implementing functional blocks as described in section 5.4. The structure of a functional block instance that is part of the protocol stack looks like the following:

```
struct fblock {
        char name[FBNAMSIZ];
        void __percpu *private_data;
        int (*netfb_rx)(const struct fblock * const self,
                        struct sk_buff * const skb,
                        enum path_type * const dir);
        int (*event_rx)(struct notifier_block *self, unsigned long cmd,
                        void *args);
        struct fblock_factory *factory;
        struct fblock_notifier *notifiers;
        struct fblock_subscrib *others;
        struct rcu_head rcu;
        atomic_t refcnt;
        idp_t idp;
        spinlock_t lock; /* Used in notifiers */
} ____cacheline_aligned;
```

This main structure is cache line aligned, so that the element `name` can be used in the crit-bit tree for functional block retrieval i.e., during requests from user space. Therefore, the `name` element must also be unique. The next structure element is `private_data` which represents a pointer to the working data of a functional block instance. This can, for instance, include the bound egress and ingress

functional blocks or other implementation-specific data. This `private_data` memory is a `__percpu` pointer, which means that the allocated memory is only accessed CPU local. This memory is allocated NUMA-aware in case of SMP systems. `netfb_rx` and `event_rx` are two receive functions that are implemented by the functional block module. These functions have a reference to their functional block structure, so that private data can be accessed within the function implementation. This access is performed directly from `netfb_rx` and indirectly via the `container_of` macro in `event_rx` through the notifier block structure. The next element is a pointer to the functional block `factory` element that is used by the functional block builder. `notifiers` and `others` are lists that manage event notification subscriptions by the functional block notifier. Locking for this is performed via the `lock` element. The `rcu` element is used to schedule the structure for the RCU deletion queue, that gets active after a RCU grace period. `refcnt` tracks the number of users of this current functional block instance, so that the block gets not cleaned up during usage. If the `refcnt` reaches the value zero, then this functional block instance will automatically be removed from the system. Finally, `idp` is the unique information dispatch point value of this functional block instance. All structure members are accessed via RCU protection. Similar to crit-bit tree nodes, the functional block main structure is allocated through an slab kernel memory cache that uses hardware alignment.

The functional block builder's `factory` element of this structure has the following layout:

```
struct fblock_factory {
        char type[TYPNAMSIZ];
        enum fblock_mode mode;
        struct module *owner;
        struct fblock *(*ctor)(char *name);
        void (*dtor)(struct fblock *fb);
        void (*dtor_outside_rcu)(struct fblock *fb);
} ____cacheline_aligned;
```

Since a crit-bit tree is used for looking up the builder's `ctor` constructor function, the structure is therefore also cache line aligned and has a unique `type` name element. For instance, for the `fb_bpf` functional block, the type name `bpf` is being used. Next to the constructor `ctor`, there are two destructor callback functions that are implemented by the functional block module, namely `dtor` and the optional `dtor_outside_rcu`. The former is a functional block instance cleanup function that is called within RCU context and the latter is a cleanup function that is called *before* the RCU context. Hence, `dtor_outside_rcu` can do some spadework where parts of this function may call the kernel scheduler e.g. through a kernel mutex triggered by changes on Ethernet devices, since this is forbidden in the non-preemptive `dtor` context. The element `owner` is a reference

to the functional block kernel module, that will get locked from unloading if at least one functional block instance is present in the LANA stack. Finally, the element `mode` gives information about the working mode of the functional block, more concrete, if it is a traffic source (`MODE_SOURCE`), a traffic sink (`MODE_SINK`), or a forwarding element (`MODE_DUAL`).

A functional block notifier element, which is used within the main structure of the functional block, namely `notifiers`, consists of the following parts:

```
struct fblock_notifier {
        struct fblock *self;
        struct notifier_block nb;
        struct fblock_notifier *next;
        idp_t remote;
};
```

In this structure, the element `self` is a back-reference to its functional block main structure, since the event callback function `event_rx` from the main structure gets the `notifier_block`'s `nb` element as an argument. By this, the `fblock_notifier` container can be fetched via the kernel's `container_of` macro. Then, `self` provides a reference to the main structure, where per-CPU `private_data` can be accessed to store or read functional block private information. The element `next` points to the next `fblock_notifier` in the list and `remote` is the information dispatch point of the functional block instance where this block is subscribed to. Summarizing, this list contains all subscriptions of this functional block to remote ones.

The element `others` of the functional block's main structure contains an atomic notifier head, namely `subscribers`:

```
struct fblock_subscrib {
       struct atomic_notifier_head subscribers;
};
```

These are other functional blocks that would like to be informed about new events of this functional block. Hence, this list contains remote `nb` elements of the structure `fblock_notifier` that are invoked by the notification chain framework of the Linux kernel.

The functional block registry is realised by two mappings: first, a crit-bit tree is used to translate the unique functional block instance name to a pointer to its main structure. Second, a kernel radix-tree is used to translate information dispatch points into a pointer to its main structure. The difference between our crit-bit tree and the kernel radix tree is, that the kernel radix tree uses `long` integers instead of character arrays as keys. Both data structures are utilizing RCU protection.

The functional block builder, that maintains the presented `fblock_factory` structure, consists of a crit-bit tree for translating the functional block type name to the corresponding pointer to its factory structure. It is then responsible for calling the constructor's callback function and returns a newly spawned functional block instance.

A selection of API calls that are related to functional blocks are:

- `inline void get_fblock(struct fblock *fb)`: Increments the functional blocks (`fb`) reference counter, hence protects the functional block instance from being freed
- `inline void put_fblock(struct fblock *fb)`: Decrements the functional blocks (`fb`) reference counter; if the count is zero, the optional `dtor_outside_rcu` destructor is invoked and the functional block is enqueued into the RCU's deletion queue
- `struct fblock *alloc_fblock(gfp_t flags)`: Allocates a new functional block structure from the functional block kernel memory cache; depending on the `flags`, the allocation can be atomic (`GFP_ATOMIC`) or preemptive (`GFP_KERNEL`) for instance
- `void kfree_fblock(struct fblock *fb)`: frees an allocated functional block structure (`fb`) by putting it back into the functional block kernel memory cache
- `int init_fblock(struct fblock *fb, char *name, void __percpu *priv)`: Initializes the elements of a newly allocated functional block structure `fb` with name `name` and private data `priv`
- `void cleanup_fblock(struct fblock *fb)`: Notifies `fb`'s subscribers that `fb` will be removed, calls `fb`'s `dtor` destructor function, and frees its notification subscriptions; mainly called from within `free_fblock_rcu`
- `void free_fblock_rcu(struct rcu_head *rp)`: If the functional blocks reference count is zero, this function is being scheduled for RCU deletion on the RCU grace period; via the `container_of` macro, the functional block structure container is fetched and `cleanup_fblock` as well as `kfree_fblock` performed
- `int register_fblock(struct fblock *fb, idp_t idp)`: Registers a functional block instance to an existing, but unused IDP into the radix tree
- `int register_fblock_namespace(struct fblock *fb)`: Registers a functional block instance to the system; a new unique IDP is acquired, the functional block instance is registered into the radix tree and into the crit-bit tree
- `void unregister_fblock(struct fblock *fb)`: Unregisters a functional block instance only from the radix tree, but not from the crit-bit tree
- `void unregister_fblock_namespace(struct fblock *fb)`: Unregisters a functional block instance from the system; the functional block instance

is removed from the radix tree, from the crit-bit tree and decrements its reference count for RCU deletion

- `void unregister_fblock_namespace_no_rcu(struct fblock *fb)`: Unregisters a functional block instance from the system; the functional block instance is removed from the radix tree, from the crit-bit tree, but does not decrement its reference count
- `struct fblock *search_fblock(idp_t idp)`: Searches a functional block instance in the radix tree by its idp `idp`
- `struct fblock *search_fblock_n(char *name)`: Searches a functional block instance in the crit-bit tree by its name `name`
- `void fblock_migrate_p(struct fblock *dst, struct fblock *src)`: Migrates the private data of the functional block instance `src` to `dst` by dropping `dst`'s data and transferring `src`'s data without copying
- `void fblock_migrate_r(struct fblock *dst, struct fblock *src)`: Migrates all data of the functional block instance except the private data of `src` to `dst` by dropping `dst`'s data and transferring `src`'s data without copying
- `int fblock_set_option(struct fblock *fb, char *opt_string)`: Notifies the functional block instance `fb` with an event message that includes a null-terminated key value option string `opt_string`
- `int fblock_bind(struct fblock *fb1, struct fblock *fb2)`: Binds two functional block instances to each other assuming that `fb1` is 'on top' of `fb2` in the protocol stack
- `int fblock_unbind(struct fblock *fb1, struct fblock *fb2)`: Unbinds two functional block instances from each other that previously have been bound to each other
- `int subscribe_to_remote_fblock(struct fblock *us, struct fblock *remote)`: Subscribes the functional block instances `us` to the remote functional block instances `remote`, so that `us` gets an event message, if `remote` notifies its subscribers
- `void unsubscribe_from_remote_fblock(struct fblock *us, struct fblock *remote)`: Unsubscribes the functional block instances `us` from the remote functional block instances `remote`, so that `us` does not get messages from `remote` anymore
- `inline int notify_fblock_subscribers(struct fblock *us, unsigned long cmd, void *arg)`: Invokes the atomic notification chain of the functional block instance `us` with the command `cmd` and payload `arg`

Next to these API functions, there are also non lock-holding variants for some cases and macros that are not further mentioned here.

All functional block instances with additional information such as their bound functional block instances are exported to the procfs file `/proc/net/lana/fblocks`.

Moreover, a procfs directory `/proc/net/lana/fblock/` is being created for functional block specific procfs files.

5.2.5 Packet Processing Engine Extension

The packet processing engine extension, used to execute one functional block receive handler after another, runs within the `ksoftirqd`, the kernel software interrupt daemon, which is a low-priority per-CPU kernel thread.

Each CPU local packet processing engine has NUMA-aware structures for packet- and functional block statistics and backlog queues. The backlog queue holds socket buffers that are scheduled indirectly for LANA stack traversal. One socket buffer at a time is usually directly processed by the packet processing engine. In case a functional block instance spawns a new socket buffer, e.g. through socket buffer cloning or through sending acknowledgement packets back to the sender, it is scheduled for later processing in the CPU local backlog queue on the next packet processing engine run.

The packet processing engine API only consists of four exported functions:

- `int process_packet(struct sk_buff *skb, enum path_type dir)`: Invokes a packet processing engine run for a specific socket buffer `skb` in a specific stack direction `dir`
- `void engine_backlog_tail(struct sk_buff *skb, enum path_type dir)`: Enqueues a socket buffer `skb` that needs to be processed by the packet processing engine in a specific stack direction `dir`; processing is not performed immediately, rather the packet will be processed on the next `process_packet` invocation; scenarios for this function call are for instance functional block instances that spawn new socket buffers during processing
- `int init_engine(void)`: Initializes internal per-CPU data structures for statistics and backlog queues
- `void cleanup_engine(void)`: Cleans up internal per-CPU data structures

Within the RCU-protected `process_packet` function, the next information dispatch point is read from the socket buffer via `read_next_idp_from_skb` and its corresponding functional block structure pointer is being looked up. On success, the functional blocks receive handler is being called via `fb->netfb_rx(fb, skb, &dir)`. The direction `dir` is handed over to the receive handler as a pointer, so that a functional block instance can also flip the processing direction. The functional block needs to take care of writing the next information dispatch point into the socket buffer via `write_next_idp_to_skb`. Thereby, the engine loops through all bound functional blocks as long as yet another information dispatch point has been encoded. After having processed a particular socket buffer, the

CPU local backlog queue is tested for waiting socket buffers. If at least one socket buffer is present, it is dequeued and processed in the same manner.

The packet processing engine exports its per-CPU statistics into the procfs file `/proc/net/lana/ppe`.

5.2.6 User-Interface Extension

Similar to the virtual link extension, the user-interface provides a Netlink group for requests from user space configuration tools. Since all possible requests are performed through this core extension, the user-interface is therefore one of the main users of the exported functional block API. Depending on the Netlink command, there are eight different possibilities of processing a message. Internal functions that can be accessed by a `switch` statement which tests for the Netlink command are:

- `static int userctl_add(struct lananlmsg *lmsg)`: Adds a new functional block instance
- `static int userctl_set(struct lananlmsg *lmsg)`: Sends an event message to a functional block instance containing a key-value option string
- `static int userctl_replace(struct lananlmsg *lmsg)`: Replaces one functional block with another; optionally, private data can be dropped instead of replaced
- `static int userctl_subscribe(struct lananlmsg *lmsg)`: Subscribes one functional block with another to receive event messages
- `static int userctl_unsubscribe(struct lananlmsg *lmsg)`: Unsubscribes one functional block with another to not receive event messages anymore
- `static int userctl_remove(struct lananlmsg *lmsg)`: Removes a functional block instance
- `static int userctl_bind(struct lananlmsg *lmsg)`: Binds one functional block instance with another
- `static int userctl_unbind(struct lananlmsg *lmsg)`: Unbinds two bound functional block instances

The `struct lananlmsg` is a generic structure such as in `struct sockaddr` and can be casted to more context-specific structures.

5.3 User Space Configuration Tools

We have implemented two small central user space applications for configuration of LANA, namely `fbctl` and `vlink`. Both send Netlink messages via the AF_NETLINK BSD socket to the user-interface extension (subsection 5.2.6) and

to the virtual link extension (subsection 5.2.3). Both tools can only be executed with a high level of privileges.

`fbctl` has equivalent commands as described in subsection 5.2.6 and is used to maintain bindings, subscriptions, and the creation of 'normal' functional blocks. `vlink` is used to create virtual networking devices, to set up instances of vlink functional blocks and to maintain redirection of traffic into the LANA stack.

5.4 Functional Block Modules

We have also implemented a small set of functional blocks. Two of them are vlink functional blocks, one implements a BSD socket family and the rest of them is about forwarding, statistics creation, packet cloning or packet filtering.

5.4.1 Ethernet, Simple

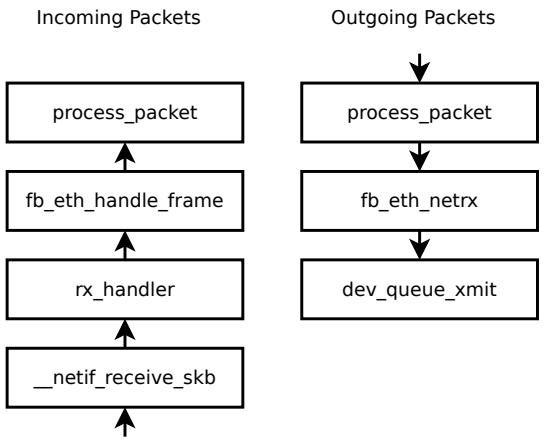

Figure 5.4: Simplified view of ingress and egress path of the Ethernet functional block

The Ethernet vlink functional block connects networking devices to the LANA stack. A single functional block instance is bound to a single networking device. One such instance for the device `eth0` is created with the vlink tool via `vlink ethernet hook eth0` and removed from the system with `vlink ethernet unhook eth0`.

The packet path can be taken from figure 5.4. Recalling section 3.1, the ingress point to LANA is located in the kernel receive function `__netif_receive_skb`. Without explicitly patching the Linux kernel, this represents the earliest possible interface in the receive path a kernel module can access. The functional block instance registers a receive handler for the networking device that is executed before the packet enters the Linux protocol stack. After having the packet processed in the packet processing engine via `fb_eth_handle_frame` which invokes

`process_packet`, the socket buffer is being dropped in every case, so that it cannot reach the Linux protocol stack anymore. `fb_eth_handle_frame` performs some basic packet checks, such as if the device origin of the socket buffer is from a hooked Ethernet device or if its MAC address is valid, for instance. Afterwards, the bound ingress information dispatch point is looked up, written to the socket buffer via `write_next_idp_to_skb` and the packet processing engine is invoked.

The egress point is of different nature (figure 5.4): since no incoming packet is handled in the functional blocks receive function, its only purpose is to transmit socket buffers to the networking device by calling `dev_queue_xmit` and setting the correct socket buffer's networking device beforehand via `skb->dev = fb_priv_cpu->dev`.

5.4.2 Ethernet, Vlink-tagged

In contrast to the Ethernet vlink functional block, the Ethernet vlink-tagged functional block can create virtual Ethernet devices that are visible to standard Linux tools such as `ethtool` or `ifconfig`.

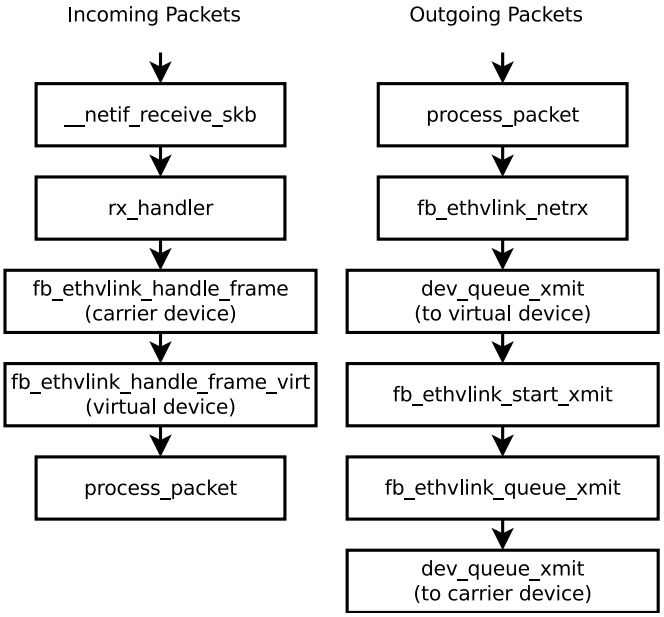

Figure 5.5: Simplified view of ingress and egress path of the Ethernet vlink-tagged functional block

Similar to VLANs, the Ethernet type field must consist of `0xAC` as the first byte and a second byte with the virtual device identifier. The virtual device identifier is set up via the `vlink` tool i.e., `vlink ethernet add mydev0 eth0 10` sets up a virtual Ethernet device that is visible under `mydev0` and a functional block instance that is bound to this device, and uses `eth0` as a carrier device. If

Ethernet traffic has an Ethernet type field of `0xAC0A`, then this frame is received on the virtual Ethernet device `mydev0`.

The ingress and egress packet path is slightly longer since the packet must pass the virtual device and the carrier device (figure 5.5). `fb_ethvlink_handle_frame`'s task is to perform basic packet checks and to redirect the frame to a possible virtual Ethernet device, which then receives the socket buffer via `fb_ethvlink_handle_frame_virt` before it passes it to the packet processing engine via `process_packet`.

In the inverse direction (figure 5.5), the socket buffer is passed to `eth_vlink_netrx`, similar as in the simple Ethernet functional block, but this time, the first `dev_queue_xmit` call dispatches the socket buffer to the xmit function of the virtual Ethernet device rather than to the carrier's driver directly.

5.4.3 Berkeley Packet Filter

We have implemented a packet filtering functional block that is based on the Berkeley Packet Filter (BPF). Syntax and semantic of Berkeley Packet Filter language is described in appendix section D.3. Once a packet filter has been loaded into a BPF functional block instance, all traffic that passes this block is evaluated through this filter and either passes it or fails it. In the latter case, the socket buffer is being dropped by this block. The evaluation of filters is done with the kernel virtual BPF machine that is invoked through the macro `SK_RUN_FILTER`. The setup of filters happens via a procfs file within the directory `/proc/net/lana/fblock/`. There, a user simply redirects the filter code into the file with the Unix tool `cat` for instance.

We have developed a Berkeley Packet Filter compiler `bpfc` (appendix section D.3) for the development of filters. `bpfc`'s output can directly be written to the procfs file like `bpfc myfilter > /proc/net/lana/fblock/myfb`.

With `bpfc`, filters can be developed by applying the Assembler-like language that is described in literature [31].

5.4.4 Tee

The tee functional block is a functional block that clones socket buffers. An incoming socket buffer is being copied to a new one, the information dispatch point of the first subsequent bound functional block instance is written to the original socket buffer, and the information dispatch point of the second subsequent bound functional block instance is written to the newly created socket buffer. The latter is scheduled for execution via the `engine_backlog_tail` function.

5.4.5 Counter

The counter functional block creates a procfs file named after its functional block instance within the directory `/proc/net/lana/fblock/`, so that the number of packets and bytes that have passed this block can be exported.

The counter itself is a per-CPU counter unit, where each CPU only updates statistics about packets that have passed this CPU, not others. Hence, during access of the procfs file with standard Unix tools such as `cat`, data from all CPUs are accumulated in kernel space.

5.4.6 Forwarding

The forwarding or dummy functional block is the simplest functional block of all. It receives a socket buffer and immediately passes it to the next bound functional block instance, without performing operations on the socket buffer.

5.4.7 PF_LANA BSD Socket

The PF_LANA functional block implements a BSD socket interface in order to allow the development of user space applications such as in section 5.5. It therefore implements basic system calls for network communication from the kernel space side, such as `socket(2)`, `close(2)`, `recvfrom(2)`, `sendto(2)` or `poll(2)`. Sockets can then be created from user space with `socket(AF_LANA, SOCK_RAW, 0)`. On successful socket creation, a functional block is implicitly instantiated that is directly bound to this socket. Hence, the functional block instance is destroyed on `close(2)`, too.

A simplified view of the packet ingress and egress path in kernel space is shown in figure 5.6. The egress path is less complex than the ingress path.

In the egress path, a `send(2)` system call is performed that switches the context to the kernel space in CPU ring 0 and triggers the execution of the PF_LANA functional block socket handler `lana_proto_sendmsg`. Through the socket structure, the corresponding functional block structure can be looked up. Basic packet and socket checks are performed in `lana_proto_sendmsg`. If these checks have been passed successfully, socket buffer is being allocated through `sock_alloc_send_skb`. Afterwards, the socket buffer structure elements are initialized and the user space buffer is being copied into the kernel space socket buffer through `memcpy_fromiovec`. Then, still in `lana_proto_sendmsg`, the bound information dispatch point is written into the socket buffer's private data area and the control is handed over to the packet processing engine in egress direction via `process_packet`.

The ingress path (figure 5.6) is far more complex and starts for this functional block through the `process_packet` function that calls the PF_LANA functional block's packet receive handler that is named `pf_lana_netrx`. Similar to the

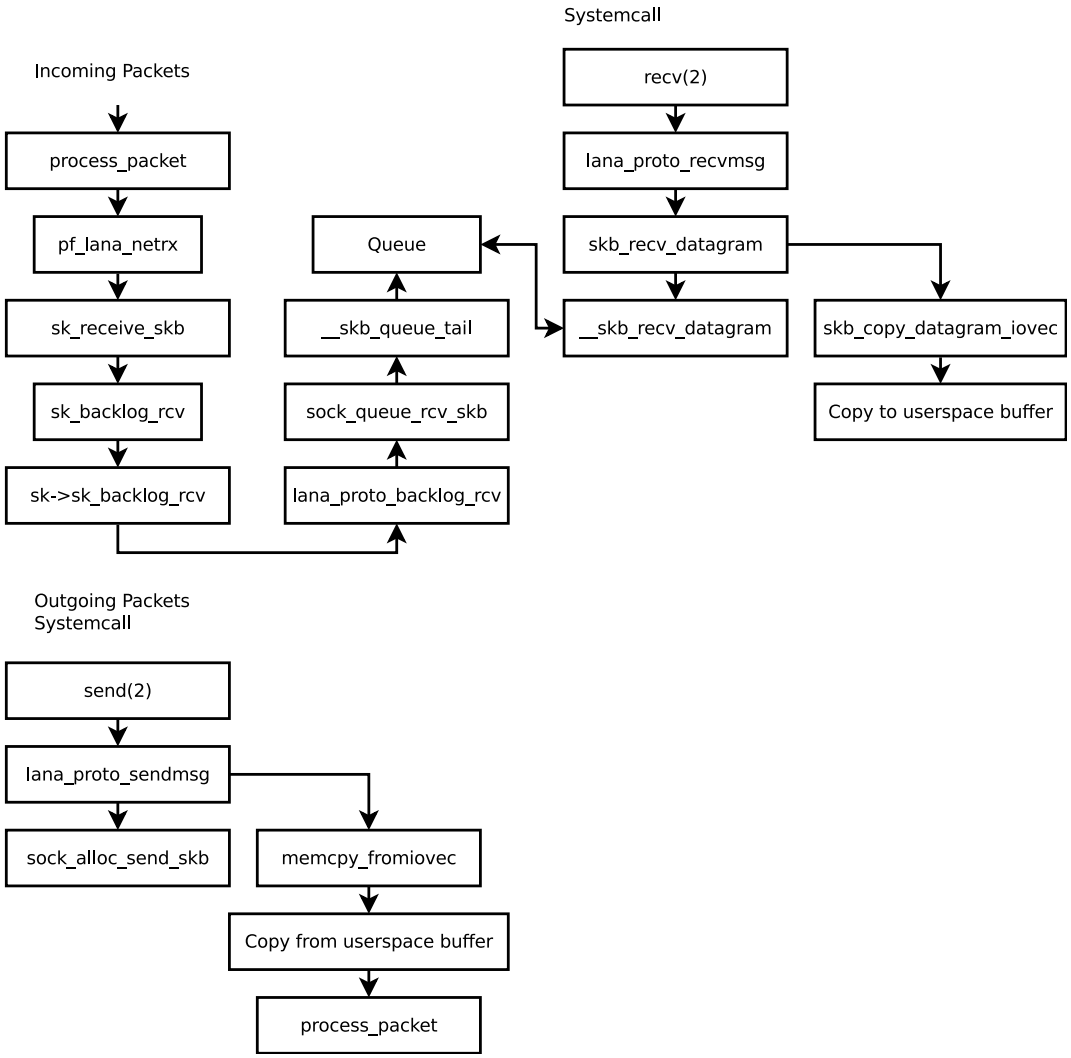

Figure 5.6: Simplified view of ingress and egress path of the PF_LANA functional block

Ethernet functional blocks, this function performs socket buffer checks, such as if the socket buffer is in shared use, so that it eventually must be cloned. The control is then handed over to `sk_receive_skb` with the socket buffer and the corresponding socket structure to this functional block instance as arguments. In `sk_receive_skb`, the Berkeley Packet Filter machine is triggered, if a filter is attached to this socket, and the socket receive queue is being checked, so that the packet is dropped if the queue is already full. If the queue is not yet full, then the function `sk_backlog_rcv` is called. This function internally calls the function pointer that is defined by the corresponding socket functions, implemented in PF_LANA. Hence, `lana_proto_backlog_rcv` is triggered.

In general, such backlog functions of socket family implementations are used

to process packets before they are enqueued into the sockets receive queue, for instance to send acknowledgement packets back to the sender.

In case of PF_LANA, we just check if the socket type is set to `SOCK_RAW` and then trigger the function `sock_queue_rcv_skb`. The main purpose of `sock_queue_rcv_skb` is to unlink the socket buffer from the networking device and to enqueue the socket buffer into the sockets receive queue via `__skb_queue_tail`. Now, the packet is only further processed if the corresponding user space process performs a system call such as `recv(2)`, that operates on the socket buffers receive queue. After the context is given to the kernel space, the `recv(2)` system call invokes the PF_LANA specific system call implementation, namely `lana_proto_recvmsg`. This function first calls `skb_recv_datagram`, which triggers `__skb_recv_datagram`. In `__skb_recv_datagram`, the socket buffer queue is being peeked via `skb_peek`. If the system call flag `MSG_PEEK` is not present, then the socket buffer is removed from the queue via `__skb_unlink` and the control returns to `lana_proto_recvmsg`. This function then checks if the packet is truncated and eventually sets proper flags for the user space caller. Afterwards, the buffer is copied to the user space address space via `skb_copy_datagram_iovec`, timestamps are added and the kernel space socket buffer is being freed.

5.5 Example Application

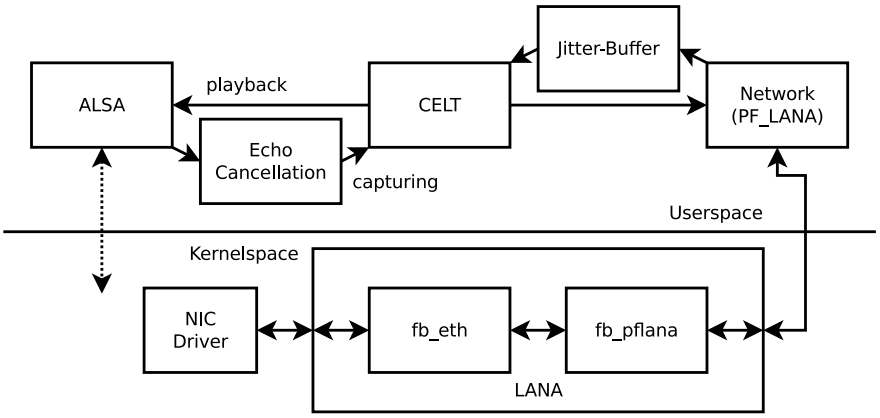

Figure 5.7: LANA example application: voice-over-Ethernet

As an example application for LANA, we have implemented a simple voice-over-Ethernet application (figure 5.7) based on the celtclient [93]. The application runs in the Linux user space and interacts with ALSA [94], the advanced Linux sound architecture, to read and write PCM (pulse code modulation) frames.

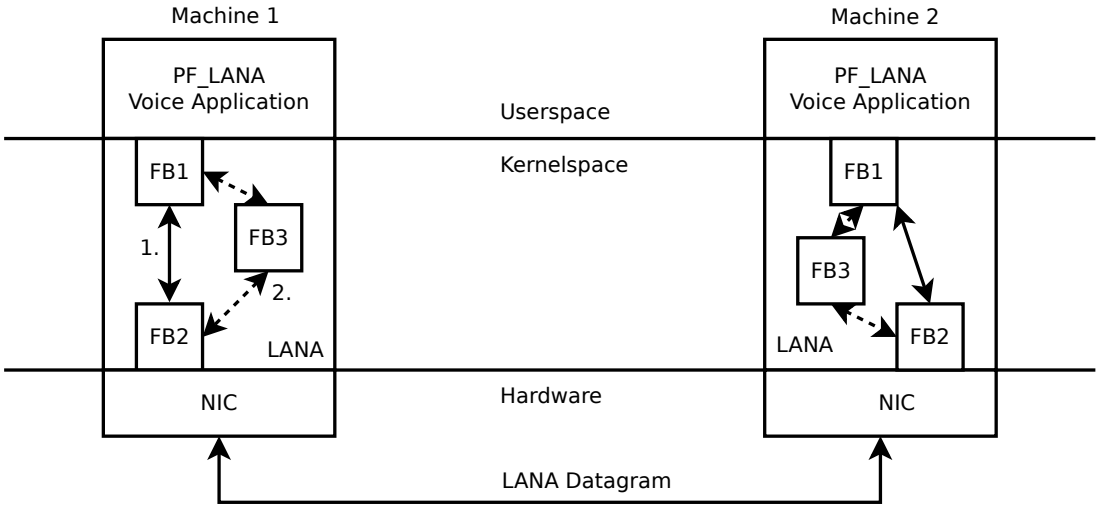

Figure 5.8: LANA voice-over-Ethernet setup between two hosts with runtime changes in the LANA stack

After capturing, the PCM frames are passed to the Speex [95] echo cancellation and are encoded through the CELT low-latency audio codec [96] before they are sent to the kernel with a PF_LANA BSD socket. The latter passes the memory through system calls to its corresponding kernel space functional block instance. The PF_LANA functional block instance then passes the created socket buffer to a bound Ethernet vlink functional block to bring the data to the network.

Another machine within the same LAN segment and the same LANA setup receives the packet and forwards it into the user space via another PF_LANA BSD socket (figure 5.8). The voice data is then being put into a jitter buffer and when the ALSA device signals that it is ready for processing the next PCM frame, data is being dequeued from the jitter buffer, decoded through CELT and written to the ALSA framework. This bidirectional data transmission through our LANA framework (figure 5.8) shows it's robustness of the code and also real-time capabilities since the ALSA buffer needs to be filled periodically to not run into an ALSA overrun or underrun.

More importantly, in the scenario of figure 5.8, we were able to change the underlying communication stack within kernel space during runtime *without* a notable voice interruption on the application side. During runtime, we inserted a forwarding functional block (FB3) between the PF_LANA functional block (FB1) and the Ethernet functional block (FB2).

Chapter 6

Performance Evaluation

Within this chapter, we evaluate LANA regarding performance in packets per second. At the beginning, we focus on functional verification of LANA and afterwards, we describe the hardware platform of our benchmarking setup. In the next sections, we explain our measurement methodology, compare LANA to the Linux networking stack, to the Click router and to the original ANA prototype. Last but not least, we provide details of how we successively optimized the LANA implementation towards its final packet processing rate.

6.1 Functional Verification

The Linux kernel does not support automatic testing frameworks such as the Test Anything Protocol [97], since they are designed for user space applications and run on top of the operating system kernel. Therefore, we have performed LANA functional tests manually without the help of an automatic testing framework. We have performed the following functional tests on our LANA implementation:

- Create a functional block instance under heavy load $\surd$
- Remove a created functional block instance under heavy load $\surd$
- Bind one functional block instance to another under heavy load $\surd$
- Unbind two bound functional block instances under heavy load $\surd$
- Subscribe one functional block instance to another under heavy load $\surd$
- Unsubscribe two subscribed functional block instances under heavy load $\surd$
- Replace one functional block instance with another under heavy load $\surd$
- Redirect traffic into the LANA stack with a functional block instance of `fb_eth` $\surd$
- Create a virtual Ethernet device with `fb_ethvlink` $\surd$
- Remove a virtual Ethernet device with `fb_ethvlink` $\surd$
- Redirect tagged traffic into the LANA stack with a functional block instance of `fb_ethvlink` $\surd$

- Filter specific traffic (i.e., ARP, IPv4) with a functional block instance of `fb_bpf` √
- Duplicate a packet with a functional block instance of `fb_tee` √
- Count the number of packets and Bytes with a functional block instance of `fb_counter` √
- Forward a packet to another functional block instance with the help of a functional block instance of `fb_dummy` √
- Transfer an incoming packet from kernel space to a `PF_LANA` user space socket with a functional block instance of `fb_pflana` √
- Transfer an outgoing packet from a `PF_LANA` user space socket to kernel space with a functional block instance of `fb_pflana` √
- Transfer datagrams between two hosts `via PF_LANA` √

6.2 Measurement Platform

Each machine in our laboratory consists of the same hardware components. The basic setup of our machines is described in the following:

Hardware:

- **CPU:** Intel Core 2 Quad, Q6600, 2.40GHz
  - 4 cores
  - L1 instruction cache: 32KB, L1 data cache: 32KB
  - L2 cache: 4096KB
- **RAM:** 4GB
- **NIC:** Intel Corporation 82566DC Gigabit Ethernet Controller
  - Onboard, PCI Express: 2.5GB/s, Width x1, supports NAPI

The basic setup for performing benchmarking consists of a traffic source that is directly connected via Ethernet to a traffic sink (figure 6.1).

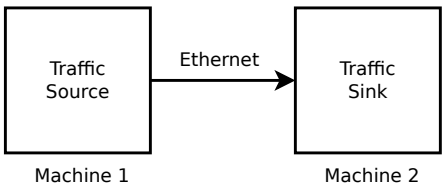

Figure 6.1: Basic benchmarking setup

6.3 Measurement Methodology

Each measurement value of section 6.4 represents the statistical median of a measurement series of a specific benchmarking setup. A measurement series consists of 7 measurement values that we recorded during our benchmark. For instance, when measuring the packets per second performance of a certain traffic sink, we started first with generating a maximum packet load from the traffic source to the traffic sink, where each packet matches a requirement like a packet size that is conform to RFC2544 [98]. These sizes include the maximum (non jumbo frame) and minimum frame sizes permitted by the Ethernet standard and a selection of sizes between these extremes with a finer granularity for the smaller frame sizes and higher frame rates [98]. Then, we monitored our traffic sink with the help of `ifpps` (appendix section D.2) and waited until the system has stabilized if possible. Afterwards, we noted the packet per second measurement value of the system that was generated by a counter application of the currently evaluated framework.

6.4 Benchmarks

We first compare LANA against the bare performance of the Linux kernel, when the kernel is configured to receive and drop packets immediately in order to gather the systems limits and how they correlate to LANA. Afterwards, we measure LANA's internal scalability up to 50 chained forwarding functional blocks. Next, we have developed a user space packet sniffing application that utilizes non-zero-copy BSD sockets provided by i) PF_SOCKET from Linux and ii) PF_LANA from LANA to determine how both relate to each other in terms of performance. Furthermore, we compare LANA with the Click modular router and LANA with the prototype of ANA.

6.4.1 LANA versus Linux and LANA's Scalability

Short summary:

- **Platform:** Linux 3.0
- **Test setup:** 2 physical machines, direct connection
- **Traffic generator:** pktgen
- **Result:** in their minimal configuration, the Linux networking subsystem and LANA have comparable packet per second performance

In our first setup, both machines from figure 6.1 run on a 64 Bit Debian GNU/Linux 6.0.2.1 with a vanilla Linux 3.0.0 kernel that we have fetched from

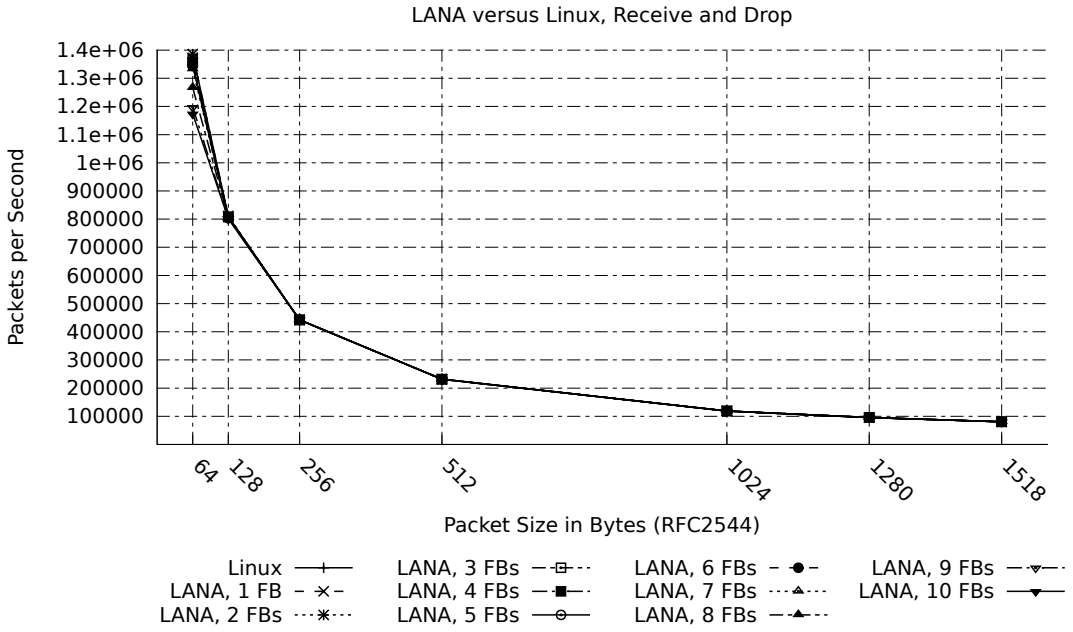

Figure 6.2: Linux versus LANA, 1-10 FBs, increment: 1 FB

Linus Torvald's Git repository [85]. On the traffic source, the kernel space `pktgen` was used as a traffic generator. The traffic sink either run i) LANA with a specific functional block configuration, or ii) the usual kernel networking subsystem. The generated packets on the traffic source were UDP packets of a specified length from RFC2544 [98] with random payload. For reception, the traffic sink was put into promiscuous mode via `ifpps` (appendix section D.2), so that the packets arrive at the Intel `e1000e` driver first and were then forwarded to one of the two networking subsystems through NAPI. Other, driver-specific interrupt throttling settings as described in [99] were not performed for the `e1000e` driver on the traffic sink. For both networking subsystems, we monitored the packets per second rate via `ifpps`. The output values of `ifpps` showed us the network driver's receive statistics that could be gathered through procfs from user space. Since both subsystems run *within* the `ksoftirqd` and have no further threads or socket buffer queues, that could fiddle the network drivers receive statistics by exiting the `ksoftirqd` earlier before all packets have been processed. We used these values to generate our graphs from figure 6.2 and figure 6.3.

Figure 6.2 shows the maximum packet per second rate that can be achieved with the traffic sink's system on Linux and LANA. In both cases packets were received and dropped immediately. In case of LANA, we determined the overhead of the framework by creating a simple forwarding functional block that receives the packet and passes it to the next one until it is dropped by the packet processing engine after it has passed the last block. The forwarding functional

block chain length reaches from only one functional block up to ten functional blocks, each scenario increased by one functional block. We assume that systems running LANA have an average of 5 till 10 functional block instances. The bare comparison of the Linux kernel's versus the LANA framework's receive-and-drop performance is represented by the graph *Linux* respectively *LANA, 1 FB* in figure 6.2. Both graphs overlap each other which means that the bare receive-and-drop performance in packets per second is alike. Speaking in numbers, we were able to process about 1,38 million 64 Byte packets per second in both cases, which is *only* 100,000 packets per second less from processing Gigabit Ethernet at wire-rates. We assume that this is rather a limitation of our network adapter and not of Linux or our LANA framework. When it comes to packet sizes equal to and larger than 128 Byte, the packets per second rate has no significant differences for all LANA configurations from figure 6.2 compared to the Linux kernel maximum. Even with a chain of 10 forwarding functional blocks in the LANA stack, a packet per second rate of roughly 1,18 million 64 Byte packets can be achieved. Furthermore, it is noteworthy that the packet processing at such high packet rates in the `ksoftirqd` and therefore in LANA, too, is CPU-friendly to that regard, that the kernel scheduler tries to distribute the load in a low-priority per-CPU `ksoftirqd` thread. Thus, other important user space processes are not further disturbed. Also, the packet per second rates of both Linux and LANA were stable, which means that during monitoring the system, there was no notably deviation of more than 20,000 pps of our measurement values. For a packet size of 1,518 Byte, a packet per second rate of approx. 81,000 was achieved.

Short summary:

- **Platform:** Linux 3.0
- **Test setup:** 2 physical machines, direct connection
- **Traffic generator:** pktgen
- **Result:** the performance of LANA with long functional block chains (> 10) drops due to blocking of the `ksoftirqd`; chains > 10 are not recommended and also rarely needed since a protocol stack is usually smaller

Since we are interested in LANA's framework scalability, we continued chaining forwarding functional blocks from 10 up to 50 functional blocks by incremental steps of 5 functional blocks. The results are presented in figure 6.3 together with the previous Linux maximum rates for comparison. Network packets with 512 Bytes or more, still have the Linux maximum rates, but in case of packets smaller than that, we notice a rather dramatical drop in performance the more functional blocks are chained together. We assume that there are two reasons for this behaviour: The first, less significant reason is assumed to be related

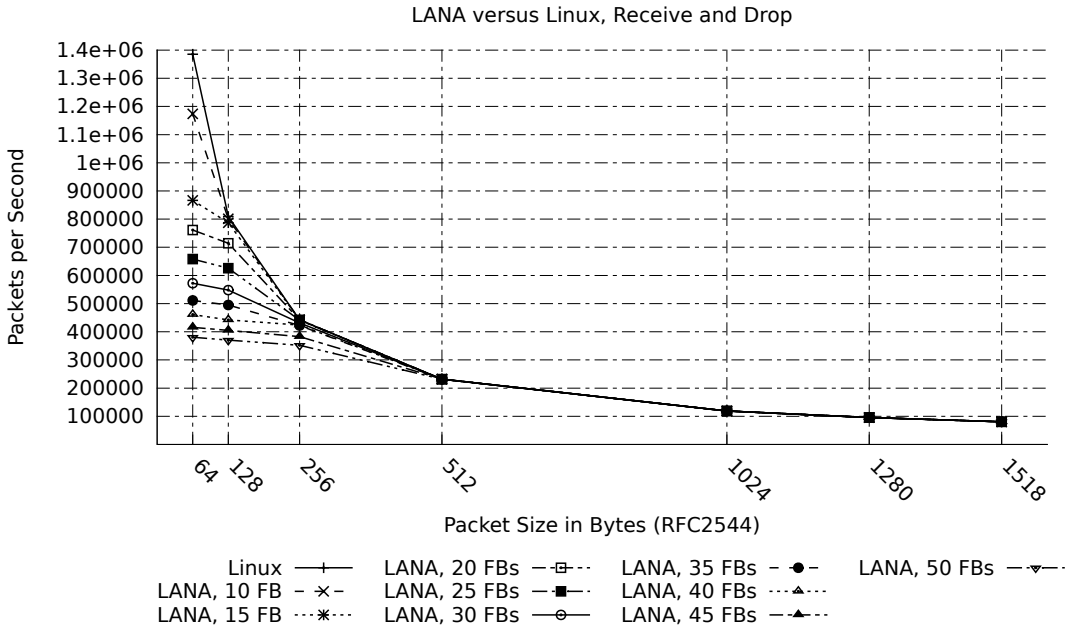

Figure 6.3: Linux versus LANA, 10-50 FBs, increment: 5 FBs

to the introduced layer of indirection, namely information dispatch points, that are looked up in LANA. Since this is deeply wired into LANA's basic idea and architecture, it is very unlikely that this layer is removed. The second reason for this observed drop in performance is assumed in the `ksoftirqd` processing. Since the packet processing engine runs directly within the `ksoftirqd`'s per-CPU threads, the `ksoftirqd` therefore needs to wait until the packet processing engine has finished. Since the packet processing engine does not run in an extra thread (section 6.5), it is at same performance rates as the Linux networking subsystem. However, it comes with the trade-off, that an expensive packet processing blocks the `ksoftirqd` from processing other packets on a specific CPU.

Our future work will therefore investigate Van Jacobson's idea of network channels [100] for our LANA framework, so that we can postpone expensive work to a later point in time to exit the packet processing engine earlier. This kind of lazy evaluation could lead to a higher packet per second rate, since then the `ksoftirqd` will be able to process more packets in a given time than the current framework.

Concluding, this benchmark demonstrated that the LANA framework has competitive performance to the Linux kernel networking subsystem when it comes to packet reception. However, we do not necessarily recommend building a packet path that must pass more than 10 functional blocks in series when there is a demand for a high packet processing performance at the same time. Nevertheless, we naturally expect a similar, but slightly less fatal performance degrade in the

classical Linux protocol stack for such a scenario.

6.4.2 LANA's PF_LANA versus Linux's PF_PACKET Socket

Short summary:

- **Platform:** Linux 3.0
- **Test setup:** 2 physical machines, direct connection
- **Traffic generator:** pktgen
- **Result:** the PF_LANA BSD socket has comparable performance to the PF_PACKET BSD socket, on some packet sizes even a slightly better packet per second performance

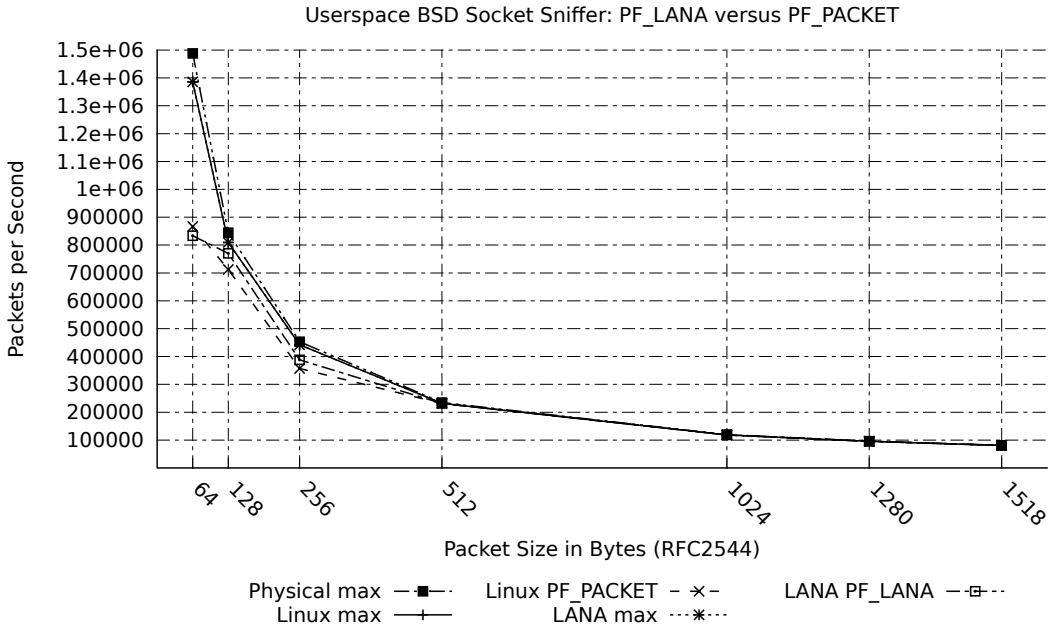

Figure 6.4: User space BSD socket sniffer: PF_LANA versus PF_PACKET

The second benchmark, presented in figure 6.4, refers to both BSD socket implementations, PF_LANA from LANA and PF_PACKET provided by the Linux networking subsystem. Our area of interest in this benchmark concerns the packet per second processing rates from user space applications. For this purpose, we have implemented two raw network packet sniffer applications which invoke the PF_LANA respectively PF_PACKET BSD socket family for capturing packets from user space, both in a non-zero-copy manner. The operating system and software distribution are the same as in benchmarks of 6.4.1, both machines run

a vanilla Linux 3.0.0 kernel and a Debian GNU/Linux 6.0.2.1 on top of it. Also in this benchmark, we have used `pktgen` as a traffic generator on our traffic source from figure 6.1. The traffic sink had either a PF_PACKET traffic sniffer application or a PF_LANA one. The latter consisted of the kernel space LANA framework which was running with the Ethernet vlink functional block `fb_eth` that was bound to the PF_LANA socket's functional block instance of `fb_pflana`. Both user space applications performed `recvfrom(2)` system calls to fetch network packets from kernel space to count received packets and bytes. Hence, we used these numbers to generate our graphs from figure 6.4. For comparison, we have also included the Linux and LANA kernel space maximum receive values, that both are equal to each other. The maximum physical Gigabit Ethernet rates have been included, too. Taken from the results, packet sizes that are larger than or equal to 512 Byte do not differ from their maximum physical values. For smaller packet sizes, the maximum values of both, PF_LANA and PF_PACKET, also differ not significantly. PF_LANA has slightly better packet per second results (roughly 20,000 packets per second more) than PF_PACKET for packet sizes of 128 and 256 Bytes. We assume that in PF_LANA, the critical receive path could be shorter than in PF_PACKET. The maximum packet per second values for PF_LANA and PF_PACKET for 64 Byte packets are near 850,000 packets per second from user space. Also, the measured results were *stable* with only minor deviations.

In conclusion to this benchmark, we have reached similar, and partly even better packet reception rates for our PF_LANA BSD socket compared to the Linux PF_PACKET socket. For LANA, this means, by having more flexibility in the underlying protocol stack, user space applications can yet expect a competitive packet performance from kernel space.

6.4.3 LANA versus Click Modular Router

Short summary:

- **Platform:** Linux 3.0 (LANA, Click in user space), Linux 2.6.38 (Click in kernel space)
- **Test setup:** 2 physical machines, direct connection
- **Traffic generator:** pktgen
- **Result:** LANA is more than twice as fast as Click running in user space, processes about 400,000 packets per second more than Click in kernel space; LANA is more resource sparing than Click

Our third benchmark focusses on the framework overhead of LANA and MIT's Click (section 3.3). For LANA, the same software setup applies as in both previous benchmarks from subsection 6.4.1 and 6.4.2. In case of Click, we had two

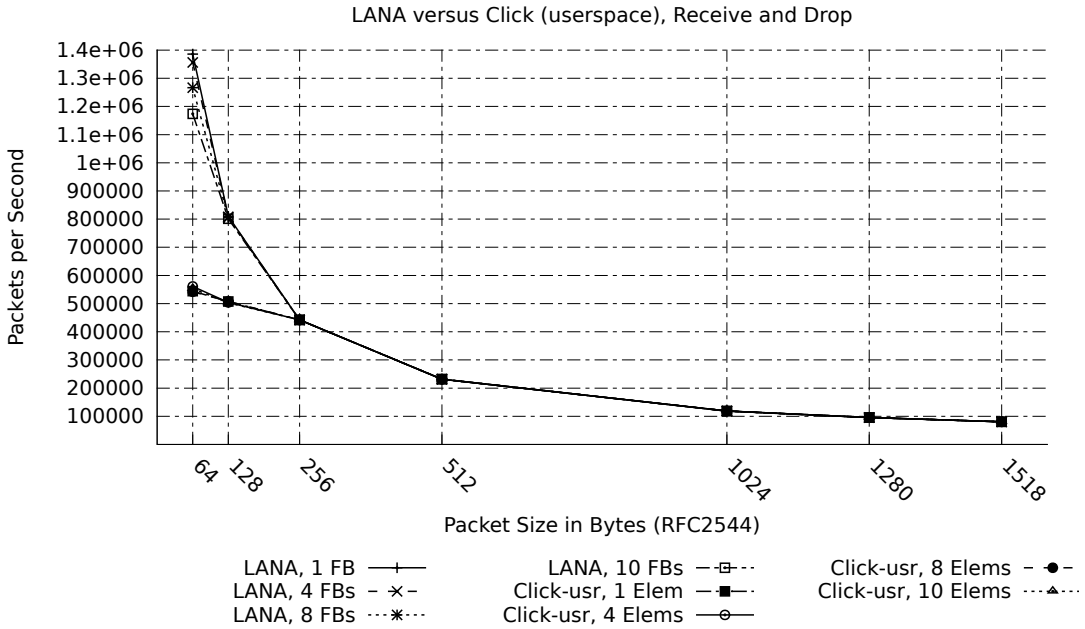

Figure 6.5: Click running in user space versus LANA

different setups, since Click can be run from user space and also from kernel space. The first evaluation whose results can be found in figure 6.5 compared LANA to Click running in user space. There, Click was compiled with multi-threaded support and also run on a vanilla Linux 3.0.0 kernel with a Debian GNU/Linux 6.0.2.1 software distribution. In this case, `pktgen` was also used by the traffic source (figure 6.1). For Click, we have written a `SimpleFwd` element (figure 6.8) that can be chained together with other Click elements, where it just forwards packets to the next bound element. Furthermore, we used elements that were already provided by the Click software distribution, such as `FromDevice(eth0)` for forwarding packets from the networking device into the Click framework, or `SimpleIdle` for dropping packets by the end of the processing scheme. In between, we utilized the `SimpleFwd` element. In our measurement, the user space variant of Click reaches a maximum of about 550,000 64 Byte packets per second with just one element (figure 6.5). Compared to that, LANA is more than twice as fast. Concerning packet sizes of 128 Bytes, LANA reaches about 800,000 packets per second and Click roughly 300,000 packets per second less from user space. We assume that the reason for this is mostly due to copying packets between address spaces and due to context switching. For all packet sizes larger than or equal to 256 Bytes, we did not find a significant difference between Click and LANA. We noted that the performance differences that relate to the number of Click elements were less significant than those of LANA's functional blocks. We assume that the reason for this - as mentioned in subsection 6.4.1 - relies on the

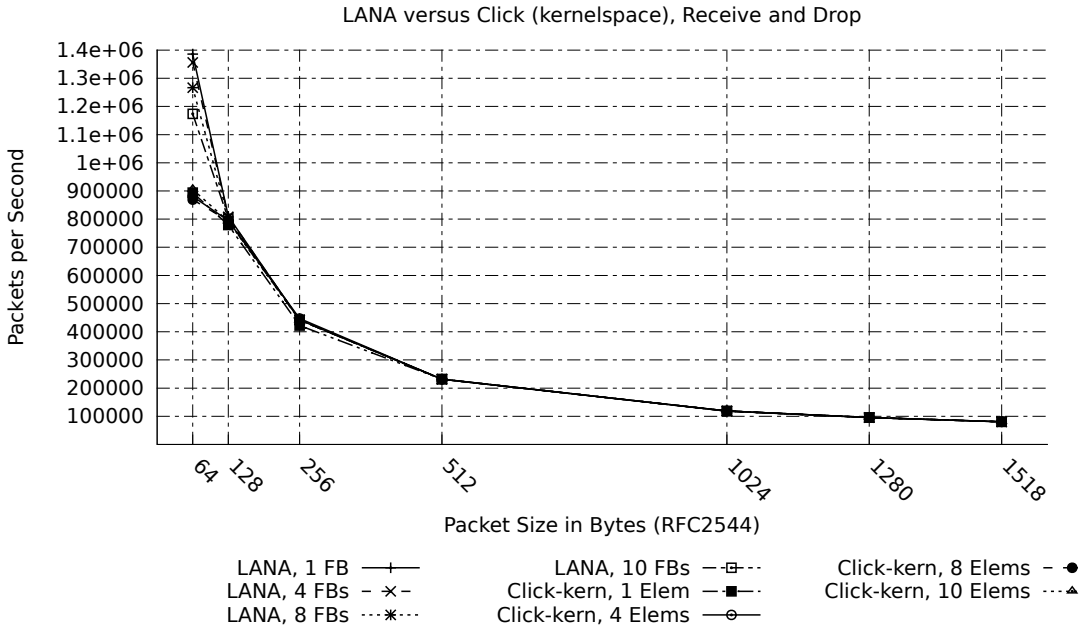

Figure 6.6: Click running in kernel space versus LANA

fact that processing packets in different threads is, on the one hand, independent from the `ksoftirqd` and therefore less disruptive, but, on the other hand, it increases the system's context switching rate, adds another layer of abstraction through threads.

When setting up the kernel space Click, we encountered quite a lot of problems to get it compiled. Since Click and all its elements are written in C++ and the Linux kernel in C and Assembler, modifications to the Linux source code were needed, because i) Click developers did some changes in the networking device structure and in some Linux network device drivers and ii) due to Click compatibility, the kernel gets compiled with G++ instead of GCC. The latter reason also brings other issues to the front, such as different syntax schemes of G++ when it comes to inline Assembler. Therefore, source code regions that do not interact with Click were changed, too. Hence, Click ships patches for the Linux source code and requires a recompilation of the kernel. These patches reach from Linux 2.2.18 until 2.6.24, but not further. Our first try was to setup an old 2.6.24 kernel on our system. We found out that it is quite problematically to i) run an old Linux kernel on a modern Debian GNU/Linux due to the lack of some file systems that are required by the Grub 2 boot loader (namely `devtmpfs`), ii) to run an old Linux kernel in combination with an old software distribution, such as Debian GNU/Linux 'Etchnhalf' due to the fact that it required drivers for our DVD drive during Live CD boot or iii) due to a CPU-related kernel panic during Live CD boot on a modern processor. In fact, we weren't able to get such an

old system running on our benchmarking hardware, until we found out after a mailing list conversation with Click developers, that there is an undocumented 'patchless' way of getting Click into kernel space. This means, we could build the Click kernel module without actually patching and recompiling an old Linux kernel. The latest possible Linux kernel for this method was Linux 2.6.38. We found out that stripping the debug symbols of the Click kernel module (`strip -g click.ko`) was helpful to reduce the image size and finally let Click run in kernel space, so that we could perform our benchmarks.

During the Click kernel space benchmark, we used the same Click configurations and packet sizes as in user space. We first stumbled upon a phenomenon that neither we nor the Click developers on the mailing list [101] were able to explain: our Click setup in kernel space reached a maximum of about 460,000 64 Byte packets per second, which is even less than Click in user space, while at the same time having context switching rates of about 200,000 context switches per second. We assume, there is an issue related to Click's scheduling unit. However, after finding a different method to resolve this issue [101], by adding an additional Click element to the end of Click's processing chain, we reached a maximum of about 900,000 64 Byte packets per second as presented in figure 6.6. In comparison to LANA, this value is quite *instable* and diverts about 100,000 packets per second in both directions. In contrast to LANA, we observed that at least one CPU was on its maximum with an approximate context switching rate of 4,000 context switches per second in Click while in LANA, we observed context switching rates of about 100 or less context switches per second on a high 64 Byte packets per second rate. Since LANA runs within `ksoftirqd`, there was no other competing kernel thread that needed a constant high scheduling rate. Thus, the context switching rate could be reduced and at the same time we were able to process more packets per second.

At the time of Linux 2.2 and 2.4, Click's own NAPI-like device driver polling approach seemed to be more successful than the Linux networking subsystem [18], since NAPI was introduced at a later point in time to reduce the NIC's interrupt load. However, by having NAPI on most common networking device drivers, nowadays it seems that Click's original approach is rather in competition to the Linux NAPI threads, so that this could explain the loss in packet per second performance and the partly odd scheduling or context switching behaviour.

Concluding, we find that our LANA framework is more than twice faster than Click in user space and is able to process nearly 400,000 64 Byte packets per second more than Click in kernel space. However, Click was more stable than LANA when it comes to the number of chained elements respectively functional blocks as shown in figure 6.7. We assume, the reason for this boils down to the fact that i) Click has no such abstraction as information dispatch points, since Click cannot be reconfigured during runtime and hence avoid this additional layer of abstraction and ii) to the kernel thread versus `ksoftirqd` trade-off. Further, we found that LANA is more CPU-friendly than Click even while being able to

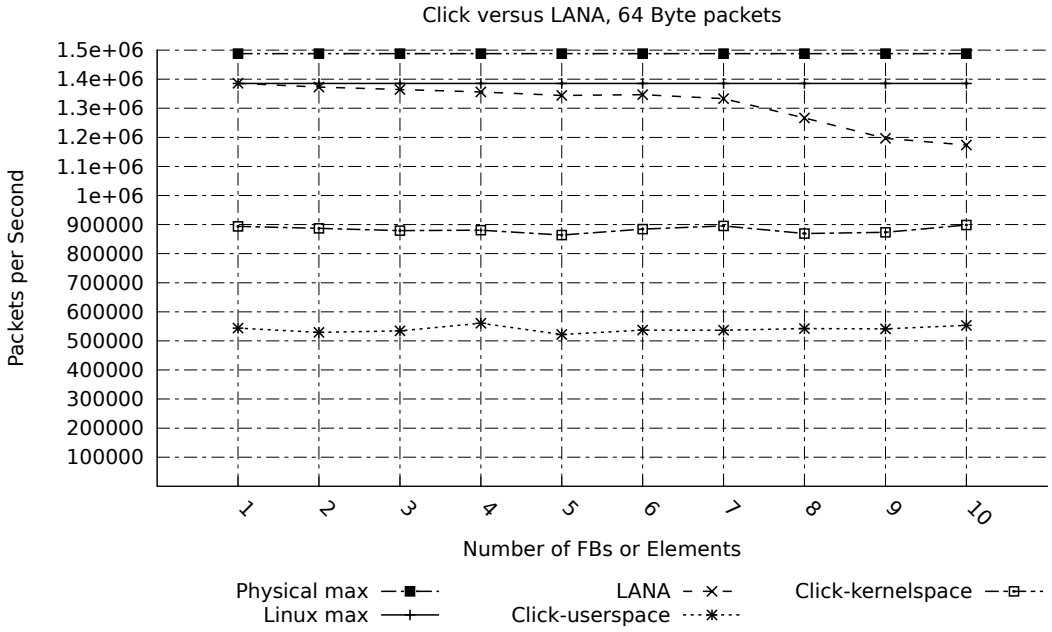

Figure 6.7: Click versus LANA, 64 Byte packets

process a significant higher number of packets per second.

File **simplefwd.hh**:

```
#ifndef CLICK_SIMPLEFWD_HH
#define CLICK_SIMPLEFWD_HH
#include <click/element.hh>
CLICK_DECLS

class SimpleFwd : public Element { public:
    SimpleFwd();
    ~SimpleFwd();

    const char *class_name() const { return "SimpleFwd"; }
    const char *port_count() const { return "-/-"; }
    const char *processing() const { return "a/a"; }
    const char *flow_code() const  { return "x/y"; }

    void push(int, Packet *);
    Packet *pull(int);
};

CLICK_ENDDECLS
#endif
```

File **simplefwd.cc**:

```
#include <click/config.h>
#include "simplefwd.hh"
CLICK_DECLS

SimpleFwd::SimpleFwd() { }

SimpleFwd::~SimpleFwd() { }

void SimpleFwd::push(int, Packet *p)
{
    output(0).push(p);
}

Packet *SimpleFwd::pull(int)
{
    Packet* p = input(0).pull();
    if (p == 0) {
        return 0;
    }
    return p;
}

CLICK_ENDDECLS
EXPORT_ELEMENT(SimpleFwd)
ELEMENT_MT_SAFE(SimpleFwd)
```

Figure 6.8: A simple Click C++ forwarding element, which implements `push` and `pull` that we have written for our benchmark

6.4.4 LANA versus ANA's Prototype

Short summary:

- **Platform:** Linux 3.0
- **Test setup:** 2 physical machines, direct connection
- **Traffic generator:** trafgen (appendix section D.1)
- **Result:** LANA outperforms the ANA prototype significantly; it is about 21 times faster than ANA

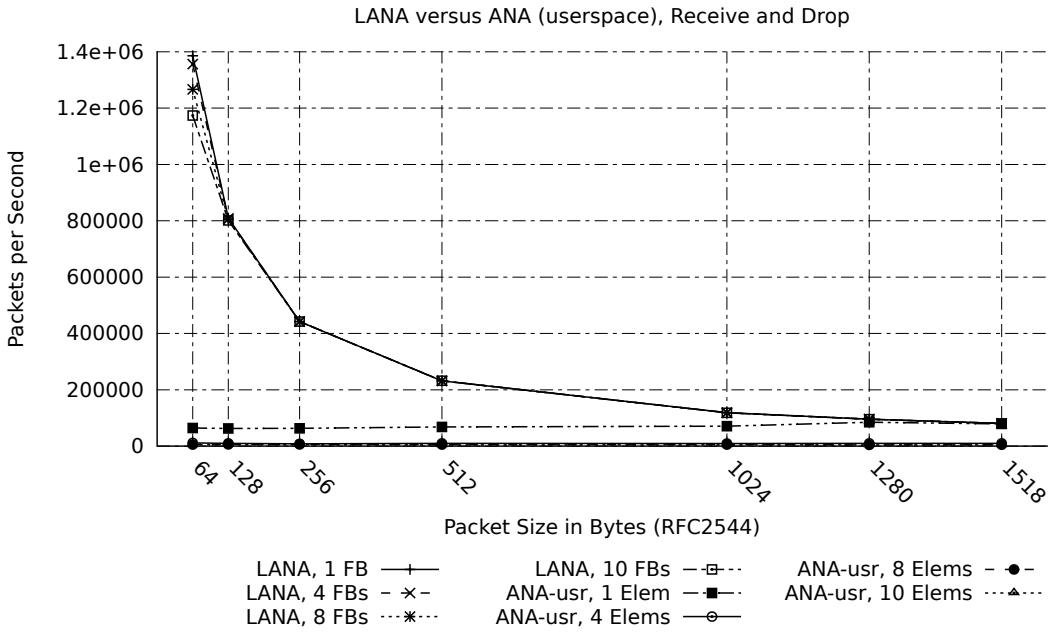

Figure 6.9: ANA running in user space versus LANA

The last benchmark takes the ANA prototype into account and compares it to LANA. Similar to Click, ANA can be run from user space as well as from kernel space. The first benchmark that is shown in figure 6.9 and in figure 6.10 includes ANA running from user space. Therefore, we were able to run a vanilla Linux 3.0.0 kernel together with a Debian GNU/Linux 6.0.2.1 on both, the traffic source and the traffic sink. Unlike LANA, Linux or Click, in this scenario we weren't able to use the kernel space `pktgen` on our traffic source, since the ANA prototype expects a specific packet payload, which cannot be configured with `pktgen`. For this benchmark, we have therefore written our own high-performance zero-copy traffic generator `trafgen` (appendix section D.1).

To bootstrap the ANA stack on the traffic sink, it needs to run on both ends, the sink and the source, since information dispatch points are dynamically created

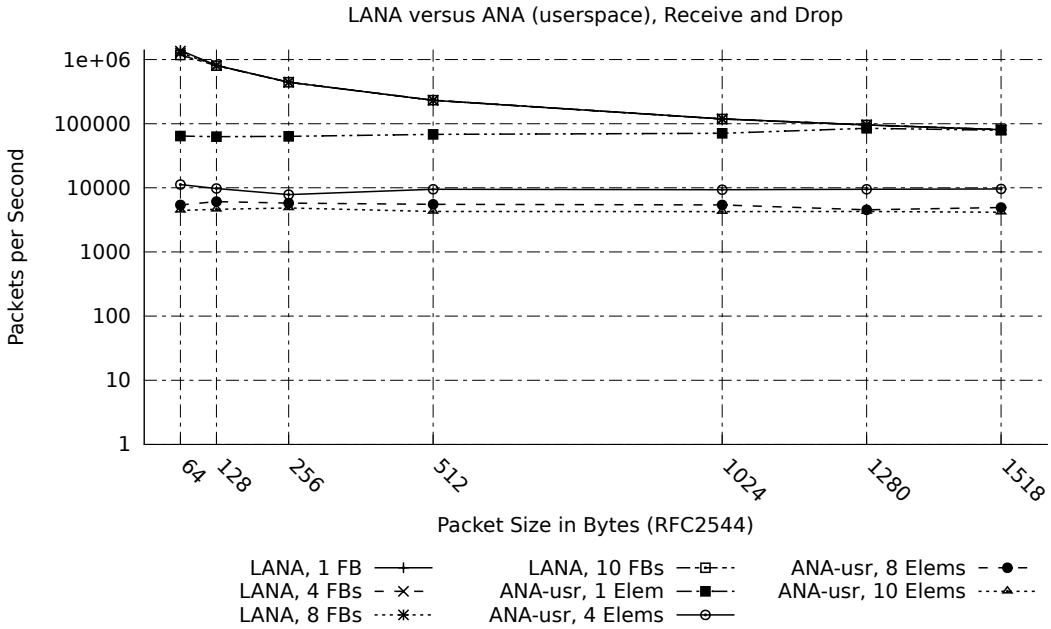

Figure 6.10: ANA running in user space versus LANA (same as figure 6.9, but with logarithmic y-axis)

by the publish subscribe mechanism of ANA. Further, we have developed simple ANA forwarding bricks for evaluating the ANA framework. Unlike LANA or Click, where we only needed to develop such a functional block or element once, in the original ANA, a forwarding brick must be developed for every new element in the forwarding chain. After setting up ANA on both sides, our procedure was to sniff the raw packet payload with `netsniff-ng -C` that was exchanged between both machines in order to gather valid ANA packets and configure a `trafgen` packet including this payload. The results from figure 6.9 show a rather weak overall performance compared to Click, LANA or Linux.

For instance, on 64 Byte packets, LANA with one functional block is about 21 times faster than ANA running in user space with one functional block. As can be seen in figure 6.10, where the y-axis has logarithmic scaling, ANA's performance dramatically drops on more than one functional block. For instance, having 10 functional blocks in ANA chained together, we only reached about 4,500 64 Byte packets per second, whereas LANA still processes 1,173,000 packets per second, which is 260 times more than ANA. Detailed maximum rates compared to the number of chained functional blocks can be found in figure 6.11. We assume that an explanation of ANA's performance mainly resides its design as discussed in chapter 3. One of the aims in ANA was to design a robust architecture, so that if one brick crashes due to a malicious behaviour, others will not be interrupted in processing. Therefore, by running ANA in user space, each brick runs as a separate process that performs interprocess communication (IPC) with ANA's

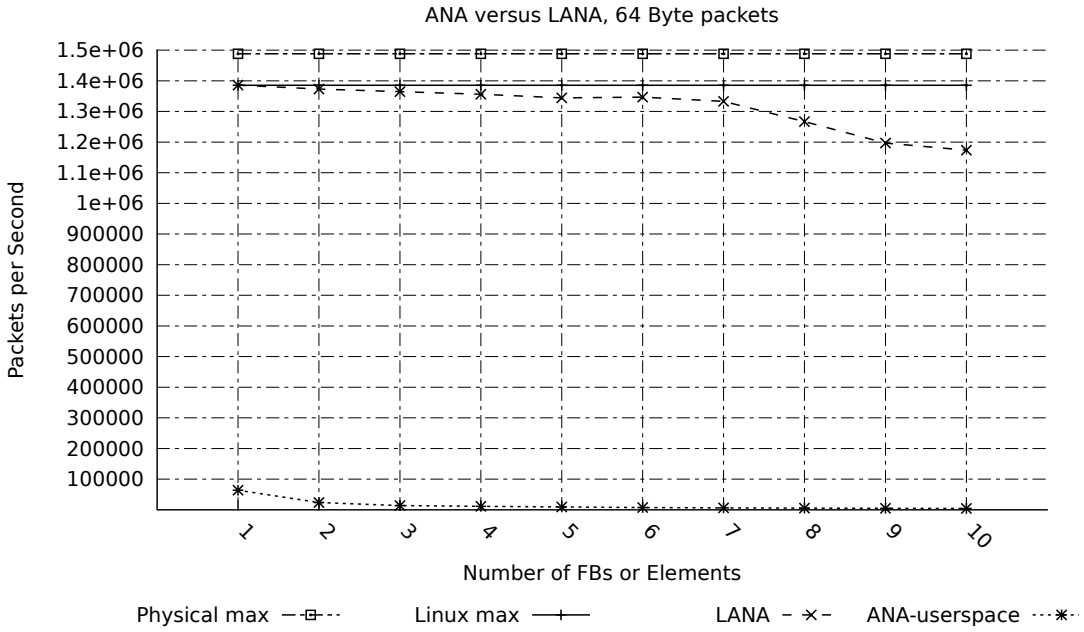

Figure 6.11: ANA versus LANA, 64 Byte packets

central MINMEX. For a single incoming network packet, the following scenario for three chained ANA bricks can be considered: 1) the packet arrives at the MINMEX first, that was copied from kernel space to user space, 2) via IPC it is sent to the first brick via the `sendto(2)` system call, 3) it is received by the first brick via the `recvfrom(2)` system call, 4) after having processed parts of the packet, the first brick sends the network packet to the information dispatch point of the second brick via the `sendto(2)` system call, 5) the MINMEX receives the network packet from the first brick via the `recvfrom(2)` system call, 6) the MINMEX looks up the corresponding brick and sends the network packet to the second brick via the `sendto(2)` system call, 7) the second brick receives the packet via the `recvfrom(2)` system call, 8) again, after having processed the packet, the second brick sends it back to the MINMEX via the `sendto(2)` system call, 9) the MINMEX receives it via `recvfrom(2)` and sends it to the third ANA brick through `sendto(2)`, 10) the third ANA brick receives the network packet via `recvfrom(2)` and finishes processing.

The described processing chain was just a simplification, but the IPC overhead for just a single incoming packet seems immense. Thus, ANA in user space results in such a performance.

For the kernel space benchmarking of ANA, we were trapped with similar issues that are described in subsection 6.4.3: getting an old Linux distribution running on our rather modern benchmarking machines. We have therefore decided to setup a kernel-based virtual machine (KVM [102]) with Intel's VT hardware

support. Within our virtual environment we were able to setup an Ubuntu 6.04 that has a Linux kernel, which is supported by ANA. We have developed the following script to start our virtual machine and to make it successfully visible to our traffic source via IP masquerading on the host machine:

```
#!/bin/sh

modprobe tun
modprobe kvm-intel

brctl addbr br0
ifconfig br0 10.0.0.254/24 up

tunctl -t tap0 -b -u root
ifconfig tap0 up 0.0.0.0 promisc

brctl addif br0 eth0
brctl addif br0 tap0

route add -host 10.0.0.1 dev br0

echo "0" > /proc/sys/net/bridge/bridge-nf-call-arptables
echo "0" > /proc/sys/net/bridge/bridge-nf-call-iptables
echo "0" > /proc/sys/net/bridge/bridge-nf-filter-vlan-tagged
echo "0" > /proc/sys/net/bridge/bridge-nf-call-ip6tables
echo "0" > /proc/sys/net/bridge/bridge-nf-filter-pppoe-tagged

echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
iptables -A FORWARD -i eth0 -o br0 -j ACCEPT
iptables -A FORWARD -i br0 -o eth0 -j ACCEPT

sleep 1

kvm -hda ./hd.img -m 2G -smp 4 -net nic -net tap,ifname=tap0,script=no \
    -cdrom Downloads/ubuntu-6.06.1-desktop-amd64.iso
```

The virtual machine was configured to have 4 (real) CPUs, 2 GB RAM with an emulated e1000 NIC through the virtual TAP device `tap0` that was bridged on the host kernel via `br0` to `eth0`. After having fixed build dependencies in the ANA kernel space build system, we were able to successfully load ANA modules into the kernel. However, in contrast to the user space ANA, the kernel space ANA failed to register the vlink module to the MINMEX. Thus, all subsequent brick modules such as our forwarding functional blocks could not register to the MINMEX, too, since the dependency of the vlink module has not been found. Finally, we weren't able to bring up ANA in kernel space mode.

Concluding, we were only able to benchmark ANA from user space with the result that our newly developed LANA is about 21 times more efficient regarding its packet per second rate than the original prototype. Also, when it comes to

CPU usage, ANA used up to 100 percent CPU resources even in idle mode when no incoming or outgoing packet was present, whereas LANA is CPU-sparing due to the adaptive scheduling behaviour of the `ksoftirqd`. We have only found one ANA kernel space functional block benchmark in literature [103] with an overall best-case result of 366,000 packets per second. Even with this result, LANA is more than 3.7 times faster compared to its original prototype implementation.

6.5 LANA's Road to 1.4 Mio Packets per Second

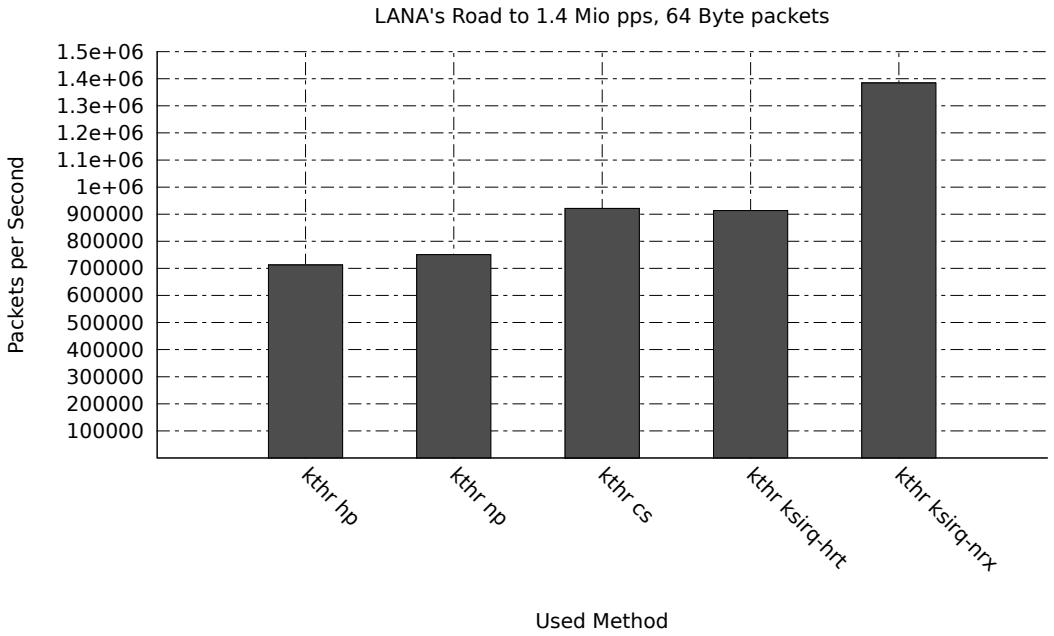

Figure 6.12: LANA's Road to 1.4 Mio pps: `kthr hp` (kernel thread, high priority), `kthr np` (kernel thread, normal priority), `kthr cs` (kernel thread, controlled scheduling), `kthr ksirq-hrt` (kernel thread, in ksoftirqd execution context through high-resolution timer), `kthr ksirq-nrx` (kernel thread, in ksoftirqd execution context through NET_RX tasklets)

During the design and implementation of LANA, we did not arrive at a packet processing rate of approximately 1.4 Mio 64 Byte packets per second immediately. It took us several stages of re-thinking and successively improving our LANA back-end through a combination of monitoring the system's behaviour with the `ifpps` (appendix section D.2) tool, when put under pressure through a high packet load.

As demonstrated in figure 6.12, we came across 5 different back-end models until we finally found the best solution that is most competitive with the

Linux networking subsystem. Our very first approach, namely `kthr hp`, was a high-priority per-CPU kernel thread, where our packet processing engines were triggered from. Each thread had a socket buffer queue assigned to it, where an upstream scheduling module enqueued an incoming socket buffer from the network driver side into one of the socket queues i.e., through a round-robin scheduling method. After performing measurements, we assumed that on a high packet rate, the performance degradation down to about 700,000 packets per second was caused due to a 'starvation' of the `ksoftirqd`, since i) context switches per second increased dramatically to more than 6000 context switches per second where usual rates reside below 500 context switches per second and ii) the rate of raised software interrupts for NET_RX dropped at the same time to a value of below 100 NET_RX software interrupts per second where usual rates are above 1000 NET_RX software interrupts per second.

This leads to the second model `kthr np` where we have reset the high thread priority, since the `ksoftirqd` only runs on a low priority to not disturb important user space processes. However, the processing rates only improved slightly to about 750,000 packets per second, since the context switching rate remained at a constantly high level. Naturally, this is due to the fact that on an otherwise idle system, only `ksoftirqd` threads and LANA threads are competing with each other, since both strive for having more CPU time for packet processing.

We then assumed that an explicit control of preemtiveness and scheduling with the LANA thread could reduce context switching rates. The result of this approach improved packet processing rates up to about 900,000 packets per second which can be seen in `kthr cs`.

Yet, the LANA threads were still competing with the lower prioritised `ksoftirqd` threads, so that our next approach was to eliminate this competition by moving the execution of LANA's packet processing engines into the `ksoftirqd` itself. The first, rather inelegant idea was to create high-resolution timer tasklets that trigger the packet processing engines within their execution. High-resolution timer tasklets are periodically handled within the `ksoftirqd`. Results of this model, namely `kthr ksirq-hrt`, did not really improve, even though the context switching rates relaxed. Besides this, it needed an adaptive algorithm that would trigger the next execution of the timer tasklet depending on the incoming packet load.

Finally, this lead to our last and most appropriate approach in model `kthr ksirq-nrx`, where we directly trigger the packet processing engine within the NET_RX software interrupt context of the `ksoftirqd`. Thus, we were finally able to reach packet processing rates of about 1.4 million packets per second.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis we have demonstrated that it is possible to combine three partially conflicting goals into a future network architecture: i) high flexibility for network programmers, ii) re-configuration of the network stack at runtime, and iii) high packet processing rates. However, it is only possible to achieve such an architecture, if great care is taken on its design with regard to principles from chapter 2, and if great care is taken on low-level implementation details.

We have not only demonstrated that our LANA framework was able to significantly outperform the previous ANA prototype implementation, we have also demonstrated that this framework has a competitive performance to the Linux kernel network stack with the advantage of high flexibility. We have shown that our LANA framework is less resource hungry than the original prototype while having higher throughput rates. Our LANA framework is even significantly faster than modular research platforms such as MIT's Click Modular Router.

Furthermore, we alleviated the development of new functional blocks by introducing basic concepts of object orientation into the framework even despite the lack of it in the underlying programming language.

A first repository of functional blocks has been implemented and we have demonstrated that we are able to send and receive network packets through our PF_LANA BSD socket family from user space via our kernel space LANA framework to the networking device drivers and vice versa. For this purpose, we have implemented a low-latency voice-over-Ethernet PF_LANA application, that utilizes ALSA and the CELT codec to bidirectionally transfer voice data. A reconfiguration of the underlying LANA communication stack did not result in a notable voice data interruption.

With this meta-architecture, we have set the foundation for i) the EPiCS networking layer, which needs to flexibly adapt to changing conditions during runtime, as well as for ii) a new future Internet architecture with a design that allows

for new innovations in all network protocol layers.

Next to this final report, the following list shortly summarizes major contributions of this thesis:

- The design and implementation of LANA's core machinery. As demonstrated in section 6.4, LANA is competitive in its processing capabilities regarding packets per second with the Linux kernel networking subsystem. Further, LANA is about 21 times faster than the original ANA prototype. LANA is also more than twice as fast than similar frameworks such as Click that runs from user space and it also outperforms it from kernel space by more than 50 percent. Yet, LANA is more resource-sparing than the original ANA prototype and Click as described in section 6.4.
- Next to this, we have implemented seven functional blocks for our LANA framework:
  - A packet counter functional block that exports its data to procfs,
  - A Berkeley Packet Filter functional block that filters network packets,
  - A simple forwarding functional block,
  - A tee functional block whose purpose is to duplicate network packets,
  - A vlink Ethernet functional block that serves as a per networking device ingress and egress point for the LANA protocol stack,
  - A vlink Ethernet functional block that allows for the creation of virtual Linux Ethernet devices that can be managed with standard Linux tools and that handle specific vlink-tagged LANA network traffic (similar as in VLANs),
  - A PF_LANA BSD socket functional block that allows for the development of LANA user space applications by offering standard system calls such as `socket(2)`, `close(2)`, `recvfrom(2)`, `sendto(2)` or `poll(2)`; buffers are then being copied into the kernel space and processed within the LANA packet processing engine. Benchmarks from section 6.4 show that it has a competitive and partly even slightly better performance than non-zero-copy PF_PACKET applications.
- The development of a voice-over-Ethernet user space sample application that works on top of ALSA and uses the low-latency CELT audio codec, Speex echo cancellation, jitter buffers and our PF_LANA socket for sending and receiving data through the LANA protocol stack. We have demonstrated that LANA is able to bidirectionally transfer real-time voice data, and that we are able to change the underlying protocol stack during runtime without voice data interruption.
- The design and implementation of a high-performance user space zero-copy traffic generator, namely `trafgen` (appendix chapter D.1), that is able to

generate up to 1.488 Mio 64 Byte packets per second on commodity Gigabit Ethernet hardware, where usual user space traffic generators only reach about 800,000 64 Byte packets per second. We have used this traffic generator to perform an evaluation of the original ANA prototype and for debugging purposes during the development of LANA.

- During the development of `trafgen`, we have found a potential null-pointer dereference in the Linux kernel's implementation of the PF_PACKET BSD socket. We have therefore created a patch [104] [105], which got accepted by David S. Miller, the Linux networking subsystem maintainer, and is scheduled for mainline inclusion for upcoming Linux kernel versions.
- The design and implementation of `bpfc` (appendix chapter D.3), a Berkeley Packet Filter compiler that is able to transform syntax as described in literature [31] into a kernel readable format, that is applied in the Berkeley Packet Filter virtual machine. This compiler can be seen as the supplement of Eric Dumazet's Berkeley Packet Filter Just-In-Time compiler [106] that has recently been included into the mainline kernel for speeding up the evaluation of filters. The output of `bpfc` can then be used by the kernel Just-In-Time compiler to generate architecture specific machine opcodes. The `bpfc` has been built as a support for our LANA BPF functional block, that can be used to filter specific network packets within the LANA protocol stack. Next to this, `bpfc` even supports the translation of undocumented BPF extensions, that are present in Linux kernels. We also have developed a Vim syntax highlighting file for the BPF syntax [31] in the VimL scripting language.
- The design and implementation of `ifpps` (appendix chapter D.2), a top-like Linux kernel networking statistics monitor. With the help of `ifpps`, we were able to track scheduling behaviour of the system in general and of networking software interrupts in particular, the networking device interrupt load as well as transferred packet and byte statistics of a specific networking device. The knowledge we have gained by using `ifpps` enabled us to derive assumptions about the system's behaviour, which lead to optimizations of our LANA core machinery implementation (section 6.5).
- The design and implementation of a Linux kernel module that is able to generate one-time stack traces (appendix chapter D.4) based on the Kprobes framework of Linux. With the aid of this module, we were able to debug or track stack frames from the invocation of a specific Linux symbol in order to examine the packet path described in section 3.1. We then used the knowledge gained from section 3.1 to find an appropriate interface or egress and ingress point to the LANA protocol stack.
- We have published a paper and presented a poster of this work at the ACM/IEEE Symposium on Architectures for Networking and Communications Systems 2011 in New York, USA (appendix section A).

- We have presented the results of this work at the EPiCS review meeting of the EU 7th Framework Programme in Brussels, Belgium as part of the EPiCS work package 4.
- We have presented the results of this work at the Intel European Research and Innovation Conference 2011 in Dublin, Ireland in the context of network infrastructures for 'smart connected devices' in embedded intelligence.

Besides these major contributions, some minor contributions of this work include things such as debugging code, setting up a Linux kernel cross-reference namely LinGrok [66], and using this reference to gather knowledge of the packet path from the device driver layer up to the BSD socket layer and vice versa, since such information is hardly covered in recent literature.

Minor contributions also include the development of forwarding functional blocks or elements in the ANA and Click framework, and debugging as well as exploring code of both frameworks for resolving setup issues.

Last but not least, a minimal raw packet sniffer application for the PF_PACKET socket family and for the PF_LANA socket family has been developed for performing benchmarks on both socket families.

7.2 Future Work

Since the LANA core machinery is a meta-architecture with no functionality regarding packet processing, we aim to complete our current functional block repository with further protocols that allow for mechanisms such as routing.

Next to this, we plan to research on a mechanism that allows an automatic configuration of LANA's functional block stack of both communication entities. Ideas for such a protocol could be based on the introduction of a qualified functional block namespace that is similar to the namespace of Java packages. There, an entity could initiate a stack negotiation by sending a specific list of qualified functional block names including their order in the communication stack that are needed for a proper communication. After having agreed over the used functional blocks, they will be initiated on both sides, so that a communication channel can be established.

Furthermore, (as already mentioned in section 6.4.1) we plan to take up Van Jacobson's ideas of network channels [100] to evaluate an improvement of LANA's framework performance for larger chains of functional blocks.

Within the EPiCS context, we plan to integrate LANA with ReconOS [107]. This will allow us to offload resource-intensive functional blocks such as encryption to hardware.

7.3 Acknowledgements

At first, I would like to thank Prof. Dr. Bernhard Plattner for the possibility to write my master's thesis at the Communication Systems Group of the Swiss Federal Institute of Technology, Zurich (ETH Zurich).

Furthermore, I am deeply grateful to both of my advisors, Ariane Keller and Dr. Wolfgang Mühlbauer for their support. During the time of my thesis, I've learned a lot of new and interesting things, had very inspiring discussions and in general a great time in Switzerland and at the Communication Systems Group. I am already looking forward to continue my research within this group after finishing my studies.

Moreover, I would like to thank Prof. Dr.-Ing. Dietmar Reimann from the HTWK Leipzig and Prof. Dr. Bernhard Plattner from the ETH Zurich for their supervision of my master's thesis.

With this master's thesis I complete my studies in computer science at the HTWK Leipzig. Therefore, I would like to thank my friends from studying and work, during my time as a student worker, for particular things and for everything: Tobias Kalbitz, Thomas Reinhardt, Rico Tilgner, Stefan Seering, Andrea Jazdzewski, Ansgar Jazdzewski, Michael Wünsch, Kerstin Erfurth, Robert Fritzsche, Jens Hadlich, Emmanuel Roullit, Lars Wächtler, Mathias Lafeldt, Ingmar Pörner, Steffen Bauch, Falk Morawitz, and all others I forgot to mention.

Last but not least, I would like to thank my girlfriend Katarzyna Czarny and my parents for their support during my studies as well as the German National Merit Foundation for providing me two scholarships.

List of Figures

References

[1] "History of IPv6 in Linux." http://tldp.org/HOWTO/Linux+IPv6-HOWTO/basic-history-ipv6-linux.html (Aug 11).

[2] Trammell, Brian, "ETH Zurich, Communication Systems Group Blog: World IPv6 Day." http://blogs.ethz.ch/csg/2011/07/01/world-ipv6-day/ (Aug 11).

[3] L. Rizzo and M. Landi, "netmap: memory mapped access to network devices," in *Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM*, SIGCOMM '11, (New York, NY, USA), pp. 422–423, ACM, 2011.

[4] "LKML discussion, Packet mmap: TX RING and zero copy." http://www.amailbox.net/mailarchive/linux-netdev/2008/9/2/3168784 (Aug 11).

[5] F. Fusco and L. Deri, "High speed network traffic analysis with commodity multi-core systems," in *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, (New York, NY, USA), pp. 218–224, ACM, 2010.

[6] David S. Miller, "Linux Multiqueue Networking." http://vger.kernel.org/~davem/davem_nyc09.pdf (Aug 11).

[7] T. Herbert, "Software Receive Packet Steering." http://lwn.net/Articles/328339/ (Aug 11).

[8] "Linux: TCP Segmentation Offload (TSO)." http://kerneltrap.org/node/397 (Aug 11).

[9] "NAPI." http://www.linuxfoundation.org/collaborate/workgroups/networking/napi (Aug 11).

[10] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: the second-generation onion router," in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2004.

[11] Satoshi Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System." `http://bitcoin.org/bitcoin.pdf` (Aug 11).

[12] "ffff-dns: P2P dns resolution service based on a DHT and ECC." `https://github.com/HarryR/ffff-dnsp2p` (Aug 11).

[13] Daniel J. Bernstein, "TCP SYN cookies." `http://cr.yp.to/syncookies.html` (Aug 11).

[14] A. Friedlander, A. Mankin, W. D. Maughan, and S. D. Crocker, "Dnssec: a protocol toward securing the internet infrastructure," *Commun. ACM*, vol. 50, pp. 44–50, June 2007.

[15] Daniel J. Bernstein et. al., "DNSCurve: Usable security for DNS." `http://dnscurve.org/` (Aug 11).

[16] L. Peterson, B. Davie, and A. Bavier, "x-kernel Tutorial." `http://www.cs.arizona.edu/projects/xkernel/www/tutorial.ps` (Aug 11).

[17] A. Cobbs, "All About Netgraph." `http://people.freebsd.org/~julian/netgraph.html` (Aug 10).

[18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[19] G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May, "The autonomic network architecture (ANA)," *JSAC special issue in Autonomic Communications*. under submission.

[20] G. Varghese, *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann, Dec. 2004.

[21] A. Akella, S. Seshan, and A. Shaikh, "An empirical evaluation of wide-area internet bottlenecks," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, pp. 316–317, June 2003.

[22] N. Hu, L. E. Li, Z. M. Mao, P. Steenkiste, and J. Wang, "Locating internet bottlenecks: algorithms, measurements, and implications," *SIGCOMM Comput. Commun. Rev.*, vol. 34, pp. 41–54, August 2004.

[23] T. E. Levin, C. E. Irvine, C. Weissman, and T. D. Nguyen, "Analysis of three multilevel security architectures," in *Proceedings of the 2007 ACM workshop on Computer security architecture*, CSAW '07, (New York, NY, USA), pp. 37–46, ACM, 2007.

[24] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 304–316, October 2002.

[25] D. Dalessandro and P. Wyckoff, "Accelerating web protocols using rdma," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, (New York, NY, USA), ACM, 2006.

[26] Felix von Leitner, "Scalable Network Programming, Or: The Quest For A Good Web Server (That Survives Slashdot)." `http://bulk.fefe.de/scalable-networking.pdf` (Aug 11).

[27] Hagen Paul Pfeifer, "Epoll and Select Overhead." `http://blog.jauu.net/2011/01/23/Epoll-and-Select-Overhead/` (Aug 11).

[28] "mmap(2) facility of the PF_PACKET socket interface in Linux." `http://lingrok.org/source/xref/linux-2.6-linus/Documentation/networking/packet_mmap.txt` (Aug 11).

[29] Johann Baudy, "Linux packet mmap to improve transmission process." `http://wiki.ipxwarzone.com/index.php5?title=Linux_packet_mmap` (Aug 11).

[30] G. Varghese and A. Lauck, "Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility," *IEEE/ACM Trans. Netw.*, vol. 5, pp. 824–834, December 1997.

[31] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture, Winter USENIX conference, Jan., 1993, San Diego, CA," 1992.

[32] A. Begel, S. McCanne, and S. L. Graham, "Bpf+: exploiting global data-flow optimization in a generalized packet filter architecture," in *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '99, (New York, NY, USA), pp. 123–134, ACM, 1999.

[33] J. Stone and C. Partridge, "When the crc and tcp checksum disagree," in *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, (New York, NY, USA), pp. 309–319, ACM, 2000.

[34] J. Engel, J. Meneskie, and T. Kocak, "Performance analysis of network protocol offload in a simulation environment," in *Proceedings of the 44th annual Southeast regional conference*, ACM-SE 44, (New York, NY, USA), pp. 762–763, ACM, 2006.

[35] R. M. Sanders and A. C. Weaver, "The xpress transfer protocol (xtp), a tutorial," *SIGCOMM Comput. Commun. Rev.*, vol. 20, pp. 67–80, October 1990.

[36] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of tcp processing overheads," *IEEE Communications Magazine*, vol. 27, pp. 23–29, June 1989.

[37] K. Karras, T. Wild, and A. Herkersdorf, "A folded pipeline network processor architecture for 100 gbit/s networks," in *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '10, (New York, NY, USA), pp. 2:1–2:11, ACM, 2010.

[38] F. Tam, "On engineering standards based carrier grade platforms," in *Proceedings of the 2007 workshop on Engineering fault tolerant systems*, EFTS '07, (New York, NY, USA), ACM, 2007.

[39] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: an aid to network processing," in *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '05, (New York, NY, USA), pp. 181–192, ACM, 2005.

[40] W. Szpankowski, "Patricia tries again revisited," *J. ACM*, vol. 37, pp. 691–711, October 1990.

[41] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, pp. 238–275, September 2005.

[42] R. G. Beausoleil, "Large-scale integrated photonics for high-performance interconnects," *J. Emerg. Technol. Comput. Syst.*, vol. 7, pp. 6:1–6:54, July 2011.

[43] Hagen Paul Pfeifer, "Head Of Line Blocking." `http://blog.jauu.net/2011/08/23/Head-Of-Line-Blocking/` (Aug 11).

[44] D. Stiliadis and A. Varma, "Efficient fair queueing algorithms for packet-switched networks," *IEEE/ACM Trans. Netw.*, vol. 6, pp. 175–185, April 1998.

[45] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage, "Inferring internet denial-of-service activity," *ACM Trans. Comput. Syst.*, vol. 24, pp. 115–139, May 2006.

[46] S. Fischer, O. Kiselyov, and C.-c. Shan, "Purely functional lazy non-deterministic programming," *SIGPLAN Not.*, vol. 44, pp. 11–22, August 2009.

[47] B. W. Lampson, "Lazy and speculative execution in computer systems," in *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, (New York, NY, USA), pp. 1–2, ACM, 2008.

[48] J. Hadi Salim, R. Olsson, and A. Kuznetsov, "Beyond Softnet." 5th Annual Linux Showcase and Conference (ALS '01). pp. 165–172. `http://www.usenix.org/publications/library/proceedings/als01/full_papers/jamal/jamal.pdf`. Retrieved 2011-03-06. The classical NAPI paper.

[49] R. Sommer and A. Feldmann, "Netflow: information loss or win?," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, IMW '02, (New York, NY, USA), pp. 173–174, ACM, 2002.

[50] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, pp. 121–137, June 1979.

[51] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," *SIGCOMM Comput. Commun. Rev.*, vol. 27, pp. 3–14, October 1997.

[52] M. Stonebraker, "Technical perspective: One size fits all: an idea whose time has come and gone," *Commun. ACM*, vol. 51, pp. 76–76, December 2008.

[53] C. Partridge and S. Pink, "Errata: A faster udp," *IEEE/ACM Trans. Netw.*, vol. 1, pp. 754–, December 1993.

[54] "Tag Switching." `http://home.earthlink.net/~hoabut/pages/tag_switching/tagswitching.html` (Aug 11).

[55] "Kernel: likely/unlikely macros." `http://kerneltrap.org/node/4705` (Aug 11).

[56] "GCC, the GNU Compiler Collection." `http://gcc.gnu.org/` (Aug 11).

[57] M. D. Smith, N. Ramsey, and G. Holloway, "A generalized algorithm for graph-coloring register allocation," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, (New York, NY, USA), pp. 277–288, ACM, 2004.

[58] E. Corwin and A. Logar, "Sorting in linear time - variations on the bucket sort," *J. Comput. Small Coll.*, vol. 20, pp. 197–202, October 2004.

[59] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, IMC '03, (New York, NY, USA), pp. 153–166, ACM, 2003.

[60] "Bitmap Allocators." `http://gcc.gnu.org/onlinedocs/libstdc++/manual/bitmap_allocator.html` (Aug 11).

[61] B. K. Bray and M. J. Flynn, "Translation hint buffers to reduce access time of physically-addressed instruction caches," in *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, (Los Alamitos, CA, USA), pp. 206–209, IEEE Computer Society Press, 1992.

[62] D. E. Knuth, *Art of Computer Programming, The, Volumes 1-3*. Addison-Wesley Professional, 3 ed., Oct. 1998.

[63] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, third edition ed., July 2009.

[64] R. Love, *Linux Kernel Development - A thorough guide to the design and implementation of the Linux kernel*. Addison Wesley, June 2010.

[65] C. Benvenuti, *Understanding Linux Network Internals, A Guided Tour to Networking on Linux*. O'Reilly, Dec. 2005.

[66] "Linux, Git source code, LinGrok X-REF of the linus-tree." `http://lingrok.org/source/xref/linux-2.6-linus/` (Aug 11).

[67] "Bridging." `http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge` (Aug 11).

[68] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly, Nov. 2005.

[69] M. A. Brown, "Traffic Control Howto, TLDP." `http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html` (Aug 11).

[70] "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers." `http://tools.ietf.org/html/rfc2474` (Aug 11).

[71] "Random Early Detection queue." `http://lingrok.org/source/xref/linux-2.6-linus/net/sched/sch_red.c` (Aug 11).

[72] "Hierarchical token bucket, feed tree version." `http://lingrok.org/source/xref/linux-2.6-linus/net/sched/sch_htb.c` (Aug 11).

[73] "Stochastic Fairness Queueing discipline." `http://lingrok.org/source/xref/linux-2.6-linus/net/sched/sch_sfq.c` (Aug 11).

[74] "Netgraph – Graph based kernel networking subsystem, FreeBSD Kernel Interfaces Manual, Section 4." http://www.freebsd.org/cgi/man.cgi?query=netgraph&sektion=4 (Aug 11).

[75] "Netgraph source code, FXR of FreeBSD 9-CURRENT (base/head)." http://fxr.watson.org/fxr/source/netgraph/ (Aug 11).

[76] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, 1991.

[77] C. Jelger, S. Schmid, and G. Bouabene, "ANA Blueprint Version 2.0." http://www.ana-project.org/deliverables/2008/ana-d1.9-final.pdf (Aug 11).

[78] "Autonomic Network Architecture - EU Project (2006-2009)." http://www.ana-project.org (Oct 09).

[79] Felix von Leitner, "Writing Small And Fast Software." http://www.fefe.de/dietlibc/diet.pdf (Aug 11).

[80] N. Brown, "Object-oriented design patterns in the kernel, part 1." http://lwn.net/Articles/444910/ (Aug 11).

[81] N. Brown, "Object-oriented design patterns in the kernel, part 2." http://lwn.net/Articles/446317/ (Aug 11).

[82] Daniel J. Bernstein, "Crit-bit trees." http://cr.yp.to/critbit.html (Aug 11).

[83] "Linux kernel coding conventions." http://lingrok.org/source/xref/linux-2.6-linus/Documentation/CodingStyle (Aug 11).

[84] Free Software Foundation, "GNU General Public License, Version 2." http://www.gnu.org/licenses/gpl-2.0.txt (Aug 11).

[85] Linus Torvalds, "Linux kernel source tree, Tag for Linux 3.0." https://github.com/torvalds/linux/tree/v3.0 (Aug 11).

[86] "Git, the fast version control system." http://git-scm.com/ (Aug 11).

[87] "Lightweight Autonomic Network Architecture." http://repo.or.cz/w/ana-net.git (Jul 11).

[88] C. C. Foster, "A generalization of avl trees," *Commun. ACM*, vol. 16, pp. 513–517, August 1973.

[89] P. E. McKenney and J. Walpole, "Introducing technology into the linux kernel: a case study," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 4–17, July 2008.

[90] J. Triplett, P. E. McKenney, and J. Walpole, "Scalable concurrent hash tables via relativistic programming," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 102–109, August 2010.

[91] Adam Langley, "Crit-bit Trees." `https://github.com/agl/critbit/raw/master/critbit.pdf` (Aug 11).

[92] M. Tim Jones, "Anatomy of the Linux slab allocator." `http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator/` (Aug 11).

[93] "Xiph celtclient.c." `http://git.xiph.org/?p=celt.git;a=tree;f=tools;h=0ab3f7d5db0677ec105434f2dec030c4d13707fe;hb=master` (Aug 11).

[94] "Advanced Linux Sound Architecture (ALSA) project." `http://www.alsa-project.org/main/index.php/Main_Page` (Aug 11).

[95] "Speex: a free codec for free speech." `http://www.speex.org/` (Aug 11).

[96] "CELT ultra-low delay audio codec." `http://www.celt-codec.org/` (Aug 11).

[97] "Test Anything Protocol." `http://testanything.org/` (Aug 11).

[98] Bradner, S. and McQuaid, J., "Benchmarking Methodology for Network Interconnect Devices." `http://tools.ietf.org/html/rfc2544` (Aug 11).

[99] "Useful kernel and driver performance tweaks for your Linux server." `http://timetobleed.com/useful-kernel-and-driver-performance-tweaks-for-your-linux-server/` (Aug 11).

[100] V. Jacobson, "Van Jacobson's network channels." `http://www.lemis.com/grog/Documentation/vj/lca06vj.pdf` (Aug 11).

[101] "Click mailing list: insmod: error inserting '/usr/local/lib/click.ko': -1 Cannot allocate memory." `http://www.mail-archive.com/click@amsterdam.lcs.mit.edu/msg05335.html` (Aug 11).

[102] "Linux Kernel Based Virtual Machine." `http://www.linux-kvm.org/page/Main_Page` (Aug 11).

[103] A. Keller, "The ANA Project: Development of the ANA-Core Software," Master's thesis, Communication Systems Group, ETH Zurich, Gloriastrasse 35, 8092 Zürich, Switzerland, 2007.

[104] D. Borkmann, "Linux Kernel Netdev: [PATCH] af_packet: tpacket_destruct_skb, deref skb after BUG_ON." `http://lists.openwall.net/netdev/2011/10/09/24` (Aug 11).

[105] D. Borkmann, "Linux Kernel Netdev: [PATCH] af_packet: remove unnecessary BUG_ON() in tpacket_destruct_skb." `http://www.spinics.net/lists/netdev/msg176599.html` (Aug 11).

[106] Eric Dumazet, "BPF JIT compiler for x86." `http://lingrok.org/source/xref/linux-2.6-linus/arch/x86/net/bpf_jit_comp.c` (Aug 11).

[107] "ReconOS, A programming model and operating system for reconfigurable hardware." `http://reconos.de/` (Aug 11).

[108] "Linux patch for the Kernel Source Level Debugger (kgdb) and the Kernel Source Level Debugger over Ethernet (kgdboe)." `http://kgdb.linsyssoft.com/intro.htm` (Aug 11).

[109] "netsniff-ng toolkit." `http://www.netsniff-ng.org/` (Aug 11).

[110] G. Muller, Y. Padioleau, J. L. Lawall, and R. R. Hansen, "Semantic patches considered helpful," *SIGOPS Oper. Syst. Rev.*, vol. 40, pp. 90–92, July 2006.

[111] Y. Padioleau, R. R. Hansen, J. L. Lawall, and G. Muller, "Semantic patches for documenting and automating collateral evolutions in linux device drivers," in *Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems*, PLOS '06, (New York, NY, USA), ACM, 2006.

[112] Y. Padioleau, J. L. Lawall, and G. Muller, "Understanding collateral evolution in linux device drivers," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, (New York, NY, USA), pp. 59–71, ACM, 2006.

[113] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in linux: ten years later," *SIGARCH Comput. Archit. News*, vol. 39, pp. 305–318, March 2011.

[114] "Coccinelle, a program matching and transformation engine which provides the Semantic Patch Language (SmPL)." `http://coccinelle.lip6.fr/` (Aug 11).

[115] "LAN Ethernet Maximum Rates, Generation, Capturing and Monitoring." http://wiki.networksecuritytoolkit.org/nstwiki/index.php/LAN_Ethernet_Maximum_Rates,_Generation,_Capturing_%26_Monitoring and http://blog.cryptoism.org/1318763742.html (Aug 11).

[116] H. Haas, "Mausezahn Traffic Generator." http://www.perihel.at/sec/mz/ (Aug 11).

[117] Robert Olsson et. al., "pktgen, the Linux kernel packet generator." http://lingrok.org/source/xref/linux-2.6-linus/net/core/pktgen.c (Aug 11).

[118] "libnet-dev project." http://libnet-dev.sourceforge.net/ (Aug 11).

[119] "netsniff-ng toolkit." http://netsniff-ng.org (Jul 11).

[120] "tcpdump and libpcap." http://www.tcpdump.org/ (Aug 11).

[121] "IPTraf, IP Network Monitoring Software." http://iptraf.seul.org/ (Aug 11).

[122] "Ncurses Library." http://www.gnu.org/software/ncurses/ncurses.html (Aug 11).

[123] "Linux kernel BPF virtual machine." http://lingrok.org/source/xref/linux-2.6-linus/net/core/filter.c (Aug 11).

[124] "tcpdump filter sheet." http://www.cs.ucr.edu/~marios/ethereal-tcpdump.pdf (Aug 11).

[125] "tcpdump, VLAN Q-in-Q or VLAN || VLAN filters not working." http://article.gmane.org/gmane.network.tcpdump.devel/5380 (Aug 11).

[126] Jonathan Corbet, "A JIT for packet filters." http://lwn.net/Articles/437981/ (Aug 11).

[127] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[128] N. Wirth, *Compiler Construction*. Addison-Wesley Professional, 1 ed., Oct. 1996.

[129] "flex: The Fast Lexical Analyzer." http://flex.sourceforge.net/ (Aug 11).

[130] "Bison - GNU parser generator." http://www.gnu.org/s/bison/ (Aug 11).

[131] D. E. Knuth, "On the translation of languages from left to right," *Information and Control*, vol. 8, pp. 607–639, October 1965.

[132] William Cohen, "Gaining insight into the Linux kernel with Kprobes." `http://www.redhat.com/magazine/005mar05/features/kprobes/` (Aug 11).

[133] S. Goswami, "An introduction to KProbes." `http://lwn.net/Articles/132196/` (Aug 11).

[134] "Kprobes documentation." `http://lingrok.org/source/xref/linux-2.6-linus/Documentation/kprobes.txt` (Aug 11).

[135] "Linux event notification chains." `http://lingrok.org/source/xref/linux-2.6-linus/kernel/notifier.c` (Aug 11).

Appendix A

Publication 'Efficient Implementation of Dynamic Protocol Stacks' at the ANCS 2011

We have presented our work at the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS) on October 3-4, 2011 in New York, USA.

There, we have participated in a poster session which is a forum for researchers to showcase and discuss their work to others. The attached paper has been published in the conference proceedings and summarizes the contents of our poster.

2011 Seventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems

Efficient Implementation of Dynamic Protocol Stacks

Ariane Keller
ETH Zurich, Switzerland
ariane.keller@tik.ee.ethz.ch

Daniel Borkmann
ETH Zurich, Switzerland
HTWK Leipzig, Germany
dborkma@tik.ee.ethz.ch

Wolfgang Mühlbauer
ETH Zurich, Switzerland
muehlbauer@tik.ee.ethz.ch

ABSTRACT

Network programming is widely understood as programming strictly defined socket interfaces. Only some frameworks have made a step towards *real* network programming by decomposing networking functionality into small modular blocks that can be assembled in a flexible manner. In this paper, we tackle the challenge of accommodating 3 partially conflicting objectives: (i) high flexibility for network programmers, (ii) re-configuration of the network stack at runtime, and (iii) high packet forwarding rates. First experiences with a prototype implementation in Linux suggest little performance overhead compared to the standard Linux protocol stack.

Categories and Subject Descriptors

C.2.1 [**Computer Communication Networks**]: Network Architecture and Design

General Terms

Design, Experimentation, Performance

Keywords

Network architecture, flexible network stacks, Future Internet experimentation, performance

1. INTRODUCTION

Beyond doubt, the Internet has grown out of its infancy and has become a critical infrastructure for private and business applications. Its success is largely due to the plethora of transport media it uses and to the rich set of network applications it offers. Yet, *network programming* is still mainly about programming sockets that form a strictly defined interface between the networking (TCP/IP) and the actual application part (Facebook, VoIP, etc.). What if designers of network applications could even tailor the networking functionality to their needs? We can just speculate about the resulting innovations.

Nowadays, changes in the configuration of a protocol stack usually require applications or even the operating system to be restarted. The need for changing the protocol stack can arise if networking functionality needs to be patched, if the used encryption method is not considered safe anymore, or when privacy concerns change. Ideally, applications should not be affected by such changes. Therefore, we advocate *run time reconfigurable* protocol stacks. For example, such protocol stacks can be useful for self-star properties in computing, since they provide an algorithm that configures and adapts the protocol stack autonomously.

Similar objectives were also followed by active networking [3], the Click modular router project[4], or OpenFlow [5], etc. Yet, we are not aware of any research that has achieved the following three partially conflicting goals:

1. Simple integration and testing of new protocols on end nodes on all layers of the protocol stack.
2. Runtime reconfiguration of the protocol stack in order to allow for even bigger flexibility.
3. High performance packet forwarding rates.

In this paper, we propose the *Lightweight Autonomic Network Architecture (LANA)*. Our architecture borrows ideas from ANA [2], where network functionality is divided into *functional blocks (FB)* that can be combined as required. Each FB implements a protocol such as *IP, UDP, or content centric routing.* ANA does not impose any protocols to be used. Rather it provides a framework that allows for the flexible composition and recomposition of FBs to a protocol stack. This allows for the experimentation with protocol stacks that are not known by today's standard operating systems, and it allows for the optimization of protocol stacks at runtime without communication tear down or application support. The existing implementation of ANA shows the feasibility of such a flexible architecture but suffers sever performance issues. In contrast to ANA, the proposed LANA architecture relies on a message passing by reference scheme, minimizes the number of threads, and uses optimized packet processing structures provided by the Linux kernel. Surprisingly, our first experiences with a prototype implementation suggest that we can offer comparable flexibility as ANA, but at packet forwarding rates comparable to those of the standard Linux networking stack.

2. LANA: APPROACH

Generally, the LANA network system is built similarly to the network subsystem of the Linux kernel. Applications can send and transmit packets via the BSD socket interface. The actual packet processing is done in a *packet processing engine (PPE)* in the kernel space. An overview of the architecture is presented in Figure 1.

The hardware and device driver interfaces are hidden from the PPE behind a *virtual link interface*, which allows for a simple integration of different underlaying networking technologies such as Ethernet, Bluetooth or InfiniBand.

Each functional block is implemented as a Linux kernel module. Upon module insertion a constructor for the cre-

978-0-7695-4521-9/11 $26.00 © 2011 IEEE
DOI 10.1109/ANCS.2011.19

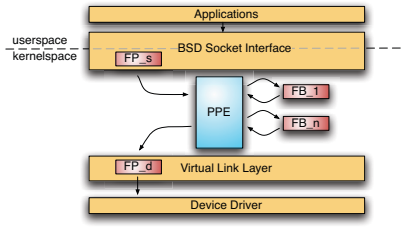

Figure 1: Packet flow in LANA

ation of an instance of the FB is registered with the LANA core. Upon configuration of the protocol stack the instances of the FBs are created. The instances register a *receive function* with the PPE. This function is called when a packet needs to be processed.

Functional blocks can either drop a packet, forward a packet to either ingress or egress direction, or duplicate a packet. After having processed a packet, the FB returns the identifier of the next FB that should process this packet. In addition, FBs belonging to the virtual link interface will queue the packets in the network drivers transmit queue and FBs communicating with BSD sockets will queue the packets in the sockets receive queue.

The PPE is responsible for calling one FB after the other and for queuing packets that need to be processed.

2.1 Implementation

The protocol stack can be configured from user space with the help of a command line tool. The most important commands are summarized below.

- `add`, `rm`: Adds (removes) an FB from the list of available FBs in the kernel.
- `set`: sets properties of an FB with a `key=value` semantic.
- `bind`, `unbind`: Binds (unbinds) an FB to another FB in order to be able to send messages to it.
- `replace`: Replaces one FB with another FB. The connections between the blocks are maintained. Private data can either be transferred to the new block or dropped.

Within the Linux kernel the notification chain framework is used to propagate those configuration messages to the individual FBs.

The current software is available under the GNU General Public License from [1]. In addition to the framework, it also includes five functional blocks: Ethernet, Berkeley Packet Filter, Tee (duplication of packets), Packet Counter and Forward (an empty block that forwards the packets to another block). The framework does not need any patching of the Linux kernel but it requires a new Linux 3.X kernel.

2.2 Improving the Performance

We have evaluated different options for the integration of the PPE with the Linux kernel. We summarize our insights to provide guidance for researchers who apply fundamental changes on the Linux protocol stack.

We compared the maximum packet reception rate of the Linux kernel while not doing any packet processing with LANA. In LANA packets are forwarded between three FBs that do only packet forwarding.

- One high priority LANA thread per CPU achieves approx. half the performance of the default Linux stack. The performance degradation is due to 'starvation' of the software interrupt handler (ksoftirqd). Changing the priority of the LANA thread only slightly increases the throughput.
- Explicit preemption and scheduling control achieves approx. two third of the performance of the default stack. The performance degradation is due to scheduling overhead.
- Execution of the PPE in ksoftirqd context achieves approx. 95% of the performance of the default stack.

The corresponding numbers are listed in Table 1.

Mechanism	Performance
Kernel threads (high priority)	700.000
Kernel threads (normal priority)	750.000
Kernel threads (controlled scheduling)	900.000
Execution in ksoftirqd	1.300.000
Linux kernel networking stack	1.380.000

Table 1: Performance evaluation in pps with 64 Byte packets. (Intel Core 2 Quad Q6600 with 2.40GHz, 4GB RAM, Intel 82566DC-2 NIC, Linux 3.0rc1)

3. CONCLUSIONS AND FUTURE WORK

We described how to implement a flexible protocol stack with similar performance as the default Linux stack. Its flexibility allows to include and test protocols, yet to be developed, and to change the protocol stack at runtime. In contrast to TCP/IP, our proposed solution allows to tailor the networking layer for the needs of a particular networking situation. In the short-term, we will compare LANA performance achieved in real scenarios with other systems (e.g., default Linux stack, Click router, etc.). In the mid-term, we plan to work on mechanisms that automatically configure protocol stacks based on the needs of applications and networks. In the long-term, we envisage a system that requires less configuration as compared to today's networks and that is able to adapt itself to changing network conditions.

4. ACKNOWLEDGMENTS

This research has received funding from the European Union 7th Framework Programme (grant n^o257906).

5. REFERENCES

[1] Lightweight Autonomic Network Architecture. `http://repo.or.cz/w/ana-net.git` (Jul 11).

[2] G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May. The autonomic network architecture (ANA). *Selected Areas in Communications, IEEE Journal on*, 28(1):4 –14, Jan. 2010.

[3] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela. A survey of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 29(2):7–23, 1999.

[4] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, March 2008.

Appendix B

Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Master Thesis

Lightweight implementation of ANA

Daniel Borkmann

Advisor: Ariane Keller, ariane.keller@tik.ee.ethz.ch
Co-Advisors: Wolfgang Mühlbauer, wolfgang.muehlbauer@tik.ee.ethz.ch
Professor: Prof. Dr. Bernhard Plattner, ETH Zurich, plattner@tik.ee.ethz.ch
Professor: Prof. Dr.-Ing. Dietmar Reimann, HTWK Leipzig,
reimann@imn.htwk-leipzig.de

Duration: 6 months (PrüfO-INM, 04 November 2009).

1 Introduction

This master thesis is in the context of the EPiCS project. The goal of the EPiCS project is to lay the foundation for engineering the novel class of proprioceptive computing systems. Proprioceptive computing systems collect and maintain information about their state and progress, which enables self-awareness by reasoning about their behaviour, and self- expression by effectively and autonomously adapting their behaviour to changing conditions. Concepts of self-awareness and self-expression are new to the domains of computing and networking; the successful transfer and development of these concepts will help create future heterogeneous and distributed systems capable of efficiently responding to a multitude of requirements with respect to functionality and flexibility, performance, resource usage and costs, reliability and safety, and security.

In this thesis we focus on the networking aspect of EPiCS. EPiCS uses the network architecture developed in the ANA project as a basis. The ANA network architecture is a novel architecture that enables flexible, dynamic, and fully autonomous formation of network nodes. While in the ANA project the focus was on developing this architecture in the EPiCS project the focus is on porting this architecture to embedded systems that consist of programmable hardware and software parts and to enhance ANA with some algorithms that adapt the current implementation to provide optimal performance.

The objective of this master thesis is to assess the ANA architecture and its current implementation with respect to suitability for use on embedded devices. Basic functionality should be identified and implemented with respect to resource limitations and performance of embedded systems.

2 Assignment

This assignment aims to outline the work to be conducted during this thesis. The assignment may need to be adapted over the course of the project.

2.1 Objectives

The goal of this master thesis is to develop a lightweight ANA architecture and implementation. The developed architecture has to follow the basic ANA principles while the implementation has to significantly outperform the previous implementation. The first step will be to identify the critical parts of the ANA architecture and to combine them in a new architecture. In the second step this architecture will be implemented. The main part of this implementation should be in the Linux Kernel space. In the third step the performance will be evaluated.

2.2 Tasks

This section gives a brief overview of the tasks the student is expected to perform towards achieving the objective outlined above. The binding project plan will be derived over the course of the first three weeks depending on the knowledge and skills the student brings into the project.

2.2.1 Familiarization

- Study the available literature on ANA [1, 2, 3].
- Setup a Linux machine on which you want to do your implementation, consider to use a vmware.
- In collaboration with the advisor, derive a project plan for your master thesis. Allow time for the design, implementation, evaluation, and documentation of your software.

2.2.2 Architecture and software design

- Determine the basic functionality of ANA.
- Determine whether the ANA-API still can be used or determine why not. If not, determine a new API.
- Split the functionality between Linux user and Linux Kernel space.
- Design the kernel space, the interface between user space and kernel space, and the user space part. The design should be extensible with respect to more advanced ANA functionality.
- Design a basic scenario that shows that your implementation follows the ANA principles.
- Think about possible test scenarios.
- Optional: Add some more advanced ANA functionality.
- Optional: Design an interface that allows to receive packets from ReconOS. [4, 5]

2.2.3 Implementation

- Determine an appropriate version control system. The EPiCS project is hosted at github. You might want to put your code in the same repository.
- Use the Linux kernel coding style.
- Implement your design.
- Provide a simple script that shows how your design can be loaded and how packets can be exchanged between different modules.
- Optional: Implement some more advanced ANA functionality.
- Optional: Implement the interface to ReconOS.

2.2.4 Validation

- Validate the correct operation of your implementation.
- Check the resilience of the implementation, including its configuration interface, to uneducated users.

2.2.5 Evaluation

- Do a performance evaluation of your implementation.
- Optional: Determine the bottlenecks of your implementation.
- Optional: Do a performance comparison between packet forwarding for different combinations of hardware, Kernel space and User space.

2.2.6 Documentation

- Appropriate source code documentation.
- Write a step-by-step how to for starting your code and for some simple packet exchange between different modules.
- Write a documentation about the design, implementation and validation of the lightweight ANA implementation.

3 Milestones

- Provide a "project plan" which identifies the mile stones.
- Two intermediate presentations: Give a presentation of 10 minutes to the professor and the advisors. In this presentation, the student presents major aspects of the ongoing work including results, obstacles, and remaining work.
- Final presentation of 15 minutes in the CSG group meeting, or, alternatively, via teleconference. The presentation should carefully introduce the setting and fundamental assumptions of the project. The main part should focus on the major results and conclusions from the work.
- Masterseminar: Intermediate presentation and discussion at HTWK Leipzig. (StudO-INM, 4 November 2009).
- Masterkolloquium at HTWK Leipzig. Presentation of 30 minutes of the master thesis with a question session of at most 60 minutes. (PrüfO-INM, 4 November 2009).
- Any software that is produced in the context of this thesis and its documentation needs to be delivered before conclusion of the thesis. This includes all source code and documentation. The source files for the final report and all data, scripts and tools developed to generate the figures of the report must be included. Preferred format for delivery is a CD-R.
- Final report. The final report must contain a summary, the assignment, the time schedule and the Declaration of Originality. Its structure should include the following sections: Introduction, Background/Related Work, Design/Methodology, Validation/Evaluation, Conclusion, and Future work. Related work must be referenced appropriately.

4 Organization

- Student and advisor hold a weekly meeting to discuss progress of work and next steps. The student should not hesitate to contact the advisor at any time. The common goal of the advisor and the student is to maximize the outcome of the project.
- The student is encouraged to write all reports in English; German is accepted as well.
- The core source code will be published under the GNU general public license.

5 References

[1] ANA Core Documentation: All you need to know to use and develop ANA software.
[2] ANA Blueprint: D.1.9 ANA Blueprint Update, D.1.9b Description of the low-level machinery of ANA
[3] http://www.ana-project.org
[4] http://www.epics-project.eu
[5] http://www.reconos.de

Appendix C

Getting Started with LANA

This chapter presents a short tutorial for building the Linux kernel, LANA modules and the LANA example application, for loading LANA into the kernel and for developing own functional blocks.

This tutorial assumes that you are using Debian GNU/Linux 6.0. The basic set of tools that are needed can be obtained via `apt-get` with the following packages:

- `build-essential` (a Debian meta-package containing basic utilities and libraries for building software like `gcc`, `make`, `g++`, `coreutils`, `bsdutils`, `util-linux`, `sed`, `tar` and others)
- `gdb` (the GNU debugger)
- `libncurses5` and `libncurses5-dev` (the graphical ncurses library, needed for `ifpps`)
- `git` (the distributed version control system [86])
- `grub` or `grub2` (a system boot loader, `grub2` is default on most installations)
- `vim`, `uemacs` or similar (a command-line text editor)
- `minicom` (terminal emulation and text-based modem, for remote debugging via RS232)
- `coccinelle` (a semantic patching tool for C)
- `flex` (a fast lexical analyzer generator, needed for `bpfc`)
- `bison` (a parser generator that is compatible with YACC, needed for `bpfc`)
- `libcelt0-0` and `libcelt-dev` (CELT codec, needed fot the LANA application)
- `libspeexdsp1` and `libspeexdsp-dev` (Speex DSP, needed fot the LANA application)
- `libasound2` and `libasound2-dev` (ALSA lib, needed fot the LANA application)

Optionally, it might be useful to port the Kernel Source Level Debugger (`kgdb`) and the Kernel Source Level Debugger over Ethernet (`kgdboe`) without too much

efforts from [108] for kernel debugging to the latest Linux release, since Linux explicitly avoids shipping a debugger within the kernel.

C.1 Building Linux and LANA

First of all, the kernel sources of Linus Torvald's Git tree are obtained with the command

```
  git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/
linux-2.6.git
  cd linux-2.6/
  git checkout v3.0
```

and then built with:

```
  make menuconfig
```

Within `menuconfig` you can optionally enable kernel hacking options, the inclusion of needed drivers, other subsystems and more. If you are not sure what you need to choose here, simply exit the menu and save the default kernel configuration file (`.config`).

```
  make -j4 && make modules -j4
```

Afterwards, an initial ramdisk is needed which is a temporary file system used for booting up the Linux kernel. It can be created by the following steps:

```
  make install && make modules_install
  cd /boot/
  mkinitramfs -o initrd.img-3.0.0 3.0.0
```

Note that the version `3.0.0` is just an example and can be different in some cases on your machine. And on the last step, the boot loader needs to be updated to recognize the newly created ramdisk:

```
  update-grub
```

We can then boot into the new kernel:

```
  reboot
```

Note that you also might want to deactivate `quiet` from the kernel command-line, in order to see `printk` outputs during the boot process. Edit `/etc/default/grub` and remove it from the command-line with `GRUB_CMDLINE_LINUX_DEFAULT=""` and do an `update-grub` afterwards.

After the reboot, the LANA repository can then be fetched via Git with the command

```
git clone git://repo.or.cz/ana-net.git
```

and compiled through the following steps:

```
cd ana-net/src
make
cd ../usr
make
cd ../app
make
```

LANA's kernel modules can be found in `src/`, user space tools in `usr/` and the LANA example application has been built in `app/`.

In case your ALSA kernel drivers have a newer version (`cat /proc/asound/version`) than the user space libraries and utilities, you eventually might want to rebuild ALSA's user space software:

```
git clone git://git.alsa-project.org/alsa-lib.git
git clone git://git.alsa-project.org/alsa-plugins.git
git clone git://git.alsa-project.org/alsa-utils.git
git clone git://git.alsa-project.org/alsa-tools.git
```

The corresponding release tag can be found within the source directory by listing the Git tags with `git tag -l` and checking out the one, that corresponds to `/proc/asound/version`. Each source directory has an Automake build system with a `configure` script. A simple test of ALSA can be performed by capturing sound via `arecord` and replay the resulting sound file with `aplay`.

C.2 Remote Debugging of LANA

If `kgdb` or `kgdboe` is not present on the system, it can be useful to redirect kernel messages to a RS232-compliant serial port for monitoring and debugging purposes on a remote machine. Just add

```
console=/dev/ttyS0,9600n8
```

to the kernel command-line, like:

```
vim /boot/grub/menu.lst
```

```
menuentry 'Debian/Linux 3.0.0' --class debian --class gnu-linux
--class gnu --class os {
  insmod part_msdos
  insmod ext2
  set root='(hd0,msdos6)'
  search --no-floppy --fs-uuid --set <...>
  linux /boot/vmlinuz-3.0.0 root=UUID=<...> ro console=/dev/ttyS0,9600n8
  initrd /boot/initrd.img-3.0.0
}
```

If you also intend to redirect the tty output of your development system to the serial port, add the following to the `/etc/inittab`:

```
s0:2345:respawn:/sbin/agetty -L ttyS0 9600 vt100
```

Both configurations assume that `ttyS0` is present as a serial port device. This can be checked on a running kernel with:

```
dmesg | grep tty
```

If you have rebooted the kernel for applying the new configuration, the remote machine can then use `minicom` with the serial port settings of `9600 bit/s`, `N` (no parity), `8` (data bits), `1` (stop bit) and flow control turned off to print out console messages of the development machine.

C.3 Setup of LANA Modules

Within this section, a simple LANA stack according to figure C.1 is being configured. This particular example receives incoming packets on an `fb_eth` block that forwards them to an `fb_tee` block, which spawns a copy of the packet to `fb3` and to `fb5`, both of type `fb_bpf`. `fb3` shall accept IPv4 typed packets and `fb5` only ARP typed packets. If packets pass the BPF blocks, then a per-CPU counter is going to accumulate statistics for IPv4 respectively ARP in `fb4` and `fb6`.

First of all, after building the modules, the LANA core kernel module is being loaded to the kernel via `insmod`:

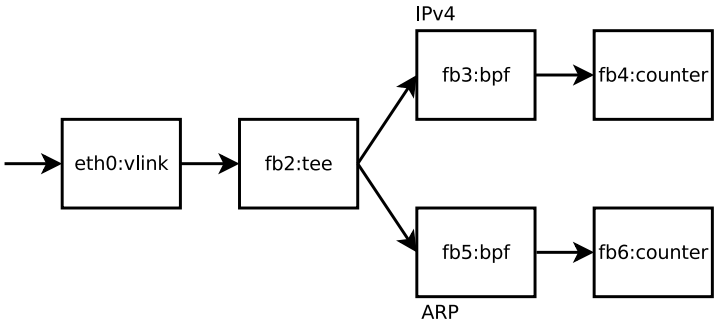

Figure C.1: LANA example configuration

```
insmod lana.ko
```

The kernel will then print a message that LANA has been loaded successfully:

`dmesg` shows:

```
[111608.202549] [lana] bootstrapping core ...
[111608.203050] [lana] core up and running!
```

Afterwards, we need to load all kernel modules for this example via `insmod`, too:

```
insmod fb_eth.ko
insmod fb_bpf.ko
insmod fb_tee.ko
insmod fb_counter.ko
```

Now none of the loaded functional blocks are active, yet. They have only been registered to a functional block builder with a constructor and destructor as well as a type identifier. To add instances of these functional blocks into the LANA stack, do the following:

```
vlink ethernet hook eth0
fbctl add fb2 tee
fbctl add fb3 bpf
fbctl add fb4 counter
fbctl add fb5 bpf
fbctl add fb6 counter
```

Now `cat /proc/net/lana/fblocks` shows functional block instances that are currently present in LANA with information about the name of the instance, its type, its kernel address, its information dispatch point, its reference count and

a list of bound or subscribed information dispatch points of functional blocks (empty at the beginning since nothing has been bound):

```
eth0 vlink ffff88007e6ee000 1 1 []
fb2 tee ffff88007e73c000 2 1 []
fb3 bpf ffff88007e749000 3 1 []
fb4 counter ffff88007e6ee0c0 4 1 []
fb5 bpf ffff88007e7490c0 5 1 []
fb6 counter ffff88007e6ee180 6 1 []
```

A `cat /proc/net/lana/ppe` shows per-CPU statistics about the packet processing engine of LANA with information about the number of received packets that entered the packet processing engine, total bytes of the packets, the number of invocations of functional block handler, the number of timer calls, the number of timer calls that were scheduled on the wrong CPU, and the length of the packet processing engine's backlog queue:

```
CPU0:  858 54837 0 3453 0 0
CPU1:  846 53536 0 2435 0 0
CPU2:  828 55750 0 2354 0 0
CPU3:  818 55813 0 3334 0 0
```

The next step before binding functional blocks is to create Berkeley Packet Filter programs that will be translated by `bpfc` (Appendix D.3 [1]) to a kernel readable bytecode format which can be piped into the BPF functional block for filter configuration:

`vim arp.bpf` with (comments start with the `;` character):

```
ldh #proto ; Load packets Ethernet type field into accumulator
jeq #0x806,L1,L2 ; Compare with Ethernet type for ARP
L1:  ret #0xffffff ; Accept packet
L2:  ret #0 ; Drop packet
```

and

`vim ipv4.bpf` with:

```
ldh #proto ; Load packets Ethernet type field into accumulator
jeq #0x800,L1,L2 ; Compare with Ethernet type for IPv4
```

[1] Note that there is also a Vim syntax highlighting file that has been developed in this work. It is part of the netsniff-ng Git repository [109].

```
L1:  ret #0xfffff ; Accept packet
L2:  ret #0 ; Drop packet
```

The resulting program can be checked by `bpfc` if the verbose mode is switched on:

`bpfc -Vi arp.bpf` will show:

```
*** Generated program:
(000) ldh [-4096]
(001) jeq #0x806 jt 2 jf 3
(002) ret #1048575
(003) ret #0
*** Validating:  is valid!
*** Result:
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 1, 0x00000806 },
{ 0x6, 0, 0, 0x000fffff },
{ 0x6, 0, 0, 0x00000000 },
```

Now, the resulting code can be piped into the BPF functional blocks with:

```
bpfc ipv4.bpf > /proc/net/lana/fblock/fb3
bpfc arp.bpf > /proc/net/lana/fblock/fb5
```

The kernel will acknowledge the filter by printing out a message (`dmesg`):

```
[177784.217556] [fb3::bpf] Parsed code:
[177784.217563] [fb3::bpf] 0:  c:0x28 jt:0 jf:0 k:0xfffff000
[177784.217566] [fb3::bpf] 1:  c:0x15 jt:0 jf:1 k:0x800
[177784.217570] [fb3::bpf] 2:  c:0x6 jt:0 jf:0 k:0xfffff
[177784.217573] [fb3::bpf] 3:  c:0x6 jt:0 jf:0 k:0x0
[177784.217581] [fb3::bpf] Filter injected!
[177788.929628] [fb5::bpf] Parsed code:
[177788.929634] [fb5::bpf] 0:  c:0x28 jt:0 jf:0 k:0xfffff000
[177788.929638] [fb5::bpf] 1:  c:0x15 jt:0 jf:1 k:0x806
[177788.929641] [fb5::bpf] 2:  c:0x6 jt:0 jf:0 k:0xfffff
[177788.929644] [fb5::bpf] 3:  c:0x6 jt:0 jf:0 k:0x0
[177788.929652] [fb5::bpf] Filter injected!
```

Now the functional blocks can be bound together in the LANA stack:

```
fbctl bind fb2 eth0
fbctl bind fb3 fb2
```

```
fbctl bind fb5 fb2
fbctl bind fb4 fb3
fbctl bind fb6 fb5
```

Again, the result can be seen in `dmesg`

```
[178962.252897] [fb2::tee] port egress bound to IDP1
[178962.252902] [eth0::vlink] port ingress bound to IDP2
[179000.733011] [fb3::bpf] port egress bound to IDP2
[179000.733016] [fb2::tee] port ingress bound to IDP3
[179005.940999] [fb5::bpf] port egress bound to IDP2
[179005.941004] [fb2::tee] port ingress bound to IDP5
[179039.581172] [fb4::counter] port egress bound to IDP3
[179039.581181] [fb3::bpf] port ingress bound to IDP4
[179048.013100] [fb6::counter] port egress bound to IDP5
[179048.013106] [fb5::bpf] port ingress bound to IDP6
```

and also in the dependency list of each functional block in `/proc/net/lana/fblocks`:

```
eth0 vlink ffff88006c0ad180 1 4 [2]
fb2 tee ffff88006c0ad0c0 2 10 [5 3 1]
fb3 bpf ffff88006c13b000 3 7 [4 2]
fb4 counter ffff88006c13b0c0 4 4 [3]
fb5 bpf ffff88006c13b180 5 7 [6 2]
fb6 counter ffff88006c13b240 6 4 [5]
```

Now, counter data can be fetched with `cat` via `/proc/net/lana/fblock/fb4` and `/proc/net/lana/fblock/fb6`. The output of both files show two numbers, the packet count and the accumulated byte count.

To remove LANA from the system, we need to unbind all functional blocks first, then we remove the instances, and finally we can remove the kernel modules:

```
fbctl unbind fb6 fb5
fbctl unbind fb4 fb3
fbctl unbind fb5 fb2
fbctl unbind fb3 fb2
fbctl unbind fb2 eth0
vlink ethernet unhook eth0
fbctl rm fb2
fbctl rm fb3
fbctl rm fb4
fbctl rm fb5
```

```
fbctl rm fb6
rmmod fb_eth.ko
rmmod fb_bpf.ko
rmmod fb_tee.ko
rmmod fb_counter.ko
rmmod lana.ko
```

C.4 Functional Block Development

If you intend to develop a new functional block, it is recommended that you are already familiar with kernel development, have a basic understanding of techniques like RCU, when to use which locking mechanisms and about per-CPU data structures.

There is also a freely available book about parallel programming written by the Linux kernel developer Paul McKenney that can be gathered via Git:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.
git
```

The usual way to start developing a new LANA functional block is to make a copy of `src/fb_dummy.c` since this is the simplest possible functional block of LANA. Replace all the `fb_dummy_*` functions to appropriate named ones.

Here, we use to call the new functional block `fb_myblock`. The most important functions of `fb_myblock` are `fb_myblock_ctor` and `fb_myblock_dtor` which are constructor respectively destructor functions of `struct fblock_factory - fb_myblock_factory`, which are called upon the request of a new instance of a functional block of type `myblock`. Next to the constructor and destructor, `fb_myblock_netrx` and `fb_myblock_event` should be implemented as well. The code from the dummy functional block gives an example and starting point on how these four functions can be implemented.

For instance, the dummy block receive function (`fb_myblock_netrx`) can simply forward a socket buffer to the next bound block:

```
struct fb_myblock_priv {
        idp_t port[2];
        seqlock_t lock;
};

static int fb_myblock_netrx(const struct fblock * const fb,
                            struct sk_buff * const skb,
                            enum path_type * const dir)
{
```

```
        int drop = 0;
        unsigned int seq;
        struct fb_myblock_priv __percpu *fb_priv_cpu;

        fb_priv_cpu = this_cpu_ptr(rcu_dereference_raw(fb->private_data));
#ifdef __DEBUG
        printk("Got skb on %p on ppe%d!\n", fb, smp_processor_id());
#endif
        prefetchw(skb->cb);
        do {
                seq = read_seqbegin(&fb_priv_cpu->lock);
                write_next_idp_to_skb(skb, fb->idp, fb_priv_cpu->port[*dir]);
                if (fb_priv_cpu->port[*dir] == IDP_UNKNOWN)
                        drop = 1;
        } while (read_seqretry(&fb_priv_cpu->lock, seq));
        if (drop) {
                kfree_skb(skb);
                return PPE_DROPPED;
        }
        return PPE_SUCCESS;
}
```

Here, CPU-local private data of this functional block instance is being fetched by a RCU dereference within `rcu_read_lock` held through the packet processing engine. This private data reads out the next IDP number of a functional block under a CPU-local sequential lock, which is lock-free under a `x86_64` architecture. The IDP is then written into the socket buffer's private data area via `write_next_idp_to_skb`. If no subsequent IDP has been set, then the socket buffer is freed and a `PPE_DROPPED` is given to the packet processing engine. Otherwise, the function returns with `PPE_SUCCESS` and the packet processing engine calls the next functional block handler according to the encoded IDP of the `skb`.

C.5 LANA Development with Coccinelle

Together with the LANA source code, a folder called `sem/` is being shipped. This folder contains programs under the GPL version 2 that mostly have been taken from the Linux kernel source. These programs are written in `SmPL` (Semantic Patch Language) and can be interpreted by Coccinelle. More information about Semantic Patches and Coccinelle can be found in Lawall et al. [110], [111], [112] and [113].

SmPL has been designed to make collateral code changes in C more easy by developing a semantic matching and transformation engine that generates automatically patches in C, if a given match has been detected in a specified code base[114].

The main purpose for its usage in the development of LANA is to find errors in the code base of LANA or to semi-automatically review the code quality of

the implemented kernel modules.

Some general use cases that are also applicable for LANA are the following (mentioned files are from `sem/`):

- **kfree.cocci:** finds a memory use after `kfree`
- **kzalloc-simple.cocci:** uses `kzalloc` rather than `kmalloc` followed by `memset` with 0
- **double_lock.cocci:** finds possible double locks
- **flags.cocci:** finds nested `lock+irqsave` functions that use the same flags variables
- **mini_lock.cocci:** finds possible missing unlocks
- **doubleinit.cocci:** finds duplicate field initializations
- **notnull.cocci:** detects `NULL` tests that can only be reached when the value is known not to be `NULL`
- **deref_null.cocci:** detects a variable that is dereference under a `NULL` test even though it is know to be `NULL`
- **drop_kmalloc_cast.cocci:** finds casts after (`void *`) values returned by `kmalloc`, since this is useless

Consider the following faulty C code:

```
#include <stdio.h>
#include <stdlib.h>

struct foo {
        int bar;
};

int main(void)
{
        struct foo *f = malloc(sizeof(*f));
        if (f == NULL) {
                return f->bar;
        }
        return 0;
}
```

If `spatch` is run against this C file by using the SmPL script `deref_null.cocci`, it will report an error that `f` is dereferenced even though `f` is known to be `NULL`:

```
  spatch -D report -local_includes -out_place -sp_file null/deref_null.cocci
test.c
```

`spatch` will then complain:

```
init_defs_builtins:  /usr//share/coccinelle/standard.h
HANDLING: test.c
test.c:12:12-15:  ERROR: f is NULL but dereferenced.
```

To build the latest Coccinelle version under Debian GNU/Linux, the following packages must be installed first:

```
apt-get install ocaml-native-compilers ocaml-findlib libpycaml-ocaml-dev
libsexplib-camlp4-dev menhir libmenhir-ocaml-dev
```

Afterwards, within the LANA repository root directory, the following commands under root build and install the latest version of Coccinelle:

```
cd sem/
make spatch
```

As a non-priviledged user, the LANA source folder can then be checked with a basic set of SmPL scripts with:

```
make
```

Appendix D

LANA-derived Development Utilities

During the process of implementation, validation and performance evaluation of LANA, we have implemented several utilities that facilitate the development of LANA from different aspects. These development utilities are shortly presented in this chapter.

D.1 High-Performance Zero-Copy Traffic Generator

For performance evaluation and LANA debugging purposes, we have implemented a fast zero-copy traffic generator, named `trafgen`. `trafgen` utilizes the `PF_PACKET` (`packet(7)`) socket interface of Linux which postpones complete control over packet data and packet headers into the user space. Since Linux 2.6.31, a new `PF_PACKET` extension has been added into the mainline kernel that is known under the term zero-copy `TX_RING` [4].

`TX_RING` is a ring buffer with virtual memory that is directly mapped into both address spaces (figure D.1). Thus, kernel space and user space can access this buffer without needing to perform system calls or additional context switches and without needing to copy buffers between address spaces. The `TX_RING` buffer is configurable in size and each ring buffer slot has a header with control information such as a status flag. The status flag provides information about the current usage of the slot. Thus, (i) the kernel knows if this slot is ready for transmission and (ii) the user space knows whether the current slot can be filled with a new packet.

If the kernel is triggered to process the `TX_RING` data, it allocates a new socket buffer structure for each filled ring slot, sets the `TX_RING` pages of the current slot as data fragments, and finally calls `dev_queue_xmit` for transmission (section 3.1.2).

For using the `TX_RING` with high-speed packet rates, network device drivers

should have NAPI (section 3.1.1) enabled to perform interrupt load mitigation. In `trafgen`, every 10 microseconds (default, can be changed via command line option), a real-time timer calls `sendto(2)` in order to trigger the kernel for processing frames of the `TX_RING`.

Via command line option, `trafgen` can also be bound to run on a specific CPU. Thus, overhead of process and cache-line migration is avoided, if the Linux process scheduler decides to migrate `trafgen` to a different CPU. Further, if `trafgen` is bound to a specific CPU, it automatically migrates the NIC's interrupt affinity to the bound CPU, too. This is done in order to avoid cache-line migration to the NIC's interrupt CPU, hence, to keep data CPU local.

The `TX_RING` size can also be configured via command line option with values ranging from megabytes to gigabytes. Furthermore, `trafgen` makes use of our own assembler-optimized `memcpy` for x86/x86_64 architectures with MMX registers in order to speed up copying the generated packet template into the `TX_RING` slot.

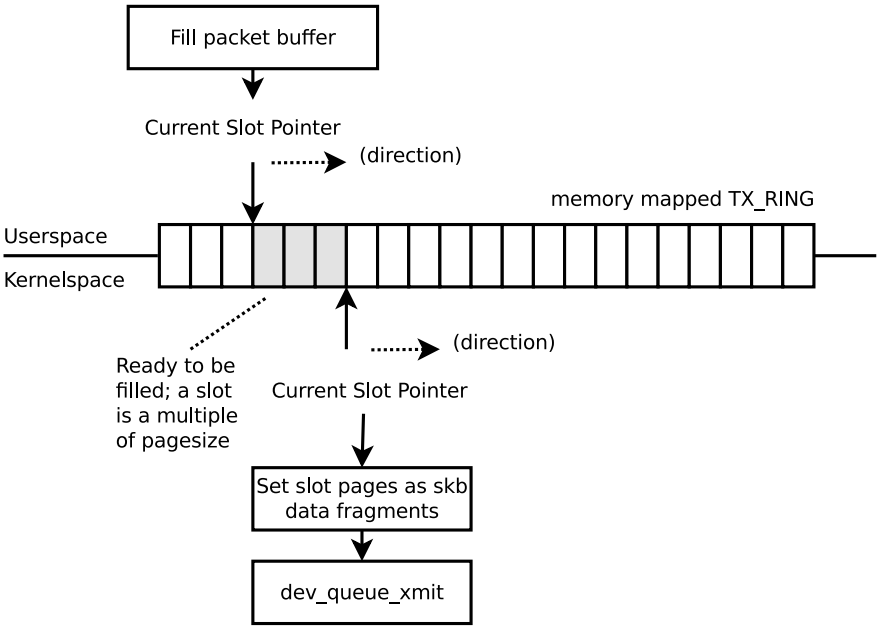

Figure D.1: `TX_RING` buffer used by `trafgen`

By exploiting the `TX_RING` for transmission, small-sized packet rates with approx. 1.25 mio pps were generated by `trafgen` on an Intel Core 2 Quad CPU with 2.40 GHz, 4 GB RAM and an Intel 82566DC-2 Gigabit Ethernet card (figure D.2). `trafgen` was bound to a single CPU and `trafgen`'s CPU interrupt migration was activated, thus NIC interrupts were received on the same CPU on which `trafgen` was bound to. An identical machine was used for packet reception, both machines were directly connected and `ifpps` (section D.2) was used on the receive-side for measurement. Since we have already published the source code

of trafgen, we attracted users to perform further benchmarks with trafgen on their hardware. We found out that the results heavily depend on the used Gigabit Ethernet adapter. For instance, Ronald W. Henderson wrote a Wiki article [115] about our trafgen where he reached the physical line rate of 1.488 mio 64 Byte pps.

With our test setup, we have compared trafgen with two other packet generators, namely mausezahn [116] and pktgen [117]. mausezahn is a fast user space packet generator that uses libnet [118], a framework for low-level network packet construction. The second traffic generator is pktgen, which is part of the Linux mainline kernel and resides in the core of the networking subsystem.

In contrast to trafgen and mausezahn, pktgen must be configured via procfs. pktgen's configuration options are limited to basic protocols like IPv4 or IPv6. As a transport layer protocol, only UDP is supported and packet payload cannot be configured at all. Figure D.2 shows that even for small packets, the kernel space pktgen is able to transmit up to 1.38 mio pps.

The kernel source code shows that one packet copy *can* be avoided in comparison to trafgen. In case of trafgen, the kernel does *not* copy the TX_RING slot data to skb->data, but sets data pages as socket buffer fragments. Hence, in dev_hard_start_xmit the buffer might need to linearize its fragments in some cases through __skb_linearize (section 3.1.2) to DMA-capable memory. pktgen on the contrary can directly allocate an already linearized and DMA-capable buffer, thus this can be the cause of trafgen's performance penalty.

However, trafgen is still up to 40 percent faster than mausezahn with the benefit of having more degrees of freedom regarding packet configuration in contrast to pktgen. On larger packet sizes, all three traffic generators have a similar pps performance in the test setup. We assume that this is mainly due to hardware and bandwidth limitations of the underlying system. On better equipped systems with e.g. 10 Gigabit Ethernet, we assume that the order of performance of the benchmarked tools looks similar to the part with smaller sized packets.

Next to the TX_RING, trafgen has a second working mode that allows the definition of inter-packet departure times, which is mainly used for debugging purposes in LANA. This method invokes system calls and the copy of packet buffers for transmission, since inter-packet departure times are not supported by the TX_RING. This is also realized using PF_PACKET sockets, but instead of allocating a TX_RING, packets are directly transmitted with sendto(2).

Furthermore, trafgen provides its own packet configuration language. By this, multiple packets can be defined in a single packet configuration file, where packet headers and packet payload are specified byte-wise. Within such a packet configuration, there can be elements like counter or random number generators, thus e.g. bytes of a source MAC address can be randomized or incremented.

trafgen is published under the GNU GPL version 2 and has been added into the netsniff-ng toolkit [119]. trafgen does not need any libraries except libc and can be built and installed with:

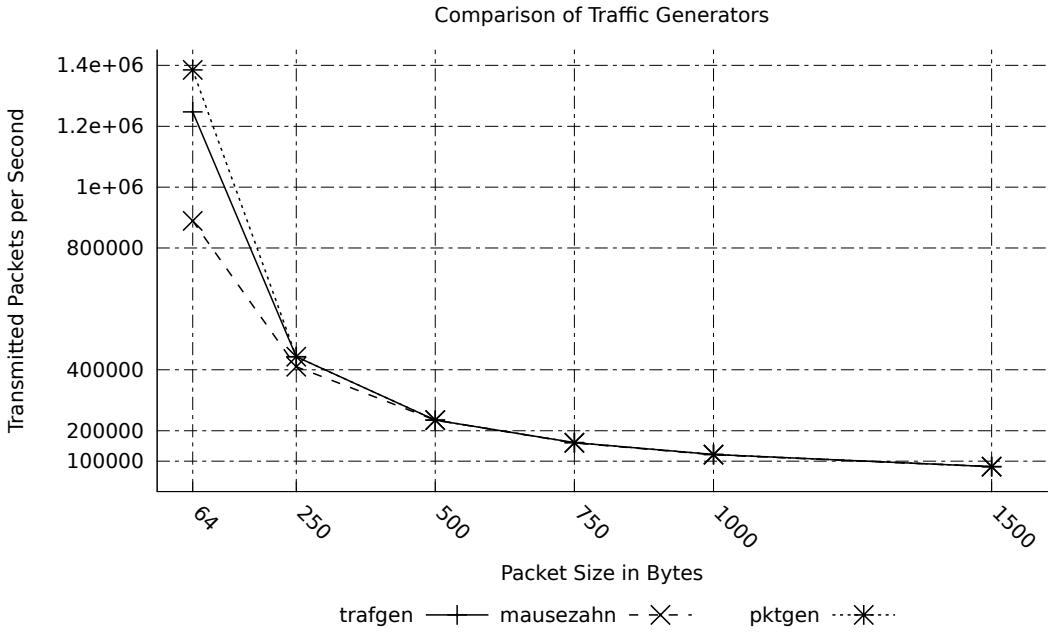

Figure D.2: `trafgen` compared to `mausezahn` and the kernel space `pktgen`

```
git clone git://repo.or.cz/netsniff-ng.git
cd netsniff-ng/src/trafgen
make && make install
```

The netsniff-ng toolkit also ships an example packet configuration file that can be used for high-speed transmissions:

```
trafgen -dev eth0 -conf ../trafgen.txf -bind 0
```

Further command line attributes are explained in `trafgen -h`.

Future work on `trafgen` will include a Cisco-like telnet front-end `tgsh` for a more user-friendly packet configuration and network device management. There, packets with LANA-specific protocols as well as traditional Internet protocols can be configured. Further, multiple user logins and network device statistics will be supported, thus `trafgen` can be used on a traffic generator appliance.

D.2 Top-like Kernel Networking Statistics

For measurement purposes, we have implemented a tool called `ifpps`, which periodically provides top-like networking and system statistics from the kernel.

`ifpps` gathers its data directly from procfs files and does not apply any user

space monitoring libraries such as libpcap [120] which is used in tools like `iptraf` [121], for instance.

The main idea behind `ifpps` is to apply principles from section 2.2.1 in order to be able to have more accurate networking statistics under a high packet load. For instance, consider the following scenario: two directly connected Linux machines with Intel Core 2 Quad Q6600 2.40GHz CPUs, 4 GB RAM, and an Intel 82566DC-2 Gigabit Ethernet NIC are used for performance evaluation. One machine generates 64 Byte network packets by using the kernel space packet generator `pktgen` with a maximum possible packet rate. The other machine displays statistics about incoming network packets by using i) `iptraf` and ii) `ifpps`. `iptraf`, that incorporates libpcap, shows an average packet rate of 246,000 pps while on the other hand `ifpps` shows an average packet rate of 1,378,000 pps. Hence, due to copying packets and deferring statistics creation into user space, a measurement error of approx. 460 per cent occurs.

Tools like `iptraf`, for instance, display much more information such as TCP per flow statistics (therefore the use of libpcap), which we have not implemented in `ifpps` because overall networking statistics are in our focus. The principle P1 in our case is applied by avoiding collecting network packets in user space for statistics creation. Further, principle P2 means that we let the kernel calculate packet statistics, for instance, within the network device drivers. With both principles applied, we fetch network driver receive and transmit statistics from procfs. Hence, the following files are of interest for `ifpps`:

- **/proc/net/dev:** network device receive and transmit statistics
- **/proc/softirqs:** per CPU statistics about scheduled `NET_RX` and `NET_TX` software interrupts (section 3.1)
- **/proc/interrupts:** per CPU network device hardware interrupts
- **/proc/stat:** per CPU statistics about time (in `USER_HZ`) spent in user, system, idle, and IO-wait mode
- **/proc/meminfo:** total and free memory statistics

Every given time interval (t, default: $t = 1s$), statistics are parsed from procfs and displayed in an ncurses-based [122] screen. An example `ifpps` output of `ifpps eth0` looks like the following:

```
Kernel net/sys statistics for eth0

RX:      0.003 MiB/t       20 pkts/t        0 drops/t        0 errors/t
TX:      0.000 MiB/t        2 pkts/t        0 drops/t        0 errors/t

RX:    226.372 MiB    657800 pkts          0 drops          0 errors
TX:     12.104 MiB    101317 pkts          0 drops          0 errors
```

```
SYS:     2160 cs/t    43.9% mem          1 running       0 iowait

CPU0:    0.0% usr/t    0.0% sys/t  100.0% idl/t      0.0% iow/t
CPU1:    0.0% usr/t    0.5% sys/t   99.5% idl/t      0.0% iow/t
CPU2:    0.5% usr/t    0.0% sys/t   99.5% idl/t      0.0% iow/t
CPU3:    4.9% usr/t    0.5% sys/t   94.6% idl/t      0.0% iow/t

CPU0:       7 irqs/t          7 soirq RX/t          0 soirq TX/t
CPU1:       8 irqs/t          8 soirq RX/t          0 soirq TX/t
CPU2:       3 irqs/t          3 soirq RX/t          0 soirq TX/t
CPU3:       3 irqs/t          4 soirq RX/t          0 soirq TX/t

CPU0:     158842 irqs
CPU1:     158704 irqs
CPU2:     159393 irqs
CPU3:     158710 irqs
```

The first two lines display received and transmitted MiB, packets, dropped packets, and errors for the network device `eth0` within a given time interval t. The next two lines show their aggregated values since boot time. Moreover, the line starting with `SYS` shows context switches per t seconds, current memory usage, currently running processes and processes waiting for I/O to complete. Furthermore, `ifpps` displays per CPU information about CPU percentage spent executing in user mode, kernel mode, idle mode and I/O wait mode. Next to this, per CPU hardware and software interrupts are shown. Displayed hardware interrupts are only hardware interrupts that were caused by the networking device `eth0`. Software interrupts are not further distinguished by devices, thus only per CPU receive and transmit overall statistics are provided. However, the last lines show aggregated `eth0` hardware interrupts since boot time.

Especially the knowledge gathered by monitoring the system's runtime behaviour regarding hardware and software interrupts, context switches and receive statistics helped in performance optimization experiments with LANA's packet processing engine as described in section 6.5.

`ifpps` is published under the GNU GPL version 2 as an extension to the netsniff-ng toolkit [119] and can be built and installed with:

```
git clone git://repo.or.cz/netsniff-ng.git
cd netsniff-ng/src/ifpps
make && make install
```

Furthermore, `ifpps` supports setting the network adapter into promiscuous mode by applying option `-promisc`, i.e. `ifpps -dev eth0 -promisc`.

D.3 Berkeley Packet Filter Compiler

Within this work, we have developed a Berkeley Packet Filter (BPF) compiler for LANA's BPF functional block. Berkeley Packet Filters as described in literature [31] are small assembler-like programs that can be interpreted by a virtual machine within the Linux (or *BSD) kernel [123]. Their main application is to filter out packets on an early point in the receive path as described in the principle of early demultiplexing in 2.1.1. However, the operating system kernel can only work with an array of translated mnemonics that operate on socket buffer memory. The canonical format of such a single translated instruction is a tuple of Bit fields of `opcode`, `jt`, `jf` and `k` and looks like:

opcode:16	**jt:8**	**jf:8**
k:32		

Here, `opcode` is the numerical representation of an instruction and has a width of 16 Bit, `jt` and `jf` are both jump offset fields (to other instruction tuples) of 8 Bit width each and `k` is an optional instruction-specific argument of 32 Bit. The BPF virtual machine construct consists of a 32 Bit accumulator register, of a 32 Bit index register and of a so-called scratch memory store that can hold up to 16 32 Bit entries. How each component can be used is described later in this section.

Since writing filter programs in this numerical format by hand is error-prone and needs lots of efforts, it is only obvious to have a compiler that translates McCanne et al.'s [31] syntax into the canonical format. To our knowledge, the only available compiler is integrated into `libpcap` [120] and can be accessed by tools like `tcpdump` for instance:

```
# tcpdump -i eth10 -dd arp
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 1, 0x00000806 },
{ 0x06, 0, 0, 0x0000ffff },
{ 0x06, 0, 0, 0x00000000 },
```

This example demonstrates how a simple filter looks like, that can be used for ARP-typed packets. While we believe that tcpdumps filter syntax [124] might be appropriate for most simple cases, it does not provide the full flexibility and expressiveness of the BPF syntax from [31] as stated in discussions on the tcpdump-workers mailing list [125]. To our knowledge, since 1992 there is no such compiler available that is able to translate from McCanne et al.'s syntax [31] into the canonical format. Even though the BPF paper is from 1992, Berkeley Packet Filters are still heavily used nowadays. Up to the latest versions of Linux, the BPF language is still the only available filter language that is supported for sockets in the kernel. In 2011, the Eric Dumazet has developed a BPF Just-In-Time compiler for the

Linux kernel [106]. With this, BPF programs that are attached to PF_PACKET sockets (`packet(7)`) are directly translated into executable x86/x86_64 or PowerPC assembler code in order to speed up the execution of filter programs. A first benchmark by Eric Dumazet showed that 50 ns are saved per filter evaluation on an Intel Xeon DP E5540 with 2.53 GHz clock rate [126]. This seems a small time interval on the first hand, but it can significantly increase processing capabilities on a high packet rate.

In order to close the gap of a BPF compiler, we have developed a compiler named `bpfc` that is able to understand McCanne et al.'s syntax and semantic. The instruction set of [31] contains the following possible instructions:

Instr.	Addressing Mode	Description
	Load Instructions	
`ldb`	`[k]` or `[x + k]`	Load (unsigned) byte into accumulator
`ldh`	`[k]` or `[x + k]`	Load (unsigned) half word into accumulator
`ld`	`#k` or `#len` or `M[k]` or `[k]` or `[x + k]`	Load (unsigned) word into accumulator
`ldx`	`#k` or `#len` or `M[k]` or `4*([k]&0xf)`	Load (unsigned) word into index register
	Store Instructions	
`st`	`M[k]`	Copy accumulator into scratch memory store
`stx`	`M[k]`	Copy index register into scratch memory store
	Branch Instructions	
`jmp`	`L`	Jump to label `L` (in every case)
`jeq`	`#k,Lt,Lf`	Jump to `Lt` if equal (to accu.), else to `Lf`
`jgt`	`#k,Lt,Lf`	Jump to `Lt` if greater than, else to `Lf`
`jge`	`#k,Lt,Lf`	Jump to `Lt` if greater than or equal, else to `Lf`
`jset`	`#k,Lt,Lf`	Jump to `Lt` if bitwise and, else to `Lf`
	ALU Instructions	
`add`	`#k` or `x`	Addition applied to accumulator
`sub`	`#k` or `x`	Subtraction applied to accumulator
`mul`	`#k` or `x`	Multiplication applied to accumulator
`div`	`#k` or `x`	Division applied to accumulator
`and`	`#k` or `x`	Bitwise and applied to accumulator
`or`	`#k` or `x`	Bitwise or applied to accumulator
`lsh`	`#k` or `x`	Left shift applied to accumulator
`rsh`	`#k` or `x`	Right shift applied to accumulator
	Return Instructions	
`ret`	`#k` or `a`	Return
	Miscellaneous Instructions	
`tax`		Copy accumulator to index register
`txa`		Copy index register to accumulator

Next to this, mentioned addressing modes have the following meaning [31]:

Mode	Description
`#k`	Literal value stored in `k`
`#len`	Length of the packet
`x`	Word stored in the index register
`a`	Word stored in the accumulator
`M[k]`	Word at offset `k` in the scratch memory store
`[k]`	Byte, halfword, or word at byte offset `k` in the packet
`[x + k]`	Byte, halfword, or word at the offset `x+k` in the packet
`L`	Offset from the current instruction to `L`
`#k,Lt,Lf`	Offset to `Lt` if the predicate is true, otherwise offset to `Lf`
`4*([k]&0xf)`	Four times the value of the lower four bits of the byte at offset `k` in the packet

Furthermore, the Linux kernel has undocumented BPF filter extensions that can be found in the virtual machine source code [123]. They are implemented as a 'hack' into the instructions `ld`, `ldh` and `ldb`. As negative (or, in unsigned 'very high') address offsets cannot be accessed from a network packet, the following values can then be loaded into the accumulator instead (`bpfc` syntax extension):

Mode	Description
`#proto`	Ethernet protocol
`#type`	Packet class[1], e.g. Broadcast, Multicast, Outgoing, ...
`#ifidx`	Network device index the packet was received on
`#nla`	Load the Netlink attribute of type X (index register)
`#nlan`	Load the nested Netlink attribute of type X (index register)
`#mark`	Generic packet mark, i.e. for netfilter
`#queue`	Queue mapping number for multiqueue devices
`#hatype`	Network device type[2] for ARP protocol hardware identifiers
`#rxhash`	The packet hash computed on reception
`#cpu`	CPU number the packet was received on

A simple example on how BPF works is demonstrated by retrieving the previous example of an ARP filter. This time, it is written in BPF language, that `bpfc` understands:

```
ldh [12]
jeq #0x806,L1,L2
L1:  ret #0xffff
L2:  ret #0
```

Here, the Ethernet type field of the received packet is loaded into the accumulator first. The next instruction compares the accumulator value with the absolute

value `0x806`, which is the Ethernet type for ARP. If both values are equal, then a jump to the next instruction plus the offset in `jt` will occur, otherwise a jump to the next instruction plus the offset in `jf` is performed. Since this syntax hides jump offsets, a label for a better comprehensibility is used instead. Hence, if both values are equal, a jump to `L1`, else a jump to `L2` is done. Instructions on both labels are return instructions, thus the virtual machine will be left. The difference of both lines is the value that is returned. The value states how many bytes of the network packet should be kept for further processing. Therefore a value of `0` drops the packet entirely and a value larger than `0` keeps and eventually truncates the last bytes of the packet. Truncating is only done if the length of the packet is larger than the value that is returned by `ret`. Else, if the returned value is larger than the packet size such as `0xffffff` Byte, then the packet is not being truncated by the kernel.

For the BPF language, we have implemented the `bpfc` compiler in a typical design that can be found in literature [127] or [128]. There, the sequence of characters is first translated into a corresponding sequence of symbols of the vocabulary or also named token of the presented BPF language. Its aim is to recognize tokens such as opcodes, labels, addressing modes, numbers or other constructs for later usage. This early phase is called *lexical analysis* (figure D.3). Afterwards, the sequence of symbols is transformed into a representation that directly mirrors the syntactic structure of the source text and lets this structure easily be recognized [128]. This process is called *syntax parsing* and is done with the help of a grammar for the BPF language, that we have developed. After this phase, the `bpfc` compiler performs *code generation*, which translates the recognized BPF instructions into the numerical representation that is readable by the kernel.

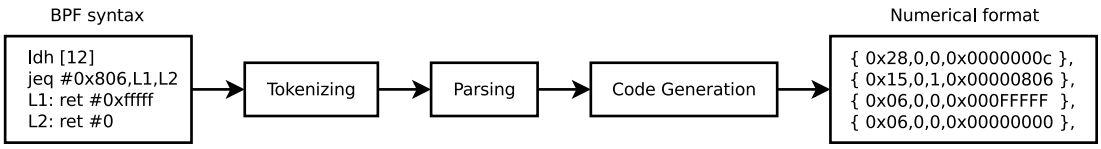

Figure D.3: bpfc phases of code translation

We have implemented `bpfc` in C as an extension for the netsniff-ng toolkit [119]. The phase of lexical analysis has been realized with `flex` [129], which is a fast lexical analyzer. There, all vocabulary of BPF is defined, partly as regular expressions. `flex` will then generate code for a lexical scanner that returns found tokens or aborts on syntax errors. The phase of syntax parsing is realized with GNU `bison` [130], which is a Yacc-like parser generator that cooperates well with `flex`. Bison then converts a context-free grammar for BPF into a deterministic LR parser [131] that is implemented in C code. There, a context-free grammar is a grammar in which every production rule is of nature $M \rightarrow w$, where M is a single nonterminal symbol and w is (i) a terminal, (ii) a nonterminal symbol

or (iii) a combination of terminals and nonterminals. In terms of `flex` and `bison`, terminals represent defined tokens of the BPF language and nonterminals are meta-variables used to describe a certain structure of the language, or how tokens are combined in the syntax. In the code generation phase, `bpfc` replaces the parsed instructions by their numerical representation. This is done in two stages. The first stage is an inline replacement of `opcode` and `k`. Both jump offsets `jt` and `jf` are ignored and left to 0 in the first stage. However, recognized labels are stored in a data structure for a later retrieval. Then, in the second code generation stage, BPF jump offsets for `jt` and `jf` are calculated.

`bpfc` also has an optional code validation check. Thus, after code generation, basic checks are performed on the generated instructions. This includes checking of jump offsets, so that jumps to non-existent instructions are prevented, for instance.

The source code of `bpfc` is released under the GNU GPL version 2, it can be fetched via Git and built with the following commands:

```
git clone git://repo.or.cz/netsniff-ng.git
cd netsniff-ng/src/bpfc
make && make install
```

In order to use `bpfc`, the ARP example can be copied into a text file that is handed over as an command-line argument:

```
bpfc arp.bpf
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 1, 0x00000806 },
{ 0x6, 0, 0, 0x000fffff },
{ 0x6, 0, 0, 0x00000000 },
```

Here, `bpfc` directly outputs the generated code and internally performs code validation in non-verbose mode. For debugging purposes, `bpfc` can also be used verbosely:

```
bpfc -Vi arp.bpf
*** Generated program:
(000) ldh [12]
(001) jeq #0x806 jt 2 jf 3
(002) ret #1048575
(003) ret #0
*** Validating:  is valid!
*** Result:
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 1, 0x00000806 },
```

```
{ 0x6, 0, 0, 0x000fffff },
{ 0x6, 0, 0, 0x00000000 },
```

D.4 Linux Kernel One-time Stacktrace Module

The Linux one-time stack trace (otst) module that we have developed within this work can be used to generate kernel stack traces during runtime for debugging purposes. A stack trace can be seen as a runtime summary of active stack frames. With the help of a short fictitious example, the use of stack traces is demonstrated. Consider the following pseudo code:

```
void bar(void)
{
}

void bart(void)
{
        printf("42\n");
}
```

```
void foo(void)
{
        bar();
        bart();
}

int main(void)
{
        foo();
        return 0;
}
```

If we now would like to see all active stack frames during runtime from within the function `bart`, we trigger a stack trace just before the `printf` call. The content of the stack during runtime would look like [] → [`main`] → [`main`, `foo`] → [`main`, `foo`, `bar`] → [`main`, `foo`] → [`main`, `foo`, `bart`] → `stack trace triggered`. Thus, we are able to see, that `bart` was called by `foo` which was called by `main`.

This example is quite clear regarding calling context. However, the function call context in the Linux kernel is not that obvious. Since none of the current literature describes the Linux 3.0 networking subsystem clearly, we have developed a Linux kernel module that can generate stack traces for arbitrary kernel symbols. We have used this and in addition a cross reference of the latest Linux Git repositories that we have set up for this work at `http://lingrok.org/` in order to examine the Linux network stack, which is described in section 3.1.

The runtime stack traces that can be generated by our kernel module apply a technique that is called Kprobes [132], [133]. The idea of Kprobes is to enable dynamically breaking into any kernel routine non-disruptively [134]. There, almost any kernel code address can be trapped with a given handler for the breakpoint. Three different types can be applied in Kprobes: `kprobes`, `jprobes` and `kretprobes`. `kprobes` can be inserted on any instruction, `jprobes` is inserted at the entry of a kernel function and further provides access to function arguments and last but not least `kretprobes` is inserted on the return of functions [134].

Our kernel module makes use of `kprobes`, so that only the concept of `kprobes` is further described in this section with reference to [134].

When a `kprobes` is registered, Kprobes makes a copy of the probed instruction and replaces the first bytes of the instruction with a breakpoint instruction such as `int3` on architectures like x86_64. If the CPU invokes the probed instruction, it hits a breakpoint instead. By doing a trap, CPU registers are saved and the control is passed to Kprobes. Kprobes first executes user-registered handlers as `pre_handler` with the help of Linux notification chains [135]. After all registered handlers have been invoked, Kprobes single-steps its copy of the probed instruction and executes user-registered handlers as `post_handler`s. Hereafter, the kernel resumes normal execution.

In our Linux one-time stack trace module, the stack trace is thrown as a `pre_handler` and the `kprobes` is disabled within the `post_handler`, since it should only be invoked on the first call of the probed instruction. To have an easy usage, we have implemented an interface to procfs. Thus, arbitrary kernel symbols can be written to a procfs driver file that triggers the registration of a `kprobes` for the given kernel symbol.

Therefore, a symbol can be written to the procfs-file `/proc/driver/otst` like:

```
echo -n "netif_rx" > /proc/driver/otst
```

Hence, on the next call to the function `netif_rx`, a stack trace is being generated *once*, thus one is able to see the calling context of the specified function. More than one function can be traced at a time. All currently probed functions can be shown with:

```
cat /proc/driver/otst
```

Our source code of `otst` is licensed under the GNU GPL version 2 and can be obtained via Git with:

```
git clone git://repo.or.cz/otst.git
cd otst/
make
insmod otst.ko
```

An example output for the symbol `netif_rx` looks like:

```
[   83.267411] otst: one-time stacktrace driver loaded!
[   95.591140] otst: symbol netif_rx registered!
[  121.770022] otst: triggered stacktrace for symbol netif_rx at
                     0xffffffff814c0050:
[  121.770027] Pid: 0, comm: kworker/0:1 Not tainted 3.0.0-rc1+ #3
[  121.770030] Call Trace:
```

```
[  121.770033] <#DB> [<ffffffffa002838b>] otst_handler+0x2b/0x30 [otst]
[  121.770047] [<ffffffff815ba4a7>] aggr_pre_handler+0x57/0xb0
[  121.770053] [<ffffffff814c0050>] ? net_rx_action+0x2e0/0x2e0
[  121.770057] [<ffffffff815b960c>] kprobe_exceptions_notify+0x3fc/0x460
[  121.770062] [<ffffffff815ba166>] notifier_call_chain+0x56/0x80
[  121.770067] [<ffffffff815ba1ca>] atomic_notifier_call_chain+0x1a/0x20
[  121.770071] [<ffffffff815ba1fe>] notify_die+0x2e/0x30
[  121.770075] [<ffffffff815b7263>] do_int3+0x63/0xd0
[  121.770079] [<ffffffff815b6a88>] int3+0x28/0x40
[  121.770083] [<ffffffff814c0051>] ? netif_rx+0x1/0x190
[  121.770086] <<EOE>> <IRQ> [<ffffffff814c0352>] ? netif_rx_ni+0x12/0x30
[  121.770094] [<ffffffff814f69b9>] ip_dev_loopback_xmit+0x79/0xa0
[  121.770098] [<ffffffff814f7930>] ip_mc_output+0x250/0x260
[  121.770102] [<ffffffff814b1353>] ? __alloc_skb+0x83/0x170
[  121.770106] [<ffffffff814f6a09>] ip_local_out+0x29/0x30
[  121.770111] [<ffffffff81528aeb>] igmp_send_report+0x1db/0x210
[  121.770117] [<ffffffff81087d78>] ? sched_clock_cpu+0xb8/0x110
[  121.770121] [<ffffffff81529380>] igmp_timer_expire+0x100/0x130
[  121.770125] [<ffffffff8104e2e2>] ? scheduler_tick+0x132/0x2b0
[  121.770130] [<ffffffff8106e8aa>] run_timer_softirq+0x16a/0x390
[  121.770134] [<ffffffff81529280>] ? ip_mc_destroy_dev+0x80/0x80
[  121.770139] [<ffffffff8102830d>] ? lapic_next_event+0x1d/0x30
[  121.770144] [<ffffffff8106577f>] __do_softirq+0xbf/0x200
[  121.770148] [<ffffffff81085967>] ? hrtimer_interrupt+0x127/0x210
[  121.770153] [<ffffffff815bf51c>] call_softirq+0x1c/0x30
[  121.770157] [<ffffffff8100d2e5>] do_softirq+0x65/0xa0
[  121.770161] [<ffffffff81065595>] irq_exit+0xb5/0xc0
[  121.770165] [<ffffffff815bfe5e>] smp_apic_timer_interrupt+0x6e/0x99
[  121.770170] [<ffffffff815becd3>] apic_timer_interrupt+0x13/0x20
[  121.770172] <EOI> [<ffffffff81013e1d>] ? mwait_idle+0xad/0x1c0
[  121.770180] [<ffffffff815ba1ca>] ? atomic_notifier_call_chain+0x1a/0x20
[  121.770185] [<ffffffff8100b0b7>] cpu_idle+0xb7/0x110
[  121.770190] [<ffffffff815ae4a1>] start_secondary+0x1c0/0x1c7
[  174.500038] otst: symbol netif_rx unregistered!
```

Appendix E

Time Schedule

Working period: March 2011 - December 2011 (42 Weeks, 10 Months)

Schedule (note that parts can be overlapping):

1. Introduction, Analysis and Design (distributed between March 2011 - June 2011)
    - Introductory Reading: 2 Weeks
    - Setup of Work Environment: $<$ 1 Week
    - ANA Compilation and Test Run: $<$ 1 Week
    - Development and Test of own Brick: $<$ 1 Week
    - Determine Basic Functionality: 3 Weeks
    - Architecture Design: 2 Months
2. Implementation (distributed between April 2011 - November 2011)
    - Core Framework Implementation and Optimization: 6 Weeks
    - User space Configuration Utilities: 1 Week
    - Example Application: 2 Weeks
    - Functional Block Development: 4 Weeks
    - Traffic Generator Development: 2 Weeks
    - Statistic Monitoring Tool: 2 Weeks
    - Berkeley Packet Filter Compiler: 1.5 Weeks
    - Stack trace Module: $<$ 1 Week
    - Testing Debugging of Software: 3 Months
3. Documentation (distributed between March 2011 - December 2011)
    - Literature Study: 4 Months
    - Performance Evaluation: 4 Weeks
    - Final Report: 5 Months (August 2011 - December 2011)
    - 3 Intermediate (2 times at ETH Zurich, one at HTWK Leipzig) and 2 Final Presentations (one at ETH Zurich, one at HTWK Leipzig): 5 Weeks

Appendix F

Content of the Attached CD

The following content can be found on the attached CD:

- The **final report** of this master's thesis in Portable Document Format and as LaTeX source code
- The **publication** and **poster** from the ACM/IEEE Symposium on Architectures for Networking and Communications Systems 2011 in Portable Document Format and as LaTeX source code
- All of the LANA **source code** for Linux
- **Source code** of the netsniff-ng toolkit for Linux that include tools from D

Appendix G

Declaration of Originality / Eidesstattliche Versicherung

Ich erkläre hiermit, dass ich diese an der HTWK Leipzig verfasste Masterarbeit selbstständig ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht.

Zurich, December 21st, 2011

_____