An Extended Berkeley Packet Filter Approach for Real-Time Applications in Edge Computing Environments

Bachelor's Thesis

Aurel Isaak Weinhold
403779

3. Juli 2023

Technische Universität Berlin
Faculty IV – Electrical Engineering and Computer Science
Department of Telecommunication Systems
Distributed and Operating Systems

First Reviewer:	Prof. Dr. habil. Odej Kao
Second Reviewer:	Prof. Dr. rer. nat. Volker Markl

Eidesstattliche Erklärung / Statutory Declaration

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Berlin, 3rd July, 2023

Aurel Weinhold

Zusammenfassung

Mit dem fortschreitenden Prozess der Digitalisierung der Welt wird der Nutzen der Digitalisierung von Alltagsgegenständen deutlich erkennbar. Dadurch wird das Internet der Dinge (IoT) zu einem integralen Bestandteil sowohl des täglichen Lebens als auch der Arbeit. Die Interaktion von Technologie in der Nähe von Menschen und anderen Maschinen in der realen Welt führt zur Einführung des Konzepts der Echtzeit-Systeme, bei denen bestimmte Aufgaben innerhalb vorgegebener Fristen erfüllt werden müssen, um sowohl nützlich als auch sicher zu sein. Oft finden sich solche Echtzeit-Systeme in Edge-Computing-Umgebungen, in denen die Edge-Knoten häufig nicht optimal für den Echtzeitbetrieb ausgelegt sind.

Die aktuelle Forschung konzentriert sich hauptsächlich auf die Netzwerkkomponente. Dies erfolgt erstens durch die Einführung neuer Technologien wie dem 5G-Netzwerk und die Bewertung von Protokollen für den Echtzeitbetrieb. Zweitens wird versucht, die Last auf die verfügbaren Edge-Knoten zu verteilen. Zuletzt wird untersucht, wie Berechnungen näher an den Datenquellen und Akteuren, in den Netzwerkkomponenten durchgeführt werden können. Dabei wird jedoch häufig das Potenzial der Edge-Knoten selbst übersehen. Diese Arbeit stellt eine Anwendung des Extended Berkeley Packet Filters (eBPF) vor, um die Antwortzeit am Rand des Netzwerkes zu verkürzen und dessen Varianz zu reduzieren, indem ein Teil der Operationen in den Kernel verschoben wird.

Der Ansatz wird in einem beispielhaften Szenario im Bereich Intelligente Transportations Management Systeme (ITMS) untersucht. Es werden zwei Konfigurationen getestet: eine, bei der nur Antworten im Kernel verarbeitet werden, und eine, bei der der Inhalt der Antworten ebenfalls im Kernel berechnet wird, wodurch höherer Aufwand pro Anfrage entsteht. Diese werden verglichen mit einer herkömmlichen Server-Anwendung im User Space. Die Experimente zeigen eine Verbesserung der Antwortzeit um den Faktor 1,76 bzw. 1,43. Darüber hinaus kann der Server mehr Clients gleichzeitig bedienen, und zwar um den Faktor 1,93 bzw. 1,95.

Die Ergebnisse verdeutlichen, dass eBPF in der Lage ist, Echtzeit-Systeme erheblich zu verbessern. Die vorliegende Arbeit liefert somit eine Grundlage für zukünftige Forschung und Anwendungen.

Abstract

With the ongoing digitization of the world, the utility of digitizing everyday objects becomes apparent. This makes Internet of Things (IoT) an integral part of everyday life and work. Having technology interact in the vicinity of people and other machines in the real world introduces the idea of Real-Time (RT) systems, where tasks have to meet specific deadlines in order to be valuable and safe. Oftentimes RT systems are found in Edge Computing environments, where the edge nodes might not be optimized for RT.

Current research mainly focuses on the network component. Firstly, by introducing new technologies, such as 5th Generation (5G) networking, and evaluating protocols for RT operation. Secondly, to balance load over the available edge nodes. Finally, to move the computation into the network components for closer operation to the data sources and actors. Often overlooked however is the potential on the edge nodes themselves. This thesis introduces an application of Extended Berkeley Packet Filters (eBPF) on the edge to reduce response time and decrease its variance by moving part of the operation into the kernel.

The approach was tested in an exemplary scenario in the domain of Intelligent Traffic Management Systems (ITMS). Two configurations, one where only responses where handled in-kernel, and one where the content of the response was also calculated in the kernel, where compared to a base-line of a regular user-space server application. The experiments yielded an improvement in response time by a factor of about 1.76 and 1.43 respectively. Additionally, the server could handle more clients simultaneously by a factor of 1.93 and 1.95.

These results show that eBPF is capable of enhancing RT systems significantly and the thesis gives a foundation for future research and application.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the ongoing digitization of the world, the utility of digitizing everyday objects becomes apparent. The integration of a digital representation and networking infrastructure enables a multitude of functionalities, ranging from simple state logging for debugging purposes to more complex abilities such as decision-making based on these states. This process of digitizing and networking objects is commonly referred to as the Internet of Things (IoT) [1]. Given the projected growth of the IoT market to an estimated 30 billion devices by 2030 [2], it is clear that improvements have a huge potential benefit. IoT devices span from household appliances like dishwashers and vacuums, over infrastructure such as traffic lights, to more industrial-grade hardware like sensors deployed in factories.

Some of these IoT devices exhibit Real-Time (RT) behavior to ensure safety and Quality of Service (QoS). RT operation is defined by the German Institute for Standardization [3] to be the operation of a computer system with programs that are at any time ready to process data in a way that the computational results are available within a given period of time. This is necessary as these actors interact with the real world, being in physical contact with each other and possibly even humans [4]. To mitigate risk for humans and to ensure QoS in the product it is strictly necessary to be predictable, as a violation might cost millions of Euros and potentially the lives of humans [5].

With the exponential growth in data generated by IoT devices, the increase in demand for RT computing and unpredictable latencies in unknown networks [6], a shift away in computing from the cloud towards the IoT networks has been made. This Edge Computing, a new paradigm in Distributed Systems, has introduced the idea to pre-compute or compute "on the edge" of the cloud [7], which significantly reduces response times by avoiding the network latencies to and from the cloud, while reducing stress on these networks and the computing resources of the cloud.

1.1 Problem Statement

Even with the computing nodes closer, actors depending on RT still can not rely on unpredictable networks. It is however possible to get closer to RT computation by using Edge Computing and new technologies.

Current research is mostly focusing on enhancing the network component using novel 5G technologies [8] and finding and evaluating new protocols [9], but research on improving response-time and dependability on the edge node itself is scarce.

There still however is potential on the edge devices, for example in the context of smart traffic systems. Frequently recurring requests, with responses only depending on application-internal states or infrequently changing variables, i.e. a tempo limit which changes depending on the number of cars, currently require redundant overhead, such as context switches. This does not only cost valuable computing power on the edge node, but also unnecessarily delays the response, meanwhile making the necessary computations in a less predictable way depending on the user-space scheduler.

This thesis considers this application and evaluates it in a real-world scenario, set in a Intelligent Traffic Management System (ITMS).

1.2 Thesis outline

At the outset, chapter 2 provides the background and motivation for the research. This is followed by chapter 3 on related work, which includes a literature review and a comparison with existing solutions to the problem at hand. Chapter 4 focuses on the design and implementation of the proposed solution, providing an overview of the architecture, design and implementation details of the different components of the system. This chapter also discusses the challenges faced during the design and implementation process and the solutions adopted to overcome them.

Chapter 5 describes the experiments to evaluate the systems feasibility in the context of RT on the edge. It includes a detailed experiment plan, cases, and performance metrics used to measure the systems performance and predictability. The data obtained in the experiments is also analyzed and evaluated in this chapter.

The thesis concludes in chapter 6 with a summary of the contributions of the research and the thesis' key points, and possible future work.

Chapter 2

Background

This chapter introduces the relevant concepts touched on in this thesis. It starts by introducing the concept of Real-Time in section 2.1. Section 2.2 gives an introduction to the Edge Computing paradigm and show its importance for RT computation in IoT networks of all kinds. The Linux kernel is introduced in section 2.3, followed by the key technology, extended Berkeley Packet Filer (eBPF) in section 2.4.

2.1 Real-Time

Computational systems interacting with the physical world are usually referred to as Cyber-Physical Systems (CPS) [4]. They are subject to more rigorous constraints when executing processes. These systems must comply with safety restrictions when interacting with humans, or in manufacturing processes, where they must adhere to precise timing requirements [4].

Real-Time [10] refers to the property of a process being capable of completing at any point within a specific time-frame, regardless of when it is initiated. RT processes are subject to a deadline, which can be categorized as either a hard or soft deadline. The value of a RT process' result diminishes when it misses a soft deadline. Conversely, when a process misses a hard deadline, the value of its result becomes null.

The primary objective of RT systems is to ensure that a maximum number of processes meet their respective deadlines. Achieving this objective typically involves performing statistical analysis on each process and the system as a whole, taking into account factors such as the expected runtime and measures of scale like the standard deviation.

Often times a part of a RT system is a networked component. In that case it is

necessary to consider the transportation protocol used. The two basic protocols, most network traffic on the internet uses today are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) [11].

TCP [12] is a reliable and connection-oriented protocol offering ordered and error-checked transmission of data potentially broken down into small packets transferring for transmission. UDP [13] is connectionless and unreliable network protocol. As there is no connection to be established, UDP has less overhead and is generally faster, however it exposes the user to any unreliability of the underlying network. Depending on the application, both can be a valid choice.

2.2 Edge Computing Paradigm

As the amount of data generated increases, the need for computing resources increases proportionally. As processors don't have the capabilities to process terabytes of data per second Cloud Computing (CC) [14] in conjunction with distributed systems emerged and providers such as Amazon Web Services, Microsoft Azure and Google Cloud Platform quickly took over the market.

This approach led to a rapid increase in network bandwidth needed and while the CC approach from a computing resource standpoint seems to be infinitely scalable, the networks, which are mostly not virtualized, simply are not [7].

In RT systems, as they are often found in IoT networks, a congested network is not feasible as dependability is key and thus a movement back from the cloud towards the on-site network was needed. This new paradigm in distributed systems was named Edge Computing [7].

Edge Computing allows computation, such as aggregation of data coming from the IoT network and control of the actors in the network, in the vicinity, the so called edge, of the IoT network, while allowing computationally expensive tasks to be offloaded to the cloud. This is done by placing less powerful servers, compared to the cloud, near the IoT, as shown in fig. 2.1.

2.3 Linux Kernel

Linux is the kernel of a family of operating systems often referred to as Gnu/Linux. It is a monolithic, modular, multitasking Unix-like kernel authored by Linus Torvalds. Gnu/Linux is by far the most used server operating system (OS) and as

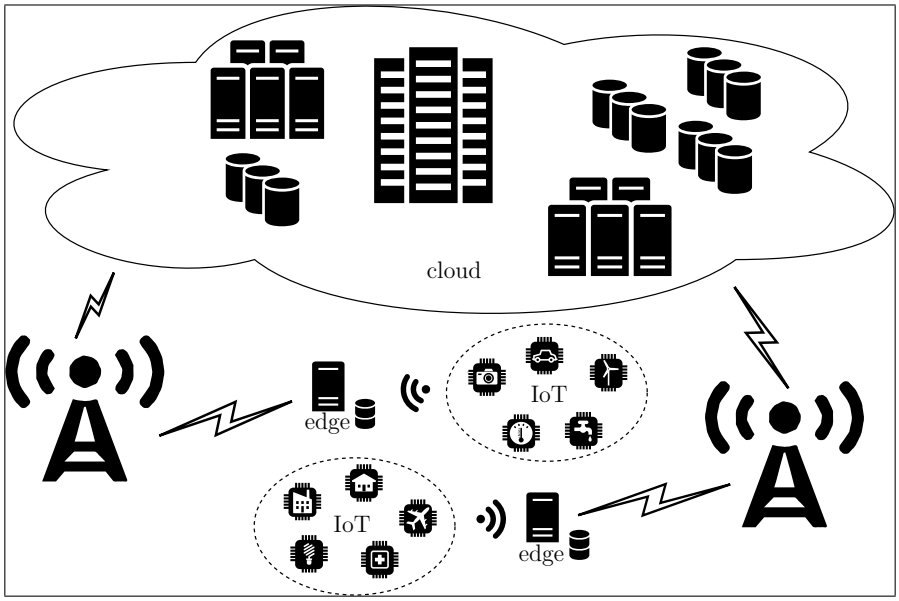


Figure 2.1: An overview of the IoT, edge, and cloud environment. The edge nodes are closer to the IoT network, while the cloud is further away. The radio towers indicate the connection to the extended internet through which edge devices and cloud are connected, while smaller mobile connections connect the IoT devices directly to the edge nodes. Additionally, the cloud resources are far more vast visually depicted by the quantity and size of servers and databases.

its kernel, Linux is responsible for interacting with the servers hardware, virtualized or not, such as the Network Interface Controller (NIC) [15].

Adhering to the TCP/IP stack, the Linux kernel implements protocols up to the transport layer. Application layer protocols, such as HTTP and SSH, are implemented in user-space abstracting the complicated networking stack using sockets. An IP socket is associated with an IP address, the transport layer protocol used, usually TCP or UDP, and a port. Transmission and reception operations are transparent to the user-space application using the socket[1].

It is the operating systems responsibility to get the incoming packets from the NIC to the corresponding socket. To achieve that incoming packets go through the stages of the Linux network stack. The stack, starting at the bottom, consists of the physical hardware, device drivers, the device agnostic interface, the network protocol, the protocol agnostic interface, the system call interface and finally the application layer. When an application makes a networking request it has to go

[1] `https://linux-kernel-labs.github.io/refs/heads/master/labs/networking.html`

from the application layer all the way down to the physical hardware at which point the packet exists on the physical networking medium.

The stages of the network stack are grouped into three: the user-space, the kernel-space and the physical layer, The top layer, the application layer is part of the user-space, the next five layers, from system call interface to the device drivers, are in the kernel-space and the physical hardware is in the physical layer.

2.4 Extended Berkeley Packet Filter

Extended Berkeley Packet Filter (eBPF) [16] refers to a versatile virtual machine that exists within the Linux kernel that can be programmed during runtime from user-space. It enables the safe execution of relatively straightforward, albeit limited, programs in the kernel-space at various hook points.

It was initially developed as an extension of the Berkeley Packet Filter, which is now usually referred to as classic Berkeley Packet Filter (cBPF). cBPF programs were limited to network applications, hooking into the eXpress Data Path (XDP), or the Traffic Control (TC) stage of the various packet processing stages in kernel-space to allow inspection and filtering of packets early on in the network pipeline. Although limited it opened to door into advanced network level monitoring on the kernel level.

Only with its extension to eBPF in recent years, researchers have tried to use it for more advanced network related tasks [17], beyond filtering, tracing, monitoring and profiling, and even started to use it in completely different applications hooking into non-network related systemcalls.

eBPF programs are usually written in a high level language similar to C and later compiled to eBPF byte-code by the clang compiler. During loading of the eBPF program they are statically verified by the in-kernel verifier[2]. This verifier checks the program against a set of restrictions, such as a limitation on the number of instructions and that all loops are bounded to ensure the safety of the kernel.

These eBPF programs are then attached to a hook. The kernel provides hook points at the ingress, where packets enter, and egress points, where packets leave the kernel, of the network interfaces, the various stages of packet processing in the kernel, such as routing and protocol handling, and the sockets API[3].

[2] `https://www.kernel.org/doc/html/latest/bpf/verifier.html`, accessed 18th April, 2023

[3] `https://devenes.medium.com/ebpf-a-powerful-technology-for-linux-kernel-networking-f615a8272d3c`, accessed 1st April, 2023

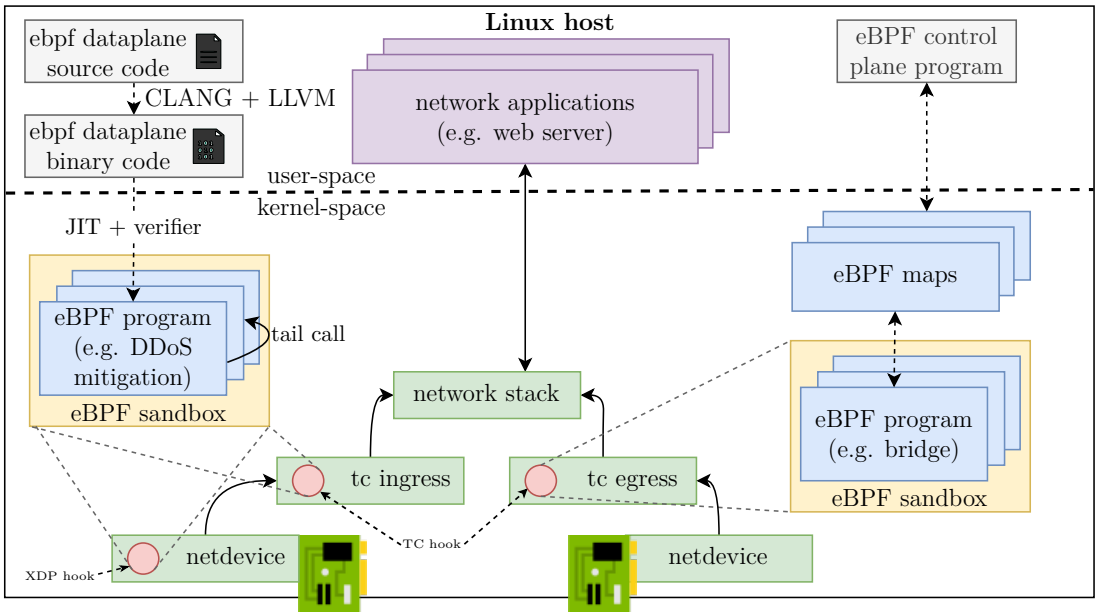


Figure 2.2: An overview of eBPF in the Linux kernel. It shows how eBPF programs are loaded, run in a sandboxed environment, the way eBPF programs communicate through maps with a control program and a brief overview of the path a packet takes through the netdevice, TC, network stack to the user-space network application. Adapted from [17][4].

It is possible to establish communication channels between multiple eBPF programs and even user-space applications. This is done through the use of so called eBPF maps. These maps provide a shared data structure stored in the kernels memory space, offering a flexible way to pass data between programs. As this memory lies in kernel-space, the eBPF programs can access the data directly, whereas a user-space program has indirect access through the use of the BPF systemcall[5]. Although hidden behind a file descriptor, access is not direct, as the values are copied over to user-space on lookup. Implemented data structures include an array, a hash map, a queue and a stack, with some including a per CPU implementation as well.

Some of these hook points as well as the maps are displayed in fig. 2.2. It also shows how the different components work together. The Linux network stack is shown and how the data traverses from the NIC and its driver (here the netdevice) over the TC into the network stack. It shows how the eBPF programs are run in

[4]The figure has been altered slightly. The memory of the eBPF maps is located in the kernel-space and so they have been moved below the dividing line.

[5]`https://man7.org/linux/man-pages/man2/bpf.2.html`, accessed 25th June, 2023

a sandboxed environment, the virtual machine, communicating with a user-space control program through maps. Finally, it shows that the eBPF code is compiled into binary code using Clang and the LLVM framework and then loaded into kernel-space.

As eBPF provides a way to change the behavior of the kernel without the need to recompile or change it, it is a gateway to many powerful abilities. With its many hook points, it allows a wide range of applications, and for custom logic to be executed at almost any imaginable point in the kernel. In networking applications it provides a way to handle packets before much processing is done, and without the need to switch contexts. Currently, eBPF is used to harden the security of the Linux system [18][19], for (Distributed) Denial of Service (DDoS) attack mitigation [20][21], to create complex network services [17] and functions [22][23] such as load balancing [24][25], to offload to hardware such as NICs [26] and more.

Chapter 3

Related Work

This chapter reviews related academic research on Edge Computing and its importance in IoT networks and RT in systems with a networked component using novel technologies, such as 5G networking, and eBPF.

3.1 Load Balancing

The distributed approach of Edge Computing offers numerous benefits for RT, low-latency applications, including reduced latency, improved scalability, enhanced privacy, and bandwidth optimization. However edge devices, such as smartphones, sensors, IoT gateways, and cars, have limited computational power, energy, storage capacity, and network bandwidth. Moreover, they operate in highly dynamic and heterogeneous environments, where resources may vary significantly in terms of availability, proximity, and capabilities. This makes it necessary to allocate the resources of these edge devices intelligently. Load balancing, one of the approaches taken, involves distributing the work load intelligently across available nodes.

Zhao et al. [27] determine, the goal of efficient resource allocation is to minimize average service response time, and focus on improving the service process delay. They introduce two algorithms, numeration-Based Optimal Edge Resource Allocation Algorithm (EOERA) and Clustering-Based Heuristic Edge Resource Allocation Algorithm (CHERA), but determine that based on computational complexity, CHERA is the better solution to the Average System Response Time Minimization (SRTM) problem. CHERA first finds the resource allocation case for each application heuristically based on the current average service response time. Secondly it initializes clusters of candidate edge servers for each application based on the result of the heuristic procedure. Third, it replaces the current edge server that allocates resources for an application in each cluster heuristically. Their tests show that both their algorithms improve the average response time compared to established

algorithms.

Their approach shows that there is still room for improvement on the algorithms used on the load-distribution level. Efficient load-balancing is essential to maximize the utilization of limited resources. However, it has no influence on the utilization of the resources on the edge node with the given tasks, which this thesis focus is placed on.

Additionally, load balancing introduces the problem that resource allocation consumes itself time as well and recent advances in eBPF have brought interest in using this rather new technology to improve load balancing.

Kogias, Iyer, and Bugnion [24] propose a Connection Redirection loAd Balancer (CRAB), which offers an alternative Layer 4 (L4) load balancing scheme that addresses latency overheads and scalability bottlenecks while enabling the deployment of complex and stateful load balancing policies. CRAB load balancers operate differently by only participating in the TCP connection establishment phase and remaining off the connections datapath. This requires a new approach, for load balancers to only depend on the data rate, instead of, traditionally, the connection bandwidth. However it enables the use of technologies such as eBPF. Their evaluation demonstrates that CRAB effectively shifts the IO bottleneck from the load balancer to the servers in scenarios where traditional L4 load balancing struggles to scale and, more importantly, it achieves end-to-end latencies comparable to direct communication while retaining the scheduling benefits associated with stateful L4 load balancing.

Lee et al. [25] follow a similar approach in implementing a containerized load balancer in eBPF for the XDP within the Linux kernel. They compared the results of their experiments to implementations using Netfliter[1] modules and the loopback interface, using the RFC 2544 [28] benchmarking standard and simulations of real-world traffic patterns. Their results demonstrated that the proposed load balancer outperformed Netfilter approaches in terms of throughput, with the performance difference increasing as packet size decreased. The divergence in performance of the proposed load balancer from the performance of the loopback device, which represents the theoretical maximum performance limit, was also minimal.

Although both these solutions implement load balancers for cloud computing environments, they showcase the feasibility of achieving load balancing with minimal resource requirements. Consequently, these findings indicate the potential applicability of load balancing techniques in Edge Computing environments as well.

[1] `https://www.netfilter.org/`, accessed 28th June, 2023

The three paper demonstrate the possibilities of utilizing eBPF and the feasibility of Edge Computing in RT, low-latency applications at the network-infrastructure level. There seems to be a gap in research towards improving near-RT performance on the Edge Computing devices themselves. This is where this thesis focus is directed towards.

3.2 In-Network Computing

Virtualized network functions have the advantage of being unified and open [29] as opposed to vendor specific hardware, which makes it more flexible [17]. While this is already done even on general-purpose Linux machines, the approaches taken usually bypass the kernel, using frameworks like the Data Plane Development Kit (DPDK)[2].

Miano et al. [22] introduce an in-kernel approach using eBPF. This proves several advantages, such as the avoidance of constant polling, which needed to be done in traditional DPDK implementations and is unnecessary with the eBPF approach which is only executed when a new packet arrives.

This paper shows the effectiveness of using eBPF in networking tasks. However it focuses on the network functions themselves that are part of the network used by the application introduced in this thesis which aims to show the more general-purpose use-case of eBPF.

This general-purpose computing is also already being investigated, specifically computing on the network devices. Rüth et al. [30] demonstrate the feasibility of using eBPF as an in-network controller for a small CPS consisting of an inverted pendulum. The controller tries to stabilize the pendulum. As the eBPF approach is placed in the network closer to the data source, being executed on the network switch between controller and CPS, it has the ability to stabilize the pendulum, while the traditional approach is too slow. However the authors are leveraging the design of the two-phase algorithm to move the computationally intensive task to an offline phase before the system is even used. The parameters gained are then used to calculate the result quickly and online. This is not what this thesis considers, being even more general and utilizing the resources of a full server.

[2] `https://www.dpdk.org/`, accessed 28th June, 2023

Chapter 4

Approach

This chapter introduces an eBPF application to achieve near RT behavior by allowing process prioritization of specific requests by avoiding the user-space and therefore context switches and less predictable scheduling.

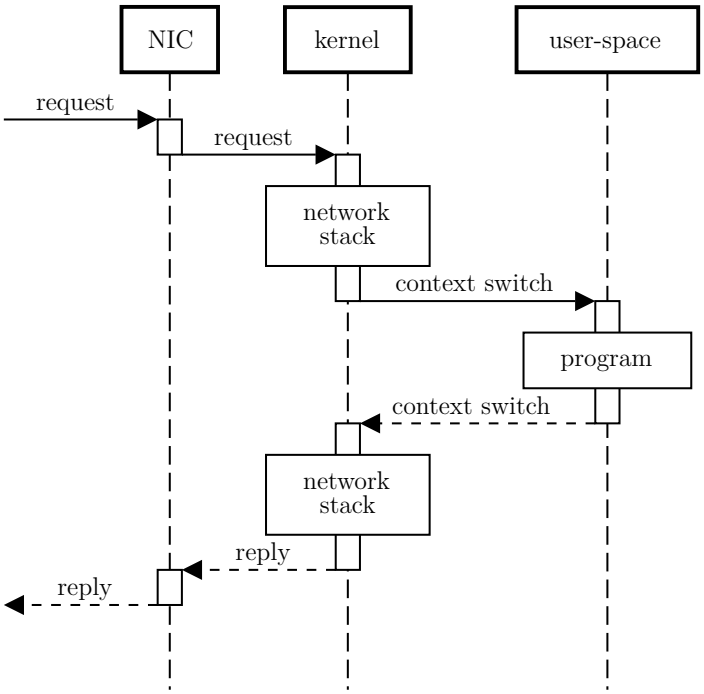
First it presents the way a regular server application runs in the Linux environment and compares that to how the server application runs using the proposed eBPF program generally. It continues by showing how the eBPF application is then integrated with this server application. Lastly it reveals some limitations this approach brings.

4.1 System Design

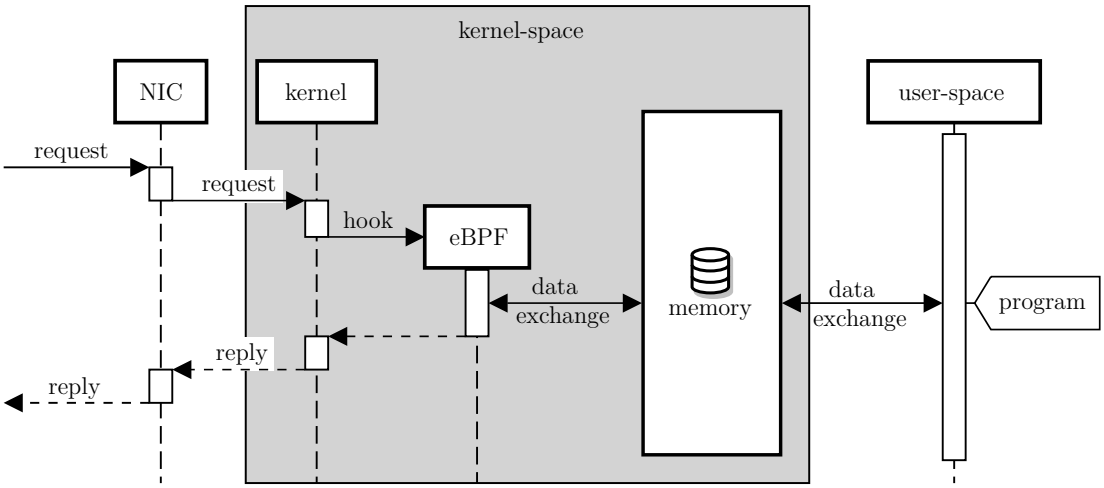
The Edge Computing environment is characterized by a distributed infrastructure consisting of multiple edge nodes positioned on the edge of IoT networks. High-speed communication links connect the IoT devices and the edge node that usually is closest in proximity. This thesis considers a single edge node and its corresponding IoT devices and focuses on enhancing it.

It thus finds its application in a situation as follows. An edge node offers a service, consisting of at least state access. IoT devices request this state from the edge node, which enables them to fulfill their tasks. This is a generally applicable client-server communication model with the edge device therefore called the server, and the IoT devices the clients. As this is set in a RT application, the goal is to not only decrease the response time and increase the amount of simultaneous connections, but also to be more predictable in execution time, i.e. to have less variance in the response time.

In this application, the communication between the server and the clients is using the UDP transport layer protocol, as it is faster and simpler than TCP and the



(a) The communication in the regular Linux environment. Packets have to go through the entire Linux network stack to be received in the user-space program.



(b) The communication in the Linux environment using eBPF. Packets are caught by the eBPF program at the hook point before going through the entire Linux network stack and responded from there.

Figure 4.1: Packet processing in the regular Linux environment, seen in fig. 4.1a, versus in an application utilizing eBPF, as seen in fig. 4.1b.

benefits TCP offers are not needed. However the approach is agnostic to the transportation protocol and thus similar enhancements can be achieved using TCP, with little change to the code.

Implementing a server application in the Linux environment usually follows the following steps: First create a socket for an IP-address family, IPv4 or IPv6, and a transport protocol, such as TCP or UDP, and bind it to a port. Then start to listen for incoming packets. When a new packet is incoming, the user-space can receive the content on the socket and start to process it. Behind the scenes however, this packet has already gone through the entire Linux kernel stack as described in section 2.3. This is shown in fig. 4.1a.

The eBPF program, after being loaded into the kernel by the user-space program starts to process incoming requests. A part of the original user space program is now being masked by this eBPF program. This means some functionality, previously handled by the original server user-space application, is now, transparent to the user-space program, being handled in the eBPF program, in kernel-space. Consequently, as shown in fig. 4.1b, the flow of the packet ends in kernel space, before going through the entire stack and to the user-space program, and is processed and replied to there.

For this application the XDP is chosen. As it is one of the first stages in the in-kernel packet processing stack described earlier, it is ensured that there is as little processing done beforehand as possible. Still, in this stage it is possible to access the entire packet, including all its headers, such as the Ethernet header.

For a packet to be processed in kernel-space alone, it needs to have access to all the data necessary. As described in section 2.4, this is done using maps. Depending on the application the underlying data structure can be chosen from a list of implemented structures, allowing for a maximum of flexibility in application.

As seen in fig. 4.1b, the memory where these maps are located, lies in kernel-space. This means that the eBPF program has immediate access to it, while the user-space interacts with it through system calls, fetching or overwriting single elements from the maps through the use of the BPF systemcall.

4.1.1 eBPF Application

With any incoming network packet, the eBPF program[1] is executed. This means the first challenge for it is to determine whether it should process the specific packet.

[1] found in `https://github.com/AurelWeinhold/bachelors_thesis/blob/8f129da29ebbcfd8f50d3cc047188741ae556b2f/server/thesis.bpf.c`

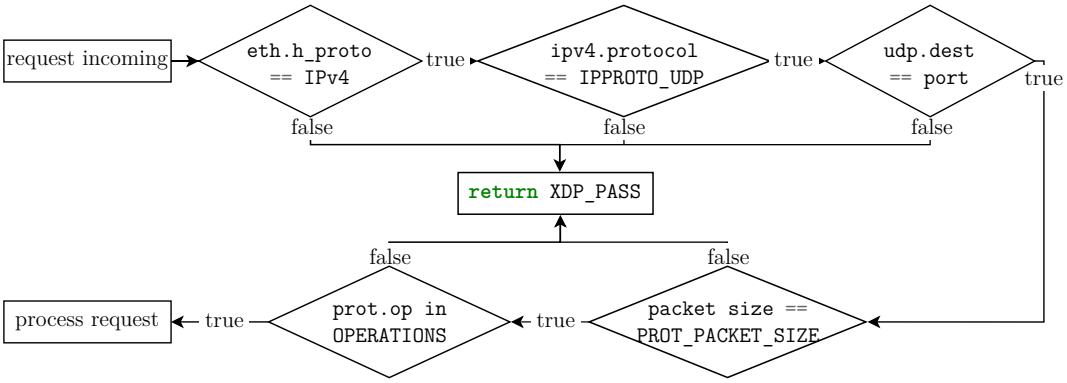


Figure 4.2: The chain of checks an eBPF program has to go through to be sure to only process packets it is supposed to. Not only general information, such as the Ethernet protocol, IP protocol and port used need to be checked, but also application specific information, such as the packet size, and the type of packet received, need to be checked.

For that, it goes through a range of checks. As seen in fig. 4.2, it first checks that the Ethernet protocol used is IPv4. This is specified in the Ethernet header in the `h_proto` field which holds specific values for IPv4 and IPv6 packets[2]. It continues to check the transportation protocol used by comparing the IPv4 headers `protocol` field with the value `IPPROTO_UDP`. This means a UDP packet has arrived. Next it checks the `dest` field in the UDP header. It indicates the destination port and it should be the same that the socket is bound to. This value has to be passed to the eBPF program. If all these have turned out true, the packet is destined to the server application and should therefore adhere to the application level protocol used there. To verify, the eBPF program checks the size of the content of the packet. It should equal the specified packet size, in fig. 4.2 described as `PACKET_SIZE`. Finally, it checks the operation specified in the packet. If it is in the list of operations, described in fig. 4.2 as `OPERATIONS`, it goes on to process the request. The specifics of the packet processing is dependent on the application.

After finishing to process the request, the eBPF program needs to return the packet to the sender. For this the MAC-address, the IP-address and the port are swapped. As the IPv4 header, the UDP header and the content of the packet change, the IPv4 and UDP checksums must be recalculated. Otherwise, these packets have invalid checksums and will be dropped at the first hop checking them. Finally, the program returns the `XDP_TX` enum value to indicate to the kernel, that the packet should not be handed up the Linux network stack, but rather be passed to the

[2] `https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml`, accessed 4th June, 2023

egress point of the XDP.

However, if any of the previous checks fail, the eBPF program returns the `XDP_PASS` enum value, to indicate to the kernel that the eBPF program does not need to do any further processing and the packet should be passed further through the network stack.

4.1.2 User-Space Application

Next to the code paths not masked by the eBPF program, which still need to be handled by the user-space application, the user-space application now has two new tasks. First it needs to load the eBPF program into the kernel and second it needs to hold a reference to all the maps needed in the application.

This new part of the application is thus responsible for loading and attaching, and unloading and detaching the eBPF program into the kernel at the appropriate hook points and for making data accessible to the eBPF program by implementing access to the state through eBPF maps. This enables both the eBPF- and user-space-program to access correct, i.e. up-to-date, data and to change state persistently.

4.2 Limitations

As there are now at least two concurrent programs running, depending on the original program, data access has to be synchronized. As the maps are in memory in the eBPF program on the kernel side this can be done by using atomic operations provided by clang and gcc such as `__sync_fetch_and_add()`. This is necessary, as even though eBPF programs are not preemptable, multiple instances of the program could run simultaneously on the different cores of a system. When accessing the data from the user-space however, it is copied over into the memory of the user-space program. As there is no direct access to the original memory, it is not possible to synchronize access. This means an increment needs to happen in three steps, copy, increment, write, where the write operation will overwrite the data on the kernel side, even if it changed in between the three operations[3].

To avoid this, a solution could be, to separate the data that is calculated in the user-space from the data it depends on, which may be gathered in kernel space. This way, calculation and changing of one particular variable is only ever done either in kernel or in user-space and only read in the other.

[3] `https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html`

Additionally, as many packets are now handled in kernel-space, the existence of these packets remains entirely unknown to the user-space application. This means recalculating states on incoming packets is not possible anymore. As there currently is no way for an eBPF program attached to the XDP to signal the user-space application from the eBPF program, these tasks must be handled on a timer. This timer needs to be set on assumptions about how often the data changes, the state depends on.

In certain applications it might be feasible to recalculate that state in the eBPF program and pass it to the user-space through a map if necessary. However, to reiterate, eBPF programs are limited in size and capabilities, as shown in section 2.4.

Thirdly, currently not implemented in eBPF is the use of floating point operations. This means more complex calculations that depend on the precision of floating point numbers simply can not be done in eBPF as of today.

Finally, as eBPF is a rather new concept in the Linux kernel. With the first eBPF in-kernel interpreter added in 2014, the interfaces are subject to change and new features are added constantly. Using a recent kernel might be necessary to facilitate an eBPF approach, but could be unavailable to some systems.

Chapter 5

Evaluation

This chapter first introduces the application example in which the approach is tested in. A detailed description of the setting, the communication parties and the communication itself is given, to show the applicability of the approach. It continues by giving a comprehensive description of the experiment setting including the experiments and the hardware on which they are run. Finally it shows the results and evaluates them in the setting of RT systems in Edge Computing environments.

5.1 Application

With the growing demand for efficient and sustainable urban transportation systems, the introduction of Intelligent Traffic Management Systems (ITMS) becomes imperative. These systems offer the potential to revolutionize how traffic is managed, optimize flow, enhance safety, and promote a greener and more livable urban environment. In an ITMS live data is used to manage the traffic, which enables it to adapt traffic control with intelligent decisions, detect incidents like crashes, integrate with public transportation, give up-to-date information like the expected time of arrival and more, and increase safety for unprotected road users.

It also brings with it challenges on the road side, where IoT devices, such as cars, communicate with road signs, the edge nodes, to enable assisted or autonomous driving. In this scenario as seen in fig. 5.1, a car approaches a street sign which regulates the speed limit depending on the amount of vehicles currently in near vicinity of the sign. The car request the speed limit of the sign which it then responds with. Another car at the same time leaves the area of effect and signals that to the sign.

To implement this, first a simple application-level protocol is designed. It consists only of two 4 B fields: the `OP` and `VALUE` field. While the `VALUE` field can hold

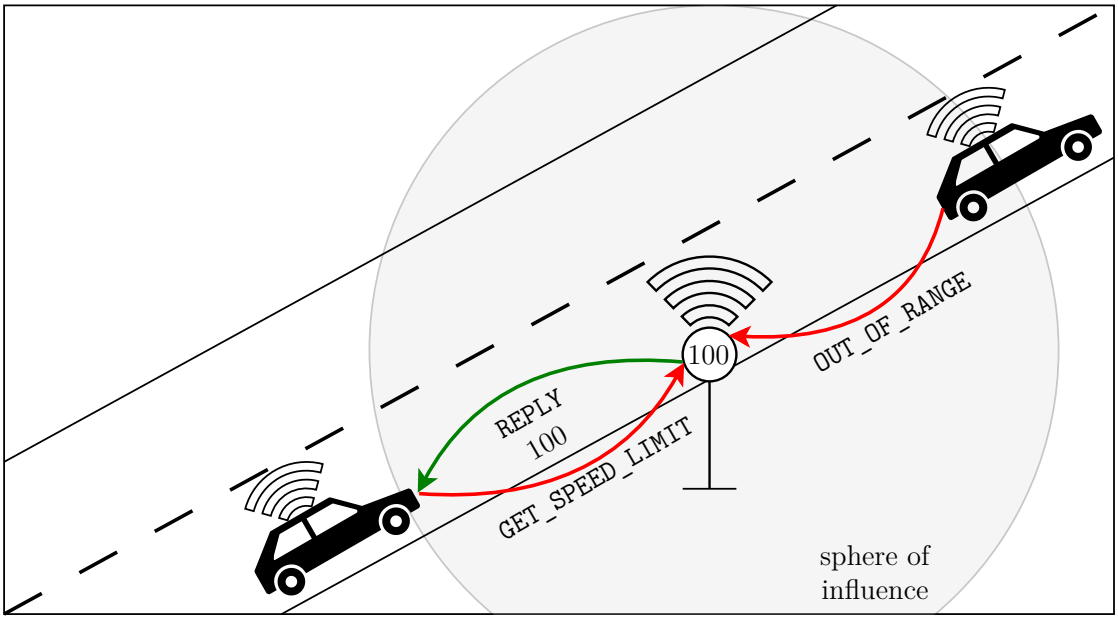


Figure 5.1: A snapshot of a situation in the evaluation application scenario. A car arrives in the regulated zone and requests the speed limit, while another car leaves the zone, sending its final message.

an arbitrary 32 bit integer value, the `OP` field is reserved for the values seen in table 5.1.

In this context, as seen in fig. 5.1, `GET_SPEED_LIMIT` is used by cars to request the current speed limit of the street sign they are approaching. The street sign then replies with a `REPLY` package and the current speed limit in the `VALUE` field of the protocol. The car proceeds on its way, adjusting its own speed if necessary, and once it reaches the end of the sphere of influence of the street sign sends a `OUT_OF_RANGE` packet.

The street sign counts the number of cars in its range by incrementing its car

Table 5.1: The relevant values the `OP` field of the protocol can hold, and their enum representations in the code.

enum name	value
`REPLY`	3
`GET_SPEED_LIMIT`	4
`OUT_OF_RANGE`	5

counter by one each time a `GET_SPEED_LIMIT` request comes in and decrements the counter by one every time a car sends an `OUT_OF_RANGE` operation. It also re-calculates the speed limit whenever a `GET_SPEED_LIMIT` requests is incoming, before replying. This ensures the cars always get the most up-to-date speed limit, as its always recalculated before sending it off to a car.

$$\text{speed limit}(c) = \begin{cases} 120 & 0 \leq c < 30 \\ 45 * \cos\left(\frac{\pi}{70} * (c - 30)\right) + 75 & 30 \leq c < 100\ ,\ c \in \mathbb{Z}^+ \\ 30 & 100 \leq c \end{cases} \tag{5.1}$$

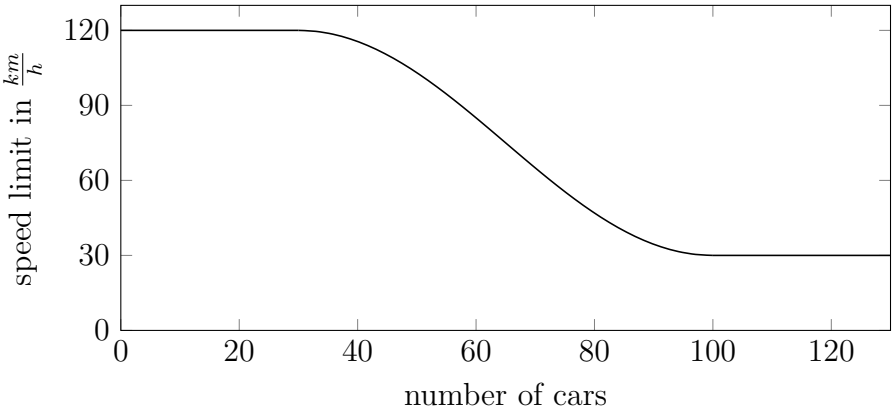


Figure 5.2: The function of the speed limit to the amount of cars in the sphere of influence as described in eq. (5.1) visualized.

The speed limit itself is calculated as a function of the amount of cars and, as seen in eq. (5.1), is itself separated into three functions. While for amounts of cars between 0 and 30 the speed limit is fixed to a maximum value of $120\,\mathrm{km\,h^{-1}}$, for amounts of cars greater or equal than 100, the speed is fixed to a minimum value of $30\,\mathrm{km\,h^{-1}}$. In between a cos function creates an even transition between the two extreme speed limits. It is visualized in fig. 5.2, however this function is not meant to be the best possible way to determine the speed limit based on the cars, as different sized and located roads might need different maximum and minimum values and possibly steeper or flatter transitions. However as this thesis aims to examine the eBPF application in a RT context, how the speed limit is calculated is not that important.

In this application scenario the eBPF program is responsible for processing the `GET_SPEED_LIMIT` and `OUT_OF_RANGE` requests. After verifying the checks 1-4 seen in fig. 4.2, in step 5 it checks the `OP` field of the packet to be the `GET_SPEED_LIMIT`

or `OUT_OF_RANGE` op-code. In this case it either increments the cars counter in the map and continues by accessing the map to fetch the speed limit to write that into the `VALUE` field and send the packet back to the sender, or it decrements the cars counter and drops the packet, depending on the op-code.

The maps need to hold all the information necessary for all of the components to be fully functional as this is the only way to communicate between them. In this application only one map of type hash map is required. It maps string keys to 32 bit unsigned integers. The three variables held in the map are the port to which the socket is bound, the current speed limit and the amount of cars currently in range of the sign.

In a real-world setting, the amount of cars that can enter the range of the sign is physically limited by the size of the road. Thus it is enough to check the cars counter and update the speed limit once every second. And both the user-space and the eBPF program always have to update the maps also whenever they change the value of the speed limit or the car counter.

However, in this specific application, as the function only depends on values known to the eBPF program and is simple to compute, it is also possible to recalculate the speed limit in the eBPF program itself. This should, on the one hand increase the precision of the value, as it is recalculated with the most up to date data, and on the other hand decrease the precision due to the unavailability of floating point operations. It does however yield an interesting data point, adding a more complex computation to the program instead of just the lookup.

5.2 Experiments

This section is divided into two parts. The first details the setup of the machines the experiments are run on, the second introduces the experiment procedures themselves.

5.2.1 Set-Up

The application is validated by recreating the scenario in two virtual machines. The first hosts the server. It has at its disposal two virtual cores. The second virtual machine hosts the client using four virtual cores. All cores are pinned to individual threads of the hosts Intel Core i7-8565U CPU clocked at 1.8 GHz. The scaling governor of the host machine for all these threads is set to “performance” mode, to ensure the frequency stays at the specified 1.8 GHz. In total the setup uses one entire physical core for the servers virtual machine and two physical

cores for the client machine. Both server and client virtual machines run the Debian[1] 12.0.0 (Bookworm) Linux distribution, with its standard Linux kernel version 6.1.0-9-amd64. The host x86-64 system runs kernel version 6.3.9-arch1-1 using the Kernel-based Virtual Machine (KVM)[2] hypervisor that is built into that Linux kernel.

The two virtual machines are connected to the hosts network through a virtual network bridge and are in one-hop distance of each other.

Table 5.2: The three configurations of the server.

shorthand	eBPF	speed-limit
user-space-only	disabled	user-space
mixed	enabled	user-space
eBPF-only	enabled	eBPF

As seen in table 5.2, the server is set-up in three different configurations. In the first, the server runs in user-space-only mode. This means there is no eBPF component and all handling of the eBPF program is also disabled. This is equivalent to the original server application as described earlier and the communication flows as shown in fig. 4.1a. It will from here on out be referred to as user-space-only. In the second and third configuration, the eBPF component is enabled, however in the second, the user-space still computes the speed-limit, while in the third, the speed-limit is computed right in the eBPF program itself. They will be referred to as mixed and eBPF-only respectively and in both configurations communication flows as shown in fig. 4.1b. To make sure the three different configurations do not have overhead, of the components not used, the code-paths not needed for them to work are not only disabled, but removed by the C-preprocessor.

5.2.2 Procedure

For the first experiment, the client sends 10 000 requests with 120 threads concurrently for each of these server configurations. Each of these threads sends 10 000 packets. This shows the difference in response time between the three approaches.

The second experiment aims to show the difference in systems resources used by testing how many threads can send packets simultaneously to the server until the

[1] `https://www.debian.org/`, accessed 3rd July, 2023
[2] `https://www.linux-kvm.org/page/Main_Page`, accessed 3rd July, 2023

Table 5.3: Enumeration of the experiments and their respective short description.

experiment	short description
experiment 1	response time of 10 000 requests on 120 threads
experiment 2	simultaneous client threads until the first packet is dropped
experiment 3	response time of 10 000 requests per thread with an increasing number of threads

UDP packets are dropped. This happens when buffers, such as the ingress network buffer, or the CPU caches fill up because the requests are handled too slowly.

To test this, the client is started repeatedly and with each consecutive start the amount of threads is incremented by one. Between each start, a one second period is awaited, where the server has time to return to a neutral state. Each client waits for the reply and if after a timeout no reply has arrived, it is killed and one less than the number of threads is logged. This is the amount of threads that could still send the packets with no issues. This experiment is repeated multiple times.

For experiment number three, the client is run with an increasing number of threads, the same way as in the second experiment, until the first packet is dropped and the response time of the 10 000 requests is measured for each thread count. This gives an idea how the response time changes depending on the amount of clients sending requests simultaneously.

The response time is measured on the clients as the difference in clock time from right before sending the first request, to the reception of the last reply.

An enumeration and short description for quick referencing is given in table 5.3.

5.3 Results

This section shows the results yielded by the three experiments and evaluates them in their respective context.

Experiment 1 Figure 5.3 shows the results of one instance of the first experiment in three box-plots, one for each configuration of user-space-only, mixed and eBPF-only. On the y-axis the response time per packet in ms is shown. Table 5.4 shows the median, mean, minimum and maximum values, and standard deviation of the same experiment. While the experiment yielded a median of 1.8868 ms for the

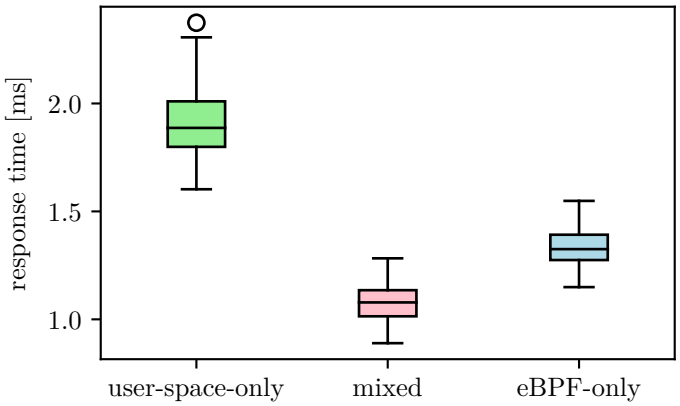


Figure 5.3: Clock time passed in ms per request for 120 threads sending 10 000 requests with no other CPU load on the server and it configured as user-space-only, mixed, and eBPF-only respectively.

established user-space-only approach, it showed significant reduction in median response time for both the novel approaches, with the mixed configuration yielding a median of 1.0784 ms and the eBPF-only approach a median of 1.3252 ms. That is a speed-up of around the factor 1.75 and 1.4 respectively.

While the minimum values of the novel approach go down to as low as 0.8896 ms and 1.1492 ms for the mixed and eBPF-only configuration respectively, even the maximum response times of 1.2826 ms and 1.5487 ms of the mixed and eBPF-only configurations stay below the minimum response time of the established user-space-only configuration of 1.6026 ms.

Table 5.4: Median, minimum and maximum values, and standard deviation of the clock time passed in ms for 120 threads sending 10 000 requests with no other CPU load on the server, and the server configured as user-space-only, mixed, and eBPF-only respectively.

configuration	median	minimum	maximum	standard deviation
user-space-only	1.8868 ms	1.6026 ms	2.3739 ms	0.1608 ms
mixed	1.0784 ms	0.8896 ms	1.2826 ms	0.0878 ms
eBPF-only	1.3252 ms	1.1492 ms	1.5487 ms	0.0818 ms

Table 5.5: Median, mean, and standard deviation of the median response time per packet in ms over 96 executions of the first experiment: 120 threads sending 10 000 requests, with the server being configured as user-space-only, mixed, and eBPF-only respectively.

configuration	median	mean	standard deviation
user-space-only	1.7476 ms	1.7514 ms	0.0187 ms
mixed	1.2010 ms	1.2011 ms	0.0111 ms
eBPF-only	1.2654 ms	1.2638 ms	0.0113 ms

The standard deviation of the novel approach is also reduced in comparison to the established solution by a factor of around 1.83 and 1.97 for mixed and eBPF-only configurations respectively with the absolute values being 0.1608, 0.0878, and 0.0818 for user-space-only, mixed, and eBPF-only respectively.

This is visualized in the box-plots. The lower whisker of the user-space-only configuration is higher than both the upper whiskers of the mixed and eBPF-only configurations. Both the entire box-plots and the boxes of the mixed and eBPF-only configurations are also visually smaller, indicating the smaller variance in response time observed in the experiment.

Although both the eBPF approaches show a significant reduction in response time compared to the base-line provided by the user-space-only configuration, the mixed configuration improves the response time even more. This is due to the fact that the eBPF-only approach also calculates the appropriate response, when needed, while the mixed approach relies on the user-space program to compute the state and thus requiring less resources.

Table 5.5 shows the median, mean, and standard deviation of the median response time over the repeated execution of the first experiment. The small values of the standard deviation show that there is not a lot of difference in the individual runs of the experiment. It also shows that, on average, the mixed configuration experiences a speed-up of a factor of about 1.76 compared to the base-line and the eBPF-only configuration a speed-up of a factor of about 1.43. Finally, the data shows, that the results of the previously shown execution of the experiment is representative among the three configurations.

Figure 5.4 shows the median and standard deviation of the response time per packet of each individual thread for the three configurations. On the y-axis the response time per packet in ms is shown, on the x-axis the individual thread IDs numbered from 0 to 119 are displayed. Although the threads executing first might have lower

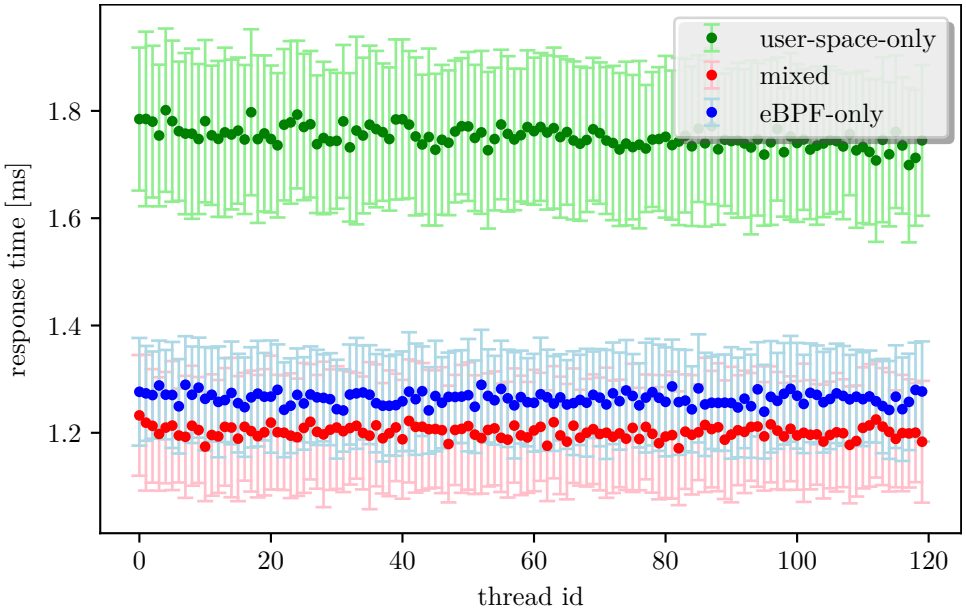


Figure 5.4: Median and standard deviation of the response time per request in ms of the individual 120 threads sending 10 000 requests in 96 executions with the server being configured as user-space-only, mixed, and eBPF-only respectively.

response times for their respective first requests, as the time over 10 000 requests is measured, the response time approaches similarity once all the threads are actively sending.

The plot shows that for all threads the response time has a similar median and standard deviation, suggesting that no prioritization is happening on the server side.

Experiment 2 The results of the second experiment are shown in fig. 5.5. It shows three box-plots one for each user-space-only, mixed, and eBPF-only configuration. On the y-axis the number of threads is displayed. The results show that both novel approaches can handle almost double the threads, and thus clients, sending requests.

The resulting values are displayed in table 5.6 and refine that while the mixed configuration can, on average, handle more packets by the factor of around 1.93, the eBPF-only approach on average even goes up to a factor of around 1.95.

The highest amount of threads simultaneously sending requests without the server dropping packets is achieved by the mixed configuration with 300 threads.

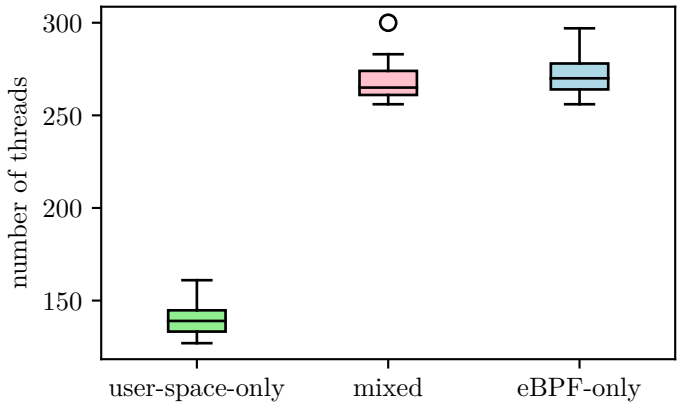


Figure 5.5: Amount of simultaneous threads sending 10 000 requests until the first packet is dropped with the server being configured as user-space-only, mixed, and eBPF-only respectively.

Experiment 3 Figure 5.6 show the results of the third experiment. On the x-axis the number of threads that are each sending 10 000 requests is shown. The y-axis shows the response time per packet in ms. The plot shows the actual response times as well as a linear regression of these for each of the configurations.

The first observation is that the user-space-only dots and linear regression stop at around 125 threads. That is, as shown in experiment 2, because the requests are dropped due to the server not being able to keep up at around that number.

Secondly, the graph shows that even for smaller amounts of threads than the previously shown 120, the response time of the user-space-only configuration is higher than for both the eBPF approaches.

The graph also shows that the response time increases linearly with the amount

Table 5.6: Median, mean, and minimum and maximum values of the amount of threads sending 10 000 requests until the first packet is dropped with the server being configured as user-space-only, mixed, and eBPF-only respectively.

configuration	median	mean	minimum	maximum
user-space-only	139	139.5	127	161
mixed	265	269.6	256	300
eBPF-only	270	271.7	256	297

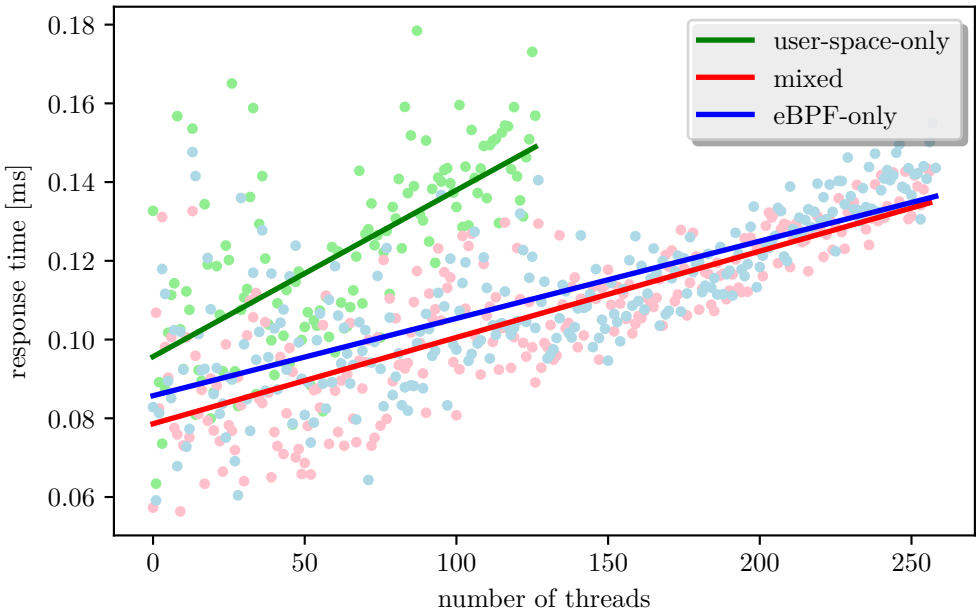


Figure 5.6: Median response time and its linear regression of sending 10 000 requests per thread as the number of threads increases with the server configured as user-space-only, mixed, and eBPF-only respectively.

of threads sending simultaneous requests. This shows that it still takes longer to process the request on the server than it takes to send out the request on the client, which is to be expected. The slightly higher response time of the eBPF-only configuration, compared to the mixed configuration is also in line with the findings from the first experiment.

Interesting is, however, that the regressions of the two eBPF configurations seem to approach each other, with only a minimal difference close to the 250 thread mark.

5.4 Contextualization

This section evaluates the results of the three experiments in the context of the thesis problem statement - Real-Time systems in Edge Computing environments.

The experiments show that using eBPF to respond early in the Linux kernel network stack has significant advantages compared to the established approach. Not only does the response time decrease, the standard deviation also decreases, and the amount of simultaneously received packets that the server can handle

increases.

Having the standard deviation decreased as demonstrated by the first experiment indicates that there is less variance in the system and the probability of having higher response times is decreased as well. This is important in the context of RT systems, where tasks have to meet deadlines to meet QoS and potentially safety regulation.

The ability to use the resources available more efficiently, which is demonstrated by the second experiment, does not only have the advantage of being cost-effective, but also allows the use of less powerful machines, and therefor consume less energy, for the same tasks. This is always beneficial but especially so in the context of the Edge Computing environment where less computing power is available compared to the cloud. Additionally, this might allow the application to run on dedicated hardware as well as using often overlooked computing resources of the environment, such as parked cars, possibly running on a battery.

The experiments however also show that there is still a choice to be made, when it comes to the location of the heavy computation. While computing the state in the kernel brings the advantage of higher precision in the content of the reply, it comes at the drawback of a higher response time. Vice versa, computing the state in the user-space on a timer lowers the precision of the replies but increases the speed at which the server can respond. Depending on the application, one or the other can be favorable and with certain limits on the amount of incoming requests, computing the state on a timer might not be as disadvantageous. For example, in the application chosen here, there is a physical limit on the amount of cars that can pass by the street sign in a time window and computing the speed limit once every second is probably plentiful.

Chapter 6

Conclusion

This thesis introduced an eBPF approach for RT applications in Edge Computing environments. The approach is capable of both reducing the total response time and its variance, and increase the amount of simultaneous request able to be handled significantly by moving the process of replying into kernel-space at an early stage in the Linux kernel network stack, specifically the XDP. Optionally, not only the reply can be created in-kernel, but also the value of the response recalculated. This increases precision of the response at the cost of response time.

Additionally, in the context of highly heterogeneous Edge Computing environments the ability of using a standard Linux kernel enables interoperability out-of-the-box for a multitude of different architectures and hardware, cuts cost and might even promote the use of Edge Computing in general.

To assess the usability of the approach, an exemplary application in the domain of ITMS is presented in which the approach is evaluated in. The results of the experiments show that the approach yields a performance increase of a factor of about 1.76 or 1.43. Additionally, around 1.93, or 1.95 times the amount of simultaneous clients could send requests until the server could not keep up anymore and packets started to be dropped. For both these metrics, the first number shows the result of the configuration with only packet processing done in-kernel while the second shows the result of the configuration with additional state computing.

These results show that using eBPF in RT systems in Edge Computing environments is feasible and should be investigated further in real applications. As the Linux kernel eBPF component is still under development and in its early stages, it is also interesting to see more capabilities introduced in the future and what can be achieved using these. Furthermore, research on how eBPF performs for tasks with higher computational cost might be interesting. With the small workload performed in the application in this thesis, the effect on the response time was already substantial, however the question arises how that changes, with higher

workload per packet. Measuring QoS of the two eBPF approaches would be interesting, especially when with added cloud control, which is especially useful in ITMS. Finally, further research on the improvements on actual CPU load and therefore also energy consumption might bring more insights into its use-case in low-power or remote and battery powered networked computing.

To conclude, the goals of this thesis where successfully implemented and although some research questions remain open, the application of eBPF in Real-Time applications in Edge Computing environments is evident.

Bibliography

[1] Pradyumna Gokhale, Omkar Bhat, and Sagar Bhat. “Introduction to IOT”. In: *International Advanced Research Journal in Science, Engineering and Technology* 5.1 (2018).

[2] Transforma Insights. *Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030 (in billions)*. 2022. URL: `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/` (visited on 06/11/2023).

[3] Deutsches Institut für Normung e.V. (Hrsg.) *DIN 44300: Informationsverarbeitung - Begriffe - Allgemeine Begriffe*. 1988.

[4] Manfred Broy, María Victoria Cengarle, and Eva Milena Geisberger. “Cyber-Physical Systems: Imminent Challenges”. In: 2012.

[5] K.G. Shin and P. Ramanathan. “Real-time computing: a new discipline of computer science and engineering”. In: *Proceedings of the IEEE* 82.1 (1994).

[6] Alexandre Marcastel et al. “Online Interference Mitigation via Learning in Dynamic IoT Environments”. In: *2016 IEEE Globecom Workshops (GC Wkshps)*. Dec. 2016, pp. 1–5.

[7] Weisong Shi et al. “Edge Computing: Vision and Challenges”. In: *IEEE Internet of Things Journal* 3.5 (2016).

[8] Swaroop Nunna et al. “Enabling Real-Time Context-Aware Collaboration through 5G and Mobile Edge Computing”. In: *2015 12th International Conference on Information Technology - New Generations*. 2015, pp. 601–605.

[9] Petcharat Suriyachai, Utz Roedig, and Andrew Scott. “A Survey of MAC Protocols for Mission-Critical Applications in Wireless Sensor Networks”. In: *IEEE Communications Surveys & Tutorials* 14.2 (2012).

[10] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Pearson Education Limited, 2006.

[11] Craig Labovitz et al. *Internet Traffic Trends - A View from 67 ISPs*. 2009.

[12] Vinton Cerf, Yogen Dalal, and Carl Sunshine. *RFC: 675 Specification of Transmission Control Protocol*. 1974.

[13] J. Postel. *RFC: 768 User Datagram Protocol*. 1980.

[14] Nick Antonoloulos and Lee Gillam. *Cloud computing*. Springer, 2010.

[15] Ellen Siever et al. *Linux in a Nutshell*. O’Reilly Media, Inc., 2005.

[16] Renzo Davoli and Michele DiStefano. “BERKELEY PACKET FILTER: theory, practice and perspectives”. MA thesis. Universita di Bologna, 2019.
[17] Sebastiano Miano et al. “Creating Complex Network Services with eBPF: Experience and Lessons Learned”. In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. 2018, pp. 1–8.
[18] Matteo Bertrone et al. “Accelerating Linux Security with eBPF iptables”. In: Association for Computing Machinery, 2018.
[19] Luca Deri, Samuele Sabella, and Simone Mainardi. “Combining System Visibility and Security Using eBPF”. In: *ITASEC 2019*. 2019.
[20] Angelo Feraudo et al. *Mitigating IoT Botnet DDos Attacks through MUD and eBPF based Traffic Filtering*. 2023.
[21] Talaya Farasat, Muhammad Ahmad Rathore, and JongWon Kim. “Securing Kubernetes Pods communicating over Weave Net through eBPF/XDP from DDoS attacks”. In: *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*. CODASPY ’23. Association for Computing Machinery, 2023.
[22] Sebastiano Miano et al. “A Framework for eBPF-Based Network Functions in an Era of Microservices”. In: *IEEE Transactions on Network and Service Management* 18.1 (2021).
[23] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. “Leveraging eBPF for programmable network functions with IPv6 segment routing”. In: *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. CoNEXT ’18. Association for Computing Machinery, 2018.
[24] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. “Bypassing the load balancer without regrets”. In: *Proceedings of the 11th ACM Symposium on Cloud Computing* (2020), pp. 193–207.
[25] Jung-Bok Lee et al. “High-Performance Software Load Balancer for Cloud-Native Architecture”. In: *IEEE Access* 9 (2021).
[26] Jakub Kicinski and Nicolaas Viljoen. “eBPF Hardware Offload to SmartNICs: cls bpf and XDP”. In: Netronome Systems, 2016.
[27] Lei Zhao et al. “Optimal Edge Resource Allocation in IoT-Based Smart Cities”. In: *IEEE Network* 33.2 (Mar. 2019), pp. 30–35. ISSN: 1558-156X. DOI: 10.1109/MNET.2019.1800221.
[28] S. Bradner. *RFC: 2544 Benchmarking Methodology for Network Interconnect Devices*. 1999.
[29] Rashid Mijumbi et al. “Network Function Virtualization: State-of-the-Art and Research Challenges”. In: *IEEE Communications Surveys & Tutorials* 18.1 (2016).

[30] Jan Rüth et al. “Demo abstract: Towards in-network processing for low-latency industrial control”. In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2018.