Design and Evaluation of an End-to-End Benchmarking and Monitoring System for 6G Networks

Presented by
Amro Hendawi, 396692
Born in Aleppo

Faculty Elektrotechnik und Informatik
Technical University - Berlin
To get the degree in

Bachelor of Science

Submitted

Board:

Chair: Prof. Dr. Thomas Magedanz (Technische Universität Berlin)
Reviewer: Prof. Dr. Thomas Magedanz (Technische Universität Berlin)
Reviewer: Prof. Dr. Axel Küpper (Technische Universität Berlin)

15. February 2022

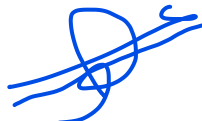
Affidavit

I hereby declare that the following thesis "Design and Evaluation of an End-to-End Benchmarking and Monitoring System for 6G Networks" has been written only by the undersigned and without any assistance from third parties.
Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Erklärung der Urheberschaft

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.

Berlin, 15. February 2022



Abstract

With the shift towards Industry 4.0 (I4.0) and the Internet of Things (IoT), ensuring guarantees is becoming more challenging as we transition towards software defined architecture. Industrial and IoT applications are safety and time critical in nature, in addition to exchanging large amounts of data they are required to make time constrained decisions. Limited bandwidth and/or untimely responses in the networks deployed in such fields may lead to undesired or even catastrophic outcomes. State-of-the-art networking technologies, e.g. Time-sensitive Networking (TSN), Fifth Network Generation (5G) and Cyber Physical System (CPS), aim towards meeting the requirements of the next generation industry as deterministic and critical machine-to-machine communications while ensuring advanced cybersecurity. However, the traditional network monitoring solutions aren't suitable for evaluating the new network technologies targeting real-time communications. We survey the state-of-the-art, review present network monitoring and benchmarking solutions, and identify open problems of monitoring real-time networks. We propose a benchmarking and monitoring system that uses the extended Berkeley Packet Filter (eBPF) technology to achieve real-time network monitoring. The proposed proof of concept implementation is analysed and the contributions are evaluated in a testbed. We found that utilizing eBPF in network monitoring systems improves performance and reduces resources overhead.

ZUSAMMENFASSUNG

Mit dem Wandel zu Industrie 4.0 (I4.0) und dem Internet der Dinge (IoT) wird es immer schwieriger, Garantien zu gewährleisten, da wir zu einer softwaredefinierten Architektur übergehen. Industrielle und IoT-Anwendungen sind sicherheits- und zeitkritisch und müssen nicht nur große Datenmengen austauschen, sondern auch zeitlich begrenzte Entscheidungen treffen. Begrenzte Bandbreite und/oder unzeitgemäße Reaktionen in den Netzwerken, die in solchen Bereichen eingesetzt werden, können zu unerwünschten oder sogar katastrophalen Resultaten führen. Modernste Netzwerktechnologien, z. B. TSN, die fünfte Generation Netzwerken (5G) und Cyber-Physische Systeme (CPS), zielen darauf ab, die Anforderungen der Industrie der nächsten Generation an eine deterministische und kritische Maschine-zu-Maschine-Kommunikation zu erfüllen und gleichzeitig fortschrittliche Cybersicherheit zu gewährleisten. Die herkömmlichen Netzwerküberwachungslösungen sind jedoch nicht für die Bewertung der neuen Netzwerktechnologien geeignet, die auf Echtzeitkommunikation abzielen. Wir geben einen Überblick über den aktuellen Stand der Technik, überprüfen die derzeitigen Netzwerküberwachungs- und Benchmarking-Lösungen und identifizieren offene Probleme bei der Überwachung von Echtzeitnetzwerken. Wir entwerfen ein Benchmarking- und Überwachungssystem, das die erweiterte Berkeley Packet Filter (eBPF)-Technologie nutzt, um eine Netzwerküberwachung in Echtzeit zu erreichen. Die entworfene Proof-of-Concept-Implementierung wird analysiert und die Beiträge werden in einem Testbed bewertet. Wir haben festgestellt, dass der Einsatz von eBPF in Netzwerküberwachungssystemen die Leistung verbessert und den Ressourcenverbrauch reduziert.

Table of Contents

List of Figures iv

1 Introduction 1
1.1 Background and Motivation . . . 1
1.2 Problem statement . . . 2
1.3 Assumptions and Scope . . . 3
1.4 Objectives and Contributions . . . 4
1.5 Methodology and Outline . . . 4

2 State of the art 5
2.1 Modern network technologies . . . 5
2.1.1 TSN . . . 6
2.1.2 5G . . . 7
2.1.3 Software-defined Networking (SDN) . . . 7
2.1.4 Fieldbus . . . 8
2.2 Network monitoring . . . 8
2.2.1 Collection layer . . . 9
2.2.2 Representation layer . . . 9
2.2.3 Report layer . . . 9
2.2.4 Analysis layer . . . 9
2.2.5 Presentation layer . . . 10
2.3 Packet filtering in Linux machines . . . 10
2.3.1 Hardware Level . . . 10
2.3.2 Network Level . . . 11
2.3.3 System Level . . . 11
2.3.4 Application Level . . . 11
2.4 The Berkeley packet filter eBPF . . . 11
2.5 The eXpress data path eXpress Data Path (XDP) . . . 12
2.6 Conclusion . . . 13

3 Requirement Analysis 17
3.1 Top-level requirements . . . 17
3.2 System-level requirements . . . 18
3.3 Feature-level requirements . . . 19

4 Concept and Approach **20**
4.0.1 Introduction . . . 20
4.1 General discussion . . . 20
4.2 Software Architecture . . . 22
4.3 Conclusion . . . 24

5 Implementation **25**
5.1 Introduction . . . 25
5.2 Software Design . . . 25
5.2.1 Real-time data collection . . . 26
5.2.2 Active probing . . . 27
5.2.3 Real-time data exporter . . . 28
5.2.4 Visualization . . . 29
5.3 Conclusion . . . 32

6 Evaluation **33**
6.1 Introduction . . . 33
6.2 Experimental validation . . . 34
6.2.1 Setup 1 . . . 35
6.2.2 Setup 2 . . . 35
6.3 Performance evaluation . . . 36
6.3.1 Evaluation on single node . . . 36
6.3.2 Stress-test evalaution . . . 37
6.4 Hardware utilization evaluation . . . 39
6.5 Comparative Analysis . . . 40
6.5.1 Requirement Evaluation . . . 40
6.6 Conclusion . . . 42

7 Summary and Further Work **43**
7.1 Overview . . . 43
7.2 Conclusions and Impact . . . 44
7.3 Outlook . . . 44

8 Specifications **I**
8.1 eBPF program examples . . . I
8.1.1 BCC programs as modules . . . I
8.2 Docker Implementation . . . III
8.3 Testbed initialization script . . . IV
8.3.1 Testbed over Docker . . . IV
8.3.2 Testbed over GCP . . . V
8.4 Performance tools bpftool . . . VI
8.5 Writing XDP programs . . . VI
8.6 FlameGraph preparation . . . VII

Bibliography **VIII**

Index **XIII**

List of Figures

2.1 the monitoring model [p7] 8
2.2 Levels of packet filtering [p5] 10
2.3 eBPF program flow [t2] 12
2.4 XDP packet processing overview [t3] 13
2.5 eBPF XDP program example in C language [p29] 13
2.6 privilige rings for the x86 available [p31] 14
2.7 Linux kernel network stack[p29] 15
2.8 bpf iptables performance comparison [p33] 15

4.1 High-level diagram of network monitoring 21
4.2 The monitoring system as a blackbox 21
4.3 The software architecture object diagram 22
4.4 Sequence diagram of the monitoring funcionality from data collection to visualization 23

5.1 The monitoring system's architecture 26
5.2 eBPF packet filtering and data exporting activity diagram 27
5.3 Activity diagram of active probing 28
5.4 Prometheus dashboard showing all connected resources of metrics . . 30
5.5 Grafana dashboard with example graphs plotting real-time data 31
5.6 Creating a new panel in Grafana 31

6.1 The testbed architecture 35
6.2 Testbed for stress testing 35
6.3 Latency comparison of eBPF example MetricCollector with non-eBPF based solution running on testbed setup 6.1 36
6.4 Latency measurements from the eBPF-based and non-eBPF tools on 3 testbeds 38
6.5 Flamegraph 39

8.1 tcprtt example output II
8.2 tcpconnlat example output III
8.3 tcplife example output III

Chapter 1

CHAPTER 1

INTRODUCTION

5 1.1 Background and Motivation 1
1.2 Problem statement 2
1.3 Assumptions and Scope 3
1.4 Objectives and Contributions 4
1.5 Methodology and Outline 4
10

1.1 Background and Motivation

Connectivity has become a pivotal driver towards digitalization and automation in
15 industrial environments. The evolution of the Internet of Things (IoT) and Cloud services has facilitated the rise of the fourth industrial generation, Industry 4.0 (I4.0), as a new trend of automation and data exchange in the manufacturing industry. This new industrial paradigm is characterised by its ability to reconfigure and often optimize autonomously.

The fourth industrial revolution I4.0 aims at transforming today's factories into
20 intelligently connected production information systems that operate well beyond the physical boundaries of the factory premises. Factories of the future leverage the smart integration of Cyber Physical System (CPS) and IoT solutions in industrial processes[p1], moving the industry to next technology while ensuring cyber security. The Fifth Network Generation (5G) networks will play a key role in enabling this integration, offering
25 programmable technology platforms able to connect a wide variety of devices in an ubiquitous manner. The 5G network infrastructure provides highly secure, reliable and resilient cellular connectivity, which is crucial for mission-critical applications.

With the increase in the amount of data captured during the manufacturing process, monitoring systems are becoming important factors in decision making for management.

30 Robust determinism and real-time (RT) performance are mandatory for process control and manufacturing systems in modern industrial automation. Many modern information systems are becoming safety-critical in a general sense because loss of life and property can result from their failure.[p2]. The number of computer systems that fall under the safety-critical category is increasing dramatically as computer systems
35 continue to be introduced into many areas that affect our lives.

Moreover, there are significant safety assurance challenges that are posed by the reconfigurable and modular nature of smart factories, which need to quickly adapt to production line changes and minimize downtime due to factory modifications. This could weaken the confidence in the safety of the factory and result in a reduction of the overall safety case.

Overall, there is a growing necessity in providing and assuring a deterministic real-time exchange of information to guarantee safe operations.

1.2 Problem statement

Network monitoring guides network operators in understanding the current behavior of a network. Therefore, accurate and efficient monitoring, e.g. collecting up-to-date information about the traffic load, performance parameters or potential problems is vital to ensure that the network operates according to the intended behavior as well as to troubleshoot any deviations. However, monitoring network can generate a large amount of data and consume network resources, especially in large and dense networks.

The amount of data generated by the network monitoring system grows with the size of the monitored network and the number of metrics collected. This issue becomes challenging in real-time networks, where network bandwidth and CPU utilization are critical resources. A significant proportion of the resource consumption of monitoring systems is related to switching between user and kernel space. More detailed explanation will follow in 2. Moreover, monitoring real-time networks deployed in mission-critical fields requires collecting more accurate measurements than conventional network setups. The traditional network monitoring systems deliver relative results that are sufficient to the regular user. However, these tools might not be suitable for I4.0 applications because they aren't built to meet industrial real-time network requirements.

To tackle these issues there have been many approaches such as utilizing machine learning models in delay prediction [p3]. Another approach proposes a monitoring model based on a subset of nodes in the network and therefore reduce the amounts on collected and processed data [p4]. These approaches do reduce the network overload and provide faster network response in some sense. However, they do not deliver absolute knowledge about the whole network topology due to the performance optimization trade-offs and therefore providing less reliable results.

In this thesis, we study utilizing the extended Berkeley Packet Filter (eBPF) technology in monitoring real-time networks [p5] to reduce resources overhead, e.g. CPU, memory, I/O operations by reducing the computations needed while also delivering results at earlier stages of packet reception following the latest methods in network monitoring [p6], addressing the challenges in monitoring real-time networks [p7].

Some of the features of an ideal real-time network include:

1. High Performance: The network must guarantee high throughput and low latency in order to meet real-time requirements of complex applications.

2. Determinism: A network is deterministic when there is little to no jitter during packet transmission.

3. Fault Tolerance: An important criterion for any safety-critical system is to have high reliability. A safety-critical network must be able to tolerate both permanent and temporary faults without leading to catastrophic results.

80 4. Unified Network: A single network is expected to carry different classes of traffic (critical, sub-critical and non-critical traffic) within the same medium.

To help quantify these essential features a real-time network monitoring system that collects and analyses measurements from the network nodes while minimizing the impact on normal traffic and ongoing network operations in order to maintain the
85 real-time network features is needed.

Network management involves controlling and monitoring packets that travel to and from a specific host and network. According to Dominik Scholz et al. [p5] the application itself should be able to determine its interpretation and processing of a network packet and what information to send back in theory. However, current implementations do not
90 do so for four reasons [p5].

Scholz' work suggests that one way to reduce overhead and traffic is to filter as early as possible. This can be done by having the applications developers bundle the policies with the application. Furthermore, the system administrator would not have to manage large amounts of varying and constantly changing configuration files, and could instead
95 focus on other important tasks. Ultimately, modern applications must meet performance and latency requirements, which can complicate policy implementation and verification.

The Linux kernel provides a framework for user-space packet filtering, which is used by the user-space firewall Iptables UFW. In the case of Iptables UFW, the Netfilter framework is used to filter packets in user space. The user-space firewall Iptables UFW
100 is a command-line tool that provides a simple interface to the Netfilter framework, which allows implementing various networking-related operations using custom handles. It is used to configure the firewall rulesets on a Linux system.

Previously, all host packet filtering rules were installed centrally in the kernel. In order to configure and maintain a centralized rule set, e.g., iptables or nftables, root
105 access is necessary. Even before rejecting a packet in kernel space, each packet is already copied to memory and undergoes basic processing. To mitigate these problems, there is a trend toward either moving the packet filtering to a lower level or separating the ruleset into parts and moving them into user space.

The extended Berkeley Packet Filter eBPF can significantly reduce the amount of
110 traffic generated in the data collection phase as well as the processing power needed in filtering and processing ingress/egress packets by extending the access to kernel-level functions and reducing the context switches since eBPF code runs in a sandbox within the kernel. This also allows capturing important packets at earlier stages. Additionally, eBPF code is event-based, rather than sampling-based. This makes eBPF-based observability
115 tools much more accurate and more efficient than traditional sampling-based alternatives because the program only runs when triggered by an event. Aside from the eBPF [p8], the integration with the Linux eXpress Data Path (XDP) for early access and dropping of incoming network packets, to gain more accurate measurements from the network while minimizing the traffic and processing load since the processing occurs in the kernel at
120 very early stages thus requires less jumping between user-space and kernel-space. More details about eBPF are found in section 2.

1.3 Assumptions and Scope

Future industrial communications involve high data rate best effort traffic working alongside real-time heterogeneous traffic for time-critical applications with hard deadlines.
125 This work addresses some challenges associated with monitoring networks of that type.

The expected result of this thesis is a software design of a benchmarking and monitoring system for real-time networks. The system benchmarks the network performance and records the results. The results can be used to identify any problems with the network and to improve the performance. This work distinguishes itself from other works by focusing on low-level operating system challenges involved in real-time (RT) network traffic filtering and by utilizing eBPF and XDP in the process of packet filtering to reduce resources consumption and gain more accurate measurements. Packet filtering follows two stages to study the packets: packet capture and packet analysis. Packet capture involves capturing packets from the network. Packet analysis is used to examine the packets and extract the relevant information. eBPF contributes to both of these stages. The proposed system has been implemented and tested on a Linux platform. Further testing on different environments hasn't been done.

1.4 Objectives and Contributions

The benchmarking and monitoring system proposed is intended to be used in future works to enable RT network monitoring. A hard real-time system guarantees that tasks will always meet deadlines, providing a high degree of reliability. The proposed concept is designed to be extended to meet different applications or needs. This system targets the measurement and control of network flow in real-time while having minor impact on hardware resources.

1.5 Methodology and Outline

The structure of the following sections is as follows:

In Chapter 1, the context, research area, and application areas of the study are described. This chapter also introduces the reader to the problem this work is dedicated to solve, the expected results and scope.

Chapter 2 reviews the relevant literature as well as the related state-of-the-art network technologies and presents existing monitoring challenges and solutions for RT networks, followed by related work helping addressing the problem.

Chapter 3 provides requirements analysis and describes the research methodology used in the study.

In chapter 4, the approach and concept are presented by describing the findings of the study, software architecture, and a summary.

Chapter 5 describes the implementation of the real-time monitoring system based on the requirements analysis. It provides a general and more in-depth overview of the software dsign, finished with conclusion.

In chapter 6 the concept and implementation are evaluated through experimental setup followed by comparative analysis based on the requirements analysis output followed by conclusion.

Chapter 7 summarizes the thesis results and provides an overview of possible future work.

165

Chapter 2

State of the Art

2.1 Modern network technologies 5
170 2.1.1 Time-sensitive Networking (TSN) 6
2.1.2 5G 7
2.1.3 Software-defined Networking (SDN) 7
2.1.4 Fieldbus 8
2.2 Network monitoring 8
175 2.2.1 Collection layer 9
2.2.2 Representation layer 9
2.2.3 Report layer 9
2.2.4 Analysis layer 9
2.2.5 Presentation layer 10
180 2.3 Packet filtering in Linux machines 10
2.3.1 Hardware Level 10
2.3.2 Network Level 11
2.3.3 System Level 11
2.3.4 Application Level 11
185 2.4 The Berkeley packet filter eBPF 11
2.5 The eXpress data path XDP 12
2.6 Conclusion 13

190

We begin this chapter with a brief introduction to the latest network technologies that could benefit from this work. The chapter also discusses modern network monitoring practices and challenges associated with monitoring RT networks. In the last section of this chapter we will discuss packet filtering on Linux machines, eBPF and XDP as
195 well as their use in RT network monitoring. The determining research question and the problem addressed in this thesis are summarized in the end of this chapter.

2.1 Modern network technologies

Industrial automation will become more efficient with the help of advancements in technology, including higher bandwidth and ultra-low latency provided by 5G and TSN.

Ethernet has been upgraded over the years to support these needs, and TSN is being developed as a more reliable, deterministic protocol than Ethernet. For real-time systems, determinism is an essential requirement, and TSN is a crucial component for managing traffic between devices. As machines become more integrated and communication between machines becomes essential, the convergence of IT and operational technology communications becomes more essential. Industrial automation will eventually benefit from the increased bandwidth with the advent of 5G and eventually 6G.

Networking technology had to evolve to keep up with the increasing demand for bandwidth. The Internet and Wi-Fi are now being used to perform a wide range of tasks, including live streaming video, virtualization, cloud computing, and IoT devices. This has resulted in a demand for more bandwidth. With 5G, the level of speeds and bandwidth necessary for supporting fields like autonomous driving and the internet of things will be greatly improved, while TSN will manage data traffic between the devices and the network.

This section presents some of the latest network technologies and the challenges associated with them. It also introduces the role of network monitoring.

2.1.1 TSN

Time-triggered communication protocols are increasingly deployed for safety critical applications because of the high predictability of message transmissions. In a time-triggered communication network, each node is assigned a time slot in which it can send a message. The messages are transmitted in a predefined order, and the receiving node can only process the messages that are received in its assigned time slot. This guarantees that the messages are processed in the order in which they are transmitted, which is essential for safety-critical applications. The time-triggered communication protocols provide a number of advantages over traditional communication protocols, such as Ethernet. First, the time-triggered protocols are deterministic, which eliminates the need for retransmissions and ensures that the messages are processed in the order in which they are transmitted. Second, the time-triggered protocols provide a high degree of reliability, as messages that are not received in the assigned time slot can be retransmitted. Finally, the time-triggered protocols are scalable, and can be used for large-scale systems. Examples of communication systems based on the time-triggered mechanisms include SAFEbus [p9], TTCAN [p10], TTEthernet [p11] and TSN [p12].

TSN is a set of standards that extends Ethernet capabilities to support hard real-time communication. To guarantee highly synchronized communication with very low latency and jitter values, IEEE 802.1 Qbv is proposed based on the time-triggered communication paradigm. Combining this with the priority classes defined in IEEE 802.1Q[p12].

TSN is mainly used in industrial and other applications that require guaranteeing real-time communication. In networked control systems, the performance of industrial Ethernet networks must be evaluated in real time. There have been efforts to monitor TSN networks. However, existing work about network monitoring cannot meet the real-time monitoring requirements of TSN, such as network state information accuracy and time synchronization [p13]. For example, the IEEE 802.1 Time-Sensitive Networking Task Group (TSN TG) is working on a standard to ensure the real-time performance of Ethernet networks. The TSN TG has developed the IEEE 802.1Qbv standard, which defines the behavior of traffic flows in TSN networks. The standard defines how to reserve bandwidth and ensure that deadlines are met. The existing work on real-time performance monitoring of TSN networks is mainly focused on the detection

of congestion and the estimation of end-to-end latency. However, these methods are not accurate enough to meet the monitoring requirements of TSN networks. In particular, the accuracy of the estimated end-to-end latency is not good enough to support the design of networked control systems. There is a lack of a comprehensive framework for the real-time performance monitoring of TSN networks. The existing work does not consider all aspects of real-time performance, such as the accuracy of state information and the synchronization of clocks. A comprehensive framework is needed to support the design of networked control systems.

2.1.2 5G

5G networks are expected to be much more complex than earlier generations of networks. They use a wider range of frequencies, and require more antennas and base stations to cover a given area. 5G networks will likely be used for a wide range of applications, including industrial IoT, autonomous vehicles, and 5G-based home broadband. This will make it more difficult to determine which applications are causing network congestion or performance problems, making monitoring 5G networks much more difficult than traditional networks. In addition, 5G networks are expected to be more dynamic, as antennas and base stations will be deployed and removed more frequently. This will make it difficult to maintain an accurate view of the network performance and identify problems. To address these challenges, network operators will need to adopt new technologies and techniques for monitoring 5G networks as existing network monitoring systems are unable to support the dynamic topologies and new technologies of modern networks, such as cloud networks and virtual environments. SDN and Network Function Virtualization Network Function Virtualisation (NFV) can be used to address these challenges and realize 5G networks by making 5G networks more flexible, scalable, open, and programmable. However, the use of SDN and NFV brings new challenges to network monitoring and troubleshooting. SDN and NFV can change the network topology and the way that traffic flows through the network. This can make it difficult to determine where problems are occurring. Additionally, SDN and NFV introduce additional vulnerabilities and risk of outages with their increased complexity and resource requirements.

Celdrán et al. [p14] propose a solution to efficiently orchestrate the monitoring services using SDN and NFV. However, this work does not discuss the issue of resource consumption due to the high level of virtualisation, especially when using OpenStack to instantiate the virtual resources. This is particularly crucial for networks with large traffic.

Another work by Thiago et al. [p15] discusses the drawbacks of using DPDK technology for fast packet processing in edge nodes running on 5G networks, in terms of excessive resource occupation. Thiago's solution investigates OS level packet processing mechanisms and proposes utilizing eBPF and XDP for lower resource consumption and higher throughput.

2.1.3 SDN

SDN technology is an approach to network management that enables dynamic, programmatically efficient network configuration, simplifying the implementation of network monitoring solutions. This technology is used in [p16] and [p17]. There have been many efforts to monitor SDN network health in real-time using different techniques. One work [p18] overloads the SDN controllers using tools such as Cbench [p19] and

hcprobe [p20]. Another work [p21] uses log data collected from OpenFlow switches, host machines and the SDN controllers in a non-invasive style to monitor SDN network health under network overload as well as a security attack. However, these works only test the network using active probing and thus cannot detect anomalous behavior in the network. Using such techniques heavily affect the network performance and thus cannot be used in a production network.

2.1.4 Fieldbus

Fieldbus delivers guarantees for safety-critical distributed real-time applications operating with strict temporal constraints that rely on deterministic networks with low latency and jitter. Traditional fieldbus networks require additional error detection and avoidance mechanisms, thus they are not suitable for safety-related controls [p22]. On the other hand, conventional networks have appropriate error detection and correction methods, yet and without modification they lack the ability to independently and rapidly detect safety failures caused by the network, the connection, or the device in between. An independent safety software layer is necessary to detect connection failures in order to be able to implement the required emergency shutdown action to avoid danger [p23].

2.2 Network monitoring

Network monitors analyze data traffic and identify problems with the network. They also help to optimize network performance and troubleshoot issues. Network monitors use a variety of tools to collect data, including packet sniffers, flow monitors, and protocol analyzers. This section is dedicated towards discussing the latest methods and challenges in network monitoring. The monitoring operations build a network model that describes the network's overall behavior and represents the operational status of the network, which is used for troubleshooting and future planning of network design or/and network configurations. Lee et al. [p7] divides the network model into five measurement phases/layers as in figure 2.1.

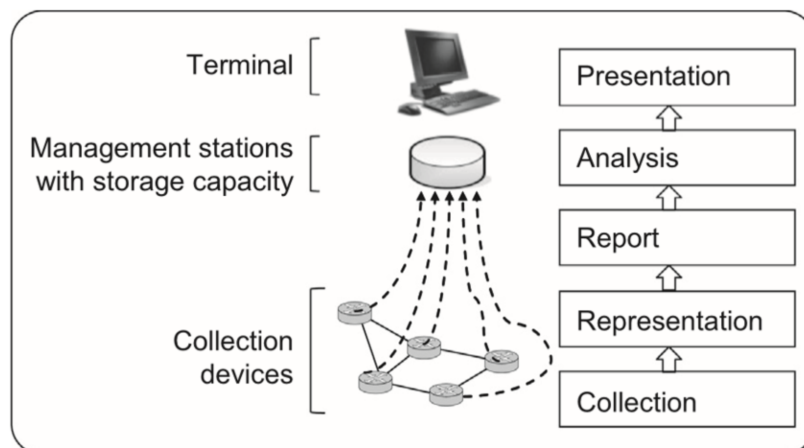


Figure 2.1: the monitoring model [p7]

320 One of the main challenges in RT network monitoring is improving efficiency of the monitoring process. Depending on the application area and size there could be a vast amounts of data to collect, analyze and store. This problem becomes more important for safety-critical applications that have less tolerance towards resources insufficiency. As mentioned in figure 2.1 generally the monitoring model consists of five layers dividing 325 the responsibilities. We discuss the roles and challenges of each layer below [p7].

2.2.1 Collection layer

Active and passive monitoring are two ways of collecting measurements from a network. In active monitoring, test traffic is injected into the network, whereas passive monitoring simply observes the traffic that is already passing through the network. Activate moni- 330 toring is useful for directly measuring certain parameters, such as bit loss, latency, and throughput. However, it can also lead to high bandwidth consumption, which can be mitigated by adjusting the size and frequency of the active probes. In passive monitoring, no additional traffic load is generated, but it can take a long time to observe the behavior of the network. Device information, such as CPU and memory usage, can be collected 335 passively. Some works use both methods to gain a more complete understanding of the network [p24] [p25].

2.2.2 Representation layer

In this layer the measurements from all contributing nodes are standardized and synchronized. NETCONF [p26], and YANG [p27] for example provide standards for this 340 purpose. One of the major issues in this layer is to synchronize the collection of measurements from heterogeneous and distributed devices in time.

2.2.3 Report layer

Data collected by the monitoring system is transformed into consumable information by the report layer. This might mean summarizing data into graphs or tables, or sending 345 alerts when thresholds are crossed. An easy-to-use and understand report layer is often an essential part of a monitoring system. In real-time networks, it can be challenging to transfer huge amounts of data at this layer. There are many ways to reduce the amount of data that needs to be sent in order to report measurements, such as aggregating similar measurements or using bandwidth-saving encodings [t1]. The frequency of polling 350 requests also needs to be considered; adjusting the frequency allows for a trade-off between accuracy and efficiency.

2.2.4 Analysis layer

Network traffic analysis is the process of inspecting and managing the traffic passing through a given network. There are a variety of different functions that can be performed 355 as part of this process, including general-purpose traffic analysis [p28], estimation of traffic demand, traffic classification per application, mining of communication patterns, fault management, and automatic updating of network documentation [p7]. By performing these functions, network administrators can better understand the traffic patterns within their network and identify any potential issues.

2.2.5 Presentation layer

Operators use various tools to visualize network traffic, depending on the type of analysis required. One of the biggest challenges is to visualize huge amounts of data in real-time without leading to a service crash or delay. Network monitors are tools used to track and display network activity in real-time. They can show where traffic is coming from, going to, and the volume of traffic. Traffic analyzers are used to examine traffic flows and performance between two points in a network. They can help identify bottlenecks and optimize network traffic. Network mapping tools help create a visual representation of the network topology, which can help with troubleshooting and security analysis.

2.3 Packet filtering in Linux machines

Linux packet filtering provides a basic layer of security for the network. By allowing or denying certain packets, it can control which traffic is allowed to pass through the network and which is not, helping to protect the network from unauthorized access and attacks. Furthermore, traffic control can also prioritize certain types of traffic over others, improving the performance of RT networks with multiple priority classes. Packets can be filtered at different stages on their way from the physical network interface until they reach the application. Dominik et al. [p5] divides packet filtering into four levels, from the lowest in abstraction to the highest as in figure 2.2.

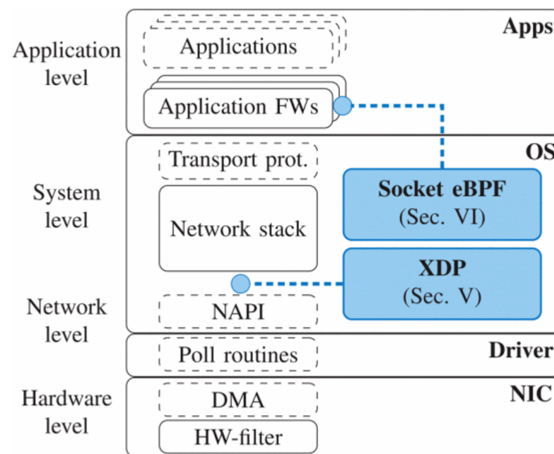


Figure 2.2: Levels of packet filtering [p5]

2.3.1 Hardware Level

First, packets are filtered on the network interface card (NIC). Modern server NICs that include hardware offloading and filtering can be configured using driver parameters and ethtool tools. Its functionality is limited and largely determined by the vendor and NIC used, which means there is no standard interface for configuring it.

2.3.2 Network Level

Network level firewalls filter packets before they are processed by the routing subsystem. Compared to a system-level firewall, this is more efficient due to fewer CPU cycles being consumed by dropped packets.

2.3.3 System Level

The second level is in the kernel, where the network stack resides. The kernel provides a set of hooks that can be used to intercept and filter packets. Input hooks are placed at different locations in the processing, such as in the routing subsystem of the Linux kernel before the application is launched. However, this must be done with root access and system-specific knowledge because different rules may interfere and the application cannot be shipped with its own packet filter. This level is well-defined and relatively easy to use, but it is also low-level and provides no help in configuring filters using a dedicated filtering device in front of the firewall, like a stateless packet filter or a Layer 4-7 switch. This has the advantage that all traffic is processed by the filter, which can lead to higher performance and lower load on the firewall. However, this approach requires more complex configuration and can be more expensive.

2.3.4 Application Level

This level is in user space, where a variety of tools can be used to configure packet filters. These tools include the ip command, iptables, and various firewall management frameworks. They provide a high-level interface for configuring filters, but they are also complex and difficult to use. A firewall at the application level looks at traffic directed to a particular application. With systemd's socket activation, application developers can include application-specific eBPF-based packet filters in sockets to allow their application to include packet filtering rules that can be deployed along with their application. This approach has the advantage that it is generally available and can be run on a wide range of devices. However, performance can be lower than with dedicated hardware filters.

2.4 The Berkeley packet filter eBPF

The Berkeley packet filter eBPF is a recent technology available in the Linux kernel, which extends the user capabilities to control kernel-level activities. It is an instruction set and an execution environment inside the Linux kernel, enabling modification, interaction, and kernel programmability at runtime. The eBPF-based packet tracing is utilized for monitoring tasks and network traffic in real-time. Although it is commonly used for building proof-of-concept applications, it has proven to be challenging to extend those applications to more complex functionality due to its limitations[p8]. eBPF allows user-space applications to inject code in the kernel at runtime, i.e., without recompiling the kernel or installing any optional kernel module. This results in a more efficient system.

The figure 2.3 shows the flow of eBPF programs from execution at user-space to the injection of network tracing points inside the kernel. The eBPF program gets compiled using Low-level virtual machine (LLVM) compiler and then attempts to load the compiled eBPF bytecode into the kernel after passing eBPF verification process first. The eBPF context runs then inside the eBPF virtual machine. The eBPF program is triggered on an event by attaching hooks in the kernel. These hooks are:

- Tracepoints: Similar to breakpoints, but instead of stopping execution when a breakpoint is hit, log information are collected for debugging reasons. Tracepoints are static locations in the kernel, e.g. disk IO operations.

- Kprobes: kprobes are a facility in the Linux kernel that allow to dynamically execute code on a kernel function. It can be used to analyze the behavior of the function, or to patch it in order to change its behavior.

- Uprobes: uprobes are similar to kprobes but allow to dynamically execute code on a user-space function.

- Perf events: Provide a command line tool and subcommands for various profiling tasks allowing to collect performance data about the running kernel by sampling or doing monitoring counts. This data can be used to identify performance bottlenecks, or to diagnose other performance issues.

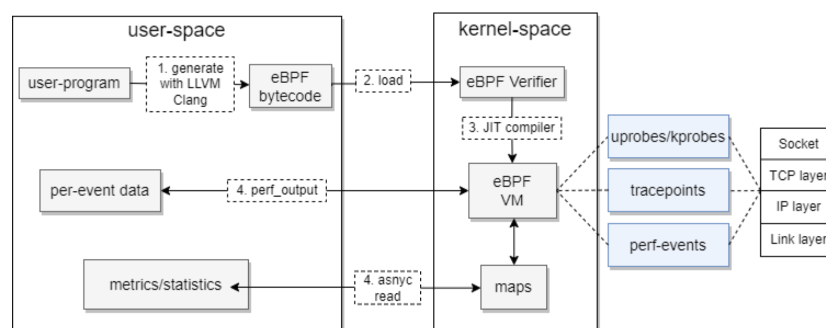


Figure 2.3: eBPF program flow [t2]

The importance of eBPF and XDP is highlighted by its fast adoption since its introduction in the Linux kernel in 2014 by both industry and academia. Their use cases have grown rapidly to include tasks such as network monitoring, network traffic handling, load balancing, and operating system insight.

2.5 The eXpress data path XDP

The eXpress data path XDP is fast programmable packet processing framework in the operating system's kernel. It is the lowest layer of Linux network stack. It allows installing programs that process packets into the Linux kernel. These programs will be called for every incoming packet.

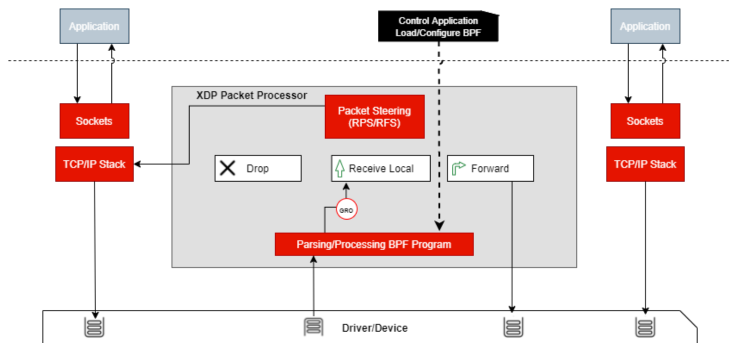


Figure 2.4: XDP packet processing overview [t3]

XDP allows adding or modifying programs without modifying the kernel source code, whereas eBPF programs modify the (programmable) kernel operation in runtime without requiring recompilation of the kernel [p29]. eBPF can also be used to program the XDP in order to process packets closer to the NIC for fast packet processing. Developers can write programs in C or P4 languages and then compile to eBPF instructions, which can be processed by the kernel or by programmable devices (e.g., SmartNICs).

```
#include <linux/bpf.h>
#include "bpf_helpers.h"

SEC("drop-all")
int drop(struct xdp_md *ctx) {
    return XDP_DROP;
}
```

No main function. Starting point indicated by ELF section to be loaded

Libraries containing most eBPF structs, definitions and helper functions

Return value defined by enum in *bpf.h*. Hook-specific.

Context passed to XDP programs. Contains pointers to packet data + metadata. Hook-specific.

Figure 2.5: eBPF XDP program example in C language [p29]

Figure 2.5 gives an overview example of an **??** program in C. This program drops all incoming packets at the driver level. A more complex program can be implemented by adding rules and conditions on packet dropping.

2.6 Conclusion

Benchmarking and monitoring RT networks require instantaneous response while ensuring minimal bandwidth, CPU, and memory consumption to provide accurate view of the network state. Some key features of a real-time monitoring system are low latency and high throughput, ability to handle large data sets, efficient use of system resources, fault tolerance and scalability. Some works suggest [p30] applying machine learning methods or reducing the amount of collected data or the number of nodes under monitoring isn't ideal in industrial networks where network communication needs to be deterministic. An optimal solution for monitoring RT networks would be to collect the necessary data without compromises that might affect the network's determinism by removing nodes from the monitoring system or applying stochastic evaluation using machine learning. Another alternative approach [p16] to reach real-time capability does not bring along general applicability and flexibility and lacks the concept of determinism.

Traffic collection and processing requires priviliged access to kernel sections from user-space. Such priviliged access is costly in terms of context switching and jumping between kernel-space and userspace. The isolation between user-space and kernel-space forms a hurdle of achieving real-time network monitoring. Each operating system or hardware handles the unprivileged access to priviliged resources differently. For example, a so called protection ring is implemented in the x86 CPUs making different levels of access to resources through hierarchical protection domains 2.6.

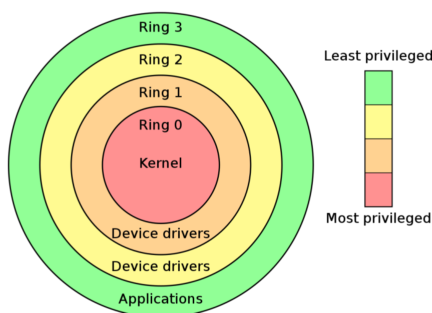


Figure 2.6: privilige rings for the x86 available [p31]

These hardware and OS-based isolation mechanisms provide security to the hardware access with the disadvantage of performance reduction. Since network adapters are IO devices they require priviliged access, thus affecting computation overhead and response speed negatively. Capturing and filtering network packets is an essential part of implementing any network monitoring system. However, doing such operations in the network require priviliged access.

The eBPF technology solves this problem with its packet tracing capabilities by allowing capturing packets at an early stage of packet reception. eBPF also offers a flexible and safe programmable environment inside the Linux kernel allowing loading and modifying eBPF programs during runtime and interacting with kernel elements such as kprobes, perf events, sockets, and routing tables [p32]. This feature increases the overall security level and makes the system more flexible by dynamically replacing programs in a live monitoring system. Additionally, The eBPF code compiles to a variety of CPU architectures, making eBPF programs scalable and interoperable.

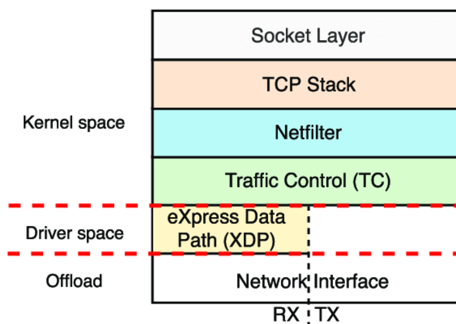


Figure 2.7: Linux kernel network stack[p29]

The figure 2.7 shows how network packets entering the operating system are processed by many layers in the kernel before reaching its destination application. Generally, higher layer has a lower performance of networking hooks. According to Vieira et al. [p29] all network packets heading to a userspace application pass through these layers and can be intercepted and modified by modules like iptables, which is found within the Netfilter layer. There are several places in the kernel where eBPF programs can be attached to enable packet filtering.

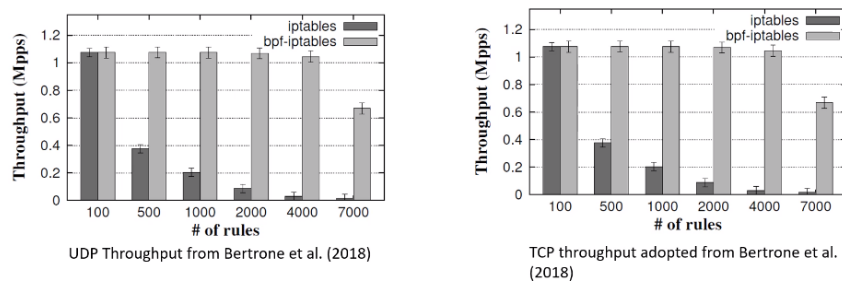


Figure 2.8: bpf iptables performance comparison [p33]

The work of Betrone et al. [p33] proves that packet filtering using eBPF brings a significant improvement on the number of packets processed per second as illustrated in figure 2.8. This reduces the resources overhead in terms of computational power as well as bandwidth consumption. Packet filtering is important in monitoring deterministic RT networks because it allows for the inspection of packets as they are transmitted on the network, which can help reducing resources overhead due to unnecessary packets or redundant requests. Additionally, packet filter allows for the identification of any issues that may occur with the transmission of packets, which can help to ensure that the network remains operational and that data is transmitted accurately. Implementations of RT network monitoring systems that focus on operating system (OS) processes such as packet filtering are still missing.

In this chapter, an overview of modern network technologies is provided, along with the role of network monitoring in designing and maintaining networks. Afterwards, network monitoring is described and a five-layer classification model of monitoring operations is provided. It includes an analysis of the monitoring functions of each layer, with particular focus on how they affect the network. Following this discussion, the paper analyzes packet filtering and its role in collecting network information for monitoring systems, as well as the contribution of eBPF and XDP to dealing with real-time monitoring issues. The next chapter presents an analysis of the requirements for implementing a system that monitors network traffic in real time, addressing the challenges from the OS side.

CHAPTER 3

REQUIREMENT ANALYSIS

520

3.1 Top-level requirements 17
3.2 System-level requirements 18
3.3 Feature-level requirements 19

525

This chapter presents a requirements analysis following a top-down approach to determine general, intermediate, and specific requirements. First, an overall high-level 530 requirement analysis identifiying the high-level requirements is presented. A more in-depth system-related requirement analysis of the approach follows. The process finishes with a less abstract explanation of each component in the approach.

3.1 Top-level requirements

Top-level requirements describe the needs of the monitoring system that are not specific 535 to any technology or implementation. For instance, the real-time network monitoring system needs to be able to identify and isolate network issues in order to allow for timely resolution. Additionally, the system needs to be able to provide an overall view of the network health in order to identify issues before they become major problems.

Table 3.1: Top-level requirements

#	Description
TL1: Real-time capabilities	Describes how the network monitoring and reporting tools need to provide performance insights into the network in real time. This helps identify performance hiccups early and avoid potential outages.
TL2: Comprehensive monitoring capabilities	This requirement implies using a single network monitoring tool to monitor a variety of network components and various operating systems, since using separate tools would require constant management and incur additional resource costs.

Table 3.1: Top-level requirements

#	Description
TL3: Scalability	A network monitoring system needs to be scalable, i.e. more flexible to changing requirements or needs without compromising on the quality of monitoring. As networks grow, their scalability helps ensure that their performance doesn't degrade significantly, regardless of their size, as productivity rises, needs change and their adaptations keep pace. Scalability is achieved through network monitoring tools that can be easily deployed and managed in a distributed environment, and can be scaled up to support a large number of devices.
TL4: Automation	Automation makes network monitoring solutions far more useful by saving time and resources. Automation allows network monitoring tools to react based on threshold values or predefined rules/criteria. By automating monitoring, the tools can spot and resolve problems automatically in a proactive fashion, send alert notifications, forecast storage growth, and more.
TL5: User management	In order to monitor the inspected network effectively a user-friendly user interface module is required. This module also allows the users through an interface to interact with the monitoring system as a whole and give the user the ability to specify own tests.
TL6: Visibility	Visibility is the ability to see what is happening in the network. It is important to have a clear view of the network traffic and the devices that are connected to it, so that users can identify and troubleshoot problems quickly.
TL7: Reporting	Reporting is a key usability aspect of network monitoring tools. Reporting is the process of generating reports based on the data collected by the network monitoring tools. Reporting is a key usability aspect of network monitoring tools, as it allows the network administrator to generate reports based on the data collected by the network monitoring tools.
TL8: Alerting	Alerting is the process of notifying the network administrator of an event. Alerting is an important aspect of network monitoring, as it notifies the administrator of a potential problem. Alerting is a key usability aspect of network monitoring tools, as it allows the network administrator to take action, based on the alert.

3.2 System-level requirements

System-level requirements are the specifications that are related to the environment the monitoring system is running in. 540

Table 3.2: System-level requirements

#	Description
SL1: Low resources consumption	Monitoring systems must not overload any of the resources. Resource unavailability could create a bottleneck in the system and cause unexpected behavior. This is crucial to ensure the stability and integrity of the monitoring system.
SL2: Interoperability	Interoperability is often achieved through the use of standards. For example, the Simple Network Management Protocol (SNMP) is a standard that allows different network management systems to exchange information.
SL3: Security	There are many aspects of network security that can be posed by the monitoring system, including hardware, software, viruses, spyware, vulnerabilities and other factors that can compromise the integrity of a network. The monitoring system needs to stay up to date with the latest security patches and software updates.

3.3 Feature-level requirements

Feature requirements are specific features or functions that a software system must provide as part of the overall software. They are used as the basis for design, development, and testing.

Table 3.3: Feature-level requirements

#	Description
FL1: Dashboard	Dashboards are a key usability aspect of network monitoring tools. Dashboards are a visual representation of the data collected by the network monitoring tools. Dashboards allow the network administrator to quickly understand the status of the network supported by visualizations.
FL2: Data persistency	In order to guarantee data persistency a database is required to be able to retrieve the data in case of unexpected failure. Time-series databases are highly optimized for real-time monitoring applications. The architecture of time-series databases makes them particularly suitable for real-time monitoring applications since they possess important architectural features such as time-stamp data storage and compression, data lifecycle management, data summarization, and time series aware queries.

Chapter 4

Concept and Approach

4.0.1 Introduction 20
4.1 General discussion 20
4.2 Software Architecture 22
4.3 Conclusion 24

Chapter 4

4.0.1 Introduction

Chapter 2 discussed the challenges of network monitoring in real time and the role that packet filters and OS low-level functions play in tackling some of these challenges. Chapter 3 outlined the requirements for implementing a real-time monitoring system. This chapter presents the concept and approach of the real-time network benchmarking and monitoring system based on the requirements listed in the previous chapter 3.

First, the general approach is described and a broad overview about the system components and their functionalities, is provided. Next, the software architecture of the real-time network benchmarking and monitoring system will be described based on the top-level requirements in chapter 3 along with the integration of eBPF and XDP into the system.

4.1 General discussion

The proposed idea is to design an end-to-end network benchmarking and monitor system relying on aggregated time series generated from network nodes. Time series hold information such as round-trip time RTT, loss, available bandwidth and additional internal timestamps. The network monitoring system is optimized for real-time communications by integrating eBPF technology in the data collection process. This system offers a data visualization service to help read the network benchmarks. The figure 4.1 depicts a general overview of network monitoring systems based on Lee's work [p7]. This figure maps some of the top-level requirements listed in requirements section 3.

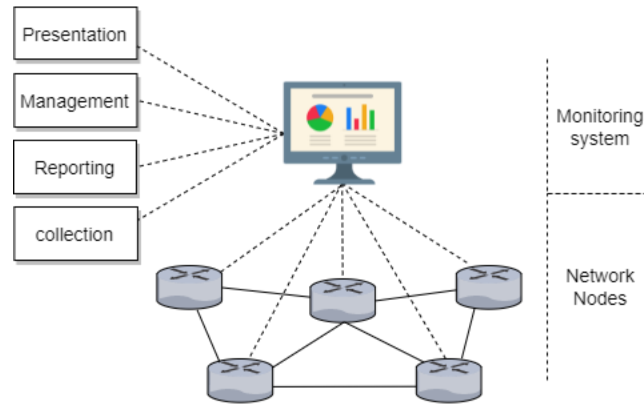


Figure 4.1: High-level diagram of network monitoring

Based on Lee's work, monitoring systems are conceptualized as a set of monitoring operations. These operations are classified into layers as a result of their hierarchical sequence. In our approach, the monitoring system is composed of four abstract layers. 580 The first one is the collection layer. This layer collects measurements from the network when new events occur, pre-processes and standardizes them. The data collection process involves gathering measurements passively from the network traffic and can also inject test traffic by creating sampling packets and sending them to available receiver nodes. The second layer is called reporting. In this layer, measurement data are exported 585 after collection and consumed asynchronously by administrative entities through data exporters. The data can be shared across multiple organizations through a virtualized infrastructure to allow scalability. In the third layer, data is managed, stored, and measurements are checked for integrity. This is the layer in which all system functions are controlled. Lastly, the presentation layer represents how the user interacts with the 590 system. According to Lee, It is easier to monitor network through visual representations, rather than through numerical data. This is because it is easier to identify issues and potential problems when they are represented visually. Additionally, it can be helpful to see how data flows through the network, and where congestion for instance is occurring. Figure 4.2 illustrates the monitoring system as a black box. More detailed insights over 595 the internal design will follow in the next section.

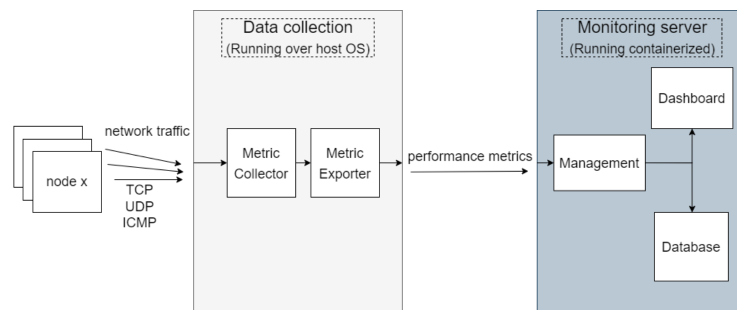


Figure 4.2: The monitoring system as a blackbox

4.2 Software Architecture

The software architecture of the monitoring system is depicted from the three levels of requirements in chapter 3. The core entities of the software are illustrated in figure 4.3. The entities are divided into two modules: DataAggregator and DataVisualizer.

The DataAggregator runs directly over the host OS and consists of four components. The kernelPerf is the metrics collection tool that is designed to measure network performance similar in its concept to netperf or tcpdump. It runs within the kernel at low-level of the OS. It consists of two main programs. The first is an eBPF-based program that injects multiple tracing points into the kernel when triggered by an event. This program passively collects network metrics only from incoming traffic. The second program utilizes XDP to drop sampled packets that have been generated by the PacketSampler for active probing purposes. As a result, redundant and unnecessary context switches within the OS along with internal network bandwidth are avoided, particularly in cases of heavy active probing.

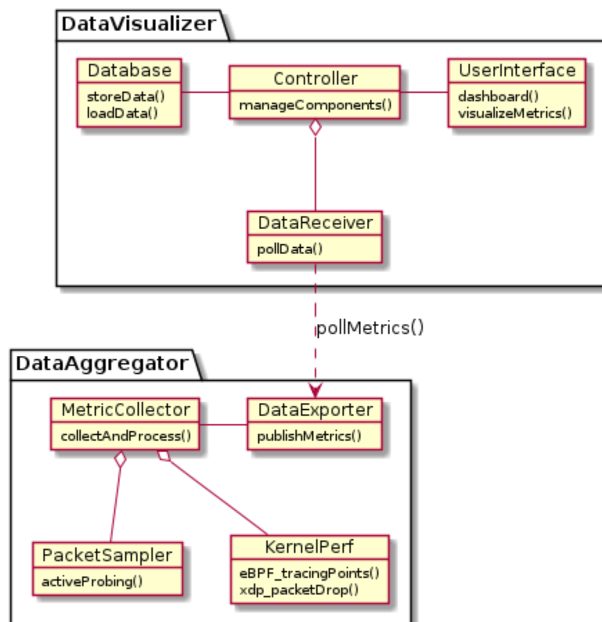


Figure 4.3: The software architecture object diagram

Since Linux kernel has only one XDP hook, only one XDP program can run at a time, so the program must own the XDP hook in order to use it. XDP is very effective in dropping packets as explained in Betrone's work [p33] in chapter 2. XDP is usually used to mitigate Distributed Denial-of-Service (DDoS) attacks and real-time intrusion detection (IDS).

The PacketSampler is called by the MetricCollector to generate and send ICMP probe packets to host nodes. The ICMP protocol is used by network devices to send error messages and control packets for debugging purposes. ICMP is therefore ideal for network monitoring in addition to not requiring any peer applications to run on nodes as in the case of TCP packets. SL2 requirement establishes the importance of

620 interoperability as a key element of a homogeneous network. In general, routers and NICs can handle ICMP packets differently, contributing to different packet loss rates.

Using active probing allows for a more accurate picture of the network state. If combined with XDP, the resources overhead can be reduced significantly. This can help meet the SL1 requirement. In addition, the MetricCollector is responsible for controlling 625 the amount of probe packets injected into the network to maintain network stability. Finally, the DataAggregator includes a DataExporter. This part collaborates with the MetricCollector to publish the collected and processed network metrics.

The DataVisualizer also consists of four main entities. The entities are managed and connected by a Controller. The Controller manages the Database where network 630 performance metrics are stored along with other metrics from the system, such as hardware resources consumption rates. The Datareceiver polls metrics published by the DataExporter periodically and feeds it to the Controller. Finally, the UserInterface gives the users the ability to manage the monitoring system through a dashboard, Create visualizations of the available data, set system alarms in case of specific events and 635 more. The interaction between the software components is illustrated in figure 4.4.

In the requirements section 3, TL3 specifies that the monitoring system needs to be scalable to accommodate various network topologies. This can be accomplished by deploying the DataVisualizer in a containerized environment, allowing for horizontal scaling by creating replicas or vertically by increasing the hardware resources as needed. 640 The entities are managed and connected by a Controller. The Controller manages the Database where network performance metrics are stored along with other metrics from the system, such as hardware resources status. The Datareceiver polls metrics published by the DataExporter periodically and asynchronously and feeds it to the Controller. Finally, the UserInterface lets users manage the monitoring system via a dashboard, 645 visualize the data, set alarms for specific events, and more. The interaction between the software components is illustrated in figure 4.4.

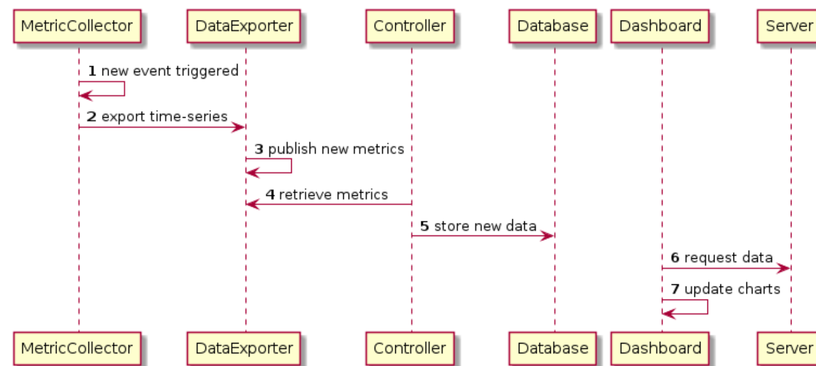


Figure 4.4: Sequence diagram of the monitoring funcionality from data collection to visualization

4.3 Conclusion

In this chapter a general approach is provided following the monitoring model of Lee et al. [p7]. The software architecture is described by presenting the core components of the software design and their interaction in details. The part of the software responsible of data collection utilizes eBPF and XDP technology as described in chapter 2. 650

The next chapter addresses the underlying problem of monitoring RT network traffic more precisely. The technologies used for the implementation are discussed, mapping the solution to the requirements defined in 3.

655

Chapter 5

Implementation

5.1 Introduction 25
660 5.2 Software Design 25
5.2.1 Real-time data collection 26
5.2.2 Active probing 27
5.2.3 Real-time data exporter 28
5.2.4 Visualization 29
665 5.3 Conclusion 32

5.1 Introduction

670 In this chapter we discuss the implementation phase following the previous chapter and the requirements set in 3. The Implementation targets the concept design and defines the steps required to turn the concept into a working system. The main goal of the implementation phase is to make sure that the system functions as intended and that all the requirements have been met. The first step in the implementation phase is to 675 create a detailed design for the system. This design will include a description of the system's architecture, the functionality of each component, and the interactions between the components.

5.2 Software Design

The implementation consists of two main modules, each module holds different com- 680 ponents as illustrated in figure 5.1. The DataAggregator consists of four components. The first component is the MetricCollector, which injects network tracing points in the kernel using eBPF. The second component is PacketSampler. This component generates probe packets. The third component is XDP-packetDrop. This component drops the probing packets at early stages of reception in the kernel. The final component is the 685 DataExporter. This component holds API endpoints to the collected network metrics.

The second module is the DataVisualizer. This module represents a Docker Compose YAML file. The DataVisualizer runs in a containerized environment over virtual

hardware, making the software scalable as required in TL3 3.1 depending on the network topology. This also gives flexibility to the user to deploy the module on any host machine. The DataVisualizer consists of four components, each of which represents a microservice. The first component is MonitoringServer. The component polls network metrics from the DataAggregator and stores them. Additionally, it has its own dashboard for various management practices, including binding metrics exporters, binding databases, and publishing data to third-party services. Next, there is the Visualization component. This component reads data from the MonitoringServer and allows the user through dashboard to analyse and visualize data in real-time. The third component is the Database. A fourth component, referred to as MetricsExporter, is abstract and unimplemented. One or more MetricsExporters may exist. New MetricsExporter services can be added to the DataVisualizer, which can then be binded to the MonitoringServer.

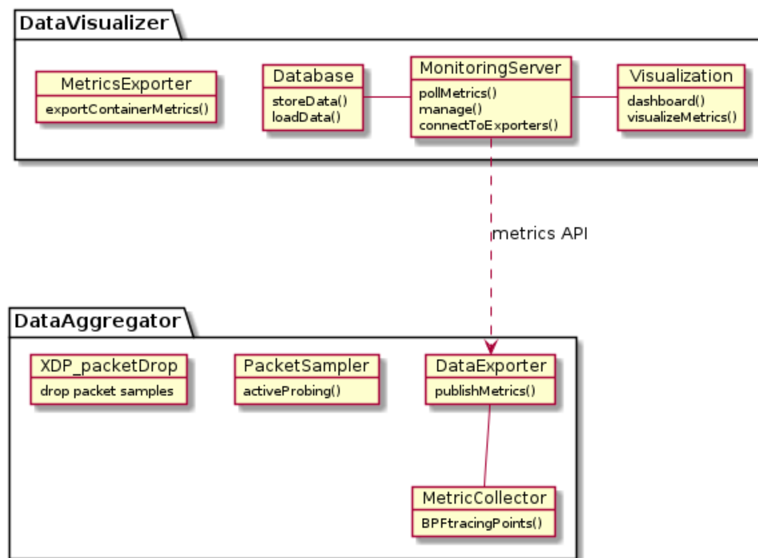


Figure 5.1: The monitoring system's architecture

5.2.1 Real-time data collection

The implementation of the MetricCollector consists of eBPF tracing programs using the BPF Compiler Collection (BCC) library running in a sandbox inside the kernel. These eBPF programs inject tracing points in the Linux kernel to track network events and are triggered on events, such as TCP connections, and process creations. eBPF programs allows reading the collected measurements from the user space, either by sending details per event or by accumulating the data and passing them via the BPF map asynchronously. BPF maps can support arrays, associative arrays, and histograms, and are suitable for passing summary statistics. The figure 5.2 illustrates how the eBPF program collaborates with the DataExporter to publish real-time network measurements from the kernel-space. These two components work in parallel.

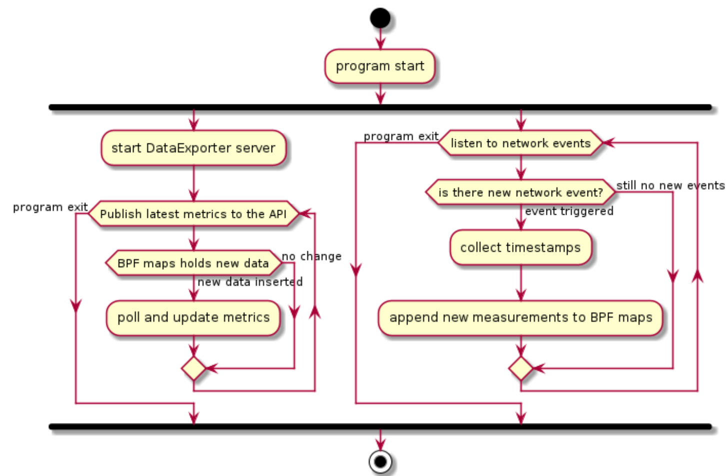


Figure 5.2: eBPF packet filtering and data exporting activity diagram

There is a variety of programming languages and frameworks to create eBPF programs. One of the most popular is called the Berkeley Compiler Collection BCC. BCC is a library used to create Berkeley Packet Filter (eBPF) programs that analyze network and OS performance without incurring overhead or posing security threats. BCC eliminates the necessity for users to know deep technical details about eBPF, and provides many ready-to-use starting points, such as the bcc-tools package containing pre-created eBPF programs. Additionally, BCC offers a Clang compiler capable of compiling BPF code at runtime, adapting it to the requirements of a particular target kernel, which facilitates development of maintainable BPF applications designed to be compatible with kernel changes. BCC allows writing eBPF programs in many languages such as C, C++, Python, Lua and go.

eBPF programs are very efficient since they operate in the kernel space and only trace kernel functions instead of tracing every packet and filtering it. They reduce time and overhead by avoiding unnecessary context switches.

5.2.2 Active probing

Due to the fluctuations in the passive network flow active probing is necessary to fill the gaps when less information about the network status is available because of lower traffic volume as Lee's work states [p7]. Active probing is essential in order to guarantee up-to-date real-time monitoring capabilities. Active probing is done by the PacketSampler. The MetricCollector informs the PacketSampler to create probe packets and send it to host node similar to pinging via ICMP packets. The MetricCollector collects network measurements such as latency, RTT, and jitter of the packets sent by the PacketSampler and the acknowledgment received back. These packets can be then dropped by the XDP-packetDrop component to save bandwidth within the network stacks and hardware resources by reducing the number of context switches in CPU. The activity diagram in figure 5.3 demonstrates the process of active probing and the components involved in it.

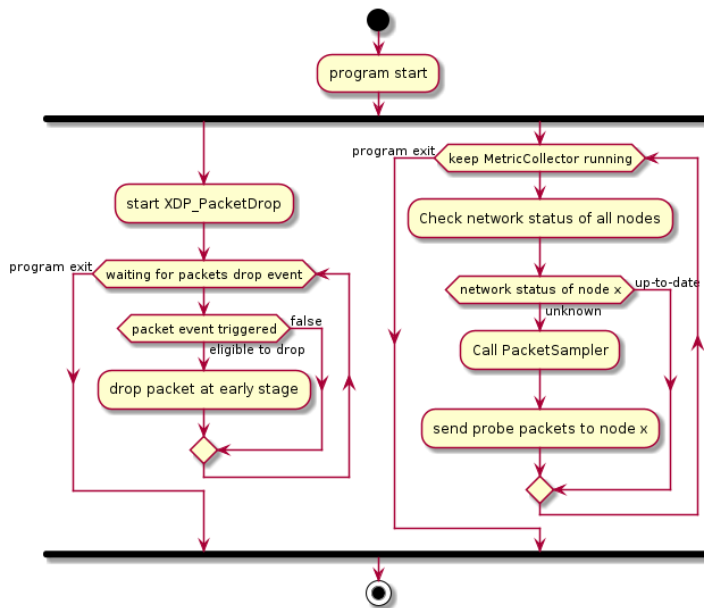


Figure 5.3: Activity diagram of active probing

5.2.3 Real-time data exporter

The DataExporter publishes up-to-date network metrics ready for external service to scrape asynchronously. The DataExporter uses Prometheus open source Monitoring framework to expose the metrics. Prometheus has a large ecosystem of off-the-shelf exporters. Prometheus exporters provide an interface between Prometheus and applications that don't export metrics in the Prometheus format. A good example is the node exporter, which exposes Linux metrics in Prometheus format. Another popular example is cAdvisor, which exports metrics from containers. An external service such as a Prometheus monitoring server reads the metrics exposed by the target using a simple text-based exposition format. When no exporter that fits the application needs is found, Prometheus provides client libraries that can be used to develop a custom exporter that translates the metrics in Prometheus format [p34]. There are official client libraries in go, python, java, and ruby. Additional unofficial client libraries for other programming languages can be also found [t4].

There are mainly four types of metrics that will help in instrumenting any application to export metrics:

- Counter: Counters represent a single monotonically increasing number that can only increase or be reset to zero upon restart, like a counter of HTTP requests.
- Gauge: Gauges are like counters except that their value can change up or down arbitrarily. For example, temperature or current memory usage.
- Histogram: A histogram collects and counts observations such as responses or request durations in configurable buckets. It also sums up all observed data points.
- Summary: A summary is similar to a histogram, except it also provides a total count of observations and a sum of all observed values. it calculates configurable quantiles over a sliding time window.

740 745 750 755 760

To instrument the data collection application with prometheus a data exporter using the official prometheus client library is designed. The data exporter establishes an http server to publish the metrics in prometheus text format as in figure 5.1. The data exporter then calls and starts the eBPF programs as imported mdules. eBPF programs require slight modification using one or more of the four metric types to aggregate data to the DataExporter.

Listing 5.1: data exporter example for eBPF metrics using Prometheus framework

```
1  from prometheus_client import start_http_server
2  import ebpf_program
3
4  # Start up the server to expose the metrics.
5  start_http_server(port_num)
6  # call eBPF program as module
7  ebpf_program.run_tcpconnlat()
```

The figure 5.2 is an example of an eBPF pseudo program modified to aggregate data to prometheus client. In the first lines Gauge is imported from prometheus-client library and an instance is created in line 2. The data collected by the eBPF program will be aggregated to the Gauge when a new event occurs.

Listing 5.2: eBPF program called by the data exporter

```
1  from prometheus_client import Gauge
2  latencyGauge = Gauge('tcp_connlat_msec', 'tcp_connection_latency',['saddr', 'daddr'] )
3
4  define BPF program context
5
6  attach probes to kernel
7
8  def run_ebpfProgram():
9      while True:
10         poll new measurements
```

The metrics will then be exposed at a specific port. The published metrics are updated in real time every time the page is reloaded. The figure 5.3 shows an example output of the format of the published metrics.

Listing 5.3: example output of published metrics

```
1  metric_x{destination_ip, source_ip"}␣25.383
2  metric_y{destination_ip,␣source_ip"} 0.106
3  # metrix_z as a counter
4  metric_z 397
5  .
6  .
7  .
```

5.2.4 Visualization

The visualization module is a combination of Prometheus server, Grafana, and database in a multi-container application. Docker-compose defines the protocol and communication between the containers. Additional metrics exporters can be implemented whether in the same docker-compose file containing the DataVisualizer module or externally and binded later with the monitoring server represented in Prometheus server. Examples of additional metric exporters are found in figure 5.4. The endpoint host.docker.internal

exports the collected metrics from the eBPF-based implementation and exposes it to docker containers to scrape from the host machine.

The MonitoringServer component stores the polled metrics from the DataExporter menitoned earler in a realational database suitable for time series using HTTP pulls. The MonitoringServer also has basic visualization service over its dashboard. Grafana service representing the Visualization component requests the stored metrics from the MonitoringServer and provides various, more advanced charts, flexible queries, real-time alerts, and other features through user-interface. Three sub-parts make up the MonitoringServer: Storage management, data retrieval, and a user-friendly dashboard as visualized in 5.1. In case of unexpected failure, persistent data storage is guaranteed as required in CL2 in 3 by binding the database storage within the docker compose environment to an external volume on the host machine.

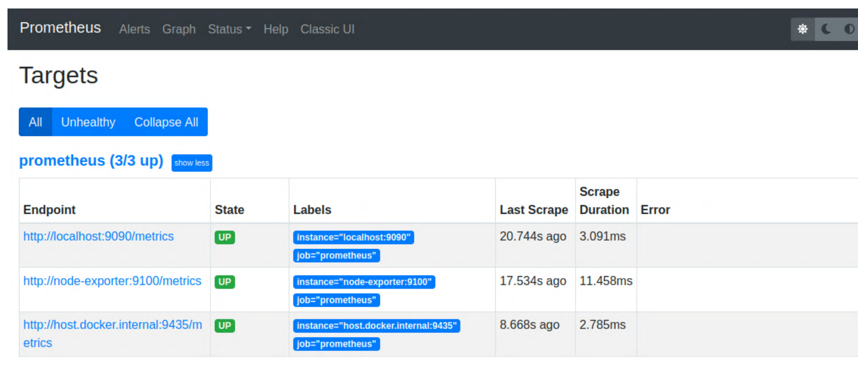


Figure 5.4: Prometheus dashboard showing all connected resources of metrics

Grafana is an open-source analytics and interactive visualization tool. It provides web charts, graphs, and alerts based on data from supported sources. Grafana comes with a built-in MySQL data source plugin. This allows the user to easily design graphs by querying and visualizing data from MySQL compatible databases.

Figure 5.5: Grafana dashboard with example graphs plotting real-time data

In Grafana, users can design graphs at their leisure by selecting metrics and time series from databases of their choosing as in 5.6. Creating a graph is as simple as creating a new panel inside a dashboard, selecting the data source and the desired metric/metrics to visualize. Users can also decide from a number of free templates according to their needs. The users can also fine-tune their queries using MySQL commands, specify legends, labels, and much more.

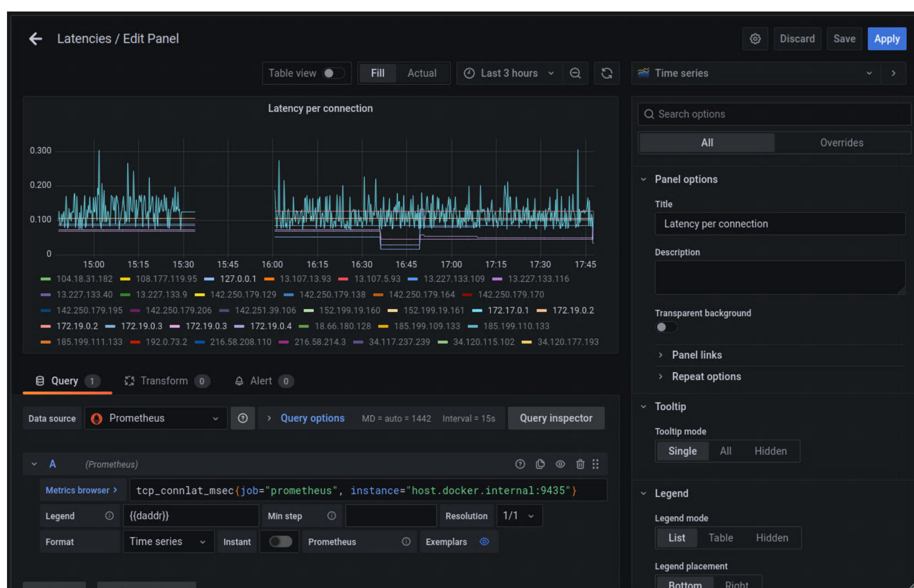


Figure 5.6: Creating a new panel in Grafana

5.3 Conclusion

In this chapter the software design of the monitoring system is presented and the core components are discussed in details. The role of eBPF and XDP in the data collection process and where the packet filtering occurs is described with code examples and diagrams. The technologies used to build up the system are presented and the reasons for picking are provided. In the next chapter the implemented monitoring system is evaluated. In this chapter, the software design of the monitoring system is presented and the core components are discussed in detail. Code examples and diagrams were used to explain how eBPF and XDP contribute to data collection and how the packet filtering is accomplished. The technologies used to create the system are described and the reasons for their selection are provided. In the next chapter, the implemented monitoring system is evaluated.

CHAPTER 6

850

EVALUATION

6.1 Introduction 33
6.2 Experimental validation 34
855 6.2.1 Setup 1 35
6.2.2 Setup 2 35
6.3 Performance evaluation 36
6.3.1 Evaluation on single node 36
6.3.2 Stress-test evalaution 37
860 6.4 Hardware utilization evaluation 39
6.5 Comparative Analysis 40
6.5.1 Requirement Evaluation 40
6.6 Conclusion 42

865

6.1 Introduction

Software testing is the process of verifying that a software program meets the requirements specified, and that it works as intended. Testing is an essential part of the software
870 development process, and it should be performed throughout the project, from the early stages of development to the final stages to see. The evaluation process determines how well the software development project met the requirements. It identifies what objectives were accomplished and what could be improved in future work.

There are many types of software testing. Functional testing for example verifies
875 that the functions of the software work as intended. Performance testing verifies that the software performs as intended under specific load conditions. Security testing verifies that the software is secure from unauthorized access, use, or modification. Unit Testing tests individual software components. System Testing tests the system as a whole.

In the first section, two configurations are described, that represent the experimental
880 setups. Based on these setups, the performance evaluations is carried out on the implemented software. The software is tested to determine the performance outcome than conventional tools on aspects of accuracy, stability, and hardware resources utilization. The goal of the performance evaluation is to study the efficacy of utilizing eBPF in

the approach. Next section conducts a comparative analysis of similar software that does not use eBPF and XDP technologies. Afterward, a detailed requirement analysis is conducted to assess the approach based on stated requirements in chapter 3. The results of the analysis are summarized in the conclusion.

6.2 Experimental validation

In this section, a performance test of the implemented software is carried out on three different testbed setups. These are, GCP, docker, and a real network consisting of devices connected to a router.

The GCP node consists of a single VM with 2 cores and 4GB of RAM. The docker node consists of a single docker container with 1 core. The real network testbed consists of n devices connected to a router.

The performance test is done to evaluate if the software provides better performance and less resource consumption. A comparative analysis of the performance of the implemented software with similar tools that do not use eBPF in packet filtering is done. In this analysis, the following factors are considered:

1. The time taken by the software to execute the task.
2. The amount of memory consumed by the software.
3. The amount of CPU utilization by the software.

An evaluation testbed consisting of n virtual machines on Google Cloud Service Provider (GCP) is created with the capability of adding more instances to the testbed. The preparation of the testbed over GCP is briefely mentioned here 8.3.2. Each node has the following specifications:

- Node series E2
- Machine type e2-standard-2
- 2 vCPU
- 4GB memory
- 10 GB SSD persistent disk
- Ubuntu 18.04 LTS with real-time kernel activation

In a similar way, a containerized evaluation testbed is created. Docker containers are used as nodes for the benchmarking and monitoring concept instead of GCP instances. As a result, unrelated factors such as outside connection drops and router bottlenecks are eliminated from evaluation processes. Docker containers represent the network nodes, while the monitoring system runs directly on the host. This allows for more accurate evaluation of the concepts. Docker provides more lightweight, agile computing resources using a container-based approach, compared with cloud computing. The process of setting up a testbed over the cloud is more complex and time-consuming. With docker, users can create a self-contained environment with all the necessary software, libraries and configurations needed for their application. With this approach, the application and all its dependencies can be run on any machine, regardless of the operating system. The concept is evaluated in docker environment, similarly to a GCP-based monitoring concept. The same performance metrics are measured, and the results are compared. The docker-based monitoring concept is found to be more efficient than the GCP-based monitoring concept. The Docker-based monitoring concept is faster and uses less

resources. The preparation of the testbed using docker containers is briefely mentioned here 8.3.1.

6.2.1 Setup 1

930 The first setup consists of 2 nodes with the configurations described in the previous section 6.2. One nodes represents a host node under monitoring and the implementation runs on the second node. In docker environment the implementation runs directly on the host machine. The following diagram shows the network topology 6.1.

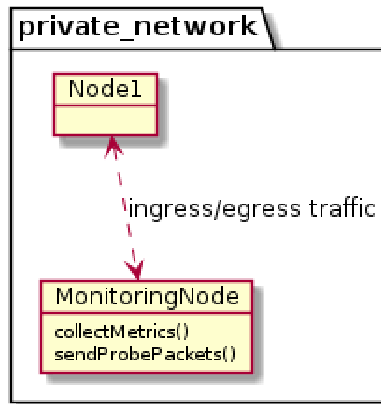


Figure 6.1: The testbed architecture

6.2.2 Setup 2

935 In the second setup, the testbed from the first setup is expanded to 5 test-nodes, totalling 5 hosts and 1 monitoring node as illustrated in figure 6.2. The nodes have similar specifications to the nodes from the first setup. This setup is designed is to examine the implementation under pressure.

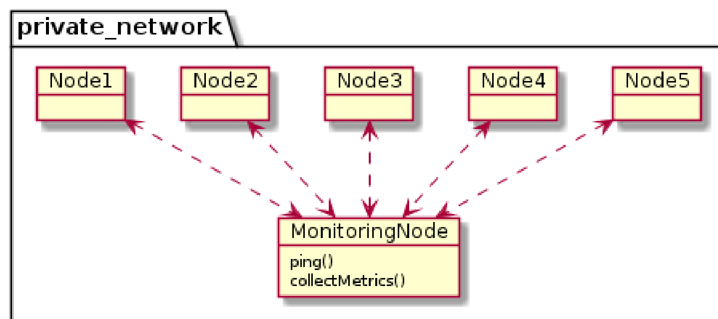


Figure 6.2: Testbed for stress testing

6.3 Performance evaluation

Evaluating the performance of the implementation is essential to understanding its capabilities and benefits. In order to do so, objective benchmarks must be used that remove any unrelated factors that might affect the results. The first benchmark evaluates the stability and accuracy of the latency measurements. This is important to ensure that the eBPF-based tool is providing accurate information in real-time. This section evaluates the robustness and accuracy of the eBPF-based approach compared to legacy tools such as ping. The testbed used n devices running on separate hardware connected to the same router.

6.3.1 Evaluation on single node

In the first part of this test, a testbed consisting of two native machines connected to a local network is set up as in figure 6.1. The monitoring node sends 100 ping requests to Node1 as illustrated in figure 6.1 at a rate of one request per 3 seconds in order to measure the round trip time (RTT) of the connection. For latency measurement, an example MetricCollector program based on eBPF is implemented similar to ping program for comparison. The test was repeated multiple times to evaluate the test results for rebustness. Figure 6.3 shows that the proposed approach using eBPF provides approximately 5 times lower latency than that of ping.

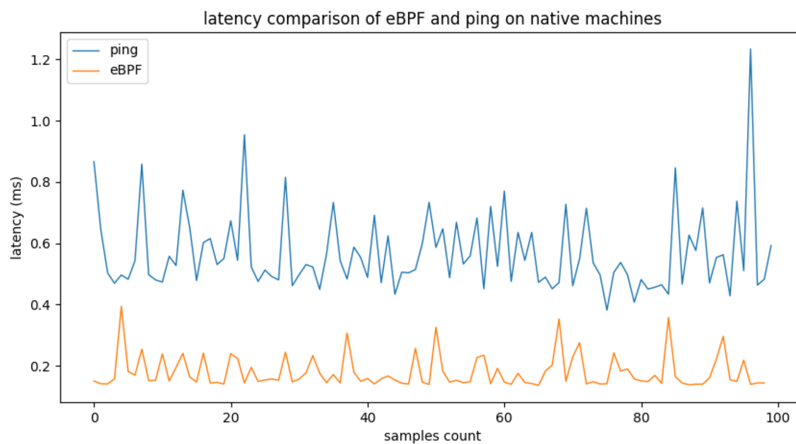


Figure 6.3: Latency comparison of eBPF example MetricCollector with non-eBPF based solution running on testbed setup 6.1

Information about the average latency and the variance of the measurements can be found in table 6.2. The standard deviation with the MetricCollector example using eBPF to capture packets is 5 times lower than with ping. Therefore, the implemented MetricCollector measurements are more consistent.

Tool	Average latency [ms]	Standard deviation [ms]
eBPF MetricCollector	0.179	±0.153
Ping	0.766	±0.528

Table 6.2: Latency results from eBPF-based MetricCollector vs. ping on Docker testbed

6.3.2 Stress-test evalaution

The next step is to stress-test the two tools, eBPF-based example MetricCollector and ping, on three different testbeds by sending large numbers of ping requests simultaneously to all host nodes from the monitoring node to n host nodes.

965 As a first testbed, docker containers running light Linux alpine images are used. In the second testbed, virtual machines are connected by one private network running on GCP, the Google cloud service. A third testbed uses real devices connected to one physical router inside a private network. The monitoring node runs on native hardware.

Table 6.3 shows that GCP and Docker produce similar results. eBPF MetricCollector 970 provides latency measurements about 7 times faster than ping in both testbeds. The standard deviation of ping is 2 times higher in the GCP testbed and 3 times higher in the Docker testbed, respectively. Using native machines for real-world testbed evaluation, the difference in performance rises significantly. The results of eBPF remain relatively constant. However, ping measurements vary greatly.

Testbed	Tool	Average latency [ms]	Standard deviation [ms]
GCP	eBPF MetricCollector	0.634	±0.1.15
	Ping	3.577	±3.258
Docker	eBPF MetricCollector	0.747	±1.851
	Ping	3.054	±5.553
Native	eBPF MetricCollector	0.193	±0.262
	Ping	51.134	56.990

Table 6.3: Average latency and standard deviation of the MetricCollector and ping on 3 different testbeds

975 An illustration of the results of the latency testing on the three testbeds can be found in Figure 6.4. During heavy active monitoring tests, both tools show fluctuation. MetricCollector, however, shows much less fluctuation thanks to the early packet capturing with eBPF. This is because the MetricCollector is able to collect data at a higher frequency at earlier stages, which allows for a more accurate evaluation of latency. By 980 running MetricCollector at a lower level in the network stack, RTT measurements can be significantly faster than those reported by ping. Additionally, XDP supports hardware timestamping on modern systems, which is a more accurate method of measuring elapsed time than software timestamping.

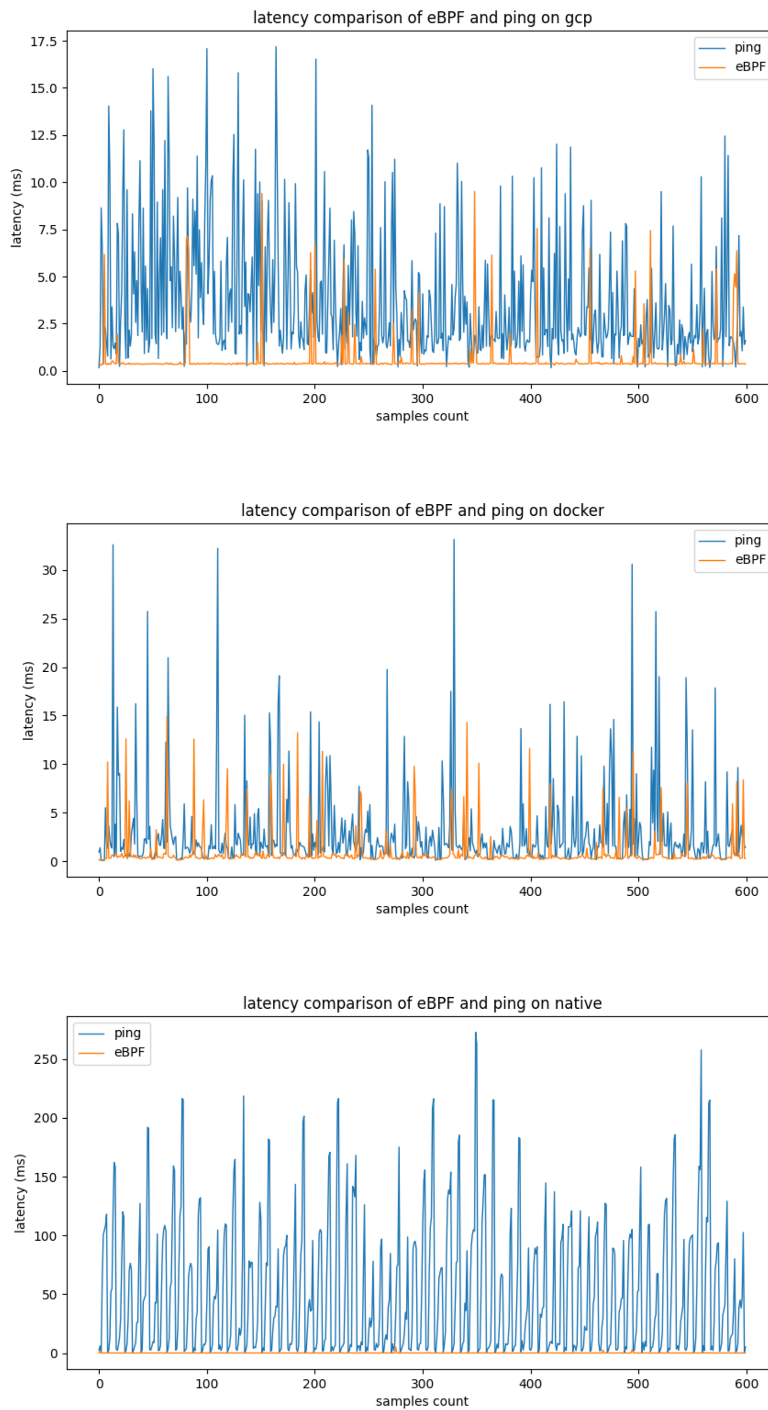


Figure 6.4: Latency measurements from the eBPF-based and non-eBPF tools on 3 testbeds

6.4 Hardware utilization evaluation

This benchmark evaluates the resources utilization of the tool. It measures how much of the system resources the concept is using in order to function. The test measures CPU utilization of the monitoring system in its entirety. To do so, the BCC tool profile.py is used to collect stack traces from the whole operating system for 60 seconds, while the monitoring system is running and collecting various hardware and network metrics. The FlameGraph tool then converts the output from the profiler into a more readable flame graph in SVG format, as shown in figure 6.5. On the x-axis, the graph shows the most frequent CPU-consuming code in the operating system in alphabetical order, and on the y-axis, it shows the stack traces ordered by the way the profiler reads them.

As visualized in figure 6.5, prometheus uses 16.40% of the CPU, while node_exporter - a non-eBPF metrics exporter - accounts for 34.57%. Meanwhile, the MetricsExporter marked in green box on the left side with the MetricCollector running in eBPF VM marked in green box on the right side use together only 5.56% of the CPU. Based on these results the MetricsExporter utilizing eBPF utilizes almost 7 times less CPU compared to node_exporter. However, node_exporter collects a greater number of metrics. Most profilers cannot trace the full stack of XDP programs. However, the XDP packet dropper running on the LLVM can still be traced on the left side marked in green. This flame graph was generated following the instructions in 8.10.

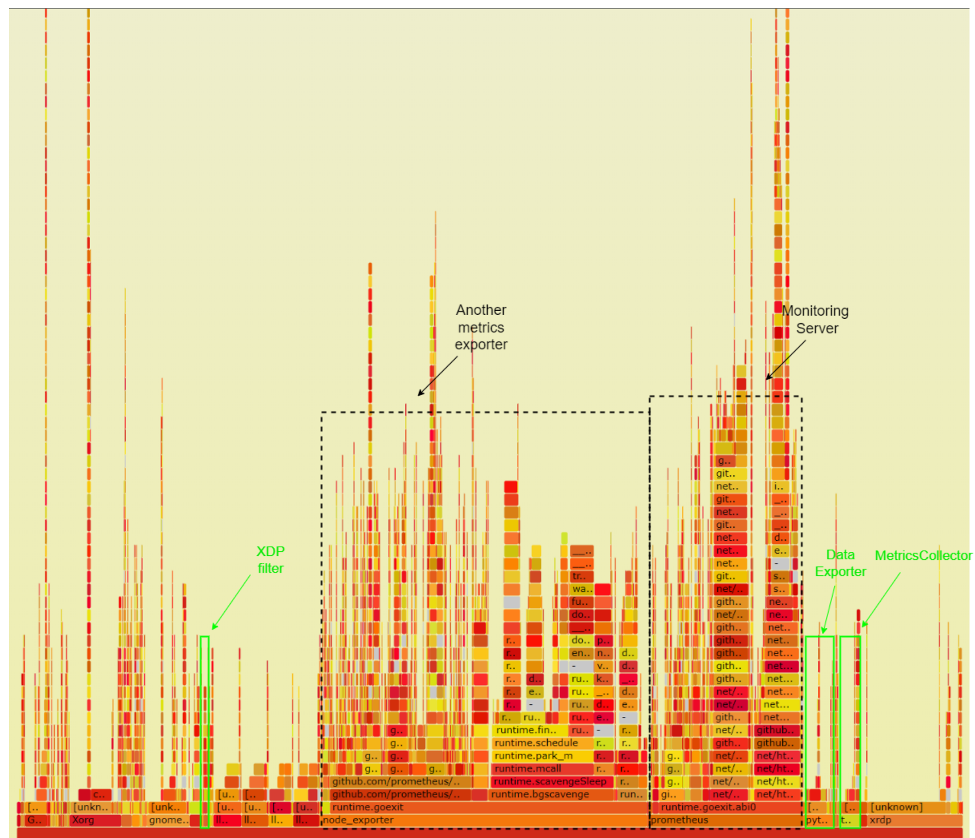


Figure 6.5: Flamegraph

6.5 Comparative Analysis

Comparative analysis is a process of reviewing and comparing two or more items to identify similarities and differences. Requirements evaluation is the process of assessing the suitability of requirements against predefined criteria. Comparative analysis can be used to assess the strengths and weaknesses of different options, to identify potential areas of agreement or disagreement, or to simply gain a better understanding of the items being evaluated.

6.5.1 Requirement Evaluation

In this section the requirements are evaluated in a table fashion to determine which requirements are satisfied and which ones are open for future work. Below is a table that evaluates the requirements in the order in which they were presented in chapter 3.

6.5.1.1 Top-level requirements

This section discusses the top-level requirements listed in section 3.1 for designing a real-time network monitoring tool. Specifically, the requirements call for a system that can continuously track network performance and identify issues in real-time. Additionally, the system should be able to provide actionable insights to help administrators optimize their networks. An evaluation of top-level requirements based on the implementation is shown in table 6.4 below.

Table 6.4: Top-level requirements

Requirement	Approach
TL9: Real-time capabilities	The real-time monitoring capability of the network was achieved by using eBPF for packet filtering. This results in an earlier detection of network events, bringing the system closer to real-time under the condition that the operating system's kernel is real-time enabled.
TL10: Comprehensive monitoring capabilities	This requirement was acheived by designing the monitoring system, so that no monitoring client needs to be installed on the nodes under monitoring. The approach collects network information from passive network traffic and using active probing which only uses ICMP protocol to send probe packets to the nodes being monitored. The firewall must be configured to allow ICMP traffic.
TL11: Scalability	The Implementation consists of two parts: A containerized DataVisualizer that uses Docker-compose and runs on virtualized hardware. This enables the system to be easily scalable by definition. The DataAggregator, which collects and processes network metrics, runs directly on the host OS and requires physical hardware resources. The DataAggregator is not designed with scaling in focus. However, it can be scaled by running multiple nodes, each exposing metrics to the DataVisualizer individually via separate DataExporter.

Table 6.4: Top-level requirements

Requirement	Approach
TL12: Automation	As the implementation does not focus on automation, it is outside the scope of this thesis. However, the open-source tools Prometheus and Grafana do allow connections to various known and customized services, so the work can be modified to meet this requirement.
TL13: User management	The implementation provides two platforms for managing the system. The first is the Prometheus dashboard, which allows users to manage data sources and databases. The dashboard also provides basic data visualization. Secondly, Grafana dashboard allows users to create and customize their own charts, providing advanced, flexible visualization options.
TL14: Visibility	This requirements is met with the implementation of the DataVisualizer module.
TL15: Reporting	Grafana dashboard allows users to create their own reports and logs via plugin scripts. The reports are synchronized to Grafana via the database connection.
TL16: Alerting	Using Grafana dashboards, the user can specify queries and expressions to generate alerts for each monitoring window.

6.5.1.2 System-level requirements

System-level requirements are the specifications that are related to the environment the monitoring system is running in. The table 6.5 below provides an evalaution of system-level requirements listed in section 3.1.

Table 6.5: System-level requirements

Requirement	Approach
SL4: Low resources consumption	Utilizing eBPF in the network traffic collection process results in reduction in resources consumption. First of all, eBPF programs only run when triggered by a predefined event. Furthermore, eBPF programs run at a very low level in the kernel, traversing less through the OS layers and requiring fewer context switches. This study [p35] contributes to this idea.
SL5: Interoperability	Interoperability is achieved by using standard communication between the nodes and the monitoring system by sending ICMP probe packets. This allows different network nodes to exchange information without further effort. Moreover, the network metrics are exposed via API endpoints at the DataExporter. This allows any service to communicate with the system via unified HTTP calls.

Table 6.5: System-level requirements

Requirement	Approach
SL6: Security	Security is enhanced in the system by dropping malicious packets at the early stages of reception through XDP filtering. The use of open-source tools to create software in general exposes the system to threats without constant patch updates. In summary, the focus of this thesis is not security, so this is outside the scope of this thesis.

6.5.1.3 Feature-level requirements

1025

Features are specific functions or features that a software system needs to provide as part of its overall functionality. They may be essential to the system's operation, or they may be desirable additions that improve the user experience or extend the system's capabilities. The feature-level requirements listed in section 3.3 are specific to the monitoring system that is the subject of this requirements document. 1030

Table 6.6: Feature-level requirements

Requirement	Approach
FL3: Dashboard	The DataVisualizer provides two main dashboards: the Prometheus monitoring server for managing the monitoring system, and the data visualization component for designing charts and setting alerts etc.
FL4: Data persistency	The DataVisualizer module holds a database running in a container and attached to external volume on the hard drive. The requirement is therefore satisfied.

6.6 Conclusion

In this chapter, relevant applied software testing methods are discussed. For performance evaluation, two setups are defined, and the performance of the implementation is evaluated for different tasks. The tasks were selected to represent different types of workloads. Among the performance aspects taken into consideration were latency, stability, and 1035 hardware utilization. The results indicated significant performance improvements with the use of eBPF and XDP. However, the evaluation results varied depending on the testbed setup. In addition, the scalability of the proposed approach was examined. Following the performance evaluation, a comparative analysis is performed by analyzing the three layers of requirements described in chapter 3. The requirements evaluation 1040 determined which requirements have been met and which require further development.

The next chapter presents a summary and conclusion of the thesis, followed by an outlook.

CHAPTER 7

SUMMARY AND FURTHER WORK

7.1 Overview 43
7.2 Conclusions and Impact 44
7.3 Outlook 44

7.1 Overview

This thesis presents a benchmarking and monitoring system for real-time networks. The work addresses the challenges of monitoring real-time networks, especially in industrial applications. The system aims to enable real-time monitoring by providing more stable monitoring, and reduce the amount of resources consumed in the process of data collection by using eBPF and XDP.

The thesis begins with an introduction to the topic and reviews the state-of-the-art in relation to the concerned research field, followed by an analysis of the requirements. In the requirements analysis process, a 3-layer model with requirements from the highest abstraction to the lowest abstraction is derived. Each layer contains a table of elements, each representing a functional or non-functional requirement. According to the assessment of requirements, this work presents the general approach following Lee's work [p7], and the concept expressed as a software architecture. The implementation of the monitoring system is based on the concept and approach to meet the defined requirements.

In addition, the integration of eBPF and XDP in the system is further explained and visualized. This implementation also includes a microservice-oriented data visualization module containerized with Docker. An evaluation of the performance of the implementation is conducted in a testbed environment. The implementation is also evaluated against the defined requirements through a comprehensive comparative analysis. Finally, a comparison of the adopted approach with alternatives is also presented.

7.2 Conclusions and Impact

This work presented a practical software-based network monitoring system that focuses on some challenges in monitoring real-time networks by significantly reducing resources consumption and capturing network events at earlier stages. The system has been implemented and evaluated on a testbed network. The results show that the proposed system is able to process more network activities than conventional solutions.

The concept of the monitoring system aims at service-oriented and containerized architecture that runs on virtual hardware, beside the data collection part that runs on the main hardware. An interface enables human-machine interactions and gives the user control over the system.

Although eBPF provides instantaneous responses from the OS, speed is not its primary feature. The idea of skipping ingress packets at the NIC driver and egress packets at the socket-level with the help of eBPF and XDP results in huge performance improvements since unnecessary computations or parts of the kernel stack are avoided. Additionally, eBPF programs detect events earlier and more accurately than traditional approaches, resulting in more accurate information. While eBPF offers better performance overall, it can be difficult to take a proof-of-concept eBPF program and extend it to a more complex program. BCC is one way of making writing eBPF programs easier, but there is still a long way to go before making writing more comprehensive eBPF programs easy.

7.3 Outlook

The monitoring system offers a proof-of-concept to demonstrate some characteristics of targeting low-level operating system operations related to network traffic collection. The presented approach can be enhanced by extending its functionalities and the implementation of more advanced eBPF applications for network monitoring. For example, using TCP over ICMP for sending probe packets is more recommended to gain a more granular image of the network state. TCP is a more common use case and thus tends to be more representative of real-world applications. The current prototype was also not evaluated in the case of long-term data collection, where a large number of measurements are taken over an extended period of time. This is an important limitation, as fluctuations in the network environment can occur and affect the accuracy of the collected data. Finally, the approach could be further optimized by incorporating more sophisticated data-collection and analysis algorithms, which would enable more in-depth insights into the network state.

The eBPF monitoring system can be further improved by adding other features, such as:

- correlating network activities with process information (e.g., PIDs) on the monitored system.
- detecting malicious activities or suspicious patterns over the network.
- providing more detailed information (e.g., packet payloads) about the monitored traffic.

CHAPTER 8

SPECIFICATIONS

8.1 eBPF program examples I
8.1.1 BCC programs as modules I
8.2 Docker Implementation III
8.3 Testbed initialization script IV
8.3.1 Testbed over Docker IV
8.3.2 Testbed over GCP V
8.4 Performance tools bpftool VI
8.5 Writing XDP programs VI
8.6 FlameGraph preparation VII

In this section a code review over the implementation, the testbed setup and evaluation is provided. The final source code is found here [m1].

8.1 eBPF program examples

This section showcases a few eBPF programs from the BCC collection that have been modified to work with the monitoring system. This displays how any network-related eBPF application can be enhanced to interact with other services and maintain a robust real-time connection.

8.1.1 BCC programs as modules

This example demostrates an example eBPF program from the BCC collection modified to be imported as a module to the prometheus_exporter. This helps exporting metrics from various eBPF programs for later scraping. The listing 8.1 shows a pseudo-code for the general structure of an eBPF program written using BCC libraries in python.

Listing 8.1: eBPF program as module

```
1 #!/usr/bin/python
2 # Copyright 2016 Netflix, Inc.
3 # Licensed under the Apache License, Version 2.0 (the "License")
4 #
```

Chapter 8

```
5   # 28−Jan−2016 Brendan Gregg Created this.
6   # 30−Mar−2016 Allan McAleavy updated for BPF_PERF_OUTPUT
7   # 23−Dec−2021 Amro Hendawi Added prometheus metrics exporter
8   from __future__ import print_function
9   from bcc import BPF
10  from prometheus_client import Gauge
11  latencyGauge = Gauge('metric_name', 'metric_description', ['attribute1', 'attribute2'] )
12  # load BPF program
13  bpf_text = """
14  #include <uapi/linux/ptrace.h>
15  #include <linux/sched.h>
16  struct val_t {
17  .
18  .
19  };
20  BPF_HASH(start, u32, struct val_t);
21  BPF_PERF_OUTPUT(events);
22  int foo(struct pt_regs *ctx) {}
23  int bar(struct pt_regs *ctx) {}
24  """
25  b = BPF(text=bpf_text)
26  b.attach_uprobe(name="c", sym="kernel_function1", fn_name="do_entry")
27  b.attach_uprobe(name="c", sym="kernel_function2", fn_name="do_entry")
28  b.attach_uretprobe(name="c", sym="kernel_function3", fn_name="do_return")
29  # header
30  def print_event(data):
31      event = b["events"].event(data)
32      latencyGauge.labels( event.host.decode('utf−8', 'replace')).set(float(event.delta) / 1000000)
33  def run_ebpf_program():
34      b["events"].open_perf_buffer(print_event)
35      while 1:
36          try:
37              b.perf_buffer_poll()
38          except KeyboardInterrupt:
39              exit()
```

One of the eBPF programs from the BCC collection used in this work is called tcprtt. This program measures the round-trip time (RTT) of TCP traffic to assess the network quality and outputs results as ready for histogram plot. It also helps us distinguish whether the network latency issue originates from the user or from the physical network.

```
# sudo ./tcprtt.py -d 10
Tracing TCP RTT... Hit Ctrl-C to end.

All Addresses = *******
     usecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 4        |*                                       |
         8 -> 15         : 21       |*******                                 |
        16 -> 31         : 110      |**************************************  |
        32 -> 63         : 114      |****************************************|
        64 -> 127        : 22       |*******                                 |
       128 -> 255        : 10       |***                                     |
       256 -> 511        : 25       |********                                |
       512 -> 1023       : 11       |***                                     |
      1024 -> 2047       : 2        |                                        |
      2048 -> 4095       : 4        |*                                       |
      4096 -> 8191       : 22       |*******                                 |
      8192 -> 16383      : 1        |                                        |
     16384 -> 32767      : 22       |*******                                 |
```

Figure 8.1: tcprtt example output

Connection latency (tcpconnlat) measures the time taken to establish a connection through TCP passively. It typically involves the TCP/IP processing at the kernel level and the network round trip time, not application runtime. tcpconnlat measures the time

between a connection and its response packet, in other words, the time from SYN sent to the response packet.

```
# sudo ./tcpconnlat.py
PID    COMM         IP SADDR           DADDR           DPORT LAT(ms)
48866  prometheus   4  172.19.0.3      172.17.0.1      9435  0.16
66191  wget         4  172.18.37.135   216.58.208.100  80    25.78
65656  Socket Threa 4  127.0.0.1       127.0.0.1       9435  0.07
65656  Socket Threa 6  ::1             ::1             9435  0.03
65656  Socket Threa 4  127.0.0.1       127.0.0.1       9435  0.07
65656  Socket Threa 4  127.0.0.1       127.0.0.1       9435  0.01
```

Figure 8.2: tcpconnlat example output

Tcplife summarizes TCP sessions that have opened and closed while tracing. It helps characterize workloads and identify what connections are taking place and what data is being transferred. Below is an example of the output when using wget to download Google's homepage.

```
# sudo ./tcplife.py
PID   COMM       LADDR           LPORT RADDR           RPORT TX_KB RX_KB MS
48866 prometheus 172.19.0.3      57304 172.17.0.1      9435      0     0 0.18
65627 wget       172.18.37.135   60504 216.58.208.100  80        0    15 127.56
48866 prometheus 172.19.0.3      57308 172.17.0.1      9435      0     0 0.10
```

Figure 8.3: tcplife example output

8.2 Docker Implementation

The Docker Compose file presented in listing 8.2 represents the DataVisualizer module in figure 5.1. It consists of prometheus server, grafana dashboard, a database integrated in prometheus service and the performance metrics exporter node_exporter.

Listing 8.2: Specification 1

```
1  version: "3.7"
2  services:
3    prometheus:
4      image: prom/prometheus:latest
5      volumes:
6        - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
7      ports:
8        - 9090:9090
9      extra_hosts:
10       - "host.docker.internal:host-gateway"
11   grafana:
12     image: grafana/grafana:latest
13     volumes:
14       - ./grafana/grafana.ini:/etc/grafana/grafana.ini
15       - ./grafana/datasource.yml:/etc/grafana/provisioning/datasources/datasource.yml
16     ports:
17       - 3000:3000
18     links:
19       - prometheus
20   node-exporter:
21     image: prom/node-exporter:latest
22     container_name: monitoring_node_exporter
23     restart: unless-stopped
24     ports:
25       - 9100:9100
```

8.3 Testbed initialization script

8.3.1 Testbed over Docker

The testbed environment prepared for the evalaution of the concept is created based on the testbed architecture in figure 6.2 using the shell script in listing 8.4. This script creates a software network bridge which with high Maximum transmission unit (MTU) rate for testing the concept under high throughput pressure will allows containers connected to the same bridge network to communicate. The script also creates 5 docker containers running basic Linux alpine image representing test network nodes. The virtual network configurations are also defined and the necessary protocols for the evaluation are enabled. Creating a testbed using docker containers eliminates unrelated outside factors such as network latency or router bottleneck.

Listing 8.3: Testbed initialization as Docker containers

```
#!/bin/bash
num_nodes=5
network_name=setup2
# cleanup from past runs
for instance in $(seq 1 $num_nodes)
do
    docker stop node−$instance
done
res=$(docker network ls | grep $network_name)
if ! [ −z "$res" ];
then
    docker network rm $network_name
fi
#####################
# create docker network with mtu higher than 1500
docker network create −−driver=bridge \
−o "com.docker.network.driver.mtu"="3000" \
−o "com.docker.network.bridge.host_binding_ipv4"="0.0.0.0" \
−o "com.docker.network.bridge.enable_icc"="true" $network_name
for instance in $(seq 1 $num_nodes)
do
    docker run −it −d −−rm −−name node−$instance −−network $network_name alpine
done
for instance in $(seq 1 $num_nodes)
do
    echo assigned IP address for node−$instance:
    docker exec node−$instance ip addr show eth0
    echo ""
done
```

The docker instances share the same docker network bridge as in the listing below 8.5. This can be reproduced by executing the following line.

Listing 8.4: Testbed initialization as Docker containers

```
    docker network inspect bridge
```

Listing 8.5: docker-network-bridge

```
[
    {
        "Name": "setup2",
        "Id": "9bde9251520aa0f16d4fcacb8e1355437f818a7460f3fa87d22034a00badfdcc",
        "Created": "2022−02−11T15:52:57.745217578+01:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
```

```
            "Config": [
                {
                    "Subnet": "172.31.0.0/16",
                    "Gateway": "172.31.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "2feae3d40780717f5419b8b17c872df077ce53a28c54d834871a8e2f1950754c": {
                "Name": "node-2",
                "EndpointID": "82d06d6542cf7aca79594ee8922c84dd22bf6bdad16748cb12322038a22dabb5",
                "MacAddress": "02:42:ac:1f:00:03",
                "IPv4Address": "172.31.0.3/16",
                "IPv6Address": ""
            },
            "747ac98846f25b50d76b05096d14a522ff3f65170adb152c31f4c4b45218756a": {
                "Name": "node-1",
                "EndpointID": "ef6591d26b5be9a066303477b93834c9622103af32333da00a8120b85eb6851c",
                "MacAddress": "02:42:ac:1f:00:02",
                "IPv4Address": "172.31.0.2/16",
                "IPv6Address": ""
            },
            "9a66c82dc179938d1f4565b0e867668dba0d751d069eabcd820bb6a108fe500d": {
                "Name": "node-3",
                "EndpointID": "fe1a04c7e5582a219f8b65eaa3d8811d54c1470ec2e4e23ef9e3534762bb9e2b",
                "MacAddress": "02:42:ac:1f:00:04",
                "IPv4Address": "172.31.0.4/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.driver.mtu": "3000"
        },
        "Labels": {}
    }
]
```

8.3.2 Testbed over GCP

The testbed environment prepared for the evalaution of the concept was created using the following shell script. It creates three n2-standard-2 VM instances on google cloud service. The virtual network and firewall rules are also defined and the necessary protocols for the evaluation are enabled. Creating a testbed over GCP differs from docker testbed in the way that the nodes run over separate machines.

Listing 8.6: Specification 1

```
#!/bin/sh
# initial variables
USER_NAME='testbed'
TAG='testbed_network'
INSTANCE_NAME='testbed-instance'
# create ssh key pair and add it to GCP account
echo -e 'y\n' | ssh-keygen -f id_rsa -C $USER_NAME
echo -n $USER_NAME":$(cat_id_rsa.pub)" > $METADATA_FILE
gcloud compute project-info add-metadata --metadata-from-file ssh-keys=$METADATA_FILE
# create the firewall rule for ssh and icmp
gcloud compute firewall-rules create allow-ssh-icmp-rule --network default --action allow --direction ingress,
        egress --rules tcp,icmp --source-ranges=0.0.0.0/0 --target-tags $TAG
# create 3 instances in the same network
for num in $(seq 1 3);
```

Chapter 8

```
14 do
15     gcloud compute instances create $INSTANCE_NAME$num --zone=europe-west3-c --machine-type n2-
           standard-2 --image-project ubuntu-os-cloud --image-family ubuntu-1804-lts --tags $TAG
16     # resize disk space of the created instance
17     echo 'y' | gcloud compute disks resize $INSTANCE_NAME$num --size=100G --zone=europe-west3-c
18 done
19 echo "success!"
```

8.4 Performance tools bpftool

bpftool are a set of tools that allow for inspection and simple manipulation of eBPF programs and maps including xdp programs. There are currently two tools in this suite:

- eBPF tool: This tool allows you to inspect and modify eBPF programs and maps. To use this tool, you must first install the eBPF kernel module.
- map tool: This tool allows you to inspect and modify maps.

Both of these tools are command-line tools. To enable the full functionality of bpftool it needs to be installed from the source code as in the listing 8.7 below.

Listing 8.7: bpftool

```
1  # install dependencies
2  sudo apt install pkg-config m4 libelf-dev libpcap-dev gcc-multilib python-docutils python-docutils
3  # update packages
4  sudo apt update && sudo apt upgrade
5  # clone libbpf repository
6  git clone https://github.com/libbpf/libbpf.git
7  # cd into directory
8  cd libbpf/src
9  # build and install libbpf
10 make
11 # download linux kernel source code
12 wget https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/snapshot/linux-5.15.tar.gz
13 # unpack and cd into directory
14 tar xvzf linux-5.15.tar.gz linux-5.15 && cd linux-5.15
15 # build and install bpftools
16 cd tools/bpf/bpftool; make; sudo make install; make doc; sudo make doc-install
```

8.5 Writing XDP programs

There are a few things to consider when writing an XDP program. XDP programs are run in kernel space, so they must be compliant with the kernel's programming model. Therefore, they must be lightweight and efficient, and they must not modify any data structures that are not explicitly shared with the kernel. In addition, XDP programs are attached to network interfaces, so they must be designed to work with the network interface's hardware. XDP programs, for instance, cannot assume they can read or write to a specific memory location on the network interface. Lastly, XDP programs can only be used to process packets received by a network interface. They cannot process packets sent by the network interface. In the following is a pseudo-XDP program designed to drop packets specified by the developer.

Listing 8.8: xdp-example

```
1 #include <linux/bpf.h>
2 #include <bpf/bpf_helpers.h>
3 #include <arpa/inet.h>
```

```
4  SEC("xdp_drop")
5  int xdp_drop_prog(struct xdp_md *ctx)
6  {
7      void *data_end = (void *)(long)ctx->data_end;
8      void *data = (void *)(long)ctx->data;
9      struct ethhdr *eth = data;
10     .
11     .
12     if(packet_eligible_for_dropping())
13         return XDP_DROP;
14     return XDP_PASS;
15 }
```

In order to build and load XDP programs, a few dependencies must be installed. The script below 8.9 illustrates how XDP programs can be executed.

Listing 8.9: xdp-setup

```
1  # clone xdp-tools repository
2  git clone https://github.com/xdp-project/xdp-tools.git
3  # cd into directory
4  cd xdp-tools
5  # build and install
6  ./configure && make
7  # compile the xdp program
8  clang -O2 -g -Wall -target bpf -c xdp_drop.c -o xdp_drop.o
9  # install xdp program to eth0
10 sudo ip link set eth0 xdpgeneric obj xdp_drop.o sec xdp_drop
11 # load xdp program using xdp-loader
12 sudo ./xdp-loader/xdp-loader load -m skb -s xdp_drop eth0 xdp_drop.o
```

8.6 FlameGraph preparation

Flamegraphs are a visualization tool used to depict the call stack of a running program. They are created by stacking vertically the functions called during a program's execution, with the function at the top of the stack being the one that was called first. The width of each function's column corresponds to how long that function was running for. The height of each function's column is proportional to the number of times that function was called. Flamegraphs can be used to quickly identify which functions are taking up the most time in a program. This information can be used to help optimize a program's code.

The table below shows how to use flamegraphs

Listing 8.10: flamegraph

```
1  # make sure you are in root user
2  sudo bash
3  # download Flamegraph
4  git clone https://github.com/brendangregg/FlameGraph
5  # download bcc open-source tools
6  git clone https://github.com/iovisor/bcc.git
7  # 1- call the eBPF-based profiler at a frequency of 99 hertz for 60 seconds
8  # 2- pipe the output to flamegraph to convert to svg flamegraph
9  sudo ./bcc/tools/profile.py -dF 99 -f 60 | ./FlameGraph/flamegraph.pl > perf.svg
```

Bibliography

References to Scientific Publications

[p1] M. Wollschlaeger, T. Sauter, and J. Jasperneite, "The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0," *IEEE Industrial Electronics Magazine*, vol. 11, no. 1, pp. 17–27, 2017. DOI: 10.1109/MIE.2017.2649104 (cit. on p. 1).

[p2] J. C. Knight, "Safety critical systems: Challenges and directions," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, 2002, pp. 547–550 (cit. on p. 1).

[p3] R. Zhohov, D. Minovski, P. Johansson, and K. Andersson, "Real-time performance evaluation of lte for iiot," in *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*, 2018, pp. 623–631. DOI: 10.1109/LCN.2018.8638081 (cit. on p. 2).

[p4] I. Palúchová, "Optimization of network monitoring" (cit. on p. 2).

[p5] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance implications of packet filtering with linux ebpf," in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 01, 2018, pp. 209–217. DOI: 10.1109/ITC30.2018.00039 (cit. on pp. 2, 3, 10).

[p6] A. Kind, X. Dimitropoulos, S. Denazis, and B. Claise, "Advanced network monitoring brings life to the awareness plane," *IEEE Communications Magazine*, vol. 46, no. 10, pp. 140–146, 2008. DOI: 10.1109/MCOM.2008.4644132 (cit. on p. 2).

[p7] S. Lee, K. Levanti, and H. S. Kim, "Network monitoring: Present and future," *Computer Networks*, vol. 65, pp. 84–98, 2014, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2014.03.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S138912861400111X (cit. on pp. 2, 8, 9, 20, 24, 27, 43).

[p8] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "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. DOI: 10.1109/HPSR.2018.8850758 (cit. on pp. 3, 11).

[p9] K. Hoyme and K. Driscoll, "Safebus," in *[1992] Proceedings IEEE/AIAA 11th Digital Avionics Systems Conference*, 1992, pp. 68–73 (cit. on p. 6).

[p10] M. Postolache, "Time-triggered canopen implementation for networked embedded systems," in *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*, Oct. 2016, pp. 168–173. DOI: 10.1109/ICSTCC.2016.7790660 (cit. on p. 6).

[p11] M. Abuteir and R. Obermaisser, "Simulation environment for time-triggered ethernet," in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, 2013, pp. 642–648 (cit. on p. 6).

[p12] M. H. Farzaneh and A. Knoll, "Time-sensitive networking (tsn): An experimental setup," in *2017 IEEE Vehicular Networking Conference (VNC)*, 2017, pp. 23–26 (cit. on p. 6).

[p13] T. Bu, Y. Yang, X. Yang, W. Quan, and Z. Sun, "Tsn-insight: An efficient network monitor for tsn networks," Apnet, 2019 (cit. on p. 6).

[p14] A. H. Celdrán, M. G. Pérez, F. J. García Clemente, and G. M. Pérez, "Automatic monitoring management for 5g mobile networks," *Procedia Computer Science*, vol. 110, pp. 328–335, 2017, 14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017) / 12th International Conference on Future Networks and Communications (FNC 2017) / Affiliated Workshops, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2017.06.102. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050917312802 (cit. on p. 7).

[p15] T. A. N. do Amaral, R. V. Rosa, D. F. C. Moura, and C. E. Rothenberg, "An in-kernel solution based on xdp for 5g upf: Design, prototype and performance evaluation," in *2021 17th International Conference on Network and Service Management (CNSM)*, 2021, pp. 146–152. DOI: 10.23919/CNSM52442.2021.9615553 (cit. on p. 7).

[p16] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "Payless: A low cost network monitoring framework for software defined networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–9. DOI: 10.1109/NOMS.2014.6838227 (cit. on pp. 7, 13).

[p17] J. Jiang, Y. Li, S. H. Hong, M. Yu, A. Xu, and M. Wei, "A simulation model for time-sensitive networking (tsn) with experimental validation," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2019, pp. 153–160. DOI: 10.1109/ETFA.2019.8869206 (cit. on p. 7).

[p18] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky, "Advanced study of sdn/openflow controllers," in *Proceedings of the 9th central & eastern european software engineering conference in russia*, 2013, pp. 1–6 (cit. on p. 7).

[p19] Open-source, "Oflops/cbench at master · andi-bigswitch/oflops," Available online (cit. on p. 7).

[p20] Open-source, "Arccn/hcprobe: Framework for testing openflow controllers," Available online (cit. on p. 8).

[p21] B. Siniarski, C. Olariu, P. Perry, T. Parsons, and J. Murphy, "Real-time monitoring of sdn networks using non-invasive cloud-based logging platforms," in *2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, 2016, pp. 1–6. DOI: 10.1109/PIMRC.2016.7794973 (cit. on p. 8).

[p22] M. Felser and T. Sauter, "The fieldbus war: History or short break between battles?" In *4th IEEE International Workshop on Factory Communication Systems*, 2002, pp. 73–80. DOI: 10.1109/WFCS.2002.1159702 (cit. on p. 8).

[p23] R. Piggin, "Fieldbus flexes for safety networks," *inTech Magazine isa.org*, 2005 (cit. on p. 8).

[p24] M. Karsten and J. Schmitt, "Packet marking for integrated load control," in *2005 9th IFIP/IEEE International Symposium on Integrated Network Management, 2005. IM 2005.*, 2005, pp. 499–512. DOI: 10.1109/INM.2005.1440821 (cit. on p. 9).

[p25] H. Sun, "An integrated network performance monitor system," in *2010 Third International Symposium on Intelligent Information Technology and Security Informatics*, 2010, pp. 88–91. DOI: 10.1109/IITSI.2010.60 (cit. on p. 9).

[p26] E. J. S. E. R. Enns. Ed. M. Bjorklund and A. B. Ed., "Hjp: Doc: Rfc 6241: Network configuration protocol (netconf)," (Accessed on 06/04/2021), Jun. 2011 (cit. on p. 9).

[p27] M. B. Ed., "Hjp: Doc: Rfc 6020: Yang - a data modeling language for the network configuration protocol (netconf)," (Accessed on 06/04/2021), Oct. 2010 (cit. on p. 9).

[p28] D. Plonka, "Flowscan: A network traffic flow reporting and visualization tool.," in *LISA*, 2000, pp. 305–317 (cit. on p. 9).

[p29] M. A. Vieira, M. S. Castanho, R. D. Pacıfico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020 (cit. on pp. 13, 15).

[p30] S. Zhao, M. Chandrashekar, Y. Lee, and D. Medhi, "Real-time network anomaly detection system using machine learning," in *2015 11th International Conference on the Design of Reliable Communication Networks (DRCN)*, 2015, pp. 267–270. DOI: 10.1109/DRCN.2015.7149025 (cit. on p. 13).

[p31] R. Pietro and F. Lombardi, "Virtualization technologies and cloud security: Advantages, issues, and perspectives," Jul. 2018 (cit. on p. 14).

[p32] G. Fournier, "Process level network security monitoring & enforcement with ebpf," (cit. on p. 14).

[p33] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, "Accelerating linux security with ebpf iptables," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, 2018, pp. 108–110 (cit. on pp. 15, 22).

[p34] M. Chakraborty and A. P. Kundan, *Prometheus*. Berkeley, CA: Apress, 2021, pp. 99–131, ISBN: 978-1-4842-6888-9. DOI: 10.1007/978-1-4842-6888-9_4. [Online]. Available: https://doi.org/10.1007/978-1-4842-6888-9_4 (cit. on p. 28).

[p35] M. Abranches, O. Michel, E. Keller, and S. Schmid, "Efficient network monitoring applications in the kernel with ebpf and xdp," in *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2021, pp. 28–34. DOI: 10.1109/NFV-SDN53031.2021.9665095 (cit. on p. 41).

Technical References

[t1] B. Claise and S. Bryant, "Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information," RFC 5101, January, Tech. Rep., 2008 (cit. on p. 9).

[t2] brendan Gregg, "Linux ebpf tracing tools," Tech. Rep., 2017 (cit. on p. 12).

[t3] I. Visor, "Xdp - io visor project," Tech. Rep. (cit. on p. 13).

[t4] "Client libraries | prometheus," Tech. Rep. (cit. on p. 28).

Miscellaneous References

[m1] A. Hendawi, "Amro.hendawi / rt monitoring system," in GitLab, 2022. [Online]. Available: https://git.tu-berlin.de/amro.hendawi/rt_monitoring_system.git (cit. on p. I).

Index

Symbols

5G . . . . . . i, 1

B

BCC 26
bpftool VI

C

Concept and Approach 20
CPS . . . . . . i, 1

D

docker III

E

eBPF i, 2 – 5, 12, 15 f, 20, 22, 24 – 27, 29, 32, 34, 40 f, 43
eBPF-examples I
Evaluation 33

F

flamegraph VII

G

GCP 34

I

I4.0 . . . . . . i f, 1 f
Implementation 25
IoT . . . . . . i, 1

L

LLVM 12

N

NFV 7

O

OS 16, 22

R

Related Area 6
Related Area 2 7
Related Area 3 8
Related Area 4 7
Requirements 17
RT . . . . . . 4 f, 9 f, 14, 16, 24

S

SDN . . . . . . 7 f
SotA 5, 14

T

testbed-spinup IV
TSN . . . . . . i f, 6

V

Validation 33
Verification 33

X

XDP . . . 3 ff, 13, 16, 20, 22 ff, 32, 34, 42 f
xdp-programs VI