ACIT5930

Master's Thesis phase III

in

Applied Computer and Information Technology (ACIT)

August 2021

Cloud-based Services and Operations

Virtualized 4G Long Term Evolution Testbed to investigate the performance impact of In-Band Network Telemetry

Vetle T. Moen

Department of Computer Science
Faculty of Technology, Art and Design

OSLOMET

Abstract

As the fifth generation of mobile networks move into the cloud, there is a need to develop new and more flexible monitoring systems. In-band Network Telemetry (INT) has begun to make its mark on the Network Monitoring field, and provides a new way of collecting and processing network telemetry data directly on programmable switches. With INT we can use the in-band network traffic itself as the carrier for monitoring metrics by leveraging the data plane of programmable switches. In this thesis we develop a fully virtualized 4G Long Term Evolution (LTE) testbed in a VirtualBox environment with INT support, and we develop an algorithm that leverages the TOS field in IP Packet Headers to detect Packet Loss and Network Delay between two P4-capable BMV2 switches. The Radio Access Network (RAN) and and Evolved Packet Core (EPC) components of the LTE network are implemented with the OpenAirInterface project, and the EPC components are deployed in a Mininet environment as containers, which are connected to P4-capable Behavioral Model V2 (BMV2) switches. INT does incur a higher cost to CPU compared to more traditional monitoring systems such as passive and active systems, but maintains a higher measured bandwidth even with this cost. INT is also able to detect network delay with high reliability, but has reduced accuracy for detecting packet loss.

Acknowledgments

I would like to thank my supervisors, Ahmed Elmokashfi, Hårek Haugerud, and Anis Yazidi, for their guidance and support throughout this project.

I would also like to express my gratitude and appreciation towards my fiancée, who has continually supported me through my endless rants of frustration, and her invaluable advice on clear language in writing.

Finally I would like to thank my friends and family for their support and sympathetic ears throughout the last 1.5 years of hard work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years there have been two major additions to the field of network monitoring. The first is the Top-Down approach, which is a new take on designing, implementing, and operating Monitoring Systems. It focuses on increased programmability and the network-wide capabilities of such systems, with the ultimate goal of becoming omniscient and omnipresent throughout the network [1]. The second addition is In-band Network Telemetry (INT), which is a method of utilizing programmable switches to gain insight into network traffic at nearly full line rate [2]. INT can be implemented with a relatively new programming language called P4, which focuses on processing packets in the ingress and egress interfaces of programmable switches [3].

The fifth generation of mobile networks have now started to be deployed on a significant scale. 5G is envisioned to bring major improvements to flexibility and capacity through the use of virtualized network functions, cloud environments, and improved base stations for radio access [4]. This generation is more Service Oriented and can be tailored to fit a specific use case, such as Massive Machine Type Communication, Critical Communication, and enhanced mobile broadband for regular users. 5G core components can be implemented as Virtual Network Functions (VNFs) and as a result several 4G and 5G testbeds have been developed and deployed in Cloud environments such as OpenStack [5, 6]. None of these testbeds, however, have integrated P4-capable switches into their environments. And seeing as 5G technology is still young, there is no standardized and agreed upon approach to network monitoring in these new environments.

With the new approaches to network monitoring and the introduction of software-defined networking to mobile networks, INT appears to be a viable candidate for Monitoring in these new cloudified mobile networks. With that there are two research questions we attempt to answer in this project. How does INT impact the network performance of a 4G Long Term Evolution (LTE) network, compared to passive and active monitoring systems? To what extent can INT be used to detect two common network problems, delay and packet loss, in such networks?

In order to investigate these questions we design and implement a fully

virtualized 4G LTE testbed with added support for P4-capable software switches. The 4G LTE network uses the OpenAirInterface (OAI) project for both the Radio Access Network (RAN) and the Evolved Packet Core (EPC) [7]. The P4 switches are implemented with the Behavior Model V2 (BMV2) project [8]. These components are integrated with each other through the Function Offloading Prototyping with P4 (FOP4) project [9], which is based on Mininet. With this testbed, we compare each monitoring system with a baseline measurement in order to investigate how INT with P4 impacts the performance of this testbed, as well as to what extent INT can detect common network problems.

1.1 Contributions

Through this project we have made the following contributions to the field of network monitoring and software-defined mobile networks:

- Created a virtual testbed with P4 switches and an open source LTE implementation that can be deployed on commercial hardware.
- Developed an algorithm in P4 that leverages the TOS field in IP Packet Headers to detect Packet Loss and Network Delay between two P4-capable BMV2 switches.
- Investigated the performance impact of INT in a virtualized 4G LTE network.
- Investigated the capabilities of INT in a virtualized 4G LTE Network.
- Discovered bug in the makefile of P4-OVS and created a Pull Request with the proposed fix.

1.2 Structure of the report

The remainder of this thesis is organized as follows:

- **Chapter 1 - Introduction:** Explains the problem, research questions and motivation behind this work.
- **Chapter 2 - Background:** Provides the necessary background knowledge of 5G Networks, Network Monitoring and the related work in this field.
- **Chapter 3 - Methodology:** Outlines the selected approach to investigate the proposed research questions.
- **Chapter 4 - Implementation:** Presents an overview of the virtualized testbed and how we arrived at the final iteration.
- **Chapter 5 - Results:** Details the performed experiments and the results gathered from these.

- **Chapter 6 - Discussion:** Discusses the results and their implication.
- **Chapter 7 - Conclusion:** Gives an overview of what we have done, our findings, and our final conclusion.

Chapter 2

Background

In this chapter we cover all the necessary technical details and background knowledge needed to understand the work done throughout the thesis. The chapter is divided into three main sections, namely 1) Mobile Networks, 2) Network Monitoring, and finally 3) Software Defined Networking. In the first section, we cover what mobile networks are and the high-level operation of these networks. In the second section, we cover new and old approaches to monitoring and some methods for collecting data. In the third section, we introduce software defined networking and In-band Network Telemetry. Finally we outline the related works to this project.

2.1 Mobile Networks

Today, cellular networks are widespread and used by virtually everyone with a smartphone of which, according to Statista [10], there are over 3.8 billion. Many IoT devices also utilize cellular networks in order to gain Internet access. The number of Internet-connected IoT devices range in the tens of billions, and is estimated to reach 43 billion by 2027 [11]. This puts tremendous pressure on the networks providing Internet access. The current infrastructure needs to be expanded and modernized in order to handle this ever-increasing demand. 5G aims to improve upon 4G with increased bandwidth and a service oriented approach to support a diverse set of services as pointed out in a survey on network slicing in 5G networks [12]. The most important improvement is the shift towards a service oriented architecture, which follows the Infrastructure as Code (IaC) [13] paradigm with increased levels of virtualisation and slicing. The IaC paradigm aims to include more DevOps in the Operations of digital infrastructure, in the form of machine-readable configuration files to deploy and run services in a virtualised environment, and other practices borrowed from Agile Development. An example is Test-Driven Development of the configuration files in order to verify that any server or software updates will not break during deployment.

Cellular networks are wide-area networks divided into cells, hence the name Cellular, where each cell has a base station to which a client

connects. The client, also known as User Equipment (UE), needs some form of authentication in order to connect to a given base station, which is usually provided through SIM Cards. These cards are pre-configured with all the information the mobile client needs to discover, connect, and authenticate itself to a cellular network. The Base Station connects to a Network Core in order to let the client communicate over the network and out to the Internet. A simplified view of the Cellular Network architecture is divided into two parts and is shown in figure 2.1, the Radio Access Network (RAN) where the Base Station lives, and the Core Network which handles all traffic to and from the RAN as well as supporting services such as mobility management, authentication, quality of service, and more. Cellular, or mobile, networks have a long and interesting history spanning back to the 1980s, but for the sake of brevity only a brief history lesson is given in the next sections.

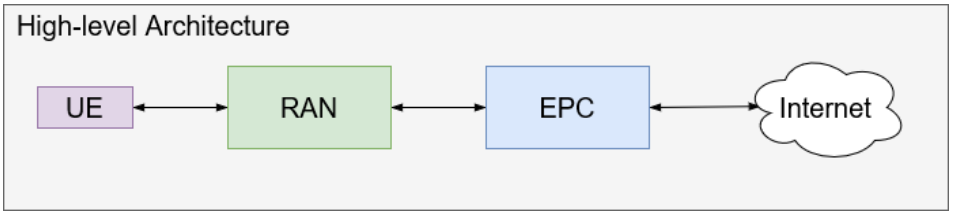


Figure 2.1: A High-level overview of the Virtual Testbed with P4-switches

2.1.1 Earlier Generations

This section briefly covers the history of cellular network evolution from analog 1G networks, all the way up to digital 4G networks with an all-IP core. This is to give the reader some background into the underlying technology in the thesis. Most of the material is based on [14] which presents a brief history of these network generations.

1G, 2G and 2.5G

The first generation of cellular networks was a completely analog system with voice-only support. These were replaced by the digital 2G, still with voice-only support. 2.5G extended the 2G standard with limited data-support in the form of an Internet-connection. The general structure of 2G networks follows the Global System for Mobile Communications (GSM) standard. According to this structure, mobile clients connect to a Base Transceiver Station (BTS). The BTS forms the cell, and is serviced by the Base Station Controller (BSC) which is responsible for providing connectivity to and from the BTS. It also handles resource allocation in the form of radio channels to mobile clients and paging which is how a BTS finds a given mobile client and its current cell. The BSC is connected to a Mobile Switching Center (MSC) which connects several BSC's together, and performs user authorization and accounting, call establishment and teardown. A provider can have multiple MSC's which contain up to five

BSC's, and a set of MSC Gateways which connect the cellular network to a larger public telephone network. Seen from another perspective, the BTS and BSC form the RAN and the MSC and MSC Gateways form the Core Network.

3G

3G supports voice and data, but enhances the data capabilities and provides higher speeds. The underlying architecture splits voice and data into two from the base station. Voice data is sent across the same network infrastructure as the old 2 and 2.5G, meaning the old telephone systems. TCP/IP data is sent to the Network Core. 3G further improves on 2G radio access by more efficient multiplexing of available radio resources to provide higher speeds. 2G and 3G lived largely in parallel.

4G LTE

4G Long-Term Evolution (LTE) introduced two important factors to the field; An all-IP Core Network and enhanced radio access. The underlying network is no longer split between voice and data, rather everything goes across the same network. Instead of splitting traffic from UE based on its content, everything is encapsulated by the eNodeB and forwarded to the Evolved Packet Core (EPC). This allows for IP address allocation to all UE and more advanced networking features such as Quality of Service (QoS) can be applied to the traffic. The Evolved Packet Core (EPC) is made up of four main components: 1) *MME*, 2) *HSS*, 3) *SPGW-U*, and 4) *SPGW-C*. Figure 2.2 shows the components and how they relate to each other.

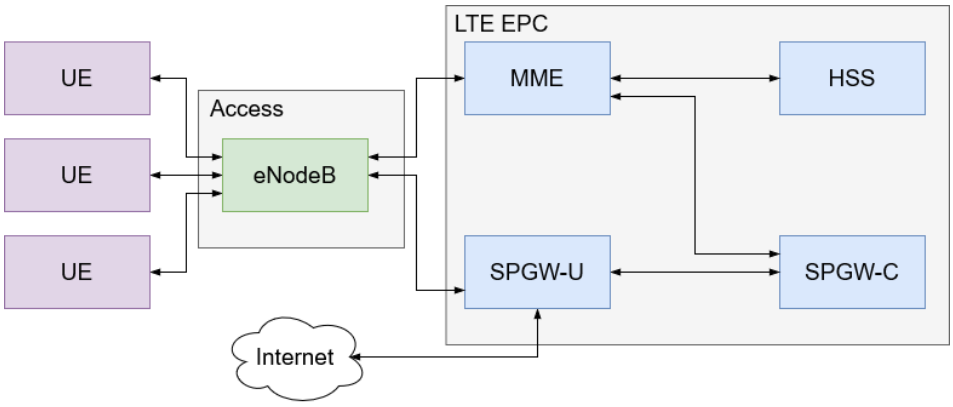


Figure 2.2: A High-level overview of the Virtual Testbed with P4-switches

The Mobility Management Entity (MME) handles connection and mobility management for all UE in its dedicated cell. The MME interacts with the Home Subscriber Server (HSS) to get the required UE information to perform its functions.

The Home Subscriber Service (HSS) acts as a database service that stores all relevant information about the service levels for each UE. This information can be QoS profiles, roaming access capabilities (the ability to

let UE from different providers use the network) and more importantly authentication information. In cases where a roaming UE attempts to attach to the network, the HSS will contact the HSS of the Service Provider to the UE and get the relevant information to decide if the UE should be authorized to attach or not.

The User Plane of the Packet Data Network Gateway (SPGW-U) will handle the user traffic to and from the Internet and between the individual UE attached to the network using GPRS Tunnelling Protocol (GTP) between the RAN and itself.

The Control Plane of the Packet Data Network Gateway (SPGW-C handles the Control Plane traffic, which is the type of traffic used to configure and control the different components of the EPC. It will communicate with the MME and send Control Traffic to the SPGW-U to let it know how to handle User Plane Traffic.

The flow of the traffic in the Core can be generalized: 1) Assuming the UE is successfully attached to the RAN, the MME will detect it through the base station and retrieve its Subscriber Identity Module (SIM) information and contact the HSS to verify if the UE should be allowed to attach to the network. 2) The UE will receive IP configuration from the SPGW-U. 3) user traffic can then flow between the RAN and SPGW-U, and to the Internet if needed.

2.1.2 5G NG

The evolution from 4G to 5G Next Generation Core (NGC) is all about enhancing existing capabilities and introducing a more service-oriented view of the network, as well as improving the EPC. According to a survey on slicing in 5G networks [12], the specific purpose of 5G is yet to be determined. While 4G LTE was focused on human to human traffic through mobile devices, 5G is envisioned to expand on this and cater to many different use-cases. There are three main use-cases identified by the ITU and 3GPP which are enhanced mobile broadband (or human to human communication, as 4G is used for today), massive machine type communication (machine to machine, IoT devices), and critical communication. Each of these can be further broken down into more fine-grained services, each with its own set of requirements such as ability to handle connection density, be highly reliable, or provide high-speed user data rates. The introduction of slicing and virtualisation to such a network makes it possible to deploy either small or large network infrastructures to facilitate a wide range of services. Slicing simply means to divide a physical infrastructure into several smaller virtualized pieces to provide specific service sets and meet diverse customer requirements, which enables the service oriented approach to networking in 5G. This relies on the underlying infrastructure to be able to handle virtualization, which can be done with OpenStack or similar cloud platforms.

An important part of what makes 5G the next logical step in the evolution is the fact that it introduces a softwarized approach to cellular networking. This allows for slicing of the underlying hardware resources

to facilitate the previously mentioned use-cases in an isolated manner on the same hardware. This largely follows the IaC paradigm where a cloud provider's hardware resources are made available for several customers and can be tailored to their needs by the customers themselves. As presented in a survey [12], the architecture of 5G Networks can be summarized as a three-layer architecture with a Management and Orchestration (MANO) entity spanning all three. These layers are 1) Service layer, 2) Network Function layer, 3) Infrastructure layer. As a result, the authors identify a number of challenge areas which is useful to facilitate future work in the field of 5G Networks.

The Service layer service concerns the description of a slice and possibly having predefined slices based on a set of requirements. The Network Function layer concerns provisioning of resources, Life-Cycle Management (LCM), configuration/control of forwarding plane. The main point to take away is the consideration of granularity. Coarse granularity represents a high-level control of functions that deals with network operation. Fine granularity takes each function and divides it into smaller pieces for a higher level of control. The Infrastructure layer concerns deployment, control, and management of the underlying infrastructure. Whenever a new slice needs to be deployed, this layer will reveal and provide available resources which comply with the given resource requirements.

There are several challenges with this new generation of cellular networks, of which three key aspects are brought up as not well-understood and should be focused on in future research. These are 1) RAN Virtualisation, 2) Service Composition with Fine-Grained Network Functions, and 3) End-to-end Slice Orchestration and Management.

RAN Virtualisation exists in the Infrastructure layer and is composed of three main topics. *Radio Resource Isolation* which is the problem of how to isolate several slices from each other on a base station. *Shared/Dedicated resource allocation*, or how to keep slices isolated while still facilitating multiple slices per station with varying SLA requirements. *RaaS*, or RAN as a Service concerns how one would best facilitate virtualised RAN instance creation on the fly and also keep different slices isolated. This challenge has in recent time been addressed by a tool named Orion[15]. This is not directly related to monitoring, but it does address the challenge of RAN Virtualisation and resource isolation. The method by which this is done is base station virtualisation to provide dynamic on-the-fly virtualisation and slicing to meet service requirements. The base station is divided into two parts, the physical layer and the virtual layer. The physical layer is considered as infrastructure and falls under the responsibility of the infrastructure provider. The virtual layer is realised as a hypervisor which manages the different slices on the base station. Isolation is guaranteed through the process of controller isolation, meaning each slice has their own controller which can be separated through well-known techniques such as KVM (Virtual Machines) or Containers (Docker, LXD). The underlying hardware resources are exposed through a novel set of abstractions. Each slice is allocated a set of resources and do not have access to any more than what they are given.

Service Composition with Fine-Grained Network Functions is in the Network Function layer and points out the fact that due to a number of different network equipment vendors, there is currently no interopable and scalable method for service composition.

End-to-End Slice Orchestration and Management exists in the Service Layer and raises the issue of how to translate a high-level service description to a concrete slice. To solve this one could look at designing and implementing a Domain-Specific Language (DSL) which allows an operator to express service characteristics, KPIs and network element capabilities and requirements, similar to how the IaC paradigm tackles this issue.

There are radical changes made to the Network Core as well, which has moved from the very static EPC design to a Service-Oriented design [4]. New components are introduced which replace existing components in the EPC and some with new functionality to complement the move to a cloud environment.

Access and Mobility Management Function (AMF) which provides Access Control and Mobility, similar to the MME node from the 4G LTE EPC. In cases where mobility of UE is not a concern, the Mobility Management is not needed in the AMF.

Session Management Function (SMF) uses network policy in order to manage sessions.

User Plane Function (UPF) acts as the main gateway for user traffic and can be deployed according to service requirements. In cases where a high connection density is expected, multiple UPFs can be deployed as edge nodes to improve throughput and capacity. This combines the SPGW-U and SPGW-C from the 4G LTE EPC.

Policy Control Function (PCF) holds all policies required by the other functions in the NG Core, such as roaming and mobility management.

Unified Data Management (UDM) acts as the database that stores subscriber data and profiles, and is similar to the HSS in the 4G LTE EPC.

NF Repository Function (NRF) provides registration and discovery functionality to the other network functions. It acts as a Service Discovery and allows for functions in the NG Core to find and communicate with other functions using their APIs.

Authentication Server Function (AUSF) is responsible for authenticating the UE that connects to the network.

Network Slice Selection Function (NSSF) maintains a list of all the defined network slices, which the AMF authorizes the use of based on the subscription information stored in the UDM.

Network Exposure Function (NEF) exposes the available APIs in the network core to the other internal functions and in some cases to external 3rd party applications.

Application Function (AF) performs many tasks, such as retrieving information from the NEF, interacting with the PCF, and exposing services to end users.

The Service-Oriented architecture is shown in figure 2.3, which provides a simplified, high-level view of the 5G NG Service-Oriented Archi-

tecture. The 5G NG Core (NGC) is made up several services realized as virtual functions, which can be tailored to fit a given set of requirements. If a network needs to be deployed in a setting where mobility is not a requirement, the AMF can be configured to not include Mobility Management. The general idea is that the NGC can be customized to fit the given requirements to provide an appropriate service level.

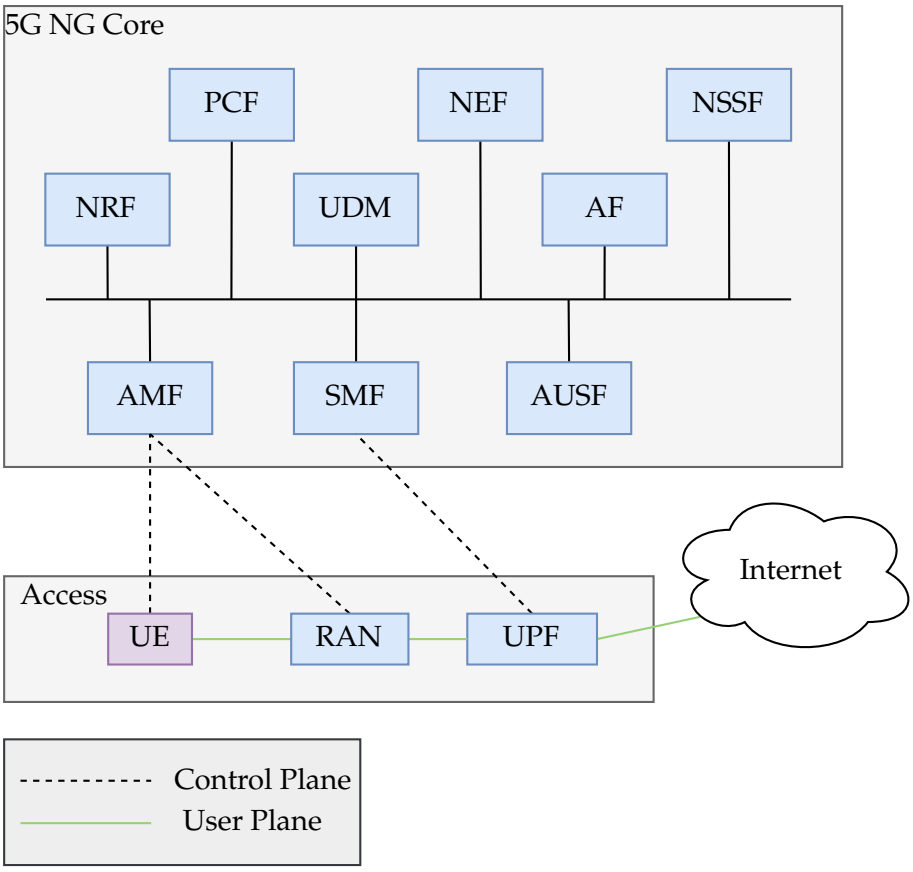


Figure 2.3: A high-level view of the 5G NG Service-Oriented Architecture

There are two main deployment strategies when it comes to 5G Networks, Non-standalone (NSA) and Standalone (SA). The NSA strategy is used in cases where there is an existing 4G LTE EPC which needs increased throughput and capacity in the access network to handle an increase in UE. This model does not use the 5G NG Core, but rather extends the 4G LTE EPC with a gNodeB (a 5G capable base station) to provide the increased performance. The SA strategy uses a complete 5G NG Core as well as an eNodeB to provide access to UE that is not 5G capable.

2.2 Network Monitoring

Monitoring is the act of collecting metrics from devices in any given infrastructure (data centers, enterprise networks, virtualized environments) for the sake of being omniscient and omnipresent. Metrics in this case are KPIs

(Key-Performance Indicators) such as network latency, jitter, delay, or they can be more specific to services, such as service response time or availability. Omniscient in this case means that the operator knows everything that happens in the infrastructure at any given time, and omnipresent means that the monitoring system will be all places all at once, always working. Without monitoring there is no way to know if a service has gone down or if it is functioning as intended. If a service is not measured it is not managed, and unmanaged services generate very little value [16].

Monitoring is largely based on the producer-consumer model in which the producer generates some output which is the input for the consumer. This model is typically expanded with a register, directory service, or broker in order to act as an intermediary node between producers and consumers. The register is also in some cases responsible for locating the producers and making them available for the consumer. This is essentially how monitoring systems work. There can be an agent (producer) which observes the network traffic or system events on a given node, looking for a specific trigger. When triggered, the producer will send a message with some information either directly to a central monitoring server (consumer), or via a message broker [17].

It is important to make the distinction between **Service** and **Network** monitoring. **Service monitoring** usually means to have more insight into an application that runs on some node, including its source code. This type of monitoring concerns itself with application layer metrics such as service response times, CPU and Memory utilization, and other metrics specific to the behaviour of the system or environment on which the application runs. **Network monitoring** however is all about insight into the network infrastructure and its behaviour and state, this also includes network device and client availability, network response times, jitter, delay, and latency [16, 18].

On a general level, monitoring systems can either be coarsely or finely granulated. Granularity refers to the level of detail and the timeliness of the data. **Coarse granularity** means that the collected data contains a wide, usually aggregated, set of metrics across longer periods of time. This reduces the strain put on any given system by monitoring it, but also reduces the level of insight. Events that span a short period of time might not be picked up by the monitoring system. **Fine granularity** is to collect highly specific metrics and events such as the specific TCP header in a given TCP Flow. The immediate concern is the amount of storage required to collect and store all of this information, as such a finely granulated monitoring system requires careful configuration, but can inevitably be more accurate and efficient compared to a coarsely granulated system [1, 19, 20].

In the next few sections, we cover the history of network monitoring and how it has moved from coarsely grained systems such as SNMP, to more fine-grained systems in datacenter environments such as Trumpet [19]. We also discuss some of the different methods used to collect data and some of their strengths and weaknesses.

2.2.1 Coarsely and Finely Grained Systems

In the early days of networking a system was needed to collect specific information from network devices. SNMP (Simple Network Management Protocol) is comprised of three main components; the SNMP agent, the SNMP manager, and MIBs. The agent lives on a device and is responsible to either respond to periodic polling from the manager, or react to a trigger on the device and send some data to the manager. The manager simply polls the agents or receives requests triggered by some event. The MIB (Management Information Base) is a collection of variables the SNMP agent can use to send information to the manager. A MIB variable can be the switch model, os version, or which interface that has gone down if triggered by such an event [16, 18]. As SNMP only periodically collects some predefined set of variables, or reacts to only certain events which must be configured manually, it has a very poor sampling rate and misses a lot of crucial information. These coarsely granulated systems are highly inefficient as the network complexity grows, and they tend to become a source of confusion due to the amount of total information collected, which is not very precise [1, 19, 21]

These old, coarsely granulated systems belong to the bottom-approach to network telemetry. It involves a high degree of configuration per network device, and in the more modern, highly complex networks of today this is simply not sustainable. These old-school systems often result in datasets with high overhead and low precision and accuracy, and they often miss out on critical events in the network.

Recently a more modern approach has begun to take hold in monitoring of datacenter and enterprise networks, where each device is no longer configured manually, but rather a network-wide system can be used to distribute queries to the appropriate agents. An early work which really embraced this is NetSight [22] which was one of the first to point out that network diagnosis tools do not have the capability to process high-level queries and do not have enough historical information to process them. The solution was to develop NetSight, a platform consisting of four applications in order to keep a detailed packet history. The goal was to have the answers to the what, where, how and why of the packet. *What* the packet headers looked like on a given switch, *where* the packet was forwarded (switches, ports), *how* the packet headers were modified if at all, and *why* it was forwarded (match-action taken on the flow and the flow table). The goal of the project was to design and implement a platform which allowed for high-level queries that would perform the brunt of the troubleshooting for an operator.

Later works include Trumpet [19] and Sonata [20]. The former is an event monitoring system that monitors every packet and produces a report at millisecond timescales. The main problem Trumpet tackles is the fact that modern monitoring solutions in data centers operate on a very coarse timescale. This is fine for human readability, but is insufficient to provide the data necessary to perform traffic engineering. The system is based on each end-host performing complete packet monitoring at full line rate and

reports events based on triggers. These triggers can be programmed, which allows the system to monitor events that are important in data centers. An event has two elements: a packet filter and a predicate. The packet filter defines what packets to monitor for specific events, the predicate defines the conditions that must be true in order for the system to detect that an event has occurred.

Sonata (Streaming Network Traffic Analysis) essentially defines a query interface to perform trigger-based monitoring on programmable switches. It will program a switch to look for traffic that matches the query which can be analysed to detect network attacks, such as SYN Flooding, or other DDoS methods. The work is experimental and tests have currently been done with two different models of switches (one using software switching, the other hardware switching). The system consists of two main parts, the runtime on a programmable switch and a stream processor. The runtime will accept a query and compile it to code that can run on the switch in order to inspect incoming traffic that matches. If the query is computing-intensive the query can be partitioned between the stream processor and switch. All matching traffic gets mirrored to the stream processor which will then perform the more demanding parts of the query. However, Sonata cannot distribute queries across multiple switches, but [23] provides a combination of Sonata and another tool named Herd to provide a scalable and network-wide telemetry system. Herd essentially distributes a query across multiple switches, a feature missing from Sonata as it was presented in the original paper. The main difference I would like to point out between Trumpet and Sonata is that the former is an end-host based monitoring system, and the latter is a switch-based monitoring system.

2.2.2 New Approaches to Monitoring

The systems discussed previously, Trumpet and Sonata, belong to the more modern Top-Down approach to network telemetry [1]. This is a new approach to designing and implementing monitoring systems. The author presents a good argument as to why the traditional way of monitoring networks, i.e. individual configuration of each device using methods with generally high overhead, should be replaced with a more modern top-down approach. The top-down approach borrows ideas from the IaC paradigm [13] such as a more programmatic approach. The approach sets some requirements that a new monitoring system should aspire to fulfill. These requirements are 1) a declarative measurement abstraction, 2) an efficient network-wide runtime system, and 3) new measurement primitives at devices.

A declarative measurement abstraction means that the operator of the monitoring system (MS) should be able to write their query in plain text, e.g. ”in region EU, watch all outgoing TCP connections with destination port 8080”, of course in a real-world setting the language would not be this plain, but it works to illustrate the point. The abstraction should be intent based (”watch all outgoing TCP connections”), use named principles (”in region EU”), be network-wide (i.e. monitor the entire communication path

until the traffic leaves the network), and support many concurrent queries to enable cases of many different management tasks and multi-tenancy or multi-application environments. A challenge with this requirement related to 5G networks is that this kind of network bring its own set of requirements regarding resource constraints, scalability and new types of queries that must be addressed when designing a new MS.

An efficient network-wide runtime system should take the abstracted measurement query and translate it to functional queries that the network devices understand. In addition, the runtime should return information in real-time, it should support different granularities (both aggregated queries and fine-grained queries), and it should also provide different levels of accuracy (meet given accuracy requirements, while staying within resource constraints). In order to achieve this, one will need new data structures and algorithms to store and analyze collected data, and of course reduce the bandwidth usage between devices and the monitoring system agent/server. Lastly the runtime should efficiently multiplex its available resources between multiple queries. A related challenge is how to handle large volumes of traffic and handling of dynamics such as changes to the routing topology and traffic patterns.

New measurement primitives at devices has two terms that need to be explained further, primitives and devices. A primitive refers to the actual metric the MS will collect, including but not limited to CPU, memory, and storage usage, additional primitives can be the current throughput on a given Network Interface Controller (NIC). A device in a network context can be one of several types, such as hosts, switches, and other programmable devices. The challenge with designing and implementing new primitives is that you cannot design the most efficient algorithm and data structure per measurement query, as new queries will be made as operators gain more familiarity with the MS and their own network. Considering this the new primitives must be generic and efficient to support many different devices and meet performance requirements. Another consideration when designing such primitives is the fact that different types of devices each have their own level on insight into network traffic. Hosts are close to a given application, whereas switches are closer to aggregate information about the general flow of network traffic and has a higher degree of insight into in-network failures. To conclude, the MS should be able to quickly correlate information from a diverse set of devices to provide a unified view of the network to the operators. One challenge pertinent to 5G networks is how to design the primitives of services in these networks, e.g. what primitives should be available for queries on eNodeB's (base stations), AMF, and PCF which are just some of the services required to provide 5G connectivity to users.

The trend of moving more computation and monitoring to the network devices, as in the case of Sonata, is clear, and some have discussed when and what parts of an application should be outsourced to network devices. [24] proposes a framework to classify different types of applications and how they would benefit from utilizing in-network processing. The reason this is relevant is because some Monitoring Systems are by nature in-

network and some of the monitoring-related tasks can be performed by other in-network devices, such as data aggregation and timestamping to name a couple. The framework presents five principles which should be followed when performing in-network computation: 1) *Offload primitives, not applications,* 2) *Make primitives reusable,* 3) *Preserve fate sharing,* 4) *Keep state out of the network,* and 5) *Minimal interference*.

Offload primitives, not applications suggests that primitives such as timestamps and aggregation could be performed on in-network devices. This means that the dedicated server running a monitoring system can skip timestamp-generation and data aggregation where possible to further improve efficiency.

Make primitives reusable means to have a standardized set of primitives that are always available on the in-network device which can be used by several applications.

Preserve fate sharing essentially means that when one device fails it should not affect the rest of the network or application, unless it is a single point of failure.

Keep state out of the network. Only maintain a soft-state in the network, lost data should be recoverable from a server.

Minimal interference should go without saying. Any application offloading some of its computation to an in-network device should not have any say on routing policies and similar network-specific functionality.

2.2.3 Active and Passive Monitoring

Before we can discuss the different data collection methods used in monitoring, there are two general types of systems that must be explained. Active and Passive systems.

Active monitoring. This type of monitoring will use an event to trigger some action. In network monitoring, the event can be a detected TCP retransmission, and the action would be to send an ICMP packet in order to discover the path of the TCP packet. This can be used for many different operations, but one notable tool is 007 which consists of a monitoring agent that runs on the end-host and a central analysis agent. There are three components to 007: *TCP monitoring agent* which monitors for retransmissions. When a retransmission is detected, the next component takes over. The *Path Discovery Agent* will identify the path of the retransmitted flow using a specialised ICMP packet and mark the path as bad. The last component, the *Analysis Agent* will tally the votes every 30s to find the most voted paths and links. 007 is a highly specific tool, used only to identify the links that cause packet drops and does so with very high accuracy. During a 2-month long trial, 007 found every problem which other similar tools also detected, but it also found previously undetected problems [21].

Passive monitoring. In this case the monitoring agent or software will simply listen and observe the system it operates on and look for certain events which it will collect and send to a central entity. Previously discussed systems that can be classified as Passive are Trumpet, Sonata,

SNMP, and NetSight. The purpose of passive monitoring is to detect and report on the current state and current problems in the network, as well as letting operators see trends and potential problems, allowing them to react to events as they happen.

2.2.4 Data Collection Methods

This section goes into some detail on the different methods for collecting data, and some strategies to consider for later chapters. Monitoring is essentially a producer-consumer type system, and as such many of the following methods will be explained using this terminology. Generally speaking the monitored node is the producer and the collection agent is the consumer.

There are two main methods for collecting data and information in any given system, push and pull. In pull-based systems the consumer is the initiator of all communication. Such systems are less precise but they have the smallest overhead, as the pulling happens in set intervals. This method uses a very low amount of bandwidth and consume the least resources (CPU, Memory, Disk IO) vs. Push-methods. There is a substantial risk that the monitoring system using this method will miss out on many critical events if the pulling-interval is low.

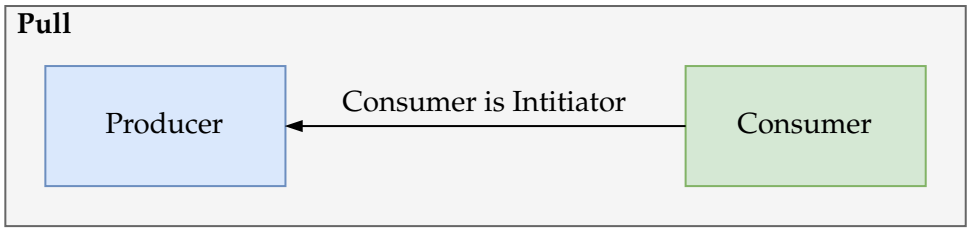


Figure 2.4: Pull-method

In push-based systems the producer is the initiator of all communication. These are more precise than pull-based systems, but also have the highest overhead, especially if the system is poorly configured. The general idea is to have a monitoring system on any given node which monitors for events that exceed some resource threshold, such as CPU utilization peaking above some given percentage. When an event has been detected it will send the relevant alert to a central entity, or if the monitoring system is an active one it will perform some action which can be to restart a service or perform a tracepath to provide additional information to the central entity.

Combination of Push and Pull: Ideally we would want an adaptive model that can decide the data collection method for each individual node. This has been explored in literature, known as the P&P model (Push and Pull) [17], which uses a variable called User Tolerant Degree (UTD) which describes how tolerant the user is to status inaccuracy of the system that is being monitored. For example if the service owner or user needs continuous updates for a critical service, UTD can be considered small. If the only requirement is a regular heartbeat from a service every fifth

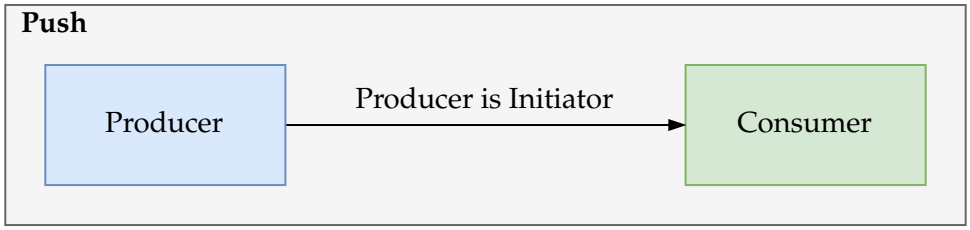


Figure 2.5: Push-method

minute, UTD is considered large. If UTD is large, the model prefers Pull, if it is small Push is preferred. If it is moderate, neither is preferred and both will be used interchangeably. This model can then form the baseline of which method that should be used in different situations.

2.2.5 Levels of insight

The level of insight a monitoring system has into a monitored node can be classified into two types: 1) *black-box*, and 2) *white-box*.

When using black-box monitoring, the system has no direct insight into the nodes it monitors as shown in figure 2.6. It can only see basic metrics such as traffic going to and from the node, it can also measure how long the node takes to respond or if it is up or down, and some basic HTTP requests can be performed. This is useful in cases where a customer denies access to the inner workings of a given network service, but still wants the infrastructure provider to have some sort of monitoring in place.

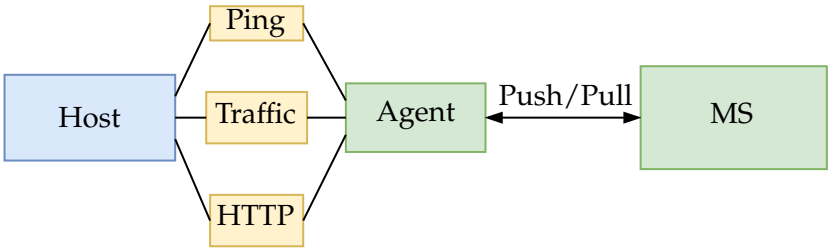


Figure 2.6: Black-box monitoring

White-box monitoring assumes nearly a full access to the node's inner workings as shown in figure 2.7. This includes resource usage (CPU, Memory, bandwidth, storage), as well as how the operating system behaves. In some cases the monitoring system can have access to logs from any running application to record errors and events, but this is usually a task for a logging system.

2.2.6 Monitoring System Placement

Next is the placement of the monitoring system and its agents in a given infrastructure. This can be explained as two main types: 1) in-network, and 2) end-host. The placement is important because it more or less dictates the level of insight into the nodes of the network. We can consider that the

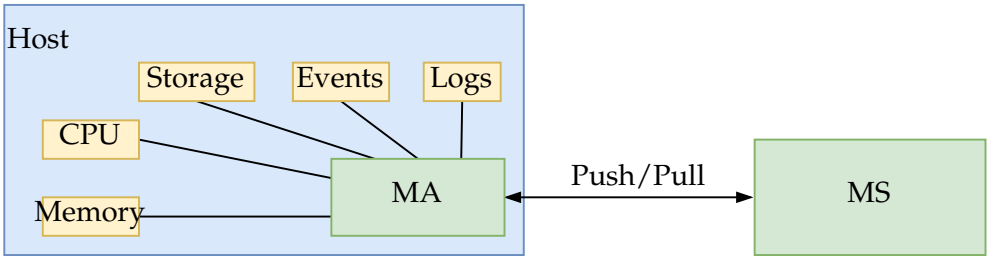


Figure 2.7: White-box monitoring

former is more akin to black-box monitoring, whereas the latter is more similar to white-box monitoring.

In-network placement as shown in figure 2.8 leverages programmable switches and port mirroring in order to observe the traffic going through the network. It allows for insight into every traffic flow through any given switch which can be used to paint a picture of the network-wide state and can be used to detect many different incidents, such as heavy hitters or multiple types of attack as shown with Sonata. In-network placement can also be used by other systems such as IDS/IPS which requires a large amount of traffic in order to analyse it for potential attacks and intrusions, in such a case all traffic would be mirrored from a given switch to a dedicated analysis node.

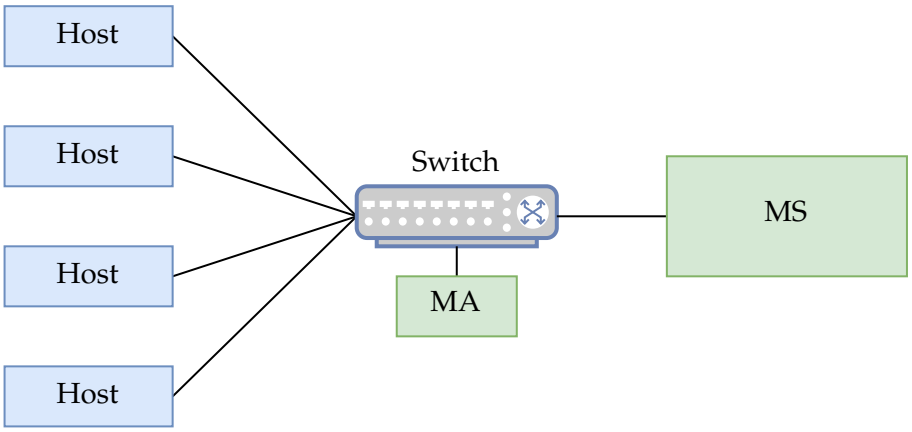


Figure 2.8: In-network monitoring

With End-host placement the monitoring agent is placed on the end-hosts throughout the network as shown in figure 2.9. This allows for much more detailed insight into the specific state of each node, as explained with white-box monitoring. Due to the flexibility of cloud environments, recent works have implemented packet capture modules on the virtual switch present on most cloud nodes in order to perform traffic sniffing and mirroring before the traffic enters the network [19, 21, 25]. Such systems utilize a push-model for collecting data and usually have a central analysis entity which does most of the heavy lifting.

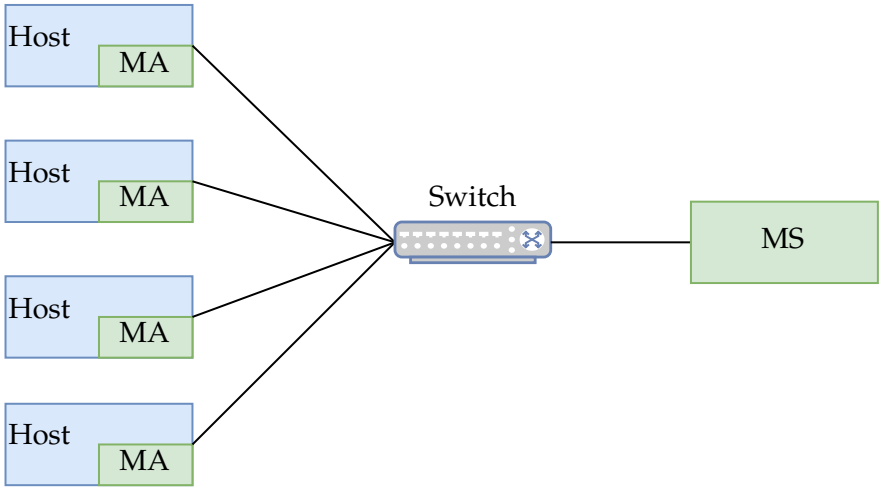


Figure 2.9: End-host monitoring

2.2.7 Use-cases for Monitoring

Now that the basics of monitoring are presented, we can discuss some of the different use cases for monitoring. There are several such cases, some are directly related to gaining insight into the network state, others are more related to network automation and security.

The main use-case for monitoring is to gain insight into the state of the network at any point in time. A network state refers to how it operates and functions throughout a given time period. To give an example, a monitoring system can be deployed on several switches in order to check if there are certain flows or paths which experience a significant packet loss or delay.

By monitoring incoming traffic load to any given application node we can use thresholds to trigger automatic scaling by adding or removing nodes. There are several ways of doing this, the simplest variant is to have a python-script scraping metrics from a load-balancer at regular intervals, and if the current number of sessions or traffic load exceeds a given threshold, the script can add or remove a node in a cloud environment. Such usage of monitoring can save some money by reducing the amount of unused resources in a pay-as-you-go cloud environment such as AWS.

Sonata can be used to discover a wide variety of different network attacks by looking directly at the flows going through a given switch. This can trigger automatic filtering of such traffic or raise an alert and let an operator investigate. IDS/IPS rely heavily on sniffed network traffic in order to generate baselines and detect anomalous traffic. **Resource placement:** Coupled with Machine Learning, monitoring can be used to find optimal resource placement in an NFVI (NFV Infrastructure). A good example of such a system is Z-TORCH[26] which combines the MANO with Monitoring and Machine Learning. Their monitoring solution uses generic KPI (Key Performance Indicators) which are relevant for any given VNF. Some of these indicators are *Network Load, Computational Burden*

on the underlying hardware, and *Storage Utilization*. Z-TORCH uses these metrics to identify which VNF profiles that are high demanding and will attempt to provide optimal placement of these VNFs in the underlying infrastructure. For this purpose they use Machine Learning and unsupervised learning to create a model of which VNF profiles that are high demanding and which that are not.

Both Active and Passive monitoring can be leveraged to introduce self-healing to a network or a service. If certain events are discovered, such as log files exceeding maximum size for a small node, the monitoring system can trigger a job to backup and reset the log file to conserve disk space.

The amount of data generated by monitoring systems is significant, some estimates show that it accounts for 25% of enterprise data storage. Machine learning models, and by extension neural networks, require a huge amount of data in order to be properly trained. By utilizing the data generated by service and network monitoring, instead of generating training data in simulated networks, the resulting accuracy of the models can potentially be increased.

2.2.8 Monitoring in 5G NG

There is no specific and standardized approach to monitoring 5G networks. We believe this is because the new generation is largely based around SDN, NFVs and deployment is done in datacenters and cloud environments, which means that monitoring approaches that work for modern service monitoring can in theory be applied to 5G NGC. There are several systems and frameworks for monitoring software-defined networks which could be applied to 5G networks.

OpenNetMon [27] is an adaptive polling scheme which leverages OpenFlow-capable switches to monitor the flow of traffic. In cases where the flow rates vary between sampling, the polling rate will be increased. When the flow rate on any given switch stabilizes, the polling rate can be reduced. This ensures a minimal level of resource usage on the network devices, but keeps a fair level of accuracy.

Software Defined Monitoring[28] is a proposed framework to integrate a monitoring controller into a software defined network. It utilizes probes on all nodes and a central collection agent called the SDM Controller. By leveraging software-defined elements, this system can be integrated into VNFs or OpenFlow capable switches.

IoT based framework for monitoring [29] leverages IoT devices to collect information about the network. It essentially treats every programmable device or node (wireless routers, phones, switches, network functions in VNFs) as a producer, each producer then sends its measurements to a broker. The broker acts as a message queue, and allows a knowledge base (which acts as the consumer) to connect and get measurements when it has available capacity.

PayLess [30] is one of the more known monitoring frameworks for SDNs and provides a RESTful API interface to query an OpenFlow controller for information about flows in the network. PayLess is a

pull-based framework and allows for developers to rapidly prototype monitoring systems on top of the framework.

DCM [31] (Distributed Collaborative Monitoring) provides a distributed system to allow for programmable switches to collaborate to achieve flow monitoring tasks and balance measurement load by leveraging two-stage bloom filters. It essentially allows the monitoring system to choose the switch that fits best for a given monitoring task.

2.2.9 Challenges

This section presents some of the challenges related to 5G Networks and networking monitoring in such an environment. Most of the challenges are related to available resources, isolation and sharing of these resources and multitenancy in cloud environments.

Resource Isolation and Sharing: As pointed out in section 2.1.2, such networks are often realized as a VNF. This means that a provider of 5G might have several VNFs existing in the same datacenter in which each VNF addresses a different use-case such as critical communication or user access. In such a case, should the monitoring system be network-wide in the sense that one system monitors several VNFs, or should it be one MS per VNF. How would multiple monitoring systems be implemented in this case? If they are in-network based systems, will they interfere with each other, and how would that impact the hardware resources made available to them.

Multitenancy: If several providers of 5G approach the same Cloud provider to host their Network Services, should each provider monitor their own systems, or should the cloud provider have their own solution made available to their clients.

Centralisation: If a 5G provider has multiple NS to handle several use-cases, should the provider monitor all of these services using one central system or implement one MS per service (also related to multitenancy, should there be a central entity for the cloud infrastructure, how should it operate, and should it have white-box knowledge on all VNFs/NS')

Resource constraints: Monitoring systems should not incur any significant cost in terms of resources and generate a minimal amount of interference on the network traffic. There must be a balance between accuracy and overhead, the former must be maintained while the latter should be minimized. Several works as previously discussed have different approaches to this, such as aggregation of metrics, adaptive polling schemes or distributed query schemes.

2.3 Software Defined Networking

Software Defined Networks (SDN) has in the last decade seen a substantial rise in popularity, mainly in data centers for public and private cloud usage [32]. Software switches such as OpenVSwitch (OVS) delivers high-end performance while being highly customizable [33]. SDN generally focuses

on increased programmability and the ability to control the network from a central controller. It allows new monitoring systems to integrate with this controller to allow for flow based monitoring, and will as a result become almost completely network wide. The consequences and implications of more SDN-dominant data centers are the clear need for monitoring systems which integrates well into these environments.

2.3.1 INT

In-band Network Telemetry has also begun to make its mark on the field [2, 34]. Languages like P4 enables an operator or technician to program a match-action pipeline for very fine grained control of how the network device forwards the data. It essentially allows for operations such as measuring time-per-switch in a packet during its traversal through the network by continually updating a custom packet field per network hop. A good example of such a system is a scheme to detect network-wide heavy-hitters [35] which uses the P4 language to essentially treat every switch in a network as one in order to discover network-wide heavy hitters. Each switch is configured to look for flows that exceed some bandwidth threshold and stores some metrics about those flows. A coordinator-node will then receive reports from all switches where a flow has exceeded the threshold and combines the count for a flow seen across multiple switches. If this count exceeds the global threshold, the coordinator will poll all the switches for their current count of the flow in order to verify that it is a network-wide heavy hitter.

In-network computation and In-band Network Telemetry (INT) are not without problems and challenges. [36] Shows that unrestrained usage of INT will substantially degrade network performance unless measures are taken to minimize overhead in the data, management and control planes. A comprehensive survey also raises several issues, such as new and unforeseen threat vectors when introducing in-network processing to enterprise networks, or how INT operations will be orchestrated [37]. INT is a major addition to network monitoring and will most likely see increased usage in the years to come.

2.3.2 P4

P4 is a data plane programming language focusing on the processing of packets in the ingress and egress directions of a programmable switch [3]. It is named P4 for Programming Protocol-Independent Packet Processors. The P4 language also introduces a major challenge of simulated test environments, as it is its own complete language and requires specialized software switches in order to run as intended. The general flow of a P4 program consists of parsing and deparsing traffic in either the ingress or egress directions of the interfaces on the programmable switch. A single parser-function can for instance be used to extract the destination IP address from the IPv4 packet, and set the desired egress interface of the switch.

A typical P4 program follows a relatively simple flow:

- **The Parser** instructs the switch how it will unpack the incoming packets, so for example extract the etherType field from the ethernet-header, and inspect it, then take further action based on this. The parser is then applied to all packets processed by the switch.
- **The ingress and egress** processors allow for definition of actions and tables. This is where the action packet processing is executed. Here the content of a given header can be read and altered, and further action decided based on the content of the header.
- **The Deparser** will restructure the packet in a very specific order, for example the ethernet-header must come before the IP-header. What this essentially does is release the packet back into the data plane in order to be forwarded with the newly updated headers.

With P4 we can also implement INT. In order to better explain how P4 and INT works, we provide here a simplified overview of packet-switched computer networks. The units of information in this type of network are called Protocol Data Units (PDU), which are single units of information that are transmitted in a network. PDUs generally consists of a set of headers with a given width or length in bits. These headers can be read and updated using Data Plane Programming. To provide more context, we have included here a simplified explanation of modern packet-switched networks using the TCP/IP model as a reference model. The TCP/IP model consists of four layers: 1) Link, 2) Network, 3) Transport, 4) Application.

The Link Layer PDU is called a MAC Frame. Simply explained it is made up by a destination and source MAC address, and a payload. The payload in packet-switched networks is usually IPv4 or IPv6 packets.

The Network Layer PDU is called a Packet and consists of a destination and source IP address, and another payload.

The Transport Layer PDU is called either a segment or a datagram, depending on whether the traffic uses TCP or UDP respectively. This consists of a destination and source port, and of course a payload. The destination port is used to identify which application on the target host will receive the data, and the source port is used to determine which port the reply should use.

The Application Layer has no specific PDU associated with it as all the protocols that operate on this layer are encapsulated in TCP or UDP PDUs, and uses the destination port to define which application that will receive the data. HTTP/S, SSH, DNS all exist in this layer and are encapsulated in either segments or datagrams.

This also explains the most common data-structure used in network monitoring, the 5-tuple, which consists of source and destination IP address, source and destination port, and transport protocol (i.e. TCP, UDP, SCTP, or similar). The source and destination MAC addresses of the MAC frame are not used as often, as these fields can change multiple times as

a PDU flows through a network. The 5-tuple is highly useful as it can be used to create a Hash which works as an ID of the traffic flow. A flow can be considered any traffic from a specific host to another specific host, on a specific set of ports, with a specific protocol. Which means that if host A opens a TCP stream to Host B, the traffic from A to B is considered one flow, and traffic from B to A is its own distinct flow.

2.3.3 Software Switches

The main enabling component of Software Defined Networks is the software switch. Most SDN implementations rely on an OpenFlow Controller to manage and control any software switch running the OpenFlow protocol [38]. The OpenVSwitch has become the de facto industry standard software switch in Software Defined Networks. In the context of this thesis it has one main limitation, which is that it is not compatible with P4 out of the box.

Behavioral Model V2

The Behavioral Model switch is the reference switch for P4 [8]. It is not intended as a production-grade switch and the performance is not at all what one can expect from more advanced software switches. It will work well to develop a prototype P4 program which can be used as a template for future development of INT in EPCs and RANs. The switch architecture does have some limitations, such as no support for floating point arithmetic and modulo operations.

OpenVSwitch

OVS is a software switch which implements most of the well-known network protocols and standards necessary to have an operational enterprise network [33]. The main use-case of OVS is in data centers and on hypervisors running multiple VMs. An instance of OVS is usually tethered to a controller using OpenFlow. This enables the switch to send the first packet in a flow to the controller and receive instructions on how the entire flow should be forwarded. This is a reactive approach in which the flow must first be observed before the switch can determine an appropriate action.

P4-OvS

P4-OvS is an extension of OVS [39]. By extending the original implementation with Berkely Packet Filter (BPF) and a P4-to-uBPF compiler, P4 code can be translated to run on this system. This allows a user to write P4 programs that operate on OVS instances with greatly increased performance compared to the BMV2 switch.

2.3.4 Network Emulation

It has become possible to emulate a fully functional enterprise network on commercial hardware with Mininet [40]. Since the introduction of Mininet, several other implementation have been developed, such as ContainerNet and FOP4. Each of these are presented here in turn, but it is important to note that ContainerNet and FOP4 are both forks of Mininet and implement the same techniques for management and orchestration.

Mininet is a system for rapidly prototyping large networks on a single host. It allows the user to deploy N emulated hosts attached to M instances of OpenVSwitches. The Mininet system is highly flexible and interactive, allowing the user to add and remove hosts at runtime. It also provides realistic results in terms of network behaviour.

ContainerNet [41] is an extension of Mininet that allows the user to deploy Docker Containers as hosts, in order to test out their own applications in an emulated network. This is also part of the OpenSource MANO (OSM) project and is frequently used in the field of network function virtualisation and cloud computing research.

FOP4 [9] is a further extension of the ContainerNet project that adds the capability to deploy BMV2 switches in a Mininet topology. This allows the user to write and test P4 programs in just about any type of deployment compatible with ContainerNet, and by extension Mininet.

2.3.5 NFVs

Network Function Virtualisation (NFV) as defined by ETSI[42] is an enabling technology to virtualize services that traditionally have been placed directly in the network as hardware appliances, such as cellular network nodes. The value of this is that Virtual Private Clouds (VPCs) and other Cloud platforms can be used to deploy complete network architectures. One immediate issue with NFVs is that there is no clear answer on whether or not the virtualized version of a service outperforms the hardware appliance. One advantage that is important to point out with NFVs is that monitoring systems that already work in datacenter environments can now be leveraged to monitor for example 5G NGC due to the fact that it can now live in a datacenter. Figure 2.10 shows the general structure of a Virtual Network Function (VNF) with its Virtual Deployment Units (VDU) and Virtual Links (VL) which connect these VDUs together.

An NFVI (NFV Infrastructure) consists of one or more VNFs which make up a Network Service. VNFs are configured and deployed by making a VNF Descriptor (VNFD) which is essentially just a YAML-file containing the necessary parameters to get the VNF up and running, such parameters can be the image of a node or how the nodes are interconnected. On top of this you can have an NSD (Network Service Descriptor) which combines multiple VNFDs to deploy a network service. There are many benefits to using NFVs, such as reduced deployment time, reduced equipment cost, flexible VNF placement, and supporting multitenancy. Figure 2.11 shows how multiple VNFs are grouped together to form a service.

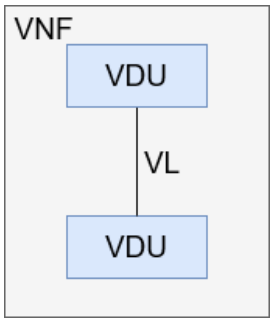


Figure 2.10: A high-level view of VNF with its VDUs and VLs

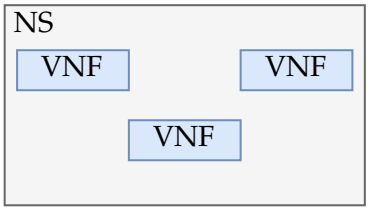


Figure 2.11: A Network Service consisting of several VNFs

2.4 OpenAirInterface

The OpenAirInterface (OAI) Consortium is a group which consists of members from all over the world and from many technology companies and firms. They have created the OAI RAN and EPC implementations for 4G LTE and are currently developing the necessary components to realize a 5G Next-Generation Core (NGC) and RAN [7]. The OAI RAN and EPC are both free and open-source with a comprehensive set of guides and tutorials for both users and developers [43, 44]. The next sections will present a brief overview of the EPC and RAN, and their components and core functionality.

EPC

The EPC consists of four main components which are shown in figure 2.12, in which we show how the RAN is related to the EPC and which components it connects to. Each component serves a distinct purpose and communicates with other components as needed. For a more detailed explanation of the EPC components, refer to section 2.1.1.

The flow of the traffic in the Core can be generalized: 1) Assuming the UE is successfully attached to the RAN, the MME will detect it through the base station and retrieve its Subscriber Identity Module (SIM) information and contact the HSS to verify if the UE should be allowed to attach to the network. 2) The UE will receive IP configuration from the SPGW-U. 3) user traffic can then flow between the RAN and SPGW-U, and to the Internet if needed. The OAI EPC project can be deployed as standalone applications on dedicated hosts or as Containers in Docker or similar Container Management Systems.

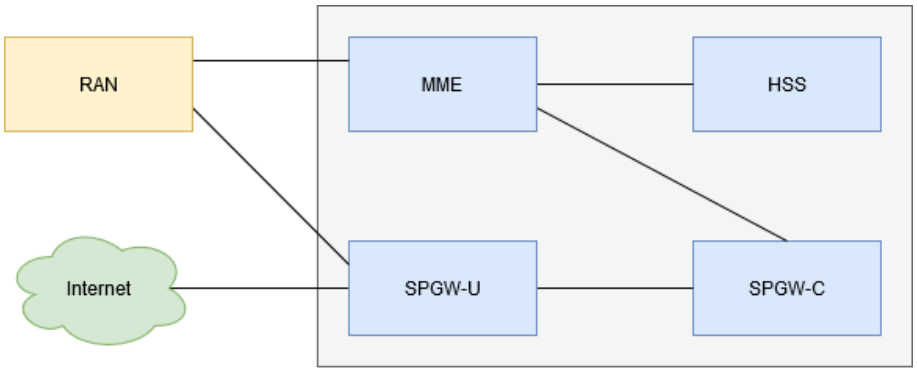


Figure 2.12: A high-level view of the 4G LTE Architecture

RAN

The OAI RAN project is an open-source platform used to deploy a mock network with a high level of realism. What that means is that we can deploy a software defined base station, which in the 4G LTE realm is the eNodeB (eNB), as well as simulate UE using different emulators. The project also includes support for many different hardware platforms in order to create a RAN to which physical UE can connect.

2.5 Related Works

There are mainly two fields that are relevant to our work here. The first is SDN, which includes by extension many of the solutions and projects that are integrated into our project. The second field is naturally Network Monitoring. Here we describe the works in these fields that closely relates the work in this project.

SDNs became increasingly popular with the introduction of the OpenFlow protocol [38]. This allowed for the design and implementation of Mininet [40], which enabled the simulation of fully functional enterprise networks in a laptop. Later, OpenVSwitch [33] was developed and integrated with Mininet. Shortly after the introduction of SDN and Mininet, ETSI proposed the NFV Standard [42] which is an enabling technology to virtualize network functions that have previously required specialised hardware, such as the different components in a 4G Core network. This also resulted in the development of Open Source MANO (OSM), which is a Management and Orchestration (MANO) system to deploy and control NFVIs in cloud environments. As a consequence of the ETSI NFV Standard, the OpenAirInterface (OAI) project [7] could begin development. The OAI project provides the necessary software in order to run a complete 4G LTE RAN and EPC network on commercial hardware. Mininet was also taken a step further with the introduction of ContainerNet [41], which added the capability to use Docker Containers as hosts in a Mininet topology. ContainerNet has also been integrated with OSM as a NFV Multi-PoP emulation platform.

As a consequence of the previously mentioned projects, standards, and innovations, there was a substantial push to create virtual testbeds for mobile radio and core networks. By utilizing the OAI Projects, several testbeds have been developed and deployed at many universities and research institutions, such as Oslo Metropolitan University, Simula Metropolitan Center for Digital Engineering, and Oulo University [6]. These are all fairly recent developments, but have been used to explore network slicing in 5G Networks and utilize various cloud environments such as OpenStack and VMWare. The MANO of these vary, SimulaMet's testbed relies on OSM [45] for the orchestration of OAIs core components. OsloMet's testbed uses OpenStack's own Heat system [46, 47] and also integrates Cisco switches into the environment for further separation with the use of VLANs, and also uses the OAI project. Finally, Oulu University's testbed is deployed to a combined environment of OpenStack and VMWare and uses OSM for orchestration. This testbed differs from the other testbeds discussed previously in that it uses the 5G Test Network (5GTN) [48] instead of the OAI project.

In the field of network monitoring there have been many additions and innovations over the past decade. With the introduction of SDNs, we have seen suggestions that a new approach is needed as a whole. The Top-Down approach suggests that monitoring should move from individual configuration of network devices and hosts, to a more programmatic approach [1]. There are several works which apply this approach with network-wide capabilities such as Trumpet and Sonata [19, 20], and further extensions of the latter have further increased its network-wide capabilites [23].

Another relatively young, but promising addition to network monitoring is In-band Network Telemetry (INT) [49]. INT aims to provide the capability to leverage the data plane of programmable switches to inspect packets in real time. This allows one to perform monitoring and telemetry collection by utilizing the packets themselves with minimal overhead compared to more traditional approaches such as passive and active monitoring. The main programming language to realize INT is P4 [50]. This enables an operator or technician to program a match-action pipeline for very fine grained control of how the network device forwards the data. It essentially allows for operations such as measuring time-per-switch in a packet during its traversal through the network by continually updating a custom packet field per network hop.

In-network computation and In-band Network Telemetry (INT) are not without problems and challenges. It has been shown that unrestrained usage of INT will substantially degrade network performance [36]. That is unless measures are taken to minimize overhead in the data, management and control planes. A comprehensive survey also raises several issues, such as new and unforeseen threat vectors when introducing in-network processing to enterprise networks, or how INT operations will be orchestrated [37]. INT is a major addition to network monitoring and will most likely see increased usage in the years to come. These problems are further supported by a paper that investigates how INT can be applied in industrial

wireless sensor networks [51]. Here the authors point out that P4 is not designed with the limitations and characteristics of such networks.

As INT and P4 has become increasingly popular, there has also risen a need to integrate P4-capable switches such as the BMV2 into the aforementioned mobile network testbeds. To the best of our knowledge there are no published works on this exact feature. There are however works to provide P4-capabilites to network emulation, such as the Function Offloading Prototyping with P4 (FOP4) project. This provides the necessary libraries and scripts to deploy a Mininet topology with Docker containers as hosts and has the option to replace the stock OVS networking with BMV2 switches, which are P4-capable [9]. Another interesting project is the P4-OvS, which aims to integrate P4 with the OvS and enable it to use much, if not all, of the functionality that P4 provides [39].

2.6 Summary

In this chapter we have outlined the background material which lays the groundwork for the rest of the thesis. Network Monitoring is a complex and large field with a lot of existing research. There currently is a paradigm shift with increased focus on software defined networking and finely grained monitoring systems. This is a direct result of the cloudification of the Internet. The consequences of which can be seen with the new 5G NGC which heavily relies on NFVs and cloud computing. The next chapter presents our methodology and approach to investigate the research questions we present in chapter 1.

Chapter 3

Methodology

This chapter presents the chosen methodology by which we will investigate the proposed research questions. In order to investigate these questions, a testbed must be designed, developed and implemented. We must also design and develop the necessary INT systems, as well as choose the appropriate systems by which we will implement Active and Passive monitoring.

3.1 The Testbed

There are several requirements to the testbed. It must support the integration of P4-capable switches, and it must be able to run the OpenAirInterface EPC and RAN. The EPC can be built and deployed as Docker Containers, and the RAN binaries can be compiled and run on a dedicated virtual machine. There is currently no official support to run the RAN components, i.e. the eNodeB and User Equipment, as containers. Ergo the testbed must have three virtual machines, which also must be connected via a common network.

Figure 3.1 shows a high-level overview of the testbed with its different virtual machines and how they are connected. The testbed itself runs in VirtualBox which is managed and orchestrated through Vagrant. The RAN consists of two virtual machines, one for the eNB base station and one which acts as the User Equipment. The EPC VM runs the EPC components as containers in Docker with the addition of P4 switches that allows for the investigation into INT and how such systems impact the general network performance.

3.2 Approach

The general approach uses Iperf3 as a traffic generator in order to measure and evaluate the general network performance. To measure this, Key Performance Indicators (KPIs) will be collected from various points in the infrastructure. The bandwidth measurements and KPIs helps to investigate and answer the first research question of how INT impacts the general

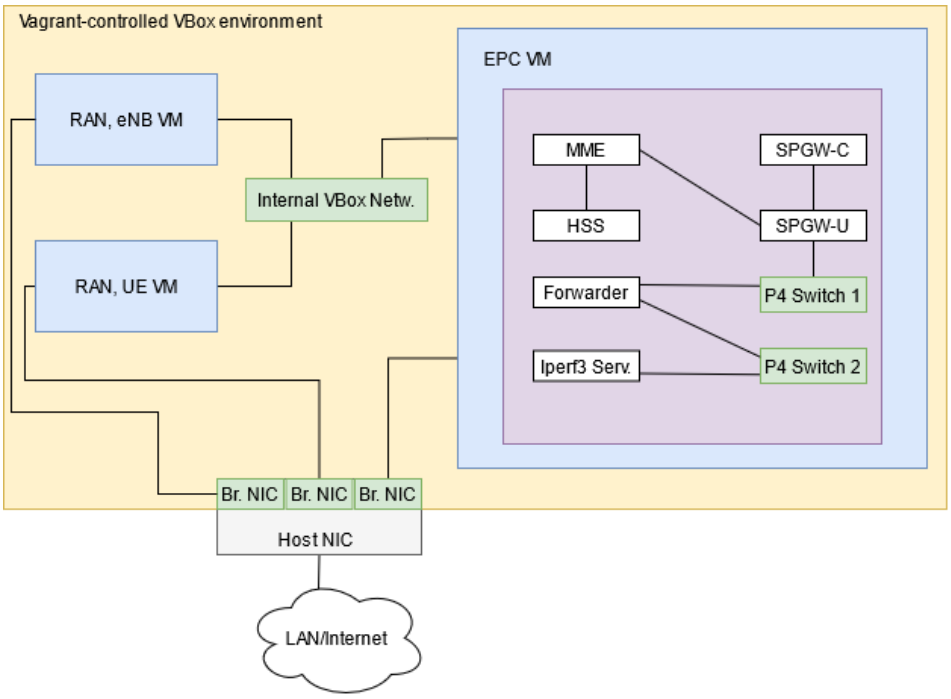


Figure 3.1: A High-level overview of the Virtual Testbed with P4-switches

network performance compared to active and passive monitoring systems. The active monitoring system is emulated with a high frequency ping on a 10 millisecond interval, which is similar to a network probe and is a well-known method for active monitoring systems. The passive monitoring system is implemented with Tshark [52], which is Wireshark's [53] command-line tool to sniff packets, and can write these to a file and further analyze the packet dump if needed.

In order to investigate the second research question, there are two INT systems implemented in this infrastructure. The first attempts to detect delay using interarrival times of sequential packets, and establishes a baseline reading which is analysed with basic linear regression to visualize the trend of the data. The second system counts packets on a given time interval in order to measure packet loss, and the results are analysed with simple percentage calculation. Both of these systems are explained more in-depth in Chapter 4.

3.2.1 Experiments

There are in total 9 experiments performed here which are divided into two groups with an associated research question. The first group consists of four experiments designed to establish a baseline network performance and then implement active, passive, and INT monitoring systems to see how each of these behave compared to the baseline. These experiments provide the data to answer the first research question. The second group of five experiments are all related to the second question. These implement

two different INT systems, each with a specific functionality. For each experiment either delay or packet loss is introduced at a specific point in the infrastructure. These provide a substantial amount of time-series data such as the number of packets seen the last ten milliseconds, or interarrival times between packets, which are further analysed to answer the second research question.

3.2.2 Data Collection

There are four KPIs that are collected from various points in the infrastructure. The bandwidth is measured from the User Equipment VM, to emulate how a user experiences the network. CPU and Memory usage is measured and collected from the P4 Switches, to see how the different monitoring systems affect these, and Disk Input/Output (IO) is collected from the EPC host machine.

Each and every experiment uses Iperf3, which is run 30 times for 30 seconds, with a 30 second pause between each run. As Iperf3 runs, the other KPIs (CPU, Memory, Disk IO) are collected from the P4 switches and EPC host, which will provide more contextual data to see how each monitoring system specifically impacts the testbed.

In order to collect the relevant data from the INT systems, each of the switches' logs are scraped for relevant output and stored locally for further analysis. The reason it is done this way is that a more comprehensive monitoring system with automatic log scraping would potentially incur a higher resource cost. The collection scripts are written to be lightweight such that the actual performance impact is incurred through the INT systems themselves, and not through the data collection scripts.

3.2.3 Data Analysis

The result from the first four experiments, i.e. the different monitoring approaches as well as the baseline, will be compared in order to see the effect each approach has on the KPIs. These are compared using box plots, where extreme outliers are pruned from the result to reduce unnecessary bias.

The result from experiments with Network Delay Detection are analyzed using simple linear regression to establish a trend of the data, and to see if an added delay in the network has any significant impact to this trend. Results from Packet Loss Detection will be presented with simple tables, and to see if the detected packet loss correlates with the induced packet loss in the network. Packet Loss detection uses simple percentage calculation to show the total number of lost packets versus the total number of seen packets.

The KPIs collected during the first group of experiments are also collected from the second group. The reason for this is that it is of interest to see if the specific functionality of the INT system, i.e. delay detection and packet loss detection, incurs similar cost to performance, or if one or the other has a significantly higher or lower cost.

The next chapter gives a more detailed explanation of the testbed, how it is configured and deployed, as well as some challenges faced during the development. Each INT system is also explained in depth.

Chapter 4

Implementation

In order to move forward with the investigation of INT, it is necessary to create a virtual testbed which allows for the integration of P4-capable switches into the EPC environment. There are several options for this purpose. OpenAirInterface provides the necessary software in order to run a 4G LTE EPC implementation. The design and implementation of such a testbed requires extensive engineering, and this chapter presents some of the challenges faced during the implementation of the testbed.

4.1 Challenges

The main challenge was that there was no documentation that describe how to integrate a P4-capable switch into the OAI EPC environment. There was the possibility of deploying each EPC component on a standalone hardware platform and connect these using physical P4-capable switches. The other option was to deploy these components as Virtual Network Functions (VNF) which has been done by SimulaMet [54]. The last option was to deploy the EPC as Docker Containers, which is a well documented approach.

4.1.1 Frequent redeployments

Another challenge that must be addressed is the number of redeployments and iterations to get the testbed up and fully functional. The main reason for these redeployments are the issues faced when integrating new components into the EPC and RAN environments. If a step in the build and compile processes fails, the most efficient solution is to record the error, delete the virtual machine, and begin again with an updated build script. When recording an error, it is also preferential to test a possible fix before deleting the virtual machine, but the machine must be deleted and redeployed to verify that a fix is successful.

Building the EPC and RAN components can take up to 30 minutes each, and building the P4 compiler and related packages can take up to 120 minutes from start to finish. In order to minimize the time spent on server configuration, we decided to manage and orchestrate the testbed

environment with a Vagrant provisioning file. Vagrant is developed by HashiCorp and allows the user to define one or several virtual machines in a VirtualBox environment much like a Dockerfile [55]. This allowed for automation of the deployment and configuration of the relevant servers and packages needed for the testbed and saved a substantial amount of time. We have made the Vagrantfile publicly available on GitHub [56].

With the Vagrant provisioning file in place, it is also possible to more easily update build scripts as these are copied to the target virtual machine by Vagrant. This is a substantial time saver as each component in the testbed have a multitude of dependencies. For instance, the P4 Compiler depends on 10 required packages, most of which have their dependencies. There are a few publicly available scripts that will set up P4 with BMV2 switches and all necessary dependencies on GitHub with associated guides and walkthroughs [57].

4.1.2 OAI 4G LTE EPC

Setting up the EPC is a complicated process, but very well documented. Each component has an associated Dockerfile for several platforms, e.g. Ubuntu 18.04, CentOS 7 and 8. Each component's Docker image must be built separately, and once the Image is built, the user must generate configuration files for the EPC components using the provided Python scripts. These scripts must be supplied with the configuration parameters for the EPC, such as authentication keys and IP addresses to each relevant EPC component. In addition to this, all of the EPC components must have a dedicated common Docker network, over which they will communicate with each other. In order to reach the EPC from outside the Docker Host machine, appropriate routing and IP forwarding must be set up and enabled on the host machine. A functional deployment of the EPC can be seen in figure 4.1, which shows the containers that are connected to the docker network, which is reachable through the other NICs on the host machine.

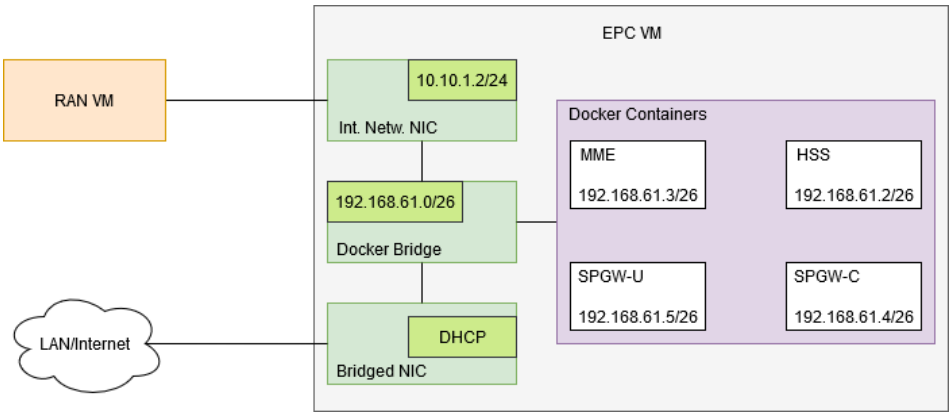


Figure 4.1: Basic Deployment of the EPC as docker containers

4.1.3 OAI RAN

The RAN is an equally complicated process to set up as it includes several radio simulators and support for many different physical radio implementations [44]. As COVID-19 related lockdowns prevented a stable access to the physical lab that SimulaMet has at its disposal, the focus was shifted towards a fully virtualized deployment. This ruled out the use of any specialized physical equipment such as Software Defined Radio hardware. In order to get a RAN operational in a virtual environment, it is required to investigate and test the available radio simulators. These simulators all aim to simplify the testing of the RAN and EPC environments. There are several simulators available, but there are two which are of interest in this thesis, the L2 NFAPI and Basic Simulator. The approaches behind them are different and their use-cases are slightly different, but the end-goal is the same; Enable a simulated UE to communicate with a simulated eNB and get a stable connection to an operational EPC. Each simulator is further discussed in the following sections. It is important to point out that there is no comprehensive documentation on the simulators readily available online.

L2 NFAPI simulator

The use-case for this simulator is to test layer 2 and above in the network stack. It uses the local loopback interface for communication between the UE and eNB, and both can be deployed on the same virtual machine. It is also meant to be used for testing more than a single UE, ideally up to 255 simulated UEs. The main problem here lies in the transport protocol used by the eNB and UE for communication, namely the Stream Control Transmission Protocol (SCTP). SCTP is a blend between UDP and TCP, it has the message-oriented feature of UDP, but provides the reliability of TCP. SCTP also implements multihoming and redundant paths. Multihoming means that the protocol will send out its packets on many different interfaces in order to reach its intended destination. This fact combined with the testbed environment, resulted in a failed deployment caused by the NAT-ed interface on each virtual host. VirtualBox uses NAT in order to let the virtual machine in question reach the internet and allow the host to connect to the VM using SSH. SCTP will flood its packets out on this NAT network, but there are no valid destination addresses for the SCTP packets on the NAT network. This will cause SCTP to fail and shut down its connection entirely. The NAT network in Virtualbox is not highly customizable, and the rules cannot be changed easily, thus configuring this network to properly handle SCTP connections could not be done in a reasonable time span.

Basic Simulator

The basic simulator can be used in situations where it's only necessary to test one UE instance. It replaces the physical radio head with a

network interface on the host machine. This removes unpredictable air perturbations, but also limits the number of UEs in the network to one, it also removes any channel. These limitations do not stand in the way of the end goal of this project. This approach is also functional in a virtualized environment and allowed us to get a basic RAN operational. The finalized version of the RAN can be seen in figure 4.2.

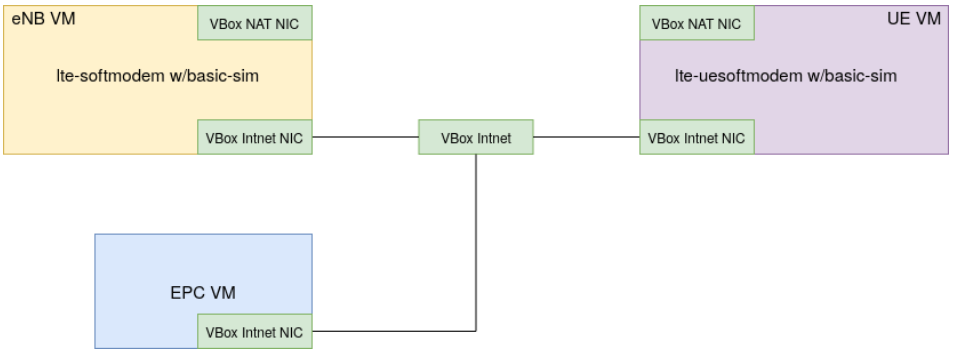


Figure 4.2: Finalized deployment of the RAN with Basic Simulator

4.1.4 Connecting the RAN to the EPC

In order to get an eNB attached to the MME in the EPC, it must have the appropriate routing from the eNB machine to the EPC. The EPC is running on its own dedicated IP subnet which is initially unreachable from any host except the EPC Host machine. It is made available by enabling IP forwarding and configuring IP routing to the EPC subnet on the EPC host machine. Any machine in the testbed environment that needs to communicate with the EPC must also be configured with the routes to this subnet via the EPC host machine. The UE VM must have a network in common with the eNB VM. Using the VirtualBox internal network, this was all possible using the same network segment. It is not, however, necessary for the UE machine to have any knowledge of the EPC network segment. Once all the necessary routing and forwarding was in place, and the correct authentication information was in place, the UE could attach to the EPC and receive an IP address from the SPGW-U.

4.1.5 Integrating P4-capable switches

The integration of a P4-capable switch is a complex task, as the available implementations all have several dependencies and requirements for the platform on which they run. The most commonly used P4-capable switch is the BMV2 [8]. It has poor performance and a limited set of features, but implements all of the well-known P4 features necessary to create a basic INT monitoring system. The initial idea was to replace the Docker networking with a BMV2 switch. The problem with this approach is that the Docker Engine, which controls the networking between containers and host machine, has no support for BMV2 switches. In order to add this

support one would have to write a new plug-in, which would further add on the complexity of the virtual testbed, and it is also outside the scope of this thesis.

P4-OvS was another option that we spent much time on. It implements all the well-known features from OvS with the addition of supporting basic P4 programs. The problem was to build and run the switch. OvS already has support for adding Docker Containers as hosts to an OvS Bridge, and as P4-OvS is a fork of OvS it also has this functionality. During the implementation phase it also became apparent that in order for P4-OvS to integrate with the Docker containers, it would require a dedicated controller such as Faucet to operate as intended. As P4-OvS is still under heavy development it had a few bugs that had to be fixed, such as missing symbols in the makefiles. A pull request has been created to fix these bugs, but it has not been approved.

From this point we had to consider rebuilding the testbed from scratch or find another way to integrate P4 switches into the testbed. Given this we looked into network emulation which could use docker containers as hosts, which led us to ContainerNet. ContainerNet allows the user to deploy docker containers in a Mininet topology, but requires the containers to have certain packages installed. ContainerNet also does not have support for BMV2 switches, but the developers created another version called FOP4 which does exactly this.

4.1.6 Network Emulation using FOP4

FOP4 as explained earlier is a fork of ContainerNet with support for BMV2 switches [9]. With this approach we could implement an extended network segment connected to the SPGW-U. This allowed us to perform basic INT on any user-generated traffic between an emulated Internet server and the UE. In order to let the EPC components be controlled by FOP4, there are a few required packages, such as iputils-ping, net-tools, and iproute2. These packages allow for basic Mininet functionality such as pingall, and also allows the FOP4 system to set up the necessary routing on the containers. We had to fork each of the EPC component repositories and expand the relevant Dockerfiles with the additional packages and build the new images, the repositories and branches used can be seen in table 4.1. The advantage of using a Mininet-based emulator is that we can define the topology however we want, which resulted in an efficient deployment of the EPC. We can also create several topology definitions for different cases, such as having no monitoring at all or including packet capturing.

Component	Version/Tag	Branch
MME	2020.w47	[58]
HSS	v1.1.1	[59]
SPGW-C	v1.1.0	[60]
SPGW-U	v1.1.0	[61]

Table 4.1: EPC component branches

4.2 The final testbed

The final version of the testbed consists of three VMs deployed in a VirtualBox environment. The UE and RAN VMs have 4GB RAM and 2 cores of a 6-core 3.7GHz Ryzen 5 2600X CPU each as well as 20GB of storage. The EPC VM has the same amount of RAM and CPU, but 50GB of storage to account for the amount of logs and packet capture files that will accumulate on this VM. The EPC components are deployed in a FOP4 controlled network, and has an extended network segment consisting of two BMV2 switches and two simple docker containers. These two containers are the Forwarder and Iperf3 server. The Forwarder will act as a simple router between the SPGW-U and the Iperf3 server. The full testbed is shown in figure 4.3.

An interesting point is that the eNB machine has no knowledge of any network except the path to the EPC and its own directly connected networks. The UE has no explicit path to the EPC network, and only knows of its directly connected networks. This includes the address pool from which it has its IP address assigned through the SPGW-U. With default routing through the oaitun_ue1 interface (the basic simulator interface), it can reach the extended network segment.

4.3 Deploying the different versions

In order to move on to experimentation, we have defined three slightly different topologies. The basic topology deploys the EPC and the extended network segment with minimal logging and monitoring, shown in the appendix A.3.1. We also defined a topology that uses packet capturing as a rudimentary bottom-up monitoring system, shown in the appendix A.3.2. This version will enable packet capturing on the switches, and every component in the EPC. We did attempt to enable packet capturing on the RAN components, but the amount of packets flowing between these VMs exceed what we can store on each machine (e.g. more than 7GB) in just a few minutes of monitoring. The third version starts up the BMV2 switches with a specialized P4 program that we'll explain the next section. The topology without any monitoring and the one with packet capturing still relies on a simple P4 program which forwards traffic out to the appropriate port on a given switch, which can be seen in the appendix A.4.1. The final version performs basic in-band network telemetry using P4 and is explained in more depth in the next section.

4.4 INT with P4

P4 and INT is introduced in section 2.3.2. Essentially what this allows one to do is to mark and inspect packets at nearly full line-rate. In order to investigate the potential performance impact of INT, we have implemented two different programs with different goals. The first program uses the interarrival time between the two last packets in any given flow to detect

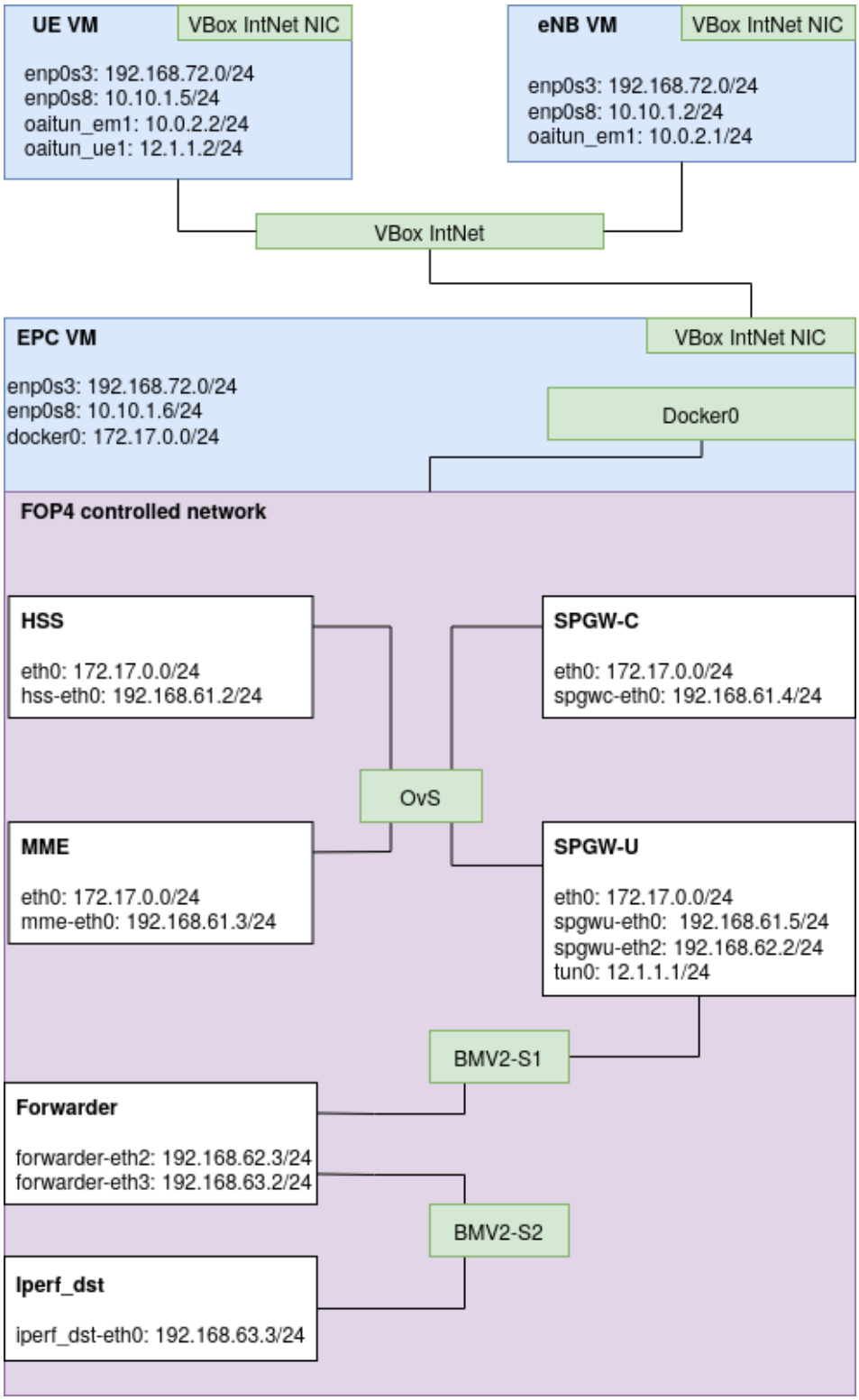


Figure 4.3: The final version of the Virtualized Testbed

delay in the network path. The second program uses packet counters in order to detect packet loss. Each program is explained in the following

sections. Both programs use the switch logs to dump the most important information and these are analyzed following an experiment. We are not writing these to be complete monitoring systems in their own right, but rather to show that INT can be used to detect these kinds of problems.

4.4.1 Delay Detection

There are several ways to measure delay in a given network path. The simplest approach is to probe it with ICMP packets (i.e. ping) and measure the round-trip time (RTT). This however falls into the active monitoring category. In order to do this with INT and P4, we can use timestamps of packets. The general idea is to calculate the interarrival time between the last two packets in a flow at any given switch in the network, and mark the packet with the result such that it can be inspected down the line in the network. The interarrival time of the last two packets will then be written to the Type of Service field in the IP Header of the current packet and sent on its way in the network. Any switch running the same program in the network path can look at this field to see the interarrival time recorded by the previous switch for the given flow. From this we can infer if there is any delay on the specific network path.

This can be shown by the following: Switch A measures the interarrival time given by $r_k = T_k - T_{k-1}$ where T_k is the kth packet, i.e. the current packet in the flow and T_{k-1} is the previous packet in the same flow. T of course is the packet's arrival time at the switch. Assume that T_k is 10, and T_{k-1} is 5, which gives the interarrival time of 5. Switch B however records an interarrival time of 20. Switch B logs its own measured interarrival time as 20 and the incoming interarrival time of 5 from Switch A. We would expect that the interarrival time is closer to equal given perfect network conditions with zero delay. If there is a substantial difference, we can infer that there is a delay on the path between switch A and B causing the interarrival time to increase.

Our approach uses registers on the BMV2 switches, which are simple key-value stores that are available to the dataplane at runtime. The key must be unique and simple to calculate at runtime. We use a hash of the 5-tuple of any flow that passes through the switch. In order to reduce the network overhead, we do not introduce new headers to the traffic as it might cause issues on the forwarder node. The available header is the Type of Service field in the IP Header, which is an 8-bit wide field. This limits the information that we can store in this field to $2^8 = 256$. The problem is that the interarrival time can be on the order of milliseconds and the timestamp is given in microseconds. One millisecond is 10^{-3} seconds and one microsecond is 10^{-6} seconds, which means that the interarrival times will almost always be greater than 256 even in near perfect network conditions. Another limitation is that the BMV2 switch architecture cannot perform floating point arithmetic, which means we can only work with integers, and the result of any division will yield an integer without decimal points.

To circumvent these limitations we can represent milliseconds and

microseconds as 4-bit values and combine them to an 8-bit representation of the interarrival time. This produces intervals which are easily convertable from the represented value to the approximated value by any subsequent switch in the network path. This solution will result in some loss of information, but it should suffice to detect delay in the network path. Microseconds are divided into intervals of 64, which are represented by values 0 to 15, where $0 \in [0, 63], 1 \in [64, 127]$, etc, resulting in a lower and an upper bound to the actual recorded microsecond value. This is done because it is expected that it will be more than a thousand microseconds between most packets in any flow. Milliseconds on the other hand will have a much more simplistic representation. Any value < 15 is written as is, whereas any value > 15 is simply represented as 15. Any value > 15 can be interpreted as a high interarrival time. When a subsequent switch in the path detects a marked TOS field, it will revert the representation to the approximate value recorded on the previous switch and write this to its logs. This is done by taking the last 4 bits in the field and multiplying by 64, and convert the first 4 bits of the field to an integer and multiplying by a thousand, and add these two values together. Even though we lose some information by compressing the data in this way, it will be sufficient to check interarrival times across two switches. It is possible to use the IP Options field, the problem with this is that it is rarely used and can potentially be marked as dangerous by some router implementations.

Both switches in the final testbed will be set up with the delay detection program, shown in its entirety in the appendix A.4.2. Because the timestamps are stored using the hashed 5-tuple, each switch will be monitoring a specific direction of the traffic. This is caused by the fact that we are using the TOS field of the IP header, and it must be marked for a switch to record its own and the incoming interarrival time.

4.4.2 Packet Loss Detection

As with delay detection, there are many different solutions to the problem of detecting packet loss. The simplest way is to use ICMP packets and send a set amount to a host. If we send a thousand packets, and only 950 receive a reply, we can assume there is a 5% loss in the network path. This falls into the active monitoring category, and we are more interested in how we can do this with INT. The obvious solution is to count packets at switch A and B, and compare these. The problem with this solution is that network traffic is rarely symmetrical, and switch A will not necessarily see the same amount of packets as switch B. One reason could be that there are more hosts connected to switch B, compared to switch A, and that there is more host-to-host communication between these. The solution is as before to count packets on a per flow basis. We can reasonably expect that the number of packets seen in flow 1, which flows from switch A to switch B, is the same for both switches.

Our approach uses counters which are stored in registers. We will use the TOS field in the IP header to send the number of packets seen in the flow. There are two counters on each switch. One that counts packets in an

epoch of 10ms and at the end of the epoch it will mark the current packet's TOS field with the count and resets the stored count to 0. The other counter, which is at the second switch, is incremented for each packet seen between marked packets. This counter is incremented until the switch detects a marked packet, and then stores the incoming packet count, its own packet count, and the difference between the two, as well as resets the counter to 0.

To give an example of how this works: Switch A records 10 packets in the last 10ms, and marks the current packet with this value in the TOS field. Switch B has, since the last time it saw a marked packet in the current flow, seen 7 packets. Switch B will then see that the incoming packet from Switch A is marked with 10, and write the following to the logs: ”pcount=7, inc_pcount=10, pcount_diff=3”, i.e. there must be 3 packets that did not make it from Switch A to B. From this we can infer that there is an approximately 30% packet loss in the network path. With a sample size this small, the detected packet loss is massive. However, for a thousand or ten thousand packets, it will in theory come closer to the actual loss rate.

4.4.3 Considerations

To be perfectly clear, we are not investing a significant amount of time into perfecting the algorithms for both loss and delay detection. We are merely interested in finding out if it's possible or not. The main focus of the thesis is to investigate the performance impact of INT compared to other approaches to monitoring. Both of the programs detailed in the previous sections will hopefully incur a measurable cost in terms of CPU, memory, disk IO, and bandwidth.

4.5 Summary

In this chapter we outlined the challenges we had to overcome in order to implement the virtualized testbed and the variations of the EPC and extended network segment that we will use for experimentation. The variations make it possible for us to test several different approximations of monitoring approaches, such as INT, packet capturing, and active monitoring. We also presented the two different INT implementations we deploy in the testbed. The next chapter explains the different experiments and presents the results.

Chapter 5

Results

This chapter presents the results from the experiments and establishes some context around the individual experiments and how each monitoring system performed in comparison with an established baseline.

5.1 Placement of the Monitoring Systems

To visualize how the different monitoring systems are placed throughout the network, a simplified network topology can be seen in figure 5.1. This figure shows the RAN and UE as a single entity, which is connected to the EPC host. On the EPC host, each individual component (MME, HSS, SPGW-U and SPGW-C) are shown as connected via a common OVS switch named S1. The SPGW-U is further connected to a BMV2 switch, S2. S2 connects to the Forwarder node, which in turn is connected to another BMV2 switch, S3. S3 is the final intermediary network node which connects the rest of the network to the Iperf3 Server. This figure will be used throughout this chapter with additional graphics to show how each monitoring system is placed in the testbed network.

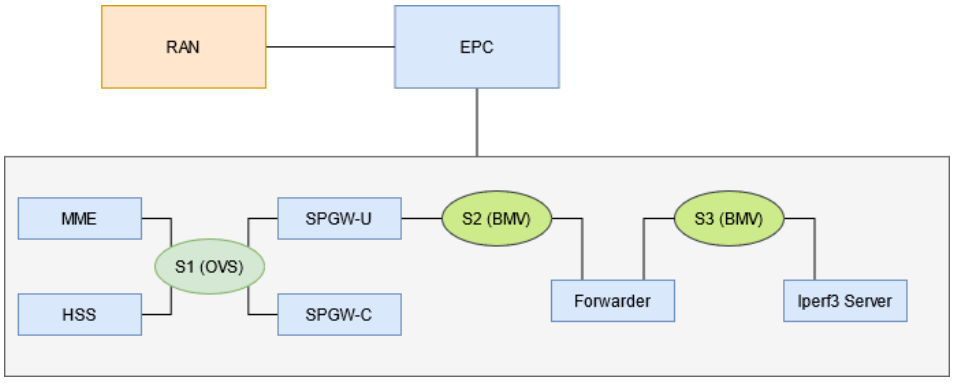


Figure 5.1: A simplified network topology of the virtual testbed.

In figure 5.2 shows where the active monitoring system is placed within a simplified network topology. It is important to note here that while the figure shows the active monitoring system on both the Iperf3 server and

the SPGW-U, it is in fact run directly from the SPGW-U node towards the Iperf3 server. The advantage this system has over both passive monitoring and INT, is that it is only placed on one node. The consequence of this fact is that the amount of configuration needed is vastly less compared to both INT, which must be tailored to fit a given use-case, and passive monitoring, which in many cases requires many different nodes to be set up with the monitoring system.

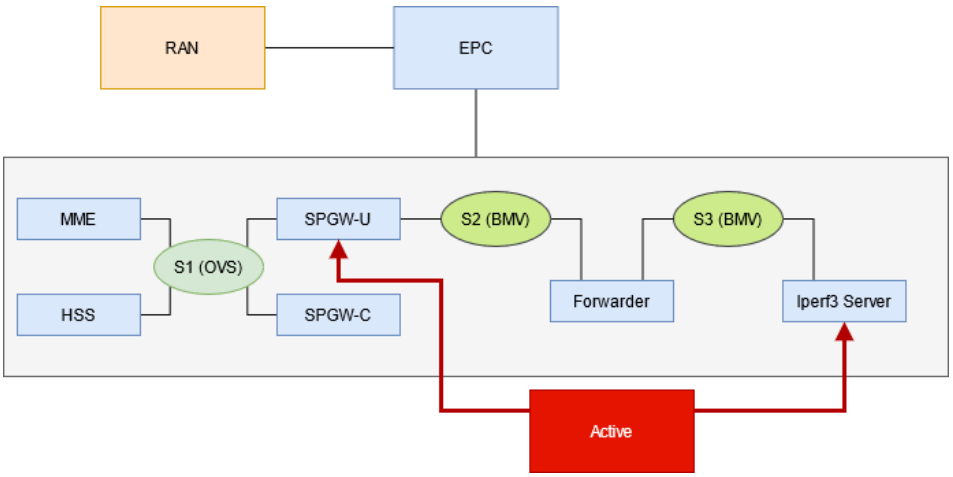


Figure 5.2: Placement of the Active Monitoring System within the testbed network.

Figure 5.3 shows how the passive monitoring system has been placed on every node in the network. This also highlights the inherent weakness of bottom-up, passive monitoring systems. In order to have a network-wide monitoring system with this approach, the system itself must be present on all nodes. It is also important to note that while the MME, HSS and SPGW-C are monitored, they do not generate a substantial amount of data while Iperf3 runs. This is caused by the fact that the traffic between these components belongs to the control plane of the network, and not the data plane.

Finally the INT system is shown in figure 5.4. Here it is shown how the INT system is placed on both of the BMV2 switches. The footprint is lower compared to the passive monitoring system, but it is important to highlight the fact that the development time of the INT system is fairly longer compared to both the passive and active monitoring systems. In order for INT to function properly, it must be thoroughly tested during its development cycle, whereas both the passive and active systems used here work out of the box. The passive system does require the operator to configure which interfaces it will monitor and where to save the captured packets, but beyond that there is little configuration.

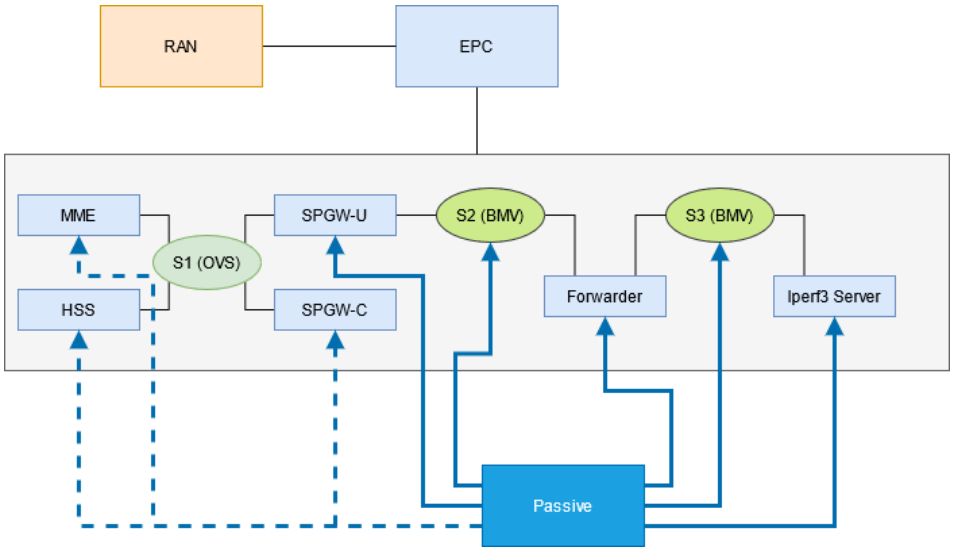


Figure 5.3: Placement of the Passive Monitoring System within the testbed network.

5.2 Comparing INT to conventional methods

Presented here is a comparison of how INT performed, plotted in a boxplot together with the baseline and with either active or passive monitoring. The INT program used for these measurements was the Delay Detection program, which has been presented in section 4.4.1. The baseline bandwidth is measured to be 15.5 Megabits per second (Mbps), the mean CPU usage is measured at 60%, the mean memory usage is at approximately 1% of 4096MB, or about 400 MB steady usage. The mean Disk IO is approximately 100 KBytes written to disk every 5 seconds. These measurements are used for all comparisons between the Baseline, INT, and Active and Passive monitoring. All of these baseline measurements are plotted from 30 runs of Iperf3.

5.2.1 INT v. Active monitoring

Figure 5.5 shows a comparison between the baseline measurement, INT, and an active monitoring approach. The active monitoring approach was implemented with High Frequency Ping (HFP) with an interval of 10msec. Figure 5.5a shows the measured bandwidth in megabits per second between the UE and Iperf3 server. The baseline measurement with no monitoring has the highest bandwidth at 15.5 Mbps, with INT slightly behind at a mean 14 Mbps, which is a 10% reduction. HFP performed the worst of these three with a mean measured bandwidth of 11.5, which is a 25.8% reduction in bandwidth. This is likely caused by excessive queuing at the SPGW-U, which lowers the overall throughput for the Iperf3 traffic.

Figure 5.5b shows the CPU usage of the BMV2 processes as a percent of two cores at 3.8GHz. INT had the highest usage at a mean 80% usage, which is not surprising as it processes each packet. HFP had the lowest

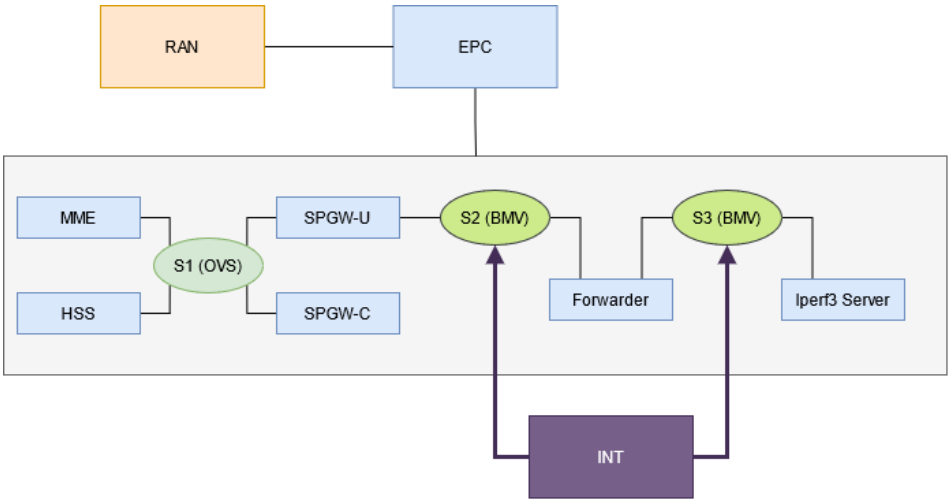


Figure 5.4: Placement of the INT system within the testbed network,

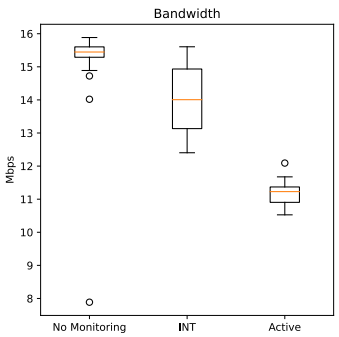
CPU usage at approximately 45%. This is 25% lower than the baseline CPU usage and it is unclear why it is that much lower. One possible explanation is that Ping uses ICMP packets. These packets exist in Layer 3. In order for the BMV2 switches to process the packets, they must have a layer 4 header such as a TCP Header. With the reduced bandwidth, there are fewer TCP packets moving through the switches, giving them less to do per flow.

Figure 5.5c shows the memory usage of the BMV2 processes as a percent of a total of 4GB. The general memory usage is fairly low, below 1.3% of 4096MB of memory, but INT has the highest usage of these three approaches. This is unsurprising as the BMV2 switches has to process each packet and read/write to its hash table of flow timestamps. This increase in usage is however insignificant when brought into a larger scale. The difference on an absolute scale of 100% is approximately 0.5% percentage points, even though the difference is 62.5% between the lowest and highest measured memory usage.

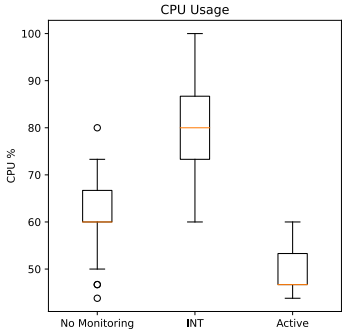
Figure 5.5d shows the kilobytes written to disk on an interval of 5 seconds. Each data point represents this average. INT on average had slightly lower disk usage compared to HFP, but also shows a higher amount of outliers. It is important to note that the scale is fairly small in terms of Kilobytes, ranging from 0 to 5000. It goes to show that the difference in Disk IO between INT and Active is fairly small.

5.2.2 INT v. Passive Monitoring

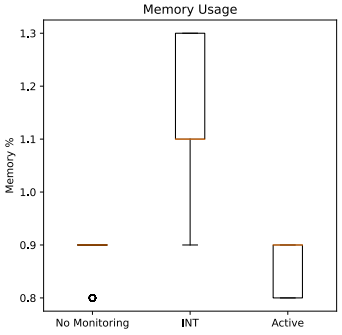
Figure 5.6 shows a comparison between no monitoring, INT, and a passive monitoring approach. Packet Capturing was selected as the preferred approach to passive monitoring. The results for INT and no monitoring are the same as the previous section, and are compared with packet capturing. Figure 5.6a shows the measured bandwidth of the three approaches. It shows that a passive monitoring approach has a significant impact on the measured bandwidth. It measured in at 12 Mbps, which is a 22.5%



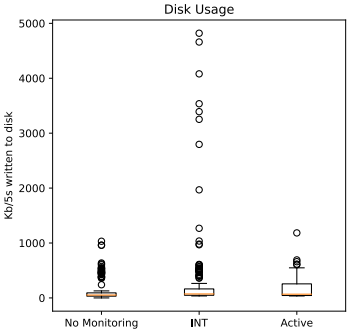
(a) Measured Bandwidth



(b) CPU Usage, percent of 2 cores @3.8GHz



(c) Memory Usage, percent of 4GB



(d) Disk Usage, KB/5s

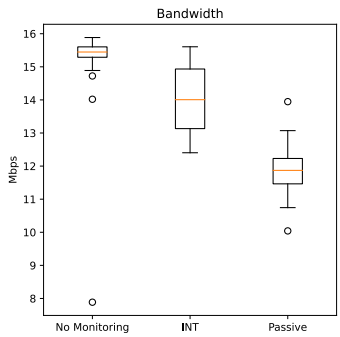
Figure 5.5: INT Versus Baseline and Active Monitoring, based on 30 seconds of iperf3 with 30 second pause, repeated 30 times.

reduction compared to the baseline of 15.5 Mbps. This is caused by the fact that each and every packet it recorded and written to file, which increases the time to process a full stream by a substantial amount.

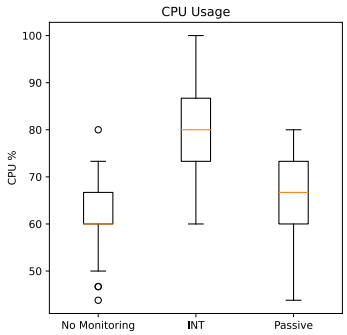
Figure 5.6b shows the measured CPU which is measured at an average of 65%. This is an 8.3% increase from the baseline measurement. Packet Capturing does have a slightly lower impact on the CPU usage compared with INT, which is caused by the fact that the BMV2 switches are not processing each individual packet to the same extent with the simple forwarder program.

Figure 5.6c shows the measured memory usage. As with the previous results, INT uses slightly more than a passive approach, but the overall memory usage is still quite low.

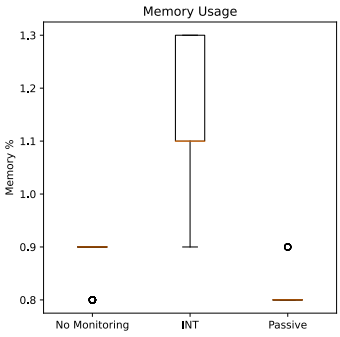
Figure 5.6d shows the kilobytes written to disk on an interval of 5 seconds. What is clear from the results is that a passive monitoring requires significantly more disk usage compared to INT. When compared, the INT disk usage is insignificant. This is again caused by the fact that each packet is written to a file, and there is one file per monitored node.



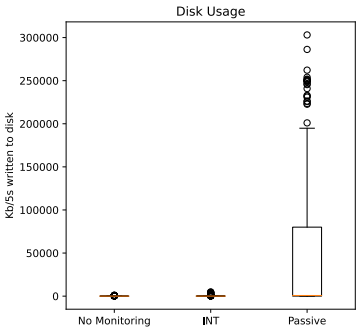
(a) Measured Bandwidth



(b) CPU Usage, percent of 2 cores @3.8GHz



(c) Memory Usage, percent of 4GB



(d) Disk Usage, KB/5s

Figure 5.6: INT Versus Baseline and Passive Monitoring, based on 30 seconds of iperf3 with 30 second pause, repeated 30 times.

5.2.3 Summary

From the results presented in the previous sections, it is clear that INT has a measurable impact on CPU usage compared to both active and passive monitoring approaches. Disk Usage was the highest with a passive monitoring approach. Even though the impact on the CPU usage is significant, INT performed the best in terms of measured bandwidth. Both the passive and active monitoring approaches had a larger impact on the bandwidth. In the next section the results from the investigation of network delay and packet loss detection will be presented.

5.3 Network Delay Detection

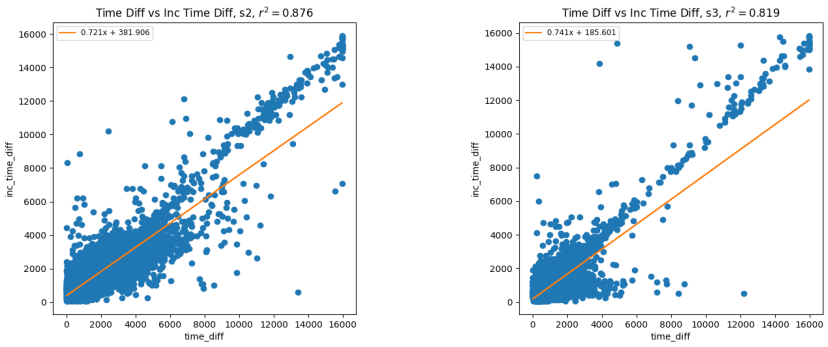
There are three cases that will be presented here: 1) No delay in the network, 2) a delay between the BMV2 switches, and 3) a delay between the UE and EPC VMs. The delay was introduced with NetEm [62], with a mean delay of 500 milliseconds with a variation of 300 milliseconds on a normal distribution. The delay was introduced on the Fowarder's

ethernet interface towards S2 in the egress. Simply put, any packet that goes towards S2 from the Forwarder will experience a delay between 200 and 800 ms, with a mean value of 500 ms.

All of the results that follow are data points plotted in a scatterplot, where the X-axis represents the measured time difference, and the Y-axis represents the incoming time difference from the other switch. For a more detailed explanation of the timestamping program, see section 4.4.1 in chapter 4. The figures are supplemented with a simple line regression and the R-squared value, which represents the overall accuracy of the model.

5.3.1 No delay

Figure 5.7 shows how the data is expected to behave. In a case where there is no added delay between two P4 switches with a delay detection system, the interarrival time recorded on both switches should be close to equal, or $Y \approx X$. The A Figure shows Switch 2, while the B figure shows Switch 3. Any point on these plots, is plotted using any given flow's incoming interarrival time in the Y axis, meaning time recorded at the other switch. While the recorded interarrival time on the current switch for any given flow is plotted in the X axis. From this we can expect any delay between S2 and S3, in S2's direction, would cause S2's recorded interarrival time to increase, and thus skew the linearly regressed line in a downwards fashion.



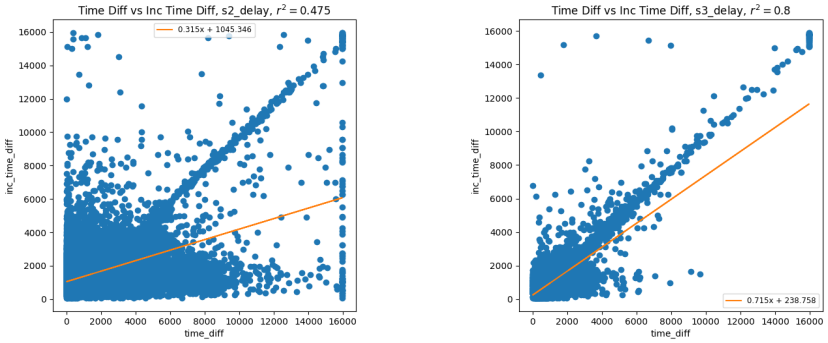
(a) S2 results (b) S3 results

Figure 5.7: No Delay between S2 and S3

5.3.2 Delay between S2 and S3

In this first case, where we have introduced a delay of 500 ms with a variation of 300 ms, we expect the interarrival time measured on S2 to be higher for any flow moving from the Iperf3 server towards the UE VM. This is because we know that the delay is only applied to egress traffic on the Forwarder node, and traffic from the UE to the Iperf3 server will not be affected. However any replies from the Iperf3 server to the UE VM will be, as these flows must go through the Forwarder towards the

S2 switch. Figure 5.8 shows how a delay between the switches affects the measured and incoming interarrival times between S2 and S3. On S2 there is a clear impact on the interarrival time. The interarrival time on S2 is higher compared to the incoming interarrival time from S3, shown in figure 5.8a, which causes a much more gentle slope and a lower R-squared value. Figure 5.8b shows little to no impact of the delay, which is explained by the fact that NetEm only applies delay to outgoing packets, and was configured only on the interface towards S2.



(a) S2 results

(b) S3 results

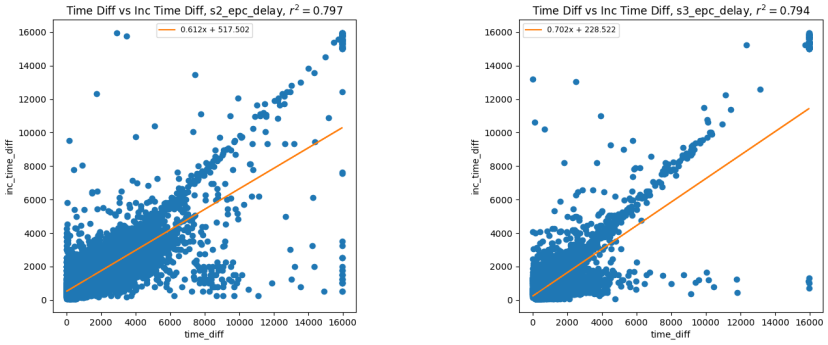
Figure 5.8: Delay between S2 and S3, in S2's direction

5.3.3 Delay between UE and EPC

In this case we have introduced the delay on the egress interface which connects the EPC VM to the eNB and UE VMs. Since there is no delay between the P4 switches, we can expect that the interarrival time to be close to equal on both switches. Figure 5.9 shows the impact of this introduced delay, which is minor. S2 shows a slight skew, meaning it did record slightly higher interarrival times for its recorded flows, but compared to the previous case it is not significant. What this tells us is that INT can be fairly imprecise if there is a network problem such as delay outside of the monitored segment, and yield a false negative when there is in fact a problem in the network. This also points towards the fact that any INT system must be network-wide to yield accurate results.

5.4 Packet Loss Detection

There are two cases that are presented here: 1) No loss in the network, and 2) a 5% loss between S2 and S3. The loss was introduced with NetEm, and as with delay, it will only be applied to outgoing packets on the interface which connects S2 and the Forwarder node. The results are presented with simple tables that show the mean loss percent for each major flow on a switch. A major flow is defined here as a flow with more than a thousand



(a) S2 results

(b) S3 results

Figure 5.9: Delay on EPC VM, outside the INT segment

packets counted. The result tables are random samples of ten rows from the relevant switch and case.

All of the tables follow the same format with six columns. The hash column contains the hash of the 5-tuple of a given flow. The total_pcount column shows the total number of counted packets across all 10 millisecond epochs during the experiments, and the total_loss column shows the total difference between the packet count and incoming packet count across all epochs. The loss_percent column is the percentage of loss measured in the given flow. The switch and case columns are added for readability in post-processing of the results, as a way to see which switch and which case the results represent. There are five columns that have been truncated, which make up the 5-tuple of the flow with source and destination IP address, source and destination port and protocol type.

5.4.1 No Loss

Tables 5.1 and 5.2 show samples of ten rows with no added packet loss in the network path between the two switches. There are some flows which experienced some minor packet loss, which should be considered normal behaviour of a fully virtualized and emulated network.

5.4.2 Loss between S2 and S3

Tables 5.3 and 5.4 show the impact of added loss in the network. All the flows monitored by S2 experienced a loss percentage quite close to the actual 5% introduced through NetEm. It is clearly not 100% accurate, but the mean loss across all flows is 4.09, which gives an accuracy of $(4.09/5) * 100 = 81.8\%$. The flows monitored by S3 did not experience any significant loss, which is explained by the direction of the introduced loss.

hash	total_pcount	total_loss	loss_percent	switch	case
437	19335	0	0.000000	s2	baseline
7067	17757	0	0.000000	s2	baseline
14	23784	14	0.058863	s2	baseline
5914	47110	0	0.000000	s2	baseline
3245	21947	0	0.000000	s2	baseline
7699	17183	0	0.000000	s2	baseline
6945	24525	0	0.000000	s2	baseline
8107	21943	0	0.000000	s2	baseline
400	20188	0	0.000000	s2	baseline
4627	21147	2	0.009458	s2	baseline

Table 5.1: Detected Packet Loss on S2 with no added loss in the network.

hash	total_pcount	total_loss	loss_percent	switch	case
5265	36068	0	0.000000	s3	baseline
6542	39673	0	0.000000	s3	baseline
5782	35574	3	0.008433	s3	baseline
1420	1243	0	0.000000	s3	baseline
8082	37295	0	0.000000	s3	baseline
4573	40281	0	0.000000	s3	baseline
340	33172	9	0.027131	s3	baseline
2689	39549	0	0.000000	s3	baseline
3083	35166	0	0.000000	s3	baseline
4851	34115	3	0.008794	s3	baseline

Table 5.2: Detected Packet Loss on S3 with no added loss in the network.

5.5 Loss v. Delay Detection

The last set of results is a comparison between the two different INT programs. Each graph shows how each experiment impacted the measurements compared with no delay or loss in the network. The tstamp label represents Delay Detection, and the Ploss label represents Packet Loss detection. The impact to the measured bandwidth, shown in figure 5.10a, does not vary much between the two methods and their related experiments. CPU Usage, shown in figure 5.10b, does not vary much. Memory usage, shown in figure 5.10c, is also fairly equal between the two methods. The only difference of some note is the Disk Usage, shown in figure 5.10d, where there are some higher values for delay detection. This is a side-effect how the delay detection program is written and how the data is collected. For every packet in any given flow, the delay detection program will calculate and record the interarrival time, which is written to the switch logs. This is a disk IO operation, which is costly in terms of time. Loss detection on the other hand operates on a 10msec epoch, while still processing each packet, but it only performs the necessary work at the end of each epoch.

hash	total_pcount	total_loss	loss_percent	switch	case
910	20911	858	4.103104	s2	with_loss
3853	18590	708	3.808499	s2	with_loss
7712	20754	879	4.235328	s2	with_loss
5771	18779	780	4.153576	s2	with_loss
2868	21109	869	4.116727	s2	with_loss
4492	19711	861	4.368119	s2	with_loss
4234	21743	926	4.258842	s2	with_loss
11	21363	841	3.936713	s2	with_loss
1177	21080	896	4.250474	s2	with_loss
6823	21670	932	4.300877	s2	with_loss

Table 5.3: Detected Packet Loss on S2 with added loss in the network

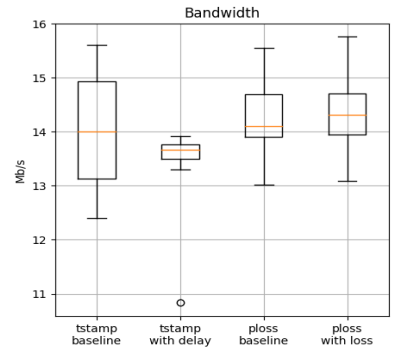
hash	total_pcount	total_loss	loss_percent	switch	case
2301	36288	0	0.00000	s3	with_loss
7402	38071	3	0.00788	s3	with_loss
3007	34571	0	0.00000	s3	with_loss
5048	1348	0	0.00000	s3	with_loss
4627	37806	0	0.00000	s3	with_loss
7069	36662	0	0.00000	s3	with_loss
2526	37486	0	0.00000	s3	with_loss
3452	36808	0	0.00000	s3	with_loss
2720	34728	0	0.00000	s3	with_loss
1797	35848	0	0.00000	s3	with_loss

Table 5.4: Detected Packet Loss on S3 with added loss in the network

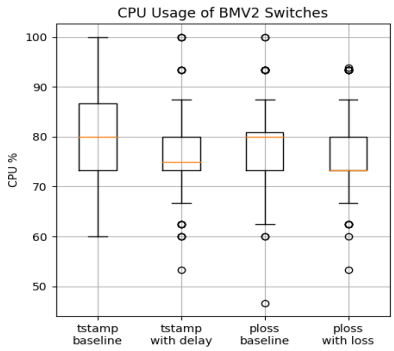
5.6 Summary

Section 5.2 presents how INT performs compared to Active and Passive monitoring. INT does incur a higher CPU cost with a mean 80% usage, but the measured bandwidth between the UE and Iperf3 server remains close to the baseline measurement with an average of 14.5 Mbps. Passive monitoring does however have the highest Disk IO cost of the three monitoring approaches, where it exceeded well over 50,000 KBytes written to disk in 5 seconds, for the duration of the experiments. This can be explained by the fact that each packet must be copied and written to disk, which will increase the processing time per packet by a substantial amount. Active Monitoring incurred the highest cost to the measured bandwidth, clocking in at approximately 11.5 Mbps, which is a 25.8% reduction compared to the baseline of 15.5 Mbps. Passive monitoring measured approximately 12 Mbps, which is a 22.5% reduction.

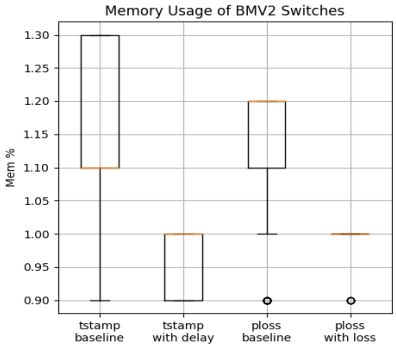
Section 5.3 presents how INT can be used to detect delay between two P4-capable switches running the same P4 program. The delay detection system itself is sensitive to direction, what that means is that if there is



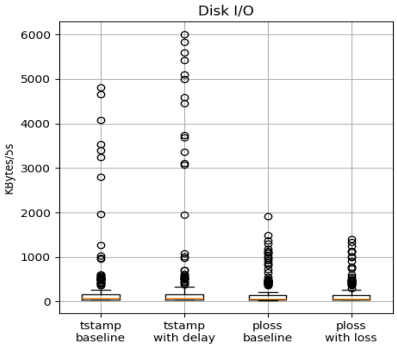
(a) Measured Bandwidth



(b) CPU Usage, percent of 2 cores @3.8GHz



(c) Memory Usage, percent of 4GB



(d) Disk Usage, KB/5s

Figure 5.10: Packet Loss versus Delay Detection, based on 30 seconds of iperf3 with 30 second pause, repeated 30 times.

a delay between S2 and S3, in the direction of S2, as the case is in these results, it will only be detectable on S2. It should also then follow that if the direction of the delay is changed to be towards S3, this switch would then detect the delay. However it did not reliably detect any delay whenever it was introduced outside of the network segment between S2 and S3.

Section 5.4 presents how INT can be used to detect packet loss between two P4-capable switches. As is the case with delay detection, the same P4 program runs on both switches. We introduced a 5% packet loss between S2 and the forwarder, in S2's direction. Similar to delay detection, packet loss detection is sensitive to the direction of the loss. It did not achieve 100% accuracy in detecting the packet loss, but it did detect 81.8% of the lost packets, which shows that INT can indeed with reasonable accuracy detect packet loss in a given network segment.

Chapter 6

Discussion

Here we discuss the findings of the results presented in the previous chapter. First we answer each research question in turn and discuss the implications of our findings. Then we briefly address some of the limitations and criticism of our work.

6.1 Findings

In chapter 1 we propose two research questions; How does INT impact the network performance of a 4G LTE Network compared to Active and Passive monitoring approaches? And to what extent can INT be used to detect two common network problems, and how accurate is it? Here we present our final answers to these questions in turn. In order to provide the reader with some additional context to understand our conclusions, we give here a brief, high-level overview of INT. INT leverages the data plane of programmable switches, which means we can use the in-band network traffic itself to shuttle monitoring data between switches. To give an example, in this thesis we use the TOS Field of the IP Header to send the interarrival time of packets from one switch to the next. This is then used on the subsequent switch to write its own measured interarrival time and the one received from the other switch to its logs. This information can be used to spot any major difference between the two, which can be used to detect network delay.

6.1.1 The Performance Impact of INT

In order to answer this question we must first discuss what performance means. In our case it means the measured bandwidth between two fixed points in the network (i.e. the UE VM and Iperf3 server) and the resource usage of selected network components. The measured bandwidth is considered as the baseline experience of the network. The resources we have measured are the CPU and Memory usage of the BMV2 switch instances and the Disk IO of the EPC VM. With the bandwidth measurements and the KPIs combined, we have a fairly solid overview of how the network performs.

The baseline measurement shows that the average bandwidth of our network is 15.5 megabits per second (Mbps), which is the maximum bandwidth that the OAI RAN can provide with the simulators. The bandwidth is considered as the main metric which we use for further comparisons between INT, active and passive monitoring systems. The CPU and memory usage are collected from the BMV2 switch instances during experiments, and tells us how each monitoring system specifically impacts the switches. These readings do not necessarily tell us how each EPC component is performing during each experiment, but coupled with the bandwidth measurements we can extrapolate how the network as a whole is performing. At the end of the day the bandwidth measurements can be used as an analogy of how a user or network host would experience the network.

From the results we can see that INT performs better than both the active and passive monitoring systems in terms of bandwidth. But this alone does not tell the whole story. The CPU cost of INT is higher than both of the other systems, while the measured bandwidth is also higher. INT has on average 14.5 Mbps of bandwidth, compared to the 11.5 Mbps of Active monitoring and the 11.9 Mbps of Passive monitoring. If we compare these bandwidth measurements to the baseline, we see that INT incurs a 10% reduction, Passive incurs a 22.2% reduction, and Active incurs a 25.8% reduction. The scale for the Memory usage is small to the point where the difference between the highest usage (INT) and the lowest usage (Passive) is approximately 0.5% percentage points. This makes the difference in Memory cost insignificant between the three approaches.

The Active monitoring system was implemented with a high frequency ping on 10 millisecond intervals. What this high frequency achieves is a higher monitoring fidelity. If the sampling rate were to be significantly lower we would run into the same issues that more coarsely granulated systems have, which is the risk of not catching any transient congestion or packet loss in the network. It is important to point out that we are increasing the total volume of traffic by a substantial amount, which causes an increased amount of work for the SPGW-U. The SPGW-U itself has a key role as the main gateway for all user-generated traffic in the network, and the added work of handling the high frequency ping operation, it should then be expected to see a drop in the measured bandwidth. We conclude that the main reason INT performs better compared to the active system is that INT does not generate any additional traffic, but rather utilizes available fields in the IP header of any packet passing through the switch.

The passive monitoring system on the other hand was implemented with Tshark, which copies each packet on any selected interfaces and writes these to a file. This causes a significant amount of Disk IO, which is one of the most time consuming processes in a modern computer. It should also be noted that the passive monitoring system was present on the SPGW-U, the Forwarder, the Iperf3 server and both of the BMV2 switches. With all of these nodes being monitored with an approach that writes a significant amount of information to disk all at the same time, a decrease in bandwidth is to be expected. The main reason that INT performs better

is that it does not write a comparable amount of information as the passive system. There is some Disk IO per BMV2 switch, but the log output generated for each packet in the delay detection system is small compared to a full packet copy with all protocol headers. Since there are only nodes generating Disk IO in the INT case, compared to the five in the passive monitoring case, we can explain the discrepancy in measured bandwidth between these two approaches. It is possible to improve the performance of passive monitoring systems by introducing a dedicated node to handle packet capturing. By implementing port mirroring on a strategically placed network node, it can send a copy of every packet to this dedicated node, which will handle packet capturing and analysis. This is incidentally how most IPS and IDS implementations function.

INT does incur a higher cost than both active and passive monitoring when it comes to CPU usage. This is caused by the added overhead of processing each packet and calculating either interarrival time or potential packet loss. The amount of calculation necessary for one packet is minimal, i.e. timestamp of current packet subtracted by the previous packet's timestamp. But when there are 20,000 packets that must be processed, it will quickly add up to more CPU time. Each switch must also have a flow table to store data about each flow, whether it is the timestamp or number of packets seen in a given time span, which also uses CPU time. The Memory cost that INT incurs is minimal in the bigger picture.

To summarize, INT does perform better than both the active and passive monitoring systems we have implemented here, but the resource cost is slightly higher in terms of CPU. We believe this is caused by the added processing each BMV2 switch must do per packet, but each switch does not write the full content of each packet to disk, nor do they generate any additional traffic. The result is a monitoring system which can have minimal impact on the bandwidth, at a slight cost to CPU. Because there is a cost to CPU, it could potentially impact an enterprise network in a more significant way, as CPU on network devices such as routers and switches is costly and strictly limited. It was briefly discussed briefly in chapter 2 that reckless use of INT will have a significant impact on the performance of any given network. INT must be implemented with care and due diligence in relation to how much information is stored per packet and how the information is stored and analyzed later on. The results here also speak to this fact with a very simple program as delay detection, the CPU usage was significantly higher than both active and passive monitoring, going up to 100% at times.

6.1.2 Detecting common Network Problems with INT

The second research question we have investigated here is to what extent can INT be used to detect two common network problems; Delay and Packet loss. To answer this we have developed two distinct INT programs, each tailored to detect a single problem. The delay detection program was also used as the INT system of choice to help answer the first research question. Here we discuss each program in turn and how well each could

detect the given problem.

The delay detection results show a clear impact on the trend of the data whenever there is an added delay in the network path. It is also sensitive to direction, meaning that each switch detects any delay in the incoming traffic, if that traffic has previously passed through another BMV2 switch. The reason for this is that the system is flow-based and relies on detecting a packet marked with the interarrival time of itself and the previous packet in the given flow. The most challenging part of this system is the analysis and detection of the actual problem. It requires that the data is first collected from the switch log, which is raw text. This must be parsed and formatted into CSV data, which can be read and parsed through the Pandas library. Finally it must be plotted and a linear regression must be performed in order to establish a trend of the data. It was however unreliable and inaccurate at best when the delay was introduced on the EPC VM, and not on an interface between the two BMV2 switches. This goes to show that if the INT system is to maintain reasonable accuracy, it must be network-wide.

The Packet Loss detection program does detect the problem with an accuracy of approximately 80%, or about 4 out of every 5 lost packets. The reason for this inaccuracy is not known for sure, but we can speculate that it is caused by transient bursts of network traffic. The Forwarder node does not have specialised switching software, and will in addition apply the 5% packet loss. This could cause some excessive queuing on the Forwarder node, causing the marked packet to get shuffled and arrive before or after other packets. The result here could be that the receiving switch would count too few or too many packets to maintain an accurate count in a given flow. In order to verify this we would need to perform more elaborate experiments. If we compare this to a simple test where we introduce the same packet loss and use ping, we will detect 100% of the loss. It does show that INT can be used to detect packet loss, but with reduced accuracy compared to simpler and more lightweight systems and tools.

6.1.3 Other Findings and Observations

The original results included the time taken for the Iperf3 client to complete a full run and the number of retransmissions recorded on the client. These were both omitted from the results chapter as they do not add any value, positive or negative, to the overall evaluation of INT. The time taken is of little interest as we did not create a file with a given size and content for Iperf3 to transmit from client to server. In retrospect this would have been valuable, as Iperf3 will attempt to max out the throughput and available bandwidth with the parameters used in the experiments.

Retransmissions on the other hand can be spurious with TCP. This can be caused by a number of different reasons, such as corrupted header fields, packet loss, or excessive delay resulting in timeouts. What we also found was that as the measured bandwidth increased, the number of retransmissions went up. In order to discover the root cause of this, we designed a simple experiment. This experiment consisted of an additional

node in the FOP4 network which was connected to the SPGW-U. The Iperf3 script was run from this node and metrics were collected from the BMV2 switches and EPC node in the same manner as earlier experiments. In this case the measured bandwidth was twice the measured bandwidth from the UE, but the number of retransmissions did not increase accordingly with the higher bandwidth. From this we can conclude that it is the connection between the UE and eNodeB which is at fault, and that an increased amount of retransmissions is expected behaviour as the measured bandwidth reaches its maximum capacity. Thus it was decided that retransmissions are of little interest in the final evaluation.

6.2 Limitations and Criticism

There are a few limitations that are important to point out in the implementation and experiments. The BMV2 software switch is first and foremost meant to be used for prototyping and experimentation with P4, it is not a production-grade switch. Its performance compared to OvS is poor. Where OvS can achieve bandwidths of several Gbps, the BMV2 can achieve up to 30 Mbps. This is however not a major limitation as the OAI RAN does not provide more than 15 Mbps. In a more ideal setting it would be advantageous to implement the P4-OvS, which has comparative performance to the stock OvS implementation.

The testbed created in this project uses solid open-source projects such as FOP4 and OpenAirInterface. In order to deploy the OAI EPC components, they do require some slight alteration of the Dockerfiles. This is done to install the necessary packages to allow the base image to be controlled by FOP4 and connect them to a OpenVSwitch instance. Because the testbed is based on Mininet, the internal performance relies on the OpenVSwitch, which is quite powerful. However the emulated radio network between the UE and RAN has its limitations and the BMV2 switches are not high performance switches compared to OpenVSwitch instances. The internal network in VirtualBox can achieve bandwidths of up to 3Gbps, which means that in a high performance environment such as a data center it will not be up to par. This leads us to considering the testbed to be an environment limited to prototyping and training.

Both of the INT systems implemented cannot be considered real-time monitoring systems, and thus they are not fully compliant with the Top-Down Approach. Neither are they network-wide, nor do they have a declarative measurement abstraction with which to dynamically adjust the use-case of the systems. Both systems however can, in theory, be implemented in Sonata [20], which does have a query-language with which to adjust what the switches should monitor. Using the approach outlined in section 4.4.1 in chapter 4 with interarrival times, it should be possible to create a delay detection query. The major limitation in this project's context is that the collected data must be first copied from the logs, then parsed and analyzed.

A natural criticism of the work is whether or not the testbed should

have been deployed in a cloud environment, rather than on a desktop computer. The defense of this choice is multi-faceted. Firstly, the COVID-19 pandemic resulted in government-ordered lockdowns, which meant that all technical work in extended periods of time had to be done from home. With that there is the uncertainty of network access to SimulaMet's cloud environment, which relies on not only the uptime of the environment itself but also the ISP. There were several short outages during the spring semester at crucial times where network access was a must, which had only a minor impact as the testbed was deployed locally. Secondly there is the consideration of deployment times and setup of automated deployments. With a cloud environment comes the requirement of more complex automation systems such as Puppet, OpenStack Heat, Ansible, or similar systems. With a more localized environment, i.e. a desktop computer, this can be done faster and more reliably, and remote work in this environment relies only on the fact that the computer is powered on.

It is also important to consider the purpose of the testbed. Other 5G and 4G testbeds focus on network slicing and are generally deployed in complex OpenStack environments with complete MANO systems. The testbed created in conjunction with this project relies on VirtualBox and has some fairly heavy resource requirements for a desktop computer. This testbed can be deployed by anyone with a similar system, but who do not necessarily have access to a much more complex and powerful cloud environment. It can also be used to test P4 prototypes or to gain familiarity with the OAI EPC and RAN. Since the EPC is deployed in a Mininet network, it can also be expanded with a fully simulated enterprise network to see how it potentially will behave in such a setting.

Chapter 7

Conclusion

The main objective of this thesis was to design and a develop a 4G LTE Testbed to investigate the performance impact of INT in such an environment, with a secondary objective to investigate how well INT can detect two common network problems in this environment. Here we have developed a testbed that deploys three virtual machines. Two of these VMs are for the OAI RAN components, the eNodeB and User Equipment, with a simulated radio connection over ethernet. The last VM runs the EPC in a FOP4 environment with all of the EPC components as containers and two BMV2 switch instances which can run P4 programs. P4 has been used to implement two different INT systems, one to detect packet loss and the other to detect network delay. We use the interarrival time between packets in a given flow to detect delay, and packet counters are used to detect packet loss. The INT system to detect delay has been compared to an Active monitoring system, which has been implemented with high frequency ping, and a Passive monitoring system, which has been implemented with Packet Capturing through Tshark. The experiments use primarily Iperf3 to generate traffic between the UE VM and an Iperf3 server located on the EPC VM in the FOP4 environment, whereas the common network problems have been introduced with NetEm on a single node between the Iperf3 server and the SPGW-U node in the EPC.

The testbed itself is flexible in that the EPC segment can be controlled through FOP4, a fork of Mininet. With this we can customize the networking between each EPC component and introduce additional network segments, which we used to deploy the Iperf3 Server. The deployment of the testbed infrastructure is managed and orchestrated with Vagrant and is publicly available on Github [56]. This testbed is, to the best of our knowledge, unique in that no other publicly available 4G LTE or 5G NGC testbeds have the capability to integrate P4 switches.

The results show that INT has a lower impact on the measured bandwidth between the UE and Iperf3 server and no significant cost to Disk IO. INT, however, incurs a higher cost to CPU and a minimal increase to the Memory cost when compared to Active and Passive monitoring. We believe this is caused by the added computation required per packet on each BMV2 switch instance. Each switch must keep a hash table keyed

on the 5-tuple of each flow, and store either the current packet count of a flow, or the previous packet's arrival time to compute the interarrival time. The resulting data from these operations is stored to the switch's logs, which causes some disk IO. Active Monitoring had the most significant impact on the measured bandwidth, which may be caused by the fact that we introduce additional traffic on top of the heavy Iperf3 traffic. Passive Monitoring had the highest measured impact on Disk IO, which is unsurprising due to the fact that Tshark will write the full content of each and every packet to a packet dump file, which causes significant Disk IO. From this we can conclude that INT is a viable monitoring system when implemented with caution, since it will incur some cost to CPU, which already is a sparse resource in the context of networking devices.

Both of the INT systems implemented here are shown to be able to detect common network problems. The delay detection system can use simple linear regression in order to detect the trend of the data. The packet loss detection system however was only able to detect 80% of lost packets. Both of these systems are flow-based and sensitive to the direction of the introduced problem. When compared to each other in terms of cost to resources and bandwidth they have similar cost. This goes to show that these types of systems, i.e. direction-sensitive flow-based INT systems, are viable systems for monitoring in the testbed environment, and both are potentially useful in larger, more complex environments.

7.1 Future Work

As it stands currently the Vagrantfile is only compatible with VirtualBox. There is the possibility of expanding or creating a similar Vagrantfile that is compatible with OpenStack. There are plugins that allow for this, and it would also mean that the testbed could be deployed in a more powerful cloud environment with more resources and a more high performance network. It would mean a more powerful testbed that can be used by students, technicians, and engineers who have access to such an environment.

Both of the INT systems can be taken several steps further. The BMV2 switch has the capability of being connected to a controller which can dynamically access counters, but not registers, at runtime. The disadvantage of using P4 counters is that they cannot be read by the switch at runtime, but they can be written to.

The INT systems can also be deployed in a more advanced testbed, such as the one created by SimulaMet, on P4-OvS switches that connect the EPC components. With this approach it is possible to also monitor the control traffic between the EPC components.

The testbed could also be used to generate datasets to train machine learning models. By expanding each component or using tools for docker containers, it is possible to collect a huge amount of information while the EPC undergoes normal operation. For example, by inducing heavy load on one specific component, how will this affect the other components.

The project relies heavily on FOP4 which has some drawbacks. One example is the fact that it does not behave exactly as a typical Python package. In order to use the libraries, the topology scripts must be placed and run from the FOP4 root directory. Ideally it would be updated to be installed as a standard Python package, similarly to ContainerNet, such that the libraries can be used from any directory on the machine which hosts the FOP4 topology.

Bibliography

[1] Minlan Yu. "Network telemetry: towards a top-down approach". In: *ACM SIGCOMM Computer Communication Review* 49.1 (2019), pp. 11–17.

[2] Changhoon Kim et al. "In-band network telemetry via programmable dataplanes". In: *ACM SIGCOMM*. Vol. 15. 2015.

[3] *P4 Language Specification*. 2020. URL: https://p4.org/p4-spec/docs/P4-16-v1.2.1.pdf.

[4] Gabriel Brown. *Service-Oriented 5G Core Networks*. Tech. rep. 2017. URL: https://www-file.huawei.com/-/media/corporate/pdf/white%20paper/heavy-reading-whitepaper--service-oriented-5g-core-networks.pdf.

[5] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. "OpenStack: toward an open-source solution for cloud computing". In: *International Journal of Computer Applications* 55.3 (2012), pp. 38–42.

[6] Ali Esmaeily and Katina Kralevska. "Small-Scale 5G Testbeds for Network Slicing Deployment: A Systematic Review". In: *Wireless Communications and Mobile Computing* 2021 (2021).

[7] Navid Nikaein et al. "OpenAirInterface: A flexible platform for 5G research". In: *ACM SIGCOMM Computer Communication Review* 44.5 (2014), pp. 33–38.

[8] *Behaviour Model (bmv2)*. URL: https://github.com/p4lang/behavioral-model.

[9] Daniele Moro et al. "Fop4: Function offloading prototyping in heterogeneous and programmable network scenarios". In: *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2019, pp. 1–6.

[10] *Smartphone Users Worldwide 2020*. 2020. URL: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

[11] Kaylie Gyarmathy. *Comprehensive Guide to IoT Statistics You Need to Know in 2020*. Mar. 2020. URL: https://www.vxchnge.com/blog/iot-statistics.

[12] Xenofon Foukas et al. "Network slicing in 5G: Survey and challenges". In: *IEEE Communications Magazine* 55.5 (2017), pp. 94–100.

[13] Kief Morris. *Infrastructure as code: managing servers in the cloud. "* O'Reilly Media, Inc.", 2016.

[14] James F Kurose and Keith W Ross. *Computer networking: a top-down approach*. Addison Wesley, 2017.

[15] Xenofon Foukas, Mahesh K Marina, and Kimon Kontovasilis. "Orion: RAN slicing for a flexible and cost-effective multi-service mobile network architecture". In: *Proceedings of the 23rd annual international conference on mobile computing and networking*. 2017, pp. 127–140.

[16] Tom Limoncelli, Christina J Hogan, and Strata R Chalup. *The practice of system and network administration*. Pearson Education, 2007.

[17] He Huang and Liqiang Wang. "P&P: A combined push-pull model for resource monitoring in cloud computing environment". In: *2010 IEEE 3rd International Conference on Cloud Computing*. IEEE. 2010, pp. 260–267.

[18] Richard Froom and Erum Frahim. *Implementing Cisco IP switched networks (SWITCH) foundation learning guide:(CCNP SWITCH 300-115)*. Cisco Press, 2015.

[19] Masoud Moshref et al. "Trumpet: Timely and precise triggers in data centers". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2016, pp. 129–143.

[20] Arpit Gupta et al. "Sonata: Query-driven streaming network telemetry". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 357–371.

[21] Behnaz Arzani et al. "007: Democratically finding the cause of packet drops". In: *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 2018, pp. 419–435.

[22] Nikhil Handigol et al. "I know what your packet did last hop: Using packet histories to troubleshoot networks". In: *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 2014, pp. 71–85.

[23] WALTER ROBERT J HARRISON et al. "Scalable, Network-Wide Telemetry with Programmable Switches". In: (2019).

[24] Dan RK Ports and Jacob Nelson. "When Should The Network Be The Computer?" In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2019, pp. 209–215.

[25] Priyanka Naik, Dilip Kumar Shaw, and Mythili Vutukuru. "NFVPerf: Online performance monitoring and bottleneck detection for NFV". In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2016, pp. 154–160.

[26] Vincenzo Sciancalepore, Faqir Zarrar Yousaf, and Xavier Costa-Perez. "z-TORCH: An automated NFV orchestration and monitoring solution". In: *IEEE Transactions on Network and Service Management* 15.4 (2018), pp. 1292–1306.

[27] Niels LM Van Adrichem, Christian Doerr, and Fernando A Kuipers. "Opennetmon: Network monitoring in openflow software-defined networks". In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE. 2014, pp. 1–8.

[28] Madhusanka Liyanage et al. "Software defined monitoring (sdm) for 5g mobile backhaul networks". In: *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE. 2017, pp. 1–6.

[29] Taras Maksymyuk et al. "An IoT based monitoring framework for software defined 5G mobile networks". In: *Proceedings of the 11th international conference on ubiquitous information management and communication*. 2017, pp. 1–4.

[30] Shihabur Rahman Chowdhury et al. "Payless: A low cost network monitoring framework for software defined networks". In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE. 2014, pp. 1–9.

[31] Ye Yu, Chen Qian, and Xin Li. "Distributed and collaborative traffic monitoring in software defined networks". In: *Proceedings of the third workshop on Hot topics in software defined networking*. 2014, pp. 85–90.

[32] Jacob H Cox et al. "Advancing software-defined networks: A survey". In: *IEEE Access* 5 (2017), pp. 25487–25526.

[33] Ben Pfaff et al. "The design and implementation of open vswitch". In: *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 2015, pp. 117–130.

[34] Frederik Hauser et al. "A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research". In: *arXiv preprint arXiv:2101.10632* (2021).

[35] Rob Harrison et al. "Network-wide heavy hitter detection with commodity switches". In: *Proceedings of the Symposium on SDN Research*. 2018, pp. 1–7.

[36] Jonatas Adilson Marques et al. "An optimization-based approach for efficient network monitoring using in-band network telemetry". In: *Journal of Internet Services and Applications* 10.1 (2019), p. 12.

[37] Lizhuang Tan et al. "In-band network telemetry: a survey". In: *Computer Networks* 186 (2021), p. 107763.

[38] Nick McKeown et al. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM computer communication review* 38.2 (2008), pp. 69–74.

[39] Tomasz Osiński et al. "P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch with P4". In: *2020 IFIP Networking Conference (Networking)*. IEEE. 2020, pp. 413–421.

[40] Bob Lantz, Brandon Heller, and Nick McKeown. "A network in a laptop: rapid prototyping for software-defined networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. 2010, pp. 1–6.

[41] M. Peuster, H. Karl, and S. van Rossem. "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments". In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. Nov. 2016, pp. 148–153. DOI: 10.1109/NFV-SDN.2016.7919490.

[42] NFVISG ETSI. "Network Functions Virtualization-Introductory White Paper". In: *SDN and OpenFlow World Congress*. 2012.

[43] *Federation of all OpenAirCN components*. URL: https://github.com/OPENAIRINTERFACE/openair-epc-fed.

[44] *OpenAirInterface5G*. URL: https://gitlab.eurecom.fr/oai/openairinterface5g/-/tree/master.

[45] Thomas Dreibholz. "Flexible 4G/5G Testbed Setup for Mobile Edge Computing using OpenAirInterface and OpenSource MANO". In: (2020).

[46] Bruno Dzogovic, Boning Feng, Thanh Van Do, et al. "Building virtualized 5G networks using open source software". In: *2018 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*. IEEE. 2018, pp. 360–366.

[47] Bruno Dzogovic et al. "Connecting remote eNodeB with containerized 5G C-RANs in OpenStack cloud". In: *2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*. IEEE. 2019, pp. 14–19.

[48] *5G Test Network - Explore 5G features and performance in a controlled, real environment*. URL: https://5gtn.fi/.

[49] Changhoon Kim et al. "In-band Network Telemetry via Programmable Dataplanes". In: 2015.

[50] Changhoon Kim et al. "In-Band Network Telemetry (INT)". In: (2016).

[51] Abdulkadir Karaagac, Eli De Poorter, and Jeroen Hoebeke. "In-band network telemetry in industrial wireless sensor networks". In: *IEEE Transactions on Network and Service Management* 17.1 (2019), pp. 517–531.

[52] *Tshark - Dump and Analyze network traffic*. URL: https://www.wireshark.org/docs/man-pages/tshark.html.

[53] *Wireshark - The world's foremost and widely-used network protocol analyzer*. URL: https://www.wireshark.org/.

[54] *5gvinni-oai-ns*. URL: https://github.com/simula/5gvinni-oai-ns.

[55] *Vagrant - the command line utility for managing the lifecycle of virtual machines*. URL: https://www.vagrantup.com/.

[56] *P4G Testbed - A virtual 4G LTE Network testbed with support for P4*. URL: https://github.com/vetletm/P4G-testbed.

[57] *Guide to p4lang repositories and some other public info about P4*. URL: https://github.com/jafingerhut/p4-guide.

[58] *OpenAir-CN - MME, modified to work with FOP4*. URL: https://github.com/vetletm/openair-mme/tree/fop4_extension_new.

[59] *OpenAir-CN - HSS, modified to work with FOP4*. URL: https://github.com/vetletm/openair-hss/tree/fop4_extension_new.

[60] *OpenAir-CN - SPGW-C, modified to work with FOP4*. URL: https://github.com/vetletm/openair-spgwc/tree/fop4_extension_new.

[61] *OpenAir-CN - SPGW-U, modified to work with FOP4*. URL: https://github.com/vetletm/openair-spgwu-tiny/tree/fop4_extension_new.

[62] Stephen Hemminger et al. "Network emulation with NetEm". In: *Linux conf au*. Vol. 5. Citeseer. 2005, p. 2005.

Appendices

Appendix A

Listings

A.1 Vagrant Provisioning

A.1.1 Vagrantfile

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

"""
This Vagrantfile will set up 3 VMs, requiring up to 90GB total storage, 12GB RAM and at least 6 available cores.

These VMs are connected using the internal VBox Network and each has a NIC connected to the VBox NAT Network for Internet Access

The eNB and UE VMs use the same script for configuration and requires the user to manually build the binaries

The EPC machine requires the user to manually run the ansible-playbook located in FOP4/ansible
  - Any topology script must be run from within the FOP4 directory

"""

Vagrant.configure("2") do |config|
  # Dedicated VM for eNB and UE
  config.vm.define "enb" do |enb|
    # Using Ubuntu 18.04 Bionic Beaver
    enb.vm.box = "ubuntu/bionic64"

    # Using vagrant-disksize plugin
    enb.disksize.size =  20GB

    # set name, ram, cpus
    enb.vm.provider "virtualbox" do |v|
      v.name = "ran"
```

```
28        v.memory = 4096
29        v.cpus = 2
30        v.customize ["modifyvm", :id, "--natnet1", "192.168.72.0/24"]
31      end
32
33      enb.vm.network "private_network", ip: "10.10.1.2", virtualbox__intnet:
        ↪  true
34
35      # Install required packages and build binaries
36      enb.vm.provision "shell", path: "scripts/bootstrap_ran.sh"
37
38      # Copy and move config files to correct locations
39      enb.vm.provision "file", source: "config/lte-fdd-basic-sim.conf",
        ↪  destination: "~/"
40      enb.vm.provision "shell", inline: "mv
        ↪  /home/vagrant/lte-fdd-basic-sim.conf
        ↪  /home/netmon/src/enb_folder/ci-scripts/conf_files/"
41
42    end
43
44    # VM for UE
45    config.vm.define "ue" do |ue|
46      ue.vm.box = "ubuntu/bionic64"
47      ue.disksize.size = "20GB"
48
49      ue.vm.provider "virtualbox" do |v|
50        v.name = "ue"
51        v.memory = 4096
52        v.cpus = 2
53        v.customize ["modifyvm", :id, "--natnet1", "192.168.72.0/24"]
54      end
55
56      ue.vm.network "private_network", ip: "10.10.1.3", virtualbox__intnet:
        ↪  true
57
58      # Install required packages and build binaries
59      ue.vm.provision "shell", path: "scripts/bootstrap_ran.sh"
60
61      # Copy and move config files to correct locations
62      ue.vm.provision "file", source: "config/ue_eurecom_test_sfr.conf",
        ↪  destination: "~/"
63      ue.vm.provision "shell", source: "mv
        ↪  /home/vagrant/ue_eurecom_test_sfr.conf
        ↪  /home/netmon/src/enb_folder/openair3/NAS/TOOLS/ue_eurecom_test_sfr.conf"
64    end
65
66    # VM for EPC and FOP4 topology
67    config.vm.define "epct" do |epct|
68      epct.vm.box =  ubuntu/bionic64
```

```
    epct.disksize.size =  50GB
    epct.vm.provider "virtualbox" do |v|
      v.name = "epct"
      v.memory = 4096
      v.cpus = 2
      v.customize ["modifyvm", :id, "--natnet1", "192.168.72.0/24"]
    end
    epct.vm.network "private_network", ip: "10.10.1.4", virtualbox__intnet:
    ↪  true
    epct.vm.provision "shell", path: "scripts/bootstrap_epc.sh"
  end
end
```

A.1.2 VM Bootstrap scripts

The following scripts are used by Vagrant to provision each VM and set them up with the necessary software packages and repositories. Both scripts follow the same structure, first they create the netmon-user and related permissions, sudo-rights, and folders. Next, they install some required packages and clone down necessary repositories. Finally they set the necessary ownership of the netmon home-folder.

RAN Bootstrap

```
#!/usr/bin/env bash

set +x

apt-get update
# Needed to install Tshark (and by extension, Wireshark) without any
↪  installation prompts
echo "wireshark-common wireshark-common/install-setuid boolean true" |
↪  debconf-set-selections
DEBIAN_FRONTEND=noninteractive apt-get install -y git tshark
↪  linux-image-5.4.0-66-lowlatency linux-headers-5.4.0-66-lowlatency iperf3

# Set up the local user with a known password and add necessary permissions,
↪  set groups, and add home folder
useradd -m -d /home/netmon -s /bin/bash netmon
echo "netmon:netmon" | chpasswd
echo "netmon ALL=(ALL) NOPASSWD:ALL" > /etc/sudoers.d/99_netmon
chmod 440 /etc/sudoers.d/99_netmon
usermod -aG vboxsf netmon

mkdir /home/netmon/src

# Pull the OpenAirInterface and checkout the required version. The eNB and
↪  UE are always in distinct folders for increased usability.
```

```
BASE_DIR="/home/netmon/src"
cd "$BASE_DIR"
git clone https://gitlab.eurecom.fr/oai/openairinterface5g.git enb_folder
cd enb_folder
git checkout v1.2.2
cd ..
cp -R enb_folder/ ue_folder

# Set necessary ownership
chown -R netmon:netmon /home/netmon/
```

EPC Bootstrap

```
#!/usr/bin/env bash

# Set up the local user with a known password and add necessary permissions, set groups, and add home folder
useradd -m -d /home/netmon -s /bin/bash netmon
echo "netmon:netmon" | chpasswd
echo "netmon ALL=(ALL) NOPASSWD:ALL" > /etc/sudoers.d/99_netmon
chmod 440 /etc/sudoers.d/99_netmon
usermod -aG vboxsf netmon

# Prepare relevant folders and permissions
mkdir -p /home/netmon/src/openair-components
mkdir -p /home/netmon/src/archives
chown -R netmon:netmon /home/netmon/src

# ENVs to properly pull and build Docker images
BASE_DIR="/home/netmon/src"
EPC_DIR="/home/netmon/src/openair-components"

sudo apt-get update
sudo apt-get install git

# Pull the required repositories to build the EPC components and the FOP4 environment
git clone --branch fop4_extension_new https://github.com/vetletm/openair-hss.git "$EPC_DIR/openair-hss"
git clone --branch fop4_extension_new https://github.com/vetletm/openair-mme.git "$EPC_DIR/openair-mme"
git clone --branch fop4_extension_new https://github.com/vetletm/openair-spgwc.git "$EPC_DIR/openair-spgwc"
git clone --branch fop4_extension_new https://github.com/vetletm/openair-spgwu-tiny.git "$EPC_DIR/openair-spgwu-tiny"
git clone --branch vetletm-fix-ansible https://github.com/vetletm/FOP4 "$BASE_DIR/FOP4"
```

```
28 git clone https://github.com/OPENAIRINTERFACE/openair-epc-fed.git
   ↪ "$BASE_DIR/openair-epc-fed"
29
30 # Build each image in turn
31 cd "$EPC_DIR"
32 cd openair-hss
33 sudo -E docker build --target oai-hss --tag oai-hss:production --file
   ↪ docker/Dockerfile.ubuntu18.04 .
34 cd "$EPC_DIR"
35
36 cd openair-mme
37 sudo -E docker build --target oai-mme --tag oai-mme:production --file
   ↪ docker/Dockerfile.ubuntu18.04
38 cd "$EPC_DIR"
39
40 cd openair-spgwc
41 sudo -E docker build --target oai-spgwc --tag oai-spgwc:production --file
   ↪ docker/Dockerfile.ubuntu18.04 .
42 cd "$EPC_DIR"
43
44 cd openair-spgwc
45 sudo -E docker build --target oai-spgwc --tag oai-spgwc:production --file
   ↪ docker/Dockerfile.ubuntu18.04 .
46 cd "$BASE_DIR"
47
48 # Pull repository needed to install P4, BMV2, Mininet and all necessary
   ↪ dependencies
49 git clone https://github.com/jafingerhut/p4-guide "$BASE_DIR/p4-guide"
50 cd "$BASE_DIR"
51 # This script calls exit on finish, nothing can be done after it through
   ↪ vagrant.
52 ./p4-guide/bin/install-p4dev-v2.sh |& tee log.txt
```

A.2 EPC Scripts

The following sections contain the scripts used to deploy, configure, start and stop the EPC network functions as containers.

A.2.1 Docker Deployment and Configuration

Here is a collection of simple management scripts used after the EPC, RAN and FOP4 environment has been fully configured.

Copy and execute configuration-scripts

```
1 #!/usr/bin/env bash
2 echo "Configuring HSS"
```

```
sudo -E docker cp ./hss-cfg.sh mn.hss:/openair-hss/scripts
sudo -E docker exec -it mn.hss /bin/bash -c "cd /openair-hss/scripts &&
↪  chmod 777 hss-cfg.sh && ./hss-cfg.sh"

echo "Configuring MME"
sudo -E docker cp ./mme-cfg.sh mn.mme:/openair-mme/scripts
sudo -E docker exec -it mn.mme /bin/bash -c "cd /openair-mme/scripts &&
↪  chmod 777 mme-cfg.sh && ./mme-cfg.sh"

echo "Configuring SPGW-C"
sudo -E docker cp ./spgwc-cfg.sh mn.spgwc:/openair-spgwc
sudo -E docker exec -it mn.spgwc /bin/bash -c "cd /openair-spgwc && chmod
↪  777 spgwc-cfg.sh && ./spgwc-cfg.sh"

echo "Configuring SPGW-U"
sudo -E docker cp ./spgwu-cfg.sh mn.spgwu:/openair-spgwu-tiny
sudo -E docker exec -it mn.spgwu /bin/bash -c "cd /openair-spgwu-tiny &&
↪  chmod 777 spgwu-cfg.sh && ./spgwu-cfg.sh"
```

Start Tshark Packet Capturing

```
#!/usr/bin/env bash
# Sets up all containers with Tshark for Packet Capturing

echo "Starting Tshark on all FOP4 network hosts: HSS, MME, SPGW-C, SPGW-U,
↪  Forwarder, Iperf_dst"
# Start network logs
sudo -E docker exec -d mn.hss /bin/bash -c "nohup tshark -i hss-eth0 -i eth0
↪  -w /tmp/hss_check_run.pcap 2>&1 > /dev/null"
sudo -E docker exec -d mn.mme /bin/bash -c "nohup tshark -i mme-eth0 -i
↪  lo:s10 -i eth0 -w /tmp/mme_check_run.pcap 2>&1 > /dev/null"
sudo -E docker exec -d mn.spgwc /bin/bash -c "nohup tshark -i spgwc-eth0 -i
↪  lo:p5c -i lo:s5c -w /tmp/spgwc_check_run.pcap 2>&1 > /dev/null"
sudo -E docker exec -d mn.spgwu /bin/bash -c "nohup tshark -i any -w
↪  /tmp/spgwu_check_run.pcap 2>&1 > /dev/null"
sudo -E docker exec -d mn.forwarder /bin/bash -c "nohup tshark -i
↪  forwarder-eth2 -i forwarder-eth3 -w /tmp/forwarder_check_run.pcap 2>&1 >
↪  /dev/null"
sudo -E docker exec -d mn.iperf_dst /bin/bash -c "nohup tshark -i
↪  iperf_dst-eth0 -w /tmp/iperf_dst_check_run.pcap 2>&1 > /dev/null"
```

Start each EPC component in turn

```
#!/usr/bin/env bash
# Starts each EPC component with a 2 second pause

```

```
echo "Starting HSS ..."
sudo -E docker exec -d mn.hss /bin/bash -c "nohup ./bin/oai_hss -j ./etc/hss_rel14.json --reloadkey true > hss_check_run.log 2>&1"
sleep 2

echo "Starting MME ..."
sudo -E docker exec -d mn.mme /bin/bash -c "nohup ./bin/oai_mme -c ./etc/mme.conf > mme_check_run.log 2>&1"
sleep 2

echo "Starting SPGW-C ..."
sudo -E docker exec -d mn.spgwc /bin/bash -c "nohup ./bin/oai_spgwc -o -c ./etc/spgw_c.conf > spgwc_check_run.log 2>&1"
sleep 2

echo "Starting SPGW-U ..."
sudo -E docker exec -d mn.spgwu /bin/bash -c "nohup ./bin/oai_spgwu -o -c ./etc/spgw_u.conf > spgwu_check_run.log 2>&1"
```

Stop each EPC component in turn and stop Tshark processes

```
#!/usr/bin/env bash
# Stops each EPC component and the Tshark process, first with SIGINT, then SIGKILL to ensure they are shut down.

echo "Sending SIGINT to EPC component processes and Tshark ..."
sudo -E docker exec -it mn.hss /bin/bash -c "killall --signal SIGINT oai_hss tshark"
sudo -E docker exec -it mn.mme /bin/bash -c "killall --signal SIGINT oai_mme tshark"
sudo -E docker exec -it mn.spgwc /bin/bash -c "killall --signal SIGINT oai_spgwc tshark"
sudo -E docker exec -it mn.spgwu /bin/bash -c "killall --signal SIGINT oai_spgwu tshark"

echo "Sleeping 10 seconds and then sending SIGKILL to EPC component processes and Tshark"
sleep 10
sudo -E docker exec -it mn.hss /bin/bash -c "killall --signal SIGKILL oai_hss tshark tcpdump"
sudo -E docker exec -it mn.mme /bin/bash -c "killall --signal SIGKILL oai_mme tshark tcpdump"
sudo -E docker exec -it mn.spgwc /bin/bash -c "killall --signal SIGKILL oai_spgwc tshark tcpdump"
sudo -E docker exec -it mn.spgwu /bin/bash -c "killall --signal SIGKILL oai_spgwu tshark tcpdump"
```

Collect all relevant files

```
#!/usr/bin/env bash
# This script collects all configuration files, log files, and packetdumps, then zips everything together.

echo "Starting application log and PCAP collection"
sudo rm -rf EPC
sudo mkdir -p EPC/oai-hss-cfg EPC/oai-mme-cfg EPC/oai-spgwc-cfg EPC/oai-spgwu-cfg EPC/hss-logs

echo "Collecting configuration files ..."
sudo -E docker cp mn.hss:/openair-hss/etc/. EPC/oai-hss-cfg
sudo -E docker cp mn.mme:/openair-mme/etc/. EPC/oai-mme-cfg
sudo -E docker cp mn.spgwc:/openair-spgwc/etc/. EPC/oai-spgwc-cfg
sudo -E docker cp mn.spgwu:/openair-spgwu-tiny/etc/. EPC/oai-spgwu-cfg

echo "Collecting log files ..."
sudo -E docker cp mn.hss:/openair-hss/hss_check_run.log EPC
sudo -E docker cp mn.hss:/openair-hss/logs/ EPC/hss-logs
sudo -E docker cp mn.mme:/openair-mme/mme_check_run.log EPC
sudo -E docker cp mn.spgwc:/openair-spgwc/spgwc_check_run.log EPC
sudo -E docker cp mn.spgwu:/openair-spgwu-tiny/spgwu_check_run.log EPC

echo "Collecting PCAP files ..."
sudo -E docker cp mn.hss:/tmp/hss_check_run.pcap EPC
sudo -E docker cp mn.mme:/tmp/mme_check_run.pcap EPC
sudo -E docker cp mn.spgwc:/tmp/spgwc_check_run.pcap EPC
sudo -E docker cp mn.spgwu:/tmp/spgwu_check_run.pcap EPC
sudo -E docker cp mn.forwarder:/tmp/forwarder_check_run.pcap EPC
sudo -E docker cp mn.iperf_dst:/tmp/iperf_dst_check_run.pcap EPC

filename="$(date  +%Y%m%d-%H%M%S )-epc-archives"
sudo -E zip -r -qq "$filename".zip EPC
echo "Saved all logs and PCAP files as archive with name: $filename"
```

A.3 FOP4 Topology definitions

A.3.1 Basic Topology with minimal monitoring

```
#!/usr/bin/python

from mininet.net import Containernet
from mininet.node import Controller, Node, OVSKernelSwitch
from mininet.cli import CLI
from mininet.link import TCLink
from mininet.log import info, setLogLevel
```

```
8  from mininet.bmv2 import Bmv2Switch, P4DockerHost
9
10
11 setLogLevel( info )
12
13
14 net = Containernet(controller=Controller)
15
16 info( *** Adding controller\n )
17 net.addController( c0 )
18
19 info( *** Adding docker containers\n )
20 # EPC
21 hss = net.addDocker( hss ,
22                      cls=P4DockerHost,
23                      ip= 192.168.61.2/24 ,
24                      dimage= oai-hss:production )
25 mme = net.addDocker( mme ,
26                      cls=P4DockerHost,
27                      ip= 192.168.61.3/24 ,
28                      dimage= oai-mme:production )
29 spgw_c = net.addDocker( spgwc ,
30                      cls=P4DockerHost,
31                      ip= 192.168.61.4/24 ,
32                      dimage= oai-spgwc:production )
33 spgw_u = net.addDocker( spgwu ,
34                      cls=P4DockerHost,
35                      ip= 192.168.61.5/24 ,
36                      dimage= oai-spgwu-tiny:production )
37 # Segment for testing monitoring
38 forwarder = net.addDocker( forwarder ,
39                      cls=P4DockerHost,
40                      ip= 192.168.62.3/24 ,
41                      mac= 00:00:00:00:00:F3 ,
42                      dimage= forwarder:1804 )
43 iperf_dst = net.addDocker( iperf_dst ,
44                      cls=P4DockerHost,
45                      ip= 192.168.63.3/24 ,
46                      mac= 00:00:00:00:00:D3 ,
47                      dimage= iperf:1804 )
48
49 info( *** Adding core switch\n )
50 s1 = net.addSwitch( s1 , cls=OVSKernelSwitch)
51
52 info( *** Adding BMV2 switches\n )
53 s2 = net.addSwitch( s2 , cls=Bmv2Switch, json= ./forwarder.json ,
   ↪  switch_config= ./s2f_commands.txt )
54 s3 = net.addSwitch( s3 , cls=Bmv2Switch, json= ./forwarder.json ,
   ↪  switch_config= ./s3f_commands.txt )
```

```
55
56  info( *** Creating links\n )
57  net.addLink(hss, s1)
58  net.addLink(mme, s1)
59  net.addLink(spgw_c, s1)
60  net.addLink(spgw_u, s1)
61  net.addLink(spgw_u, s2, intfName1= spgwu-eth2 , port1=2, port2=1)
62  net.addLink(forwarder, s2, intfName1= forwarder-eth2 , port1=1, port2=2)
63  net.addLink(forwarder, s3, intfName1= forwarder-eth3 , port1=2, port2=1)
64  net.addLink(iperf_dst, s3, port2=2)
65
66  info( *** Setting up additional interfaces on: forwarder, spgwu_u\n )
67  # Set MAC and IP on new interfaces
68  spgw_u.setMAC(mac= 00:00:00:00:00:F2 , intf= spgwu-eth2 )
69  spgw_u.setIP(ip= 192.168.62.2 , prefixLen=24, intf= spgwu-eth2 )
70  forwarder.setMAC(mac= 00:00:00:00:00:D2 , intf= forwarder-eth3 )
71  forwarder.setIP(ip= 192.168.63.2 , prefixLen=24, intf= forwarder-eth3 )
72
73  info( *** Setting up forwarding on: forwarder\n )
74  # set up forwarding
75  forwarder.cmd( iptables -P FORWARD ACCEPT )
76  forwarder.cmd( sysctl net.ipv4.conf.all.forwarding=1 )
77
78  info( *** Starting network\n )
79  net.start()
80  net.staticArp()
81
82  info( *** Setting up additional ARP\n )
83  # Some ARP entries must be manually added:
84  forwarder.setARP( 192.168.62.2 ,  00:00:00:00:00:F2 )
85  forwarder.setARP( 192.168.63.3 ,  00:00:00:00:00:D3 )
86  iperf_dst.setARP( 192.168.63.2 ,  00:00:00:00:00:D2 )
87
88  info( *** Setting up additional routing\n )
89  # Set up appropriate routing for hosts connected to more than one network
90  spgw_u.cmd( ip route add 192.168.63.0/24 via 192.168.62.3 )
91  forwarder.cmd( ip route add 12.1.1.0/24 via 192.168.62.2 )
92  iperf_dst.cmd( ip route add 192.168.62.0/24 via 192.168.63.2 )
93  iperf_dst.cmd( ip route add 12.1.1.0/24 via 192.168.63.2 )
94
95  info( *** Disabling TCP checksum verification on hosts: iperf_dst,
    ↪  forwarder, spgw_u\n )
96  # Don't verify TCP checksums, as BMV2 switches change this up and causes TCP
    ↪  packets to be dropped by the kernel:
97  iperf_dst.cmd( ethtool --offload iperf_dst-eth0 rx off tx off )
98  forwarder.cmd( ethtool --offload forwarder-eth2 rx off tx off )
99  forwarder.cmd( ethtool --offload forwarder-eth3 rx off tx off )
100 spgw_u.cmd( ethtool --offload spgwu-eth2 rx off tx off )
101
```

```
info( *** Running CLI\n )
CLI(net)

info( *** Stopping network )
net.stop()
```

A.3.2 Topology with Packet Capturing

```
#!/usr/bin/python

from mininet.net import Containernet
from mininet.node import Controller, Node, OVSKernelSwitch
from mininet.cli import CLI
from mininet.link import TCLink
from mininet.log import info, setLogLevel
from mininet.bmv2 import Bmv2Switch, P4DockerHost


setLogLevel( info )


net = Containernet(controller=Controller)

info( *** Adding controller\n )
net.addController( c0 )

info( *** Adding docker containers\n )
# EPC
hss = net.addDocker( hss ,
                     cls=P4DockerHost,
                     ip= 192.168.61.2/24 ,
                     dimage= oai-hss:production )
mme = net.addDocker( mme ,
                     cls=P4DockerHost,
                     ip= 192.168.61.3/24 ,
                     dimage= oai-mme:production )
spgw_c = net.addDocker( spgwc ,
                     cls=P4DockerHost,
                     ip= 192.168.61.4/24 ,
                     dimage= oai-spgwc:production )
spgw_u = net.addDocker( spgwu ,
                     cls=P4DockerHost,
                     ip= 192.168.61.5/24 ,
                     dimage= oai-spgwu-tiny:production )
# Segment for testing monitoring
forwarder = net.addDocker( forwarder ,
                     cls=P4DockerHost,
```

```
                    ip= 192.168.62.3/24 ,
                    mac= 00:00:00:00:00:F3 ,
                    dimage= forwarder:1804 )
iperf_dst = net.addDocker( iperf_dst ,
                    cls=P4DockerHost,
                    ip= 192.168.63.3/24 ,
                    mac= 00:00:00:00:00:D3 ,
                    dimage= iperf:1804 )

info( *** Adding core switch\n )
s1 = net.addSwitch( s1 , cls=OVSKernelSwitch)

info( *** Adding BMV2 switches\n )
s2 = net.addSwitch( s2 , cls=Bmv2Switch, json= ./forwarder.json ,
                   loglevel= debug , pktdump=True,
                   ↪  switch_config= ./s2f_commands.txt )
s3 = net.addSwitch( s3 , cls=Bmv2Switch, json= ./forwarder.json ,
                   loglevel= debug , pktdump=True,
                   ↪  switch_config= ./s3f_commands.txt )

info( *** Creating links\n )
net.addLink(hss, s1)
net.addLink(mme, s1)
net.addLink(spgw_c, s1)
net.addLink(spgw_u, s1)
net.addLink(spgw_u, s2, intfName1= spgwu-eth2 , port1=2, port2=1)
net.addLink(forwarder, s2, intfName1= forwarder-eth2 , port1=1, port2=2)
net.addLink(forwarder, s3, intfName1= forwarder-eth3 , port1=2, port2=1)
net.addLink(iperf_dst, s3, port2=2)

info( *** Setting up additional interfaces on: forwarder, spgwu_u\n )
# Set MAC and IP on new interfaces
spgw_u.setMAC(mac= 00:00:00:00:00:F2 , intf= spgwu-eth2 )
spgw_u.setIP(ip= 192.168.62.2 , prefixLen=24, intf= spgwu-eth2 )
forwarder.setMAC(mac= 00:00:00:00:00:D2 , intf= forwarder-eth3 )
forwarder.setIP(ip= 192.168.63.2 , prefixLen=24, intf= forwarder-eth3 )

info( *** Setting up forwarding on: forwarder\n )
# set up forwarding
forwarder.cmd( iptables -P FORWARD ACCEPT )
forwarder.cmd( sysctl net.ipv4.conf.all.forwarding=1 )

info( *** Starting network\n )
net.start()
net.staticArp()

info( *** Setting up additional ARP\n )
# Some ARP entries must be manually added:
forwarder.setARP( 192.168.62.2 ,  00:00:00:00:00:F2 )
```

```
forwarder.setARP( 192.168.63.3 ,  00:00:00:00:00:D3 )
iperf_dst.setARP( 192.168.63.2 ,  00:00:00:00:00:D2 )

info( *** Setting up additional routing\n )
# Set up appropriate routing for hosts connected to more than one network
spgw_u.cmd( ip route add 192.168.63.0/24 via 192.168.62.3 )
forwarder.cmd( ip route add 12.1.1.0/24 via 192.168.62.2 )
iperf_dst.cmd( ip route add 192.168.62.0/24 via 192.168.63.2 )
iperf_dst.cmd( ip route add 12.1.1.0/24 via 192.168.63.2 )

info( *** Disabling TCP checksum verification on hosts: iperf_dst,
↪  forwarder, spgw_u\n )
# Don't verify TCP checksums, as BMV2 switches change this up and causes TCP
↪  packets to be dropped by the kernel:
iperf_dst.cmd( ethtool --offload iperf_dst-eth0 rx off tx off )
forwarder.cmd( ethtool --offload forwarder-eth2 rx off tx off )
forwarder.cmd( ethtool --offload forwarder-eth3 rx off tx off )
spgw_u.cmd( ethtool --offload spgwu-eth2 rx off tx off )

info( *** Running CLI\n )
CLI(net)

info( *** Stopping network )
net.stop()
```

A.3.3 Topology with P4 INT

```
#!/usr/bin/python

from mininet.net import Containernet
from mininet.node import Controller, Node, OVSKernelSwitch
from mininet.cli import CLI
from mininet.link import TCLink
from mininet.log import info, setLogLevel
from mininet.bmv2 import Bmv2Switch, P4DockerHost


setLogLevel( info )


net = Containernet(controller=Controller)

info( *** Adding controller\n )
net.addController( c0 )

info( *** Adding docker containers\n )
# EPC
```

```
hss = net.addDocker( hss ,
                     cls=P4DockerHost,
                     ip= 192.168.61.2/24 ,
                     dimage= oai-hss:production )
mme = net.addDocker( mme ,
                     cls=P4DockerHost,
                     ip= 192.168.61.3/24 ,
                     dimage= oai-mme:production )
spgw_c = net.addDocker( spgwc ,
                     cls=P4DockerHost,
                     ip= 192.168.61.4/24 ,
                     dimage= oai-spgwc:production )
spgw_u = net.addDocker( spgwu ,
                     cls=P4DockerHost,
                     ip= 192.168.61.5/24 ,
                     dimage= oai-spgwu-tiny:production )
# Segment for testing monitoring
forwarder = net.addDocker( forwarder ,
                     cls=P4DockerHost,
                     ip= 192.168.62.3/24 ,
                     mac= 00:00:00:00:00:F3 ,
                     dimage= forwarder:1804 )
iperf_dst = net.addDocker( iperf_dst ,
                     cls=P4DockerHost,
                     ip= 192.168.63.3/24 ,
                     mac= 00:00:00:00:00:D3 ,
                     dimage= iperf:1804 )

info( *** Adding core switch\n )
s1 = net.addSwitch( s1 , cls=OVSKernelSwitch)

info( *** Adding BMV2 switches\n )
s2 = net.addSwitch( s2 , cls=Bmv2Switch, json= ./timestamping_s2.json ,
                    loglevel= info , switch_config= ./s2f_commands.txt )
s3 = net.addSwitch( s3 , cls=Bmv2Switch, json= ./timestamping_s3.json ,
                    loglevel= info , switch_config= ./s3f_commands.txt )

info( *** Creating links\n )
net.addLink(hss, s1)
net.addLink(mme, s1)
net.addLink(spgw_c, s1)
net.addLink(spgw_u, s1)
net.addLink(spgw_u, s2, intfName1= spgwu-eth2 , port1=2, port2=1)
net.addLink(forwarder, s2, intfName1= forwarder-eth2 , port1=1, port2=2)
net.addLink(forwarder, s3, intfName1= forwarder-eth3 , port1=2, port2=1)
net.addLink(iperf_dst, s3, port2=2)

info( *** Setting up additional interfaces on: forwarder, spgwu_u\n )
# Set MAC and IP on new interfaces
```

```
70  spgw_u.setMAC(mac= 00:00:00:00:00:F2 , intf= spgwu-eth2 )
71  spgw_u.setIP(ip= 192.168.62.2 , prefixLen=24, intf= spgwu-eth2 )
72  forwarder.setMAC(mac= 00:00:00:00:00:D2 , intf= forwarder-eth3 )
73  forwarder.setIP(ip= 192.168.63.2 , prefixLen=24, intf= forwarder-eth3 )
74
75  info( *** Setting up forwarding on: forwarder\n )
76  # set up forwarding
77  forwarder.cmd( iptables -P FORWARD ACCEPT )
78  forwarder.cmd( sysctl net.ipv4.conf.all.forwarding=1 )
79
80  info( *** Starting network\n )
81  net.start()
82  net.staticArp()
83
84  info( *** Setting up additional ARP\n )
85  # Some ARP entries must be manually added:
86  forwarder.setARP( 192.168.62.2 ,  00:00:00:00:00:F2 )
87  forwarder.setARP( 192.168.63.3 ,  00:00:00:00:00:D3 )
88  iperf_dst.setARP( 192.168.63.2 ,  00:00:00:00:00:D2 )
89
90  info( *** Setting up additional routing\n )
91  # Set up appropriate routing for hosts connected to more than one network
92  spgw_u.cmd( ip route add 192.168.63.0/24 via 192.168.62.3 )
93  forwarder.cmd( ip route add 12.1.1.0/24 via 192.168.62.2 )
94  iperf_dst.cmd( ip route add 192.168.62.0/24 via 192.168.63.2 )
95  iperf_dst.cmd( ip route add 12.1.1.0/24 via 192.168.63.2 )
96
97  info( *** Disabling TCP checksum verification on hosts: iperf_dst,
    ↪  forwarder, spgw_u\n )
98  # Don't verify TCP checksums, as BMV2 switches change this up and causes TCP
    ↪  packets to be dropped by the kernel:
99  iperf_dst.cmd( ethtool --offload iperf_dst-eth0 rx off tx off )
100 forwarder.cmd( ethtool --offload forwarder-eth2 rx off tx off )
101 forwarder.cmd( ethtool --offload forwarder-eth3 rx off tx off )
102 spgw_u.cmd( ethtool --offload spgwu-eth2 rx off tx off )
103
104 info( *** Running CLI\n )
105 CLI(net)
106
107 info( *** Stopping network )
108 net.stop()
```

A.4 P4 Source

A.4.1 Basic Forwarder

```
/* -*- P4_16 -*- */
/* Disclaimer: This p4-code is copied from p4lang/tutorials (exercises/basic/solution) to save time, and I've modified it to reduce TTL by 5 instead of 1 to make it explicitly clear when it has run. */
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;

/*************************************************************************
*********************** H E A D E R S  ***********************************
*************************************************************************/

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv; // TOS / DSCP
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

struct metadata {
    /* empty */
}

struct headers {
    ethernet_t   ethernet;
```

```
43      ipv4_t       ipv4;
44  }
45
46  /*************************************************************************
47  ********************** P A R S E R  ***************************************
48  *************************************************************************/
49
50  parser MyParser(packet_in packet,
51                  out headers hdr,
52                  inout metadata meta,
53                  inout standard_metadata_t standard_metadata) {
54
55      state start {
56          transition parse_ethernet;
57      }
58
59      state parse_ethernet {
60          packet.extract(hdr.ethernet);
61          transition select(hdr.ethernet.etherType) {
62              TYPE_IPV4: parse_ipv4;
63              default: accept;
64          }
65      }
66
67      state parse_ipv4 {
68          packet.extract(hdr.ipv4);
69          transition accept;
70      }
71
72  }
73
74  /*************************************************************************
75  ************   C H E C K S U M    V E R I F I C A T I O N   *************
76  *************************************************************************/
77
78  control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
79      apply {  }
80  }
81
82
83  /*************************************************************************
84  **************  I N G R E S S   P R O C E S S I N G   *******************
85  *************************************************************************/
86
87  control MyIngress(inout headers hdr,
88                    inout metadata meta,
89                    inout standard_metadata_t standard_metadata) {
90      action drop() {
91          mark_to_drop(standard_metadata);
```

```
92      }
93
94      action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
95          standard_metadata.egress_spec = port;
96          hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
97          hdr.ethernet.dstAddr = dstAddr;
98          hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
99      }
100
101     table ipv4_lpm {
102         key = {
103             hdr.ipv4.dstAddr: lpm;
104         }
105         actions = {
106             ipv4_forward;
107             drop;
108             NoAction;
109         }
110         size = 1024;
111         default_action = drop();
112     }
113
114     apply {
115         if (hdr.ipv4.isValid()) {
116             ipv4_lpm.apply();
117         }
118     }
119 }
120
121 /*************************************************************************
122 ***************  E G R E S S   P R O C E S S I N G   ******************
123 *************************************************************************/
124
125 control MyEgress(inout headers hdr,
126                  inout metadata meta,
127                  inout standard_metadata_t standard_metadata) {
128     apply {  }
129 }
130
131 /*************************************************************************
132 *************   C H E C K S U M   C O M P U T A T I O N   **************
133 *************************************************************************/
134
135 control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
136      apply {
137         update_checksum(
138         // If checksum is valid, update with the following fields
139             hdr.ipv4.isValid(),
140         { hdr.ipv4.version,
```

```
            hdr.ipv4.ihl,
            hdr.ipv4.diffserv,
            hdr.ipv4.totalLen,
            hdr.ipv4.identification,
            hdr.ipv4.flags,
            hdr.ipv4.fragOffset,
            hdr.ipv4.ttl,
            hdr.ipv4.protocol,
            hdr.ipv4.srcAddr,
            hdr.ipv4.dstAddr },
        // Update the checksum if the header is valid
        hdr.ipv4.hdrChecksum,
        // Update with the following algorithm
        HashAlgorithm.csum16);
    }
}

/*************************************************************************
**********************  D E P A R S E R  ******************************
*************************************************************************/

control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}

/*************************************************************************
**********************  S W I T C H  ******************************
*************************************************************************/

V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

A.4.2 Basic Forwarder with Timestamping

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

```

```
5  const bit<16> TYPE_IPV4 = 0x800;
6  const bit<8>  TYPE_TCP  = 6;
7
8  /*************************************************************************
9  ********************** H E A D E R S  ***********************************
10 *************************************************************************/
11
12 typedef bit<9>  egressSpec_t;
13 typedef bit<48> macAddr_t;
14 typedef bit<32> ip4Addr_t;
15
16 header ethernet_t {
17     macAddr_t dstAddr;
18     macAddr_t srcAddr;
19     bit<16>   etherType;
20 }
21
22 header ipv4_t {
23     bit<4>    version;
24     bit<4>    ihl;
25     bit<8>    diffserv;
26     bit<16>   totalLen;
27     bit<16>   identification;
28     bit<3>    flags;
29     bit<13>   fragOffset;
30     bit<8>    ttl;
31     bit<8>    protocol;
32     bit<16>   hdrChecksum;
33     ip4Addr_t srcAddr;
34     ip4Addr_t dstAddr;
35 }
36
37 header tcp_t {
38     bit<16> srcPort;
39     bit<16> dstPort;
40     bit<32> seqNo;
41     bit<32> ackNo;
42     bit<4>  dataOffset;
43     bit<3>  res;
44     bit<3>  ecn;
45     bit<6>  ctrl;
46     bit<16> window;
47     bit<16> checksum;
48     bit<16> urgentPtr;
49 }
50
51 struct metadata {
52     bit<32> flow_hash;
53     bit<48> flow_tstamp;
```

```
    bit<48> flow_tstamp_previous;
    bit<48> time_now;
    bit<48> time_diff;
    bit<10> microsecs;
    bit<10> millisecs;
    bit<8>  diff_repr;
    bit<4>  micro_hex;
    bit<4>  milli_hex;
    bit<8>  inc_diff_repr;
    bit<48> inc_time_diff;
    bit<16> inc_milli;
    bit<16> inc_micro;
    bit<48> jitter;
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
    tcp_t        tcp;
}

/*************************************************************************
********************** P A R S E R  ***********************************
*************************************************************************/

parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            TYPE_TCP: parse_tcp;
            default: accept;
        }
    }
```

```
103
104     state parse_tcp {
105         packet.extract(hdr.tcp);
106         transition accept;
107     }
108
109 }
110
111 /*************************************************************************
112 ************   C H E C K S U M    V E R I F I C A T I O N   *************
113 *************************************************************************/
114
115 control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
116     apply {  }
117 }
118
119
120 /*************************************************************************
121 **************  I N G R E S S   P R O C E S S I N G   *******************
122 *************************************************************************/
123
124 control MyIngress(inout headers hdr, inout metadata meta, inout
    ↪  standard_metadata_t standard_metadata) {
125     /* index: flow hash, value: last timestamp */
126     register<bit<48>>(8192) tstamp_register;
127
128     // Calculating time_diff
129     action calc_time_diff(){
130         // Use built-in metadata for timestamps, given in microseconds
131         meta.time_now = standard_metadata.ingress_global_timestamp;
132
133         // If timestamp of previous is zero, let time_diff be 0
134         if (meta.flow_tstamp == 0) {
135             meta.time_diff = 0;
136         } else {
137             // Else if not zero, assume current time is larger than previous
                ↪  and get difference
138             meta.time_diff = meta.time_now - meta.flow_tstamp;
139         }
140         // Set stored timestamp as previous and update to be current
            ↪  timestamp
141         meta.flow_tstamp_previous = meta.flow_tstamp;
142         meta.flow_tstamp = meta.time_now;
143
144         // Bit-slice to get bits containing milli and microseconds
145         // Causes some loss of information as time_diff increases
146         meta.millisecs = meta.time_diff[19:10];
147         meta.microsecs = meta.time_diff[9:0];
148
```

```
149         // All millisec-values <= 15 can be directly represented
150         // All millisec-values > 15 will be represented as 15, can be
            ↪ interpreted as significant time_diff
151         if (meta.millisecs <= 15) {
152             meta.milli_hex = (bit<4>) meta.millisecs;
153         } else {
154             meta.milli_hex = (bit<4>) 15;
155         }
156         // produces intervals of 64, starting at 0 and going up to 1000'ish
157         meta.micro_hex = (bit<4>) (meta.microsecs / 64);
158
159         // Produce a composite of each value
160         // To check: Slice and multiply microseconds with 64
161         // Lower bound: micro * 64
162         // Upper bound: (micro * 64) + 63
163         meta.diff_repr[7:4] = meta.milli_hex;
164         meta.diff_repr[3:0] = meta.micro_hex;
165
166         // Set the TOS/diffserv field to be the difference representation
167         hdr.ipv4.diffserv = meta.diff_repr;
168     }
169
170     action store_time_diffs() {
171         // reverse representation to get approximate time_diff from other
            ↪ switch
172         meta.inc_milli = (bit<16>) meta.inc_diff_repr[7:4];
173         meta.inc_micro = (bit<16>) meta.inc_diff_repr[3:0];
174         meta.inc_time_diff = (bit<48>) ((meta.inc_milli * 1000) +
            ↪ (meta.inc_micro * 64));
175
176         // Write crucial information to switch logs for further processing
177         log_msg("TIMEDIFFS: flow_hash={}, ip_src={}, time_diff={},
            ↪ inc_time_diff={}",
178                 {meta.flow_hash, hdr.ipv4.srcAddr, meta.time_diff,
                    ↪ meta.inc_time_diff}
179         );
180     }
181
182     action drop() {
183         mark_to_drop(standard_metadata);
184     }
185
186     // Basic forwarding logic
187     action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
188         standard_metadata.egress_spec = port;
189         hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
190         hdr.ethernet.dstAddr = dstAddr;
191         hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
192     }
```

```
193
194     table ipv4_lpm {
195         key = {
196             hdr.ipv4.dstAddr: lpm;
197         }
198         actions = {
199             ipv4_forward;
200             drop;
201             NoAction;
202         }
203         size = 1024;
204         default_action = drop();
205     }
206
207     apply {
208         if (hdr.ipv4.isValid()) {
209             ipv4_lpm.apply();
210         }
211         if (hdr.tcp.isValid()) {
212             @atomic {
213                 hash(meta.flow_hash,
214                     HashAlgorithm.crc16,
215                     (bit<32>)0,
216                     { hdr.ipv4.srcAddr, hdr.ipv4.dstAddr, hdr.ipv4.protocol,
                        ↪  hdr.tcp.srcPort, hdr.tcp.dstPort },
217                     (bit<32>) 8192);
218
219                 // Ensure import variables are 0 for any given packet
220                 meta.millisecs = 0;
221                 meta.microsecs = 0;
222                 meta.diff_repr = 0;
223                 meta.flow_tstamp = 0;                    // Explicitly set to
                    ↪  0 to ensure correct calculation
224                 meta.inc_diff_repr = hdr.ipv4.diffserv;
225
226                 // Read the previously stored timestamp
227                 tstamp_register.read(meta.flow_tstamp, (bit<32>)
                    ↪  meta.flow_hash);
228                 // Calculate time difference between previous and current
229                 calc_time_diff();
230                 // Store the current timestamp
231                 tstamp_register.write((bit<32>) meta.flow_hash,
                    ↪  meta.flow_tstamp);
232
233                 // If TOS/diffserv field is not 0, assume it's used for
                    ↪  time_diff
234                 if (meta.inc_diff_repr > 0) {
235                     store_time_diffs();
236                 }
```

```
            }
        }
    }
}

/*************************************************************************
***************  E G R E S S   P R O C E S S I N G   ******************
*************************************************************************/

control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {  }
}

/*************************************************************************
*************   C H E C K S U M    C O M P U T A T I O N   **************
*************************************************************************/

control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
     apply {
        update_checksum(
        // If checksum is valid, update with the following fields
            hdr.ipv4.isValid(),
        { hdr.ipv4.version,
          hdr.ipv4.ihl,
          hdr.ipv4.diffserv,
          hdr.ipv4.totalLen,
          hdr.ipv4.identification,
          hdr.ipv4.flags,
          hdr.ipv4.fragOffset,
          hdr.ipv4.ttl,
          hdr.ipv4.protocol,
          hdr.ipv4.srcAddr,
          hdr.ipv4.dstAddr },
        // Update the checksum if the header is valid
        hdr.ipv4.hdrChecksum,
        // Update with the following algorithm
        HashAlgorithm.csum16);
    }
}

/*************************************************************************
***********************  D E P A R S E R  *******************************
*************************************************************************/

control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
```

```
286         packet.emit(hdr.ipv4);
287         packet.emit(hdr.tcp);
288     }
289 }
290
291 /*************************************************************************
292 ***********************  S W I T C H  *******************************
293 *************************************************************************/
294
295 V1Switch(
296 MyParser(),
297 MyVerifyChecksum(),
298 MyIngress(),
299 MyEgress(),
300 MyComputeChecksum(),
301 MyDeparser()
302 ) main;
```

A.4.3 Basic Forwarder with Packet Loss Detection

```
1  /* -*- P4_16 -*- */
2  #include <core.p4>
3  #include <v1model.p4>
4
5  const bit<16> TYPE_IPV4 = 0x800;
6  const bit<8>  TYPE_TCP  = 6;
7
8  /*************************************************************************
9  *********************** H E A D E R S  ***********************************
10 *************************************************************************/
11
12 typedef bit<9>  egressSpec_t;
13 typedef bit<48> macAddr_t;
14 typedef bit<32> ip4Addr_t;
15
16 header ethernet_t {
17     macAddr_t dstAddr;
18     macAddr_t srcAddr;
19     bit<16>   etherType;
20 }
21
22 header ipv4_t {
23     bit<4>    version;
24     bit<4>    ihl;
25     bit<8>    diffserv; // TOS / DSCP
26     bit<16>   totalLen;
27     bit<16>   identification;
```

```
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

header tcp_t {
    bit<16> srcPort;
    bit<16> dstPort;
    bit<32> seqNo;
    bit<32> ackNo;
    bit<4>  dataOffset;
    bit<3>  res;
    bit<3>  ecn;
    bit<6>  ctrl;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}

struct metadata {
    bit<32> flow_hash;
    bit<48> flow_tstamp;
    bit<48> time_now;
    bit<48> time_diff;
    bit<8>  pcount;
    bit<8>  inc_pcount;
    bit<8>  pcount_diff;
    bit<8>  ploss_count;
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
    tcp_t        tcp;
}

/*************************************************************************
********************** P A R S E R  ***********************************
*************************************************************************/

parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

```

```
77      state start {
78          transition parse_ethernet;
79      }
80
81      state parse_ethernet {
82          packet.extract(hdr.ethernet);
83          transition select(hdr.ethernet.etherType) {
84              TYPE_IPV4: parse_ipv4;
85              default: accept;
86          }
87      }
88
89      state parse_ipv4 {
90          packet.extract(hdr.ipv4);
91          transition select(hdr.ipv4.protocol) {
92              TYPE_TCP: parse_tcp;
93              default: accept;
94          }
95      }
96
97      state parse_tcp {
98          packet.extract(hdr.tcp);
99          transition accept;
100     }
101
102 }
103
104 /*************************************************************************
105 ************   C H E C K S U M    V E R I F I C A T I O N   *************
106 *************************************************************************/
107
108 control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
109     apply {  }
110 }
111
112
113 /*************************************************************************
114 **************  I N G R E S S   P R O C E S S I N G   *******************
115 *************************************************************************/
116
117 control MyIngress(inout headers hdr, inout metadata meta, inout
    ↪  standard_metadata_t standard_metadata) {
118     /* index: flow_hash, value: first timestamp */
119     register<bit<48>>(8192) tstamp_register;
120     /* index: flow_hash, value: packet counter */
121     register<bit<8>>(8192) pcount_register;
122     /* index: flow_hash, value: ploss counter */
123     register<bit<8>>(8192) ploss_register;
124
```

```
125     action drop() {
126         mark_to_drop(standard_metadata);
127     }
128 
129     action check_time() {
130         // Checks if time since first packet in epoc is >10,000 (i.e. 10
            ↪  milliseconds)
131         meta.time_now = standard_metadata.ingress_global_timestamp;
132 
133         // If current timestamp is 10msec after first tstamp, reset and
            ↪  write packet_count to packet
134         meta.time_diff = meta.time_now - meta.flow_tstamp;
135         if (meta.time_diff > 10000) {
136             meta.flow_tstamp = meta.time_now;
137             hdr.ipv4.diffserv = meta.pcount;
138             meta.pcount = 0;
139         }
140     }
141 
142     action store_ploss() {
143         // Stores the difference between current packet counter and incoming
            ↪  packet counter
144         if (meta.ploss_count < meta.inc_pcount) {
145             meta.pcount_diff = meta.inc_pcount - meta.ploss_count;
146         }
147         log_msg("PLOSS: hash = {}, sAddr = {}, dAddr = {}, prot = {}, sPort
            ↪  = {}, dPort = {}, ploss_count = {}, inc_pcount = {}, pcount_diff
            ↪  = {}",
148                 {   meta.flow_hash, hdr.ipv4.srcAddr, hdr.ipv4.dstAddr,
149                     hdr.ipv4.protocol, hdr.tcp.srcPort, hdr.tcp.dstPort,
150                     meta.ploss_count, meta.inc_pcount, meta.pcount_diff
151                 }
152         );
153         // Reset ploss to 0 after receiving a packet count from another
            ↪  switch in given flow
154         meta.ploss_count = 0;
155     }
156 
157     action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
158         standard_metadata.egress_spec = port;
159         hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
160         hdr.ethernet.dstAddr = dstAddr;
161         hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
162     }
163 
164     table ipv4_lpm {
165         key = {
166             hdr.ipv4.dstAddr: lpm;
167         }
```

```
168         actions = {
169             ipv4_forward;
170             drop;
171             NoAction;
172         }
173         size = 1024;
174         default_action = drop();
175     }
176 
177     apply {
178         if (hdr.ipv4.isValid()) {
179             ipv4_lpm.apply();
180         }
181         if (hdr.tcp.isValid()) {
182             @atomic {
183                 // Hash the 5-tuple
184                 hash(meta.flow_hash,
185                     HashAlgorithm.crc16,
186                     (bit<32>)0,
187                     { hdr.ipv4.srcAddr, hdr.ipv4.dstAddr, hdr.ipv4.protocol,
                        ↪  hdr.tcp.srcPort, hdr.tcp.dstPort },
188                     (bit<32>) 8192);
189 
190                 // read timestamp, pcount, ploss and tcount from register
191                 tstamp_register.read(meta.flow_tstamp, (bit<32>)
                    ↪  meta.flow_hash);
192                 pcount_register.read(meta.pcount, (bit<32>) meta.flow_hash);
193                 ploss_register.read(meta.ploss_count, (bit<32>)
                    ↪  meta.flow_hash);
194 
195                 // increment pcount and ploss
196                 meta.pcount = meta.pcount + 1;
197                 meta.ploss_count = meta.ploss_count + 1;
198 
199                 // check mark, if mark seen, write ploss to log and reset
                    ↪  ploss
200                 meta.inc_pcount = hdr.ipv4.diffserv;
201                 if (meta.inc_pcount > 0) {
202                     store_ploss();
203                     // Reset TOS field
204                     hdr.ipv4.diffserv = 0;
205                 }
206 
207                 // check time, if end of epoch, mark packet with pcount and
                    ↪  reset pcount
208                 if (meta.flow_tstamp == 0) {
209                     tstamp_register.write((bit<32>) meta.flow_hash,
                        ↪  standard_metadata.ingress_global_timestamp);
210                 } else {
```

```
211                     // If not first packet in flow, check time
212                     check_time();
213                     tstamp_register.write((bit<32>) meta.flow_hash,
                        ↪  meta.flow_tstamp);
214                 }
215 
216                 // if ploss > 240, reset ploss to avoid overflows
217                 if (meta.ploss_count > 240) {
218                     meta.ploss_count = 0;
219                 }
220 
221                 // Write pcount, ploss and tcount to registers
222                 pcount_register.write((bit<32>) meta.flow_hash,
                    ↪  meta.pcount);
223                 ploss_register.write((bit<32>) meta.flow_hash,
                    ↪  meta.ploss_count);
224             }
225         }
226     }
227 }
228 
229 /*************************************************************************
230 ***************  E G R E S S   P R O C E S S I N G   *******************
231 *************************************************************************/
232 
233 control MyEgress(inout headers hdr,
234                  inout metadata meta,
235                  inout standard_metadata_t standard_metadata) {
236     apply {  }
237 }
238 
239 /*************************************************************************
240 *************   C H E C K S U M    C O M P U T A T I O N   **************
241 *************************************************************************/
242 
243 control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
244      apply {
245         update_checksum(
246         // If checksum is valid, update with the following fields
247             hdr.ipv4.isValid(),
248         { hdr.ipv4.version,
249           hdr.ipv4.ihl,
250           hdr.ipv4.diffserv,
251           hdr.ipv4.totalLen,
252           hdr.ipv4.identification,
253           hdr.ipv4.flags,
254           hdr.ipv4.fragOffset,
255           hdr.ipv4.ttl,
256           hdr.ipv4.protocol,
```

```
257             hdr.ipv4.srcAddr,
258             hdr.ipv4.dstAddr },
259         // Update the checksum if the header is valid
260         hdr.ipv4.hdrChecksum,
261         // Update with the following algorithm
262         HashAlgorithm.csum16);
263     }
264 }
265
266 /*************************************************************************
267 ***********************  D E P A R S E R  ********************************
268 *************************************************************************/
269
270 control MyDeparser(packet_out packet, in headers hdr) {
271     apply {
272         packet.emit(hdr.ethernet);
273         packet.emit(hdr.ipv4);
274         packet.emit(hdr.tcp);
275     }
276 }
277
278 /*************************************************************************
279 ***********************  S W I T C H  ************************************
280 *************************************************************************/
281
282 V1Switch(
283 MyParser(),
284 MyVerifyChecksum(),
285 MyIngress(),
286 MyEgress(),
287 MyComputeChecksum(),
288 MyDeparser()
289 ) main;
```

A.4.4 BMV2 Switch Commands

S2 Commands

```
1 table_add ipv4_lpm ipv4_forward 192.168.62.2/32 => 00:00:00:00:00:F2 1
2 table_add ipv4_lpm ipv4_forward 192.168.62.3/32 => 00:00:00:00:00:F3 2
3 table_add ipv4_lpm ipv4_forward 192.168.63.0/24 => 00:00:00:00:00:F3 2
4 table_add ipv4_lpm ipv4_forward 12.1.1.0/24 => 00:00:00:00:00:F2 1
```

S3 Commands

```
table_add ipv4_lpm ipv4_forward 192.168.63.2/32 => 00:00:00:00:00:D2 1
table_add ipv4_lpm ipv4_forward 192.168.63.3/32 => 00:00:00:00:00:D3 2
table_add ipv4_lpm ipv4_forward 192.168.62.0/24 => 00:00:00:00:00:D2 1
table_add ipv4_lpm ipv4_forward 12.1.1.0/24 => 00:00:00:00:00:D2 1
```

A.5 Iperf3 script and metric collection

A.5.1 Iperf3 command-line tool

```
#!/usr/bin/env python3
import argparse
import json
from statistics import mean
from time import time, sleep, strftime
from typing import Dict, Any

import iperf3

"""
Using a script to automate several tests with iperf3. Ten total runs, save output of each run
"""


def run_test(bind_addr: str =  127.0.0.1 ,
             srv_addr: str =  127.0.0.1 ,
             port: int = 5201,
             duration: int = 30,
             zerocopy: bool = False
             ) -> Dict[str, Any]:
    """
    Simple method to automate testing, returns a list of KPIs, i.e. Mbps, retransmits, time, CPU usage
    :param bind_addr: Local address used by client
    :param srv_addr: Server address
    :param port: Server address port
    :param duration: How long to perform test, default is 30s
    :param zerocopy: Use zerocopy to reduce CPU load, default is False
    :param run_no: ID for the current run, useful for later parsing of the results
    :return: Dict of KPIs
    """
    client = iperf3.Client()

    client.bind_address = bind_addr
```

```
34      client.server_hostname = srv_addr
35      client.port = port
36      client.duration = duration
37      client.zerocopy = zerocopy
38
39      time_start = time()
40      timestamp = strftime( %Y%m%d-%H%M%S )
41      result = client.run()
42      time_end = time()
43
44      total_time = time_end - time_start
45      to_return = {
46           timestamp : timestamp,
47           sent_mbps : result.sent_Mbps,
48           retransmits : result.retransmits,
49           cpu_load : result.local_cpu_total,
50           total_time : total_time
51      }
52
53      return to_return
54
55
56  def read_results(filename: str):
57      with open(filename,  r ) as f:
58          to_return = json.loads(f.read())
59      return to_return
60
61
62  def main():
63      parser = argparse.ArgumentParser(description= Perform several runs of
        ↪  Iperf3 )
64      parser.add_argument( -B ,  --bind_addr ,
65                           help= Client Address ,
66                           default= 127.0.0.1 ,
67                           type=str)
68      parser.add_argument( -s ,  --srv_addr ,
69                           help= Server Address ,
70                           default= 127.0.0.1 ,
71                           type=str)
72      parser.add_argument( -r ,  --runs ,
73                           help= Number of runs to perform ,
74                           default=1,
75                           type=int)
76      parser.add_argument( -P ,  --pause ,
77                           help= How long to pause between runs in seconds ,
78                           default=30,
79                           type=int)
80      parser.add_argument( -t ,  --duration ,
81                           help= How long to perform each run in seconds ,
```

```
                        default=5,
                        type=int)
    parser.add_argument( -Z ,  --zerocopy ,
                        help= Use zerocopy method, see Iperf3 docs. ,
                        action= store_true )
    parser.add_argument( -p ,  --port ,
                        help= Bind to specific port or default 5201 ,
                        default=5201,
                        type=int)
    parser.add_argument( -e ,  --suffix ,
                        help= Suffix to add to results file ,
                        default=  ,
                        type=str)
    parser.add_argument( -R ,  --read ,
                        help= Start script in read-mode, opens a file with
                        ↪  given filename using -F flag ,
                        action= store_true )
    parser.add_argument( -F ,  --filename ,
                        help= Specify filename to read ,
                        default=  ,
                        type=str)

    args = parser.parse_args()

    arg_runs = args.runs
    arg_pause = args.pause
    arg_bind = args.bind_addr
    arg_srv = args.srv_addr
    arg_port = args.port
    arg_duration = args.duration
    arg_zerocopy = args.zerocopy
    arg_suffix = args.suffix
    arg_read = args.read
    arg_filename = args.filename

    if arg_read:
        if not arg_filename:
            print( A filename must be provided if using the -R flag )
            exit(1)
        else:
            results = read_results(arg_filename)
            for item in results:
                print(item)
            exit(0)

    filename = f {strftime("%Y%m%d-%H%M")}-iperf-results
    if arg_suffix:
        filename = filename + f -{arg_suffix}
    filename = filename +  .json
```

```
130
131     print(f Will perform {arg_runs} runs with {arg_duration} seconds
        ↪  duration with
132           f  {arg_pause} seconds pause between each run and store results to
              ↪  {filename} )
133     results = []
134     for i in range(1, arg_runs + 1):
135         # Wrap with try-except block to ensure results are written to file
            ↪  even in failure
136         try:
137             print(f performing run {i} of {arg_runs} )
138             to_add = run_test(bind_addr=arg_bind,
139                               srv_addr=arg_srv,
140                               port=arg_port,
141                               duration=arg_duration,
142                               zerocopy=arg_zerocopy)
143             results.append(to_add)
144             if arg_runs > 1:
145                 sleep(arg_pause)
146         except Exception as err:
147             print(err)
148             break
149
150     with open(filename,  w ) as f:
151         f.write(json.dumps(results))
152
153
154 if __name__ ==  __main__ :
155     main()
```

A.5.2 Metric collection

```
1  #!/usr/bin/env bash
2
3  # This script collects CPU and Memory usage from the BMV2 switch instances
   ↪  and the Disk IO of the EPC VM
4  # and writes these to a CSV file on a 5 second interval until keyboard
   ↪  interrupt.
5  # Disk IO is the difference between each 5 second measurement.
6
7  csv_filename="$(date +"%Y%m%d-%H%M")-metrics.csv"
8  touch $csv_filename
9  echo "timestamp,pid1,cpu1,mem1,pid2,cpu2,mem2,kb_wrtn" >> $csv_filename
10
11 # First time reading Disk IO.
12 disk_io_previous=$(iostat -d sda | tail -n 2 | xargs | awk  {print $6} )
13
```

```
# Until keyboard interrupt
while true
do
  # Current time
  timestamp=$(date +"%Y%m%d-%H%M%S")

  # CPU and Memory usage of BMV2 switch instances
  cpu_mem_usage=$(pgrep simple_switch | xargs -I % top -b -n 1 -p % | grep
  ↪  simple_switch | awk  {print $1 "," $9 "," $10 ","}  | tr -d  \n  | sed
  ↪   s/.$// )

  # Current Disk IO
  disk_io_current=$(iostat -d sda | tail -n 2 | xargs | awk  {print $6} )

  # Difference since last measurement
  disk_io_diff=$(($disk_io_current-$disk_io_previous))

  # Write timestamp and metrics to CSV file
  echo "$timestamp,$cpu_mem_usage,$disk_io_diff" >> $csv_filename

  # Set current Disk IO measurement as previous
  disk_io_previous=$disk_io_current

  sleep 5
done
```

A.6 Data Analysis Scripts

Here is a collection of the data analysis scripts used to parse and analyze the different experiment results. The data set is available per request to the author.

A.6.1 Comparing baseline, INT, Active, and Passive results

```
import pandas as pd

from utils import save_boxplot, interleave_lists

nm_bw1 =
↪  pd.read_json( data/baseline/no_monitoring/20210417-0927-iperf-results-metric-collection-no-monitorin
nm_bw2 =
↪  pd.read_json( data/baseline/no_monitoring/20210417-0944-iperf-results-metric-collection-no-monitorin
nm_d_bw1 = pd.read_json( data/with_delay/no_monitoring/

                        ↪   20210414-1237-iperf-results-metric-collection-no-monitoring-with-delay.jso
nm_d_bw2 = pd.read_json( data/with_delay/no_monitoring/
```

```
                        ↪  20210414-1256-iperf-results-metric-collection-no-monitoring-with-delay

hfp_bw1 = ↪  pd.read_json( data/baseline/high_freq_ping/20210413-1650-iperf-results-metric-collection-hfp.jso
hfp_bw2 = ↪  pd.read_json( data/baseline/high_freq_ping/20210413-1707-iperf-results-metric-collection-hfp.jso
hfp_d_bw1 = ↪  pd.read_json( data/with_delay/hfp/20210414-1409-iperf-results-metric-collection-hfp-delay.json )
hfp_d_bw2 = ↪  pd.read_json( data/with_delay/hfp/20210414-1432-iperf-results-metric-collection-hfp-delay.json )

pcap_bw1 = ↪  pd.read_json( data/baseline/pcaps/20210414-1325-iperf-results-metric-collection-pcap.json )
pcap_bw2 = ↪  pd.read_json( data/baseline/pcaps/20210414-1344-iperf-results-metric-collection-pcap.json )
pcap_d_bw1 = ↪  pd.read_json( data/with_delay/pcaps/20210414-1505-iperf-results-metric-collection-pcaps-delay.js
pcap_d_bw2 = ↪  pd.read_json( data/with_delay/pcaps/20210414-1525-iperf-results-metric-collection-pcaps-delay.js

int_bw1 = ↪  pd.read_json( data/baseline/int/20210413-1829-iperf-results-metric-collection-int.json )
int_bw2 = ↪  pd.read_json( data/baseline/int/20210413-1851-iperf-results-metric-collection-int.json )
int_d_bw1 = ↪  pd.read_json( data/with_delay/int/20210415-0927-iperf-results-metric-collection-int-delay.json )
int_d_bw2 = ↪  pd.read_json( data/with_delay/int/20210415-0952-iperf-results-metric-collection-int-delay.json )

nm_metrics1 = ↪  pd.read_csv( data/baseline/no_monitoring/20210417-0927-metrics.csv )
nm_metrics2 = ↪  pd.read_csv( data/baseline/no_monitoring/20210417-0944-metrics.csv )
nm_d_metrics1 = ↪  pd.read_csv( data/with_delay/no_monitoring/20210414-1237-metrics.csv )
nm_d_metrics2 = ↪  pd.read_csv( data/with_delay/no_monitoring/20210414-1256-metrics.csv )

hfp_metrics1 = ↪  pd.read_csv( data/baseline/high_freq_ping/20210413-1652-metrics.csv )
hfp_metrics2 = ↪  pd.read_csv( data/baseline/high_freq_ping/20210413-1709-metrics.csv )
hfp_d_metrics1 = ↪  pd.read_csv( data/with_delay/hfp/20210414-1409-metrics.csv )
hfp_d_metrics2 = ↪  pd.read_csv( data/with_delay/hfp/20210414-1432-metrics.csv )

pcap_metrics1 = pd.read_csv( data/baseline/pcaps/20210414-1325-metrics.csv )
```

```
pcap_metrics2 = pd.read_csv( data/baseline/pcaps/20210414-1344-metrics.csv )
pcap_d_metrics1 =
↪  pd.read_csv( data/with_delay/pcaps/20210414-1505-metrics.csv )
pcap_d_metrics2 =
↪  pd.read_csv( data/with_delay/pcaps/20210414-1525-metrics.csv )

int_metrics1 = pd.read_csv( data/baseline/int/20210413-1831-metrics.csv )
int_metrics2 = pd.read_csv( data/baseline/int/20210413-1853-metrics.csv )
int_d_metrics1 =
↪  pd.read_csv( data/with_delay/int/20210415-0927-metrics.csv )
int_d_metrics2 =
↪  pd.read_csv( data/with_delay/int/20210415-0952-metrics.csv )

# Combine to a single frame
nm_frames = [nm_bw1, nm_bw2]
nm_d_frames = [nm_d_bw1, nm_d_bw2]

hfp_frames = [hfp_bw1, hfp_bw2]
hfp_d_frames = [hfp_d_bw1, hfp_d_bw2]

pcap_frames = [pcap_bw1, pcap_bw2]
pcap_d_frames = [pcap_d_bw1, pcap_d_bw2]

int_frames = [int_bw1, int_bw2]
int_d_frames = [int_d_bw1, int_d_bw2]

nm_metric_frames = [nm_metrics1, nm_metrics2]
nm_metric_d_frames = [nm_d_metrics1, nm_d_metrics2]

hfp_metric_frames = [hfp_metrics1, hfp_metrics2]
hfp_metric_d_frames = [hfp_d_metrics1, hfp_d_metrics2]

pcap_metric_frames = [pcap_metrics1, pcap_metrics2]
pcap_metric_d_frames = [pcap_d_metrics1, pcap_d_metrics2]

int_metric_frames = [int_metrics1, int_metrics2]
int_metric_d_frames = [int_d_metrics1, int_d_metrics2]

nm_result = pd.concat(nm_frames, ignore_index=True)
nm_d_result = pd.concat(nm_d_frames, ignore_index=True)

hfp_result = pd.concat(hfp_frames, ignore_index=True)
hfp_d_result = pd.concat(hfp_d_frames, ignore_index=True)

pcap_result = pd.concat(pcap_frames, ignore_index=True)
pcap_d_result = pd.concat(pcap_d_frames, ignore_index=True)

int_result = pd.concat(int_frames, ignore_index=True)
int_d_result = pd.concat(int_d_frames, ignore_index=True)
```

```
83
84  nm_metric_result = pd.concat(nm_metric_frames, ignore_index=True)
85  nm_metric_d_result = pd.concat(nm_metric_d_frames, ignore_index=True)
86
87  hfp_metric_result = pd.concat(hfp_metric_frames, ignore_index=True)
88  hfp_metric_d_result = pd.concat(hfp_metric_d_frames, ignore_index=True)
89
90  pcap_metric_result = pd.concat(pcap_metric_frames, ignore_index=True)
91  pcap_metric_d_result = pd.concat(pcap_metric_d_frames, ignore_index=True)
92
93  int_metric_result = pd.concat(int_metric_frames, ignore_index=True)
94  int_metric_d_result = pd.concat(int_metric_d_frames, ignore_index=True)
95
96  # Filter out results with unexplainably low bandwidth (edge cases)
97  nm_result = nm_result[nm_result[ sent_mbps ] > 2]
98  nm_d_result = nm_d_result[nm_d_result[ sent_mbps ] > 2]
99
100 hfp_result = hfp_result[hfp_result[ sent_mbps ] > 2]
101 hfp_d_result = hfp_d_result[hfp_d_result[ sent_mbps ] > 2]
102
103 pcap_result = pcap_result[pcap_result[ sent_mbps ] > 2]
104 pcap_d_result = pcap_d_result[pcap_d_result[ sent_mbps ] > 2]
105
106 int_result = int_result[int_result[ sent_mbps ] > 2]
107 int_d_result = int_d_result[int_d_result[ sent_mbps ] > 2]
108
109 # Filter out results with less than 40% CPU utilization, only keep periods
    ↪  where Iperf3 is running
110 nm_metric_result = nm_metric_result[nm_metric_result[ cpu1 ] > 40]
111 nm_metric_d_result = nm_metric_d_result[nm_metric_d_result[ cpu1 ] > 40]
112
113 hfp_metric_result = hfp_metric_result[hfp_metric_result[ cpu1 ] > 40]
114 hfp_metric_d_result = hfp_metric_d_result[hfp_metric_d_result[ cpu1 ] > 40]
115
116 pcap_metric_result = pcap_metric_result[pcap_metric_result[ cpu1 ] > 40]
117 pcap_metric_d_result = pcap_metric_d_result[pcap_metric_d_result[ cpu1 ] >
    ↪  40]
118
119 int_metric_result = int_metric_result[int_metric_result[ cpu1 ] > 40]
120 int_metric_d_result = int_metric_d_result[int_metric_d_result[ cpu1 ] > 40]
121
122 # Merge related frames together
123 baseline_bw = [
124     nm_result[ sent_mbps ], hfp_result[ sent_mbps ],
125     pcap_result[ sent_mbps ], int_result[ sent_mbps ]
126 ]
127 baseline_retr = [
128     nm_result[ retransmits ], hfp_result[ retransmits ],
129     pcap_result[ retransmits ], int_result[ retransmits ]
```

```
]
baseline_cpu = [
    nm_metric_result[ cpu1 ], hfp_metric_result[ cpu1 ],
    pcap_metric_result[ cpu1 ], int_metric_result[ cpu1 ]
]
baseline_mem = [
    nm_metric_result[ mem1 ], hfp_metric_result[ mem1 ],
    pcap_metric_result[ mem1 ], int_metric_result[ mem1 ]
]
baseline_disk = [
    nm_metric_result[ kb_wrtn ], hfp_metric_result[ kb_wrtn ],
    pcap_metric_result[ kb_wrtn ], int_metric_result[ kb_wrtn ]
]
baseline_time = [
    nm_result[ total_time ], hfp_result[ total_time ],
    pcap_result[ total_time ], int_result[ total_time ]
]
delay_bw = [
    nm_d_result[ sent_mbps ], hfp_d_result[ sent_mbps ],
    pcap_d_result[ sent_mbps ], int_d_result[ sent_mbps ]
]
delay_retr = [
    nm_d_result[ retransmits ], hfp_d_result[ retransmits ],
    pcap_d_result[ retransmits ], int_d_result[ retransmits ]
]
delay_cpu = [
    nm_metric_d_result[ cpu1 ], hfp_metric_d_result[ cpu1 ],
    pcap_metric_d_result[ cpu1 ], int_metric_d_result[ cpu1 ]
]
delay_mem = [
    nm_metric_d_result[ mem1 ], hfp_metric_d_result[ mem1 ],
    pcap_metric_d_result[ mem1 ], int_metric_d_result[ mem1 ]
]
delay_disk = [
    nm_metric_d_result[ kb_wrtn ], hfp_metric_d_result[ kb_wrtn ],
    pcap_metric_d_result[ kb_wrtn ], int_metric_d_result[ kb_wrtn ]
]
delay_time = [
    nm_d_result[ total_time ], hfp_d_result[ total_time ],
    pcap_d_result[ total_time ], int_d_result[ total_time ]
]

comparison_nm_bw = [
    nm_result[ sent_mbps ], nm_d_result[ sent_mbps ]
]
comparison_hfp_bw = [
    hfp_result[ sent_mbps ], hfp_d_result[ sent_mbps ]
]

```

```
comparison_pcap_bw = [
    pcap_result[ sent_mbps ], pcap_d_result[ sent_mbps ]
]

comparison_int_bw = [
    int_result[ sent_mbps ], int_d_result[ sent_mbps ]
]

# Create interleaved lists of results with and without delay
comparison_all_bw = interleave_lists(baseline_bw, delay_bw)
comparison_retr = interleave_lists(baseline_retr, delay_retr)
comparison_cpu = interleave_lists(baseline_cpu, delay_cpu)
comparison_mem = interleave_lists(baseline_mem, delay_mem)
comparison_disk = interleave_lists(baseline_disk, delay_disk)
comparison_time = interleave_lists(baseline_time, delay_time)

# Define content of figures

labels = [ no_monitoring ,  HFP ,  PCAPs ,  INT ]
baseline_figs = [
    [baseline_bw,  Baseline Bandwidth ,  Mb/s ,
    ↪   figures/no_delay/baseline_bw ],
    [baseline_retr,  Baseline Retransmissions ,  Retransmissions ,
    ↪   figures/no_delay/baseline_retr ],
    [baseline_cpu,  Baseline CPU Usage of BMV2 Switches ,  CPU % ,
    ↪   figures/no_delay/baseline_cpu ],
    [baseline_mem,  Baseline Memory Usage of BMV2 Switches ,  Mem % ,
    ↪   figures/no_delay/baseline_mem ],
    [baseline_disk,  Baseline Disk I/O ,  KBytes/5s ,
    ↪   figures/no_delay/baseline_disk ],
    [baseline_time,  Baseline Time Usage, Iperf3 Client ,  Seconds ,
    ↪   figures/no_delay/baseline_time ]
]

# Draw and save figures
for item in baseline_figs:
    save_boxplot(data=item[0], fig_labels=labels, title=item[1],
    ↪  ylabel=item[2], fig_name=item[3])

delay_figs = [
    [delay_bw,  Delay Bandwidth ,  Mb/s ,  figures/with_delay/delay_bw ],
    [delay_retr,  Delay Retransmissions ,  Retransmissions ,
    ↪   figures/with_delay/delay_retr ],
    [delay_cpu,  Delay CPU Usage of BMV2 Switches ,  CPU % ,
    ↪   figures/with_delay/delay_cpu ],
    [delay_mem,  Delay Memory Usage of BMV2 Switches ,  Mem % ,
    ↪   figures/with_delay/delay_mem ],
    [delay_disk,  Delay Disk I/O ,  KBytes/5s ,
    ↪   figures/with_delay/delay_disk ],
```

```
217     [delay_time,  Delay Time Usage, Iperf3 Client ,  Seconds ,
        ↪  figures/with_delay/delay_time ]
218 ]
219
220 # Draw and save figures
221 for item in delay_figs:
222     save_boxplot(data=item[0], fig_labels=labels, title=item[1],
        ↪  ylabel=item[2], fig_name=item[3])
223
224 # Prepare figure data for comparisons
225 labels = [ Without Delay ,  With Delay ]
226 to_compare = [
227     [
228         comparison_nm_bw,
229          Bandwidth, with and without Delay, No Monitoring ,
230          Mb/s ,
231          figures/comparisons/comp_bw_nm
232     ],
233     [
234         comparison_hfp_bw,
235          Bandwidth, with and without Delay, High Frequency Ping ,
236          Mb/s ,
237          figures/comparisons/comp_bw_hfp
238     ],
239     [
240         comparison_pcap_bw,
241          Bandwidth, with and without Delay, Packet Capturing ,
242          Mb/s ,
243          figures/comparisons/comp_bw_pcap
244     ],
245     [
246         comparison_int_bw,
247          Bandwidth, with and without Delay, In-band Network Telemetry ,
248          Mb/s ,
249          figures/comparisons/comp_bw_int
250     ],
251 ]
252 for item in to_compare:
253     save_boxplot(data=item[0], fig_labels=labels, title=item[1],
        ↪  ylabel=item[2], fig_name=item[3])
254
255 labels = [ nm ,  nm_d ,  hfp ,  hfp_d ,  pcap ,  pcap_d ,  int ,  int_d ]
256 to_compare = [
257     [
258         comparison_all_bw,
259          Bandwidth, With and Without Delay, all methods ,
260          Mb/s ,
261          figures/comparisons/comp_bw_all
262     ],
```

```
    [
        comparison_retr,
         Retransmissions, With and Without Delay, all methods ,
          ,
         figures/comparisons/comp_retr_all
    ],
    [
        comparison_cpu,
         CPU Usage of BMV2 Switches, With and Without Delay, all methods ,
         CPU % ,
         figures/comparisons/comp_cpu_all
    ],
    [
        comparison_mem,
         Memory Usage of BMV2 Switches, With and Without Delay, all
        ↪  methods ,
         Mem % ,
         figures/comparisons/comp_mem_all
    ],
    [
        comparison_disk,
         Disk Usage of EPC VM, With and Without Delay, all methods ,
         KBytes/5sec ,
         figures/comparisons/comp_disk_all
    ],
    [
        comparison_time,
         Iperf3 Time usage, With and Without Delay, all methods ,
         Seconds ,
         figures/comparisons/comp_time_all
    ]
]
for item in to_compare:
    save_boxplot(data=item[0], fig_labels=labels, title=item[1],
    ↪  ylabel=item[2], fig_name=item[3])
```

A.6.2 Delay Detection Analysis

Parsing

```
def parse_timediffs(filename):
    to_return = []
    with open(filename,  r ) as f:
        lines = f.readlines()
        for line in lines:
            first_half = line.split( TIMEDIFFS: )[0]
            second_half = line.split( TIMEDIFFS: )[1]
            to_add = {
```

```
                timestamp : first_half.split(   )[0].strip( [ ).strip( ] ),
                flow_hash : second_half.split( , )[0].split( = )[1],
                ip_src : second_half.split( , )[1].split( = )[1],
                time_diff : second_half.split( , )[2].split( = )[1],
                inc_time_diff :
                    second_half.split( , )[3].split( = )[1].strip()
            }
            to_return.append(to_add)
    return to_return


def dec_to_ip(decimal_representation):
    hex_repr = hex(int(decimal_representation))
    hex_repr = hex_repr[2:]
    # pad with extra 0 if hex-representation is less than 8 (will only
    ↪  happen in cases where first octet is <15)
    if len(hex_repr) < 8:
        hex_repr =   .join(( 0 , hex_repr))
    ip_addr =  . .join([str(int(hex_repr[i:i+2], 16)) for i in range(0,
    ↪  len(hex_repr), 2)])
    return ip_addr


s2_timediffs = parse_timediffs( data/baseline/int/s2_timediffs.txt )
s3_timediffs = parse_timediffs( data/baseline/int/s3_timediffs.txt )

# for item in s2_timediffs:
#     item['ip_src'] = dec_to_ip(item['ip_src'])
#
# for item in s3_timediffs:
#     item['ip_src'] = dec_to_ip(item['ip_src'])
#
for item in s2_timediffs[:10]:
    print(item[ ip_src ], dec_to_ip(item[ ip_src ]))
for item in s3_timediffs[:10]:
    print(item[ ip_src ], dec_to_ip(item[ ip_src ]))
```

Analysis

```
import pandas as pd
import numpy as np

from datetime import datetime
from scipy import stats

from utils import save_scatter_with_line_regr

```

```
9
10  print(datetime.now(),  Starting parsing )
11  # These are more than 580,000 lines each, be patient
12  s2_timediffs =
    ↪  pd.read_json( data/baseline/int/s2_timediffs.json ).sample(50000)
13  s3_timediffs =
    ↪  pd.read_json( data/baseline/int/s3_timediffs.json ).sample(50000)
14  print(datetime.now(),  Parsed INT baseline, 100000 samples )
15
16  s2_timediffs_d =
    ↪  pd.read_json( data/with_delay/int/s2_timediffs.json ).sample(50000)
17  s3_timediffs_d =
    ↪  pd.read_json( data/with_delay/int/s3_timediffs.json ).sample(50000)
18  print(datetime.now(),  Parsed INT with delay, 100000 samples )
19
20  s2_timediffs_ed =
    ↪  pd.read_json( data/with_delay/int_epc_delay/s2_timediffs.json ).sample(50000)
21  s3_timediffs_ed =
    ↪  pd.read_json( data/with_delay/int_epc_delay/s3_timediffs.json ).sample(50000)
22  print(datetime.now(),  Parsed INT with delay at EPC, 100000 samples )
23
24  items = [
25      ( s2 , s2_timediffs), ( s3 , s3_timediffs),
26      ( s2_d , s2_timediffs_d), ( s3_d , s3_timediffs_d),
27      ( s2_ed , s2_timediffs_ed), ( s3_ed , s3_timediffs_ed)
28  ]
29
30  print(datetime.now(),  Starting to draw and save figures )
31  for item in items:
32      name = item[0]
33      df = item[1]
34
35      # Replace all values in X that are larger than 15960 to ensure similar
        ↪  scaling
36      a = np.array(df[ time_diff ].values.tolist())
37      df[ time_diff ] = np.where(a > 15960, 15960, a).tolist()
38      x = df[ time_diff ]
39      y = df[ inc_time_diff ]
40
41      slope, intercept, r_value, p_value, std_err = stats.linregress(x, y)
42      eq_label = f {round(slope, 3)}x + {round(intercept, 3)}
43      title = f Time Diff vs Inc Time Diff, {name},   \
44              +  $r^{2}=  + str(round(r_value, 3)) +  $
45      # Draw and save figures
46      # save_histogram(x, y, 30, title, 'Millisecs',
        ↪  f'figures/int/hists/{name}_td_v_itd_hist')
47      save_scatter_with_line_regr(x, y, slope, intercept, title,
        ↪   inc_time_diff ,  time_diff , eq_label,
```

```
                                    ↪  f figures/testing/{name}_td_v_itd_scatter_lineregr_rsqrd )
    print(datetime.now(), f --- Drawn scatterplott of {name} )

print(datetime.now(),  Finished parsing and drawing )
```

A.6.3 Packet Loss Analysis

Parsing

```
import json


def parse_ploss(filename):
    to_return = []

    with open(filename,  r ) as f:
        lines = f.readlines()
        for line in lines:
            # Split line on PLOSS, select last item and split again on ','
            row = line.split( PLOSS: )[1].split( , )
            to_add = {}
            # Go through each item in each row
            for item in row:
                # Split each item on = to get a list of two elements
                item = item.split( = )
                # Set first element as key, second element as value
                key, val = item[0].strip(), item[1].strip()
                # Add to dictionary
                to_add[key] = val
            # Append the row as a dictionary to the final list of
            ↪  dictionaries
            to_return.append(to_add)

    return to_return


def as_json_file(filename, obj):
    with open(filename,  w ) as file:
        file.write(json.dumps(obj))


filenames = [
     data/ploss/baseline/s2_loss ,
     data/ploss/baseline/s3_loss ,
     data/ploss/with_loss/s2_loss ,
     data/ploss/with_loss/s3_loss ,
]
```

```

for name in filenames:
    item = parse_ploss(name +  .txt )
    as_json_file(name +  .json , item)
```

Analysis

```
from datetime import datetime

import pandas as pd
import numpy as np

from utils import dec_to_ip

names = [
     data/ploss/baseline/s2_loss.json ,
     data/ploss/baseline/s3_loss.json ,
     data/ploss/with_loss/s2_loss.json ,
     data/ploss/with_loss/s3_loss.json ,
]
print(datetime.now(),  Starting parsing )
items = []
for name in names:
    item = pd.read_json(name)
    switch_name = name.split( / )[-1].split( _ )[0]
    case = name.split( / )[2]
    items.append((item, switch_name, case))

print(datetime.now(),  Finished parsing )

# Transform integer notated IP address to dotted decimal form
for item in items:
    for key in [ sAddr ,  dAddr ]:
        item[0][key] = item[0][key].apply(lambda x: dec_to_ip(x))

# Create new aggregated dataframes with the most important information about
↪  the results
results = []
for item in items:
    result = item[0].groupby( hash ).agg(
        total_pcount=( ploss_count , sum),
        total_loss=( pcount_diff , sum),
        sAddr=( sAddr , set),
        dAddr=( dAddr , set),
        sPort=( sPort , set),
        dPort=( dPort , set),
        prot=( prot , set)
```

```
    )
    result[ loss_percent ] = (result[ total_loss ] / result[ total_pcount ])
    ↪  * 100
    result[ switch ] = item[1]
    result[ case ] = item[2]
    result = result[result[ total_pcount ] > 1000]
    results.append(result)

pd.set_option( display.max_columns , None)
pd.set_option( display.max_colwidth , None)
pd.set_option( display.width , 2000)
for item in results:
    # Extract all possible hash collisions, i.e. 2 or more elements in sPort
    ↪  or dPort
    dport_collisions = item.loc[item[ dPort ].apply(lambda x: len(list(x)) >
    ↪  1)]
    sport_collisions = item.loc[item[ sPort ].apply(lambda x: len(list(x)) >
    ↪  1)]
    hash_collisions = pd.concat([dport_collisions, sport_collisions],
    ↪  ignore_index=True)
    # Clean up the final tables to be printed, i.e. all sets are converted
    ↪  to comma-separated values in a string
    to_clean = [ sAddr ,  dAddr ,  sPort ,  dPort ,  prot ]
    for column in to_clean:
        item[column] = item[column].apply(lambda x:  , .join(str(i) for i in
        ↪  x))

    print(item.sample(5))
    print( mean loss: , round(item[ loss_percent ].mean(), 3))
    if not hash_collisions.empty:
        print( hash collisions: )
        print(hash_collisions)
    print()
```

A.6.4 Supporting scripts

Utils

```
# import matplotlib
import matplotlib.pyplot as plt

# Use SVG as default renderer
# matplotlib.use('png')


def save_boxplot(data, fig_labels, title, ylabel, fig_name):
    """
```

```
    Draws and saves figure with given data, labels, title, and given figure
    name
    :param data: Dataframe to draw
    :param fig_labels: Labels along the X axis, representing each column
    drawn
    :param title: Title of the figure
    :param ylabel: Scale, metric, or similar
    :param fig_name: Name to save the figure as on disk
    :return: None
    """
    fig = plt.figure(figsize=(10, 10))
    plt.boxplot(data, labels=fig_labels)
    plt.title(title)
    plt.grid()
    plt.ylabel(ylabel)
    plt.savefig(fig_name)
    plt.close(fig)


def save_scatterplot(x, y, title, ylabel, xlabel, fig_name):
    """
    Draws and saves a scatterplot with given parameters
    :param x: Data to plot in x
    :param y: Data to plot in y
    :param title: Title of the figure
    :param ylabel: Y label
    :param xlabel: X label
    :param fig_name: Name to save as
    :return: None
    """
    fig = plt.figure(figsize=(10, 10))
    plt.scatter(x, y)
    plt.title(title)
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)
    plt.savefig(fig_name)
    plt.close(fig)


def save_scatter_with_line_regr(x, y, m, b, title, ylabel, xlabel, eq_label,
    fig_name):
    """
    Takes x, y and draws scatter plot with line regression
    :param x: Data to plot in x
    :param y: Data to plot in y
    :param m: Slope
    :param b: Intercept
    :param title: Title of the figure
    :param ylabel: Y label
```

```
56      :param xlabel: X label
57      :param eq_label: Equation of fit
58      :param fig_name: Name to save as
59      :return: None
60      """
61      fig = plt.figure(figsize=(5, 5))
62      plt.plot(x, y,  o )
63      plt.plot(x, m*x + b, label=eq_label)
64      plt.xlabel(xlabel)
65      plt.ylabel(ylabel)
66      plt.title(title)
67      plt.grid(True)
68      plt.legend(fontsize= small )
69      plt.savefig(fig_name)
70      plt.close(fig)
71
72
73  def save_histogram(x, y, bins, title, xlabel, fig_name):
74      """
75      Draws two histograms side by side
76      :param x: Data for first
77      :param y: Data for second
78      :param bins: N bins
79      :param title: Title of the histograms
80      :param xlabel: Label of X
81      :param fig_name: Name to save figure as on disk
82      :return: None
83      """
84      fig, axs = plt.subplots(1, 2, sharey=True, sharex=True,
        ↪  tight_layout=True)
85      axs[0].hist(x, bins=bins)
86      axs[1].hist(y, bins=bins)
87      axs[0].set_title( time_diff )
88      axs[0].set_xlabel(xlabel)
89      axs[1].set_title( inc_time_diff )
90      axs[1].set_xlabel(xlabel)
91      plt.savefig(fig_name)
92      plt.close(fig)
93
94
95  def interleave_lists(l1, l2):
96      """
97      Will interleave two lists of the same length.
98      If l1 = [a, b, c], l2 = [1, 2, 3]
99      result = [a, 1, b, 2, c, 3]
100     :param l1: List with arbitrary elements
101     :param l2: List with arbitrary elements and equal length as l1
102     :return: Interleaved list of l1 + l2
103     """
```

```
104     return [val for pair in zip(l1, l2) for val in pair]
105
106
107 def dec_to_ip(decimal_representation):
108     """
109     Takes IP represented by integers (which are a result of hex-values) and
    ↪  returns the dotted decimal form
110     :param decimal_representation: IP as integer, i.e. '3232251651' or
    ↪  '3232251394'
111     :return: Dotted decimal form
112     """
113     hex_repr = hex(int(decimal_representation))
114     hex_repr = hex_repr[2:]
115     # pad with extra 0 if hex-representation is less than 8 (will only
        ↪  happen in cases where first octet is <15)
116     if len(hex_repr) < 8:
117         hex_repr =   .join(( 0 , hex_repr))
118     ip_addr =  . .join([str(int(hex_repr[i:i + 2], 16)) for i in range(0,
        ↪  len(hex_repr), 2)])
119     return ip_addr
```