CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Real Time 5G Simulator

Master's thesis in Computer science and engineering

ADI HRUSTIC

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022

Master's thesis 2022

Real Time 5G Simulator

ADI HRUSTIC

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2022

Real Time 5G Simulator

ADI HRUSTIC

Advisor: Olof Düsterdieck, Ericsson
Advisor: Giovanni Viola, Ericsson
Examiner: Marina Papatriantafilou, Networks and Systems
Supervisor: Romaric Duvignau, Networks and Systems

Master's Thesis 2022
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Real Time 5G Simulator

Adi Hrustic
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

As we shift towards the introduction of 5G in the world, it is indisputable that tools for continuously testing such an environment are needed in order to ensure its success. Ericsson wants to become one of the leaders of 5G solutions with their Evolved Packet Gateway (EPG), which acts as a gateway between a radio network and the Internet. A core component of the EPG is the User Plane (UP), who is responsible for inspecting and transferring payload data of network packets, one of the process-heaviest parts of the entire communication. To test the efficacy of the UP, Ericsson has several simulators at their disposal depending on the intended goal. *Ericload* is one such simulator that can send bidirectional data to the UP as a means to load test it. Currently, Ericload is able to generate model-based traffic, i.e., traffic calculated using mathematical models. While this approach is sensible, it does not entirely reflect the common properties found in real world traffic, such as self-similarity and long-range dependency. Trace-based modelling is a second approach of traffic generation, whose goal is to introduce the real world traffic properties into a simulation by replaying already captured network traces back to networks. The purpose of this thesis is to extend Ericload and implement scalable trace-based traffic modelling and load testing, as well analyze how this adage compares to model-based load testing. The analysis was done using metrics such as packet and data throughput, latency, as well as several verification approaches to ensure a correct trace-based simulation. The results conclude that the implementation of a basic trace-based traffic modelling was successful, and performs load tests well exclusively by itself or together with model-based traffic, without causing any severe performance issues. The results also shows, theoretically, how scaling captured traces affects its self-similar properties, and how to take these effects into consideration when making any further future improvements.

Keywords: packet capture, pcap, 5g, simulator, replay, trace-based, load testing.

Acknowledgements

Thanks to Olof and Giovanni at Ericsson, Romaric and Marina at Chalmers, for their time and contribution to this thesis. Special thanks to my Mother and Father for granting me my life without my permission.

Adi Hrustic, Gothenburg, July 2022

Contents

List of Figures

List of Tables

1

Introduction

1.1 Introduction

Going through the history of the past century, our increasing dependency for telecommunication seems almost self-evident. Since the discovery of radio waves and their first use in wireless communication, the advances in telecommunications technology have progressed so fast, that we are only a few years away from stable interconnectivity of any electronic device capable of connecting to the Internet, however small and redundant. Today, 5G technology is the standard promising such a solution. With its increased bandwidth and speed compared to its predecessor 4G, it is expected to be a serious competitor to existing solutions relying on wired connections.

The company Ericsson has taken upon itself to become one of the leading distributors of these 5G solutions. One of the products in Ericsson's portfolio is the Evolved Packet Gateway, EPG, which acts as a gateway between the radio network and the Internet. During the transition to 5G, the EPG will be split into a Control Plane Function (CPF) (previously Session Management Function or SMF) and a User Plane Function (UPF) to better meet the high requirement on scalability, robustness, and performance. The purpose of CPF and UPF is to handle session management and packet routing, respectively [8]. The CPF and UPF are sometimes also referred to as Control Plane (CP) and User Plane (UP).

As with any commercial solution, significant testing needs to be done before a product can be released to the public. Ericsson uses their own 5G cloud native simulators to test the performance of their 5G network, one of these is named *Ericload*. A previous thesis has been successful in extending and using Ericload to load test the UP at Ericsson [9]. The authors have mentioned that, while their work managed to produce simulations based on several common traffic models, the models themselves do not accurately represent how real world data would look like. These models include setting packet sizes to a fixed rate and size, either manually or based on Poisson processes.

The problem with using Poisson processes is mainly that the packet arrival process is memory-less, while real network traffic is shown to be time dependent and exhibit long-term memory, also called long-range dependence (LRD). The traffic is also self-similar, meaning that the behavior of the traffic is preserved irrespective of scaling in space or time. In other words, while Poisson processes assume that traffic is memory-less and smooth, empirical evidence shows that traffic is actually memory dependent and comes in bursts [12, 13].

1.2 Background

Figure 2.6 shows a simplified[1] version of the data flow when a client tries to connect to the Internet. The flow of data can go two ways: in the uplink direction where the client sends data to servers on the Internet, and the downlink direction where the servers send data to the client. When a client sends data in the uplink direction, it has to do so by first connecting to a base station nearby. The station assigns a Tunnel Endpoint Identifier (TEID) to the data to have it tunneled to a UP, along with an ARP request for mapping the MAC addresses to IP addresses. The UP itself can distinguish traffic to and from users by their user sessions. Several of these sessions can be maintained in parallel by the UP, but the tunnels themselves are unique per each established session. The data transfer is done by using the General Packet Radio Service Tunneling Protocol (GTP).

What makes UP effective is that it can run on virtual containers deployed on different platforms with the role of forwarding data packets via the Internet. One UP consists of several worker threads mapped each to a separate CPU core that await and process packets, and their routes, in parallel. The routing information for the packets is received from the CP and the most efficient route is then calculated. The UP and CP communicate with each other with the Packet Forwarding Control Protocol (PFCP) using Sx interfaces. The downlink direction is technically very similar. The only difference is that since the client is mobile and allowed to change base stations depending on its movement, the destination of packets needs to be checked regularly. This is made possible by using a Mobility Management Entity (MME) which manages clients and their connected gateways.

At its current state, the Ericload simulator can generate traffic for the UP using model-based load testing methods such as a basic steady-rate traffic, step-wise rate and Poisson processes. This leaves the potential to extend the simulator even more, by introducing traffic generation based off of reading and replaying historical data, instead of relying on mathematical models to generate traffic.

1.3 Aim

The overall aim of this project is to answer whether it is possible to load test a 5G simulator by replaying captured real time traffic data. The following subsections will present what research questions are proposed, along with how the available technology will be used to answer them.

1.3.1 Research Questions

- What are the approaches needed to implement scalable trace-based traffic modelling and load testing?

[1] Simplified as in that there otherwise can exist several UP-components between the base station and the final UP, forwarding the packets to the Internet

- How does trace-based load testing compare to model-based, in terms of producing realistic and configurable traffic?

1.3.2 Goals

The simulator at Ericsson will be improved with the aim to answer the above-mentioned research questions. More specifically, the goals are expressed as follows:

- Captured data comes from packet capture (`pcap`) files. An implementation to automate the extraction of necessary data from these files must be created.
- The simulator must be improved to be able to import captured real traffic data, recreate the packets and inject them into another network.
- Analyze old and new performance data. Find ways to plot the output data in a relevant way for analysis with old and new data, will be implemented. This will most likely involve using programs providing graphical tools, in order to help clarify the data.
- General basic code improvements of old code that could yield better overall performance.

1.4 Related Work

Previous research on 5G simulators is relatively slim. Given that the technology is still in its early stages, companies are not incentivized to publish work around it, as it could hurt their position on the competitive stage. A few open source simulators do exist, however. One of these is the *GTEC 5G Link-Level Simulator*, an open source simulator based on `MATLAB` which offers highly flexible implementations based on modules, that allows it to simulate different wireless communications standards, including 5G [7]. The downside is that it focuses on single-link performance and does not support multi-user or multi-transmitter (multi base stations) scenarios, making it too narrow of a use case for this project. Another simulator is the Vienna Cellular Communications Simulator (VCCS) suite that offers Link-Level (LL) and System-Level (SL) simulation [18, 22]. The main difference between a LL- and an SL-simulator is the former focuses on the physical layer of wireless communication allowing for the investigation of issues such as Multiple-Input Multiple-Output (MIMO) gains or modeling of channel encoding and decoding, while the latter focuses more on network-related issues like interference, scheduling and mobility handling. The SL-simulator acts, in other words, on top of the LL-simulator, using the LL-simulators inputs. However, the SL-simulator is not build to work with existing 5G components.

The ns-3 simulator is another type of simulator that uses open source modules to form a generic network simulator [16]. The module focuses mainly on the physical level of wireless communication but, unlike the Vienna simulators, does not distinguish pure LL-simulation from SL-simulation. This creates an inevitable degree of abstraction. The 5G K-SimNet is another network simulator that extends the ns-3 simulator to allow for layer simulation of the 5G network stack [2]. One of these extensions are the management of UPF and SMF functionality. However, it is not

constructed to work with real UPF-components. Furthermore, the simulator can only simulation traffic in the down-link direction and not the up-link direction. So while the ns-3 simulator has more in common with Ericload than the others mentioned, it too can only act as a reference.

Finally, since this project is a continuation on *5G User Plane Load Simulator* [9], many implementation solutions and measurements are inevitably inspired by the paper. Core components of Ericload, such as setting up proper network connections towards the EPG, remain the same and will not be changed for this project unless deemed necessary. The main focus will be on extension and adding traffic replay functionality to the simulators runtime phase.

1.5 Limitations

The biggest limitation of this projects comes to the traffic data used. There is both a privacy and confidentiality issue with using and publishing real data traces that Ericsson has captured. Instead, the traces mainly stem from our own personal networks using our own personal devices. Expanding on this, the scope of this thesis also limits the total amount of traces that can be used, making the thesis a proof of concept implementation at best. Ideally, a larger amount of `pcap` traces would be needed to better confirm the results. Lastly, further tests on larger UP nodes will also be needed to better assess the results in more depth.

1.6 Outline

Chapter 2 consists of several parts. The first gives explanation to basic definitions used throughout the paper. The second part gives a detailed introduction to traffic modeling. The third and final part introduces the 5G architecture at Ericsson and how it evolved from its predecessor 4G, as well as an introduction to the main simulator used: Ericload. Chapter 3 describes the implementations done to Ericload in order to grant it packet replay functionality. Chapter 4 discusses what results are expected from the simulator, and Chapter 5 presents what was actually obtained. The two final chapters, Chapter 6 and 7 analyzes and discusses the results obtained and then presents a final conclusion to the thesis.

2
Background

This chapter consists of three parts. The first explains reoccurring definitions used throughout this thesis. The second goes deeper into the theory of traffic modeling and data replay. The final part describes the network architecture Ericsson uses for its 5G traffic simulator Ericload, as well an introduction to the simulator itself.

2.1 Definitions

2.1.1 Packet Capture

Packet capture is the process of intercepting and logging traffic that passes over a computer network or part of a network. The data of the network packet is a formatted unit of data that consists of control information and user data (payload). The control information of packets contain data for delivering the payload and are typically found in their headers. What determines the arrangements inside the data packet is the communication protocol used, which is simply a system of rules that allows two or more machines over a network to reliably transfer information to each other. Figure 2.1 shows the Transmission Control Protocol (TCP), which is one of the main protocols used on the Internet.

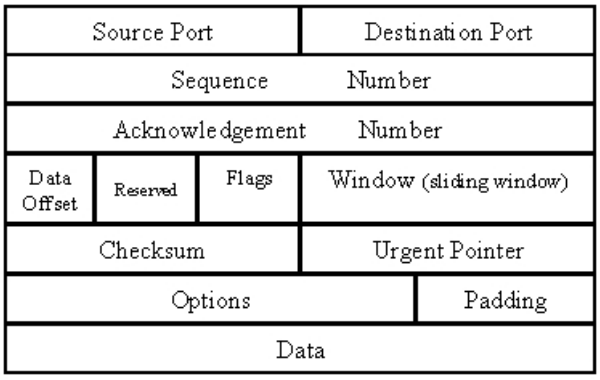

Figure 2.1: The anatomy of a TCP packet. The width of this packet is 32 bits. Every field except for the *Data* field is part of its header.

The tools used for capturing packet are called packet analyzers or packet sniffers. As data signals flow across a network, the analyzers capture each packet in the flow and decode their raw data, showing the values mapped to different fields. *pcap* [23] is an API written in C for capturing network traffic, that also allows the reading or

writing of data from or into `.pcap` files. The simulator used in this thesis, which runs on a Linux system, implements packet capture from the standard Linux library *libpcap*. This library contains an extensive collection of functions that can be used. However, this thesis uses only a handful of them, deemed necessary for improving the simulator. A full listing of relevant functions can be found in both Listing 4 and 5.

Most of the data in the packets is preserved during replay, since the intention is to essentially have a complete mirror of the original traffic. However, the IP addresses inevitably need to be changed in order to send the packets on another network. Further, the content of the packet payloads is not of importance for this thesis. It's critical that the size of the payloads remain the same for performance measurements, but its content might as well be random data or a corrupted version of the original. This last point is taken advantage of for latency calculations and is further explained in Chapter 3.

2.1.2 Cloud Native

The simulator used for this project is cloud native based. Cloud native refers to the property of being able to run (or virtualize) any application on a machine, regardless of the operating system (OS) and/or system architecture it uses. This is in one case made possible by providing an encapsulating platform called *containers* for the applications. Containers include all the code, libraries, and dependencies of the application. Similarly, using *Virtual Machines (VMs)* is another approach of achieving cloud based properties. VM's rely on a piece of software called a hypervisor, which is a small layer enabling multiple OS's to run alongside each other, sharing the same physical resources. VM's can therefore been seen as emulation of physical computers. In comparison, (see Figure 2.2), containers do not rely on a hypervisor and are therefore considered to be more light-weight, since the bottom-up approach of virtualizing underlying hardware and an OS is a more complex and resource intense strategy. The encapsulation of applications into one or more containers is also sometimes referred to as a microservice.

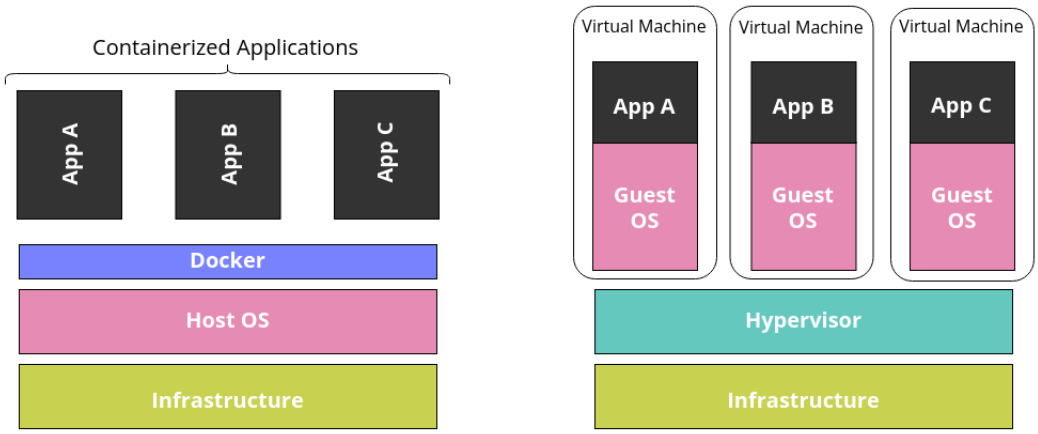

Figure 2.2: Difference between containers and virtual machines.

Ericload achieves a containerized architecture by using Docker [6, 9]. Docker creates containers at runtime by using so-called Docker images, which are executable packages of software including all necessary building blocks to run an application: code, runtime, system tools, system libraries and settings. The instructions for building a Docker image comes from a Dockerfile, which is essentially a text file that contains all the commands a user could call on the command line to assemble an image. These commands can be anything from port configuration to directory permission settings.

2.1.3 Configuration File

YAML [25] is used for setting up configuration files, with its main advantage being ease of use and readability compared to similar languages. A YAML configuration file is processed by the simulator during run time. The simulator sets up load parameters and the network environment by communicating with the connected network [9]. Parsing the YAML file into the C language of the simulator, involves simple mapping of key/value pairs using an appropriate data structure. In the usual case for C, the composite data type declaration *struct* is used. Listing 1 and 2 shows how such a translation could look like.

```
session_group:
    name:                   base
    start_ue_address:       22.64.0.1
    nr_sessions:            2
    start_teid:             839248
    predefined_type:        simplest
    core_network_instance:  sgi1
    node_id_fqdn:           pgwc.com
```

Listing 1: Excerpt from the Ericload YAML-configuration file.

```
typedef struct session_group {
    char*       name;
    uint8_t     start_ue_address;
    uint32_t    nr_sessions;
    uint32_t    start_teid;
    char*       predefined_type;
    char*       core_network_instance;
    char*       node_id_fqdn;
} session_group_t;
```

Listing 2: Data structure translation to C from the YAML-configuration file in Listing 1.

2.2 Traffic Modeling

There are usually two different approaches of generating test traffic: so called the analytic model-based traffic generation and trace-based traffic replaying [11]. This section explains the differences between both, their pros and cons, and why the latter approach is used for this thesis.

2.2.1 Analytic Model-Based Traffic Generation

The analytic model-based approach uses mathematical models for different traffic characteristics and generates the traffic to adhere to the models [11]. This can be tricky since the characteristics must be empirically measured beforehand while knowing what specific characteristics are considered important. Thus, while more simple models such as steady rate increases might be a reasonable tool for load testing systems, they do not necessarily mimic the behavior of actual traffic data [9]. For a very long time, the use of Poisson processes were a common approach to traffic modelling [12, 13].

Poisson processes are processes where discrete events at irregular times are observed over a continuous time interval. Packet arrival times are a plausible representation for this in network traffic [17]. A characteristic of Poisson processes are that the events are statistically independent, meaning that past and future events are not dependent on each other. Hence, as Poisson process is described as *memoryless*. Another property is that the more traffic sources are introduced and aggregated, the more *smooth* the traffic becomes. However, in 1993, Leland and colleagues discovered that this was not at all the case for LAN traffic [13]. What they found was that the traffic was rather *self-similar*, characterized by bursts, and long-range dependent (LRD) instead of memoryless. Later, the failure of Poisson modeling in wide-area traffic would also be discovered, calling for the abandonment of Poisson-based modeling altogether in traffic modeling [21].

Self-Similarity and Long-Range Dependence

In short: self-similarity describes the phenomenon where process behavior is preserved irrespective of scaling in space and time [13]. LRD means that the behavior of a time-dependent process shows statistically significant correlations across large time scales [12]. What was found was that traffic would come in so-called fractal-like *bursts*. At every timescale, ranging from milliseconds up to hours, this property was still evident. Also, aggregating traffic streams would only intensify this self-similar characteristic instead of smoothing it.

To describe these two properties in the context of time-series analysis, consider the stochastic process $X(t)$. Sometimes X can represent discrete time series $\{X_t\}, t = 0, 1, \ldots, N$, through either periodic sampling or by averaging its value across a series of fixed intervals. In the case of network traffic X describes the volume of packets observed in a link every time interval t. It is also possible to characterize the

dependence between the process's values at different times by evaluating the process's *auto-correlation function* (ACF), which is $\rho(k)$. The ACF measures similarity between X_t and a shifted version X_{t+k}:

$$\rho(k) = \frac{E[(X_t - \mu)(X_{t+k} - \mu)]}{\sigma^2} \tag{2.1}$$

where μ and σ are the mean and standard deviation. If we have that:

$$\sum_{k=1}^{\infty} |\rho(k)| = \infty \tag{2.2}$$

then we say that the stationary process X is long-range dependent. For self-similarity, we say that the stochastic process X distributed over time is self-similar if:

$$X(at) = a^H X(t), a > 0 \tag{2.3}$$

a is a scaling factor and H is the *Hurst exponent*. Should the Hurst exponent be in the interval $0.5 < H < 1$, we say that the process exhibits long range dependency [13, 20]. To estimate the Hurst exponent, we can use two different type of models: time domain operators and the frequency or wavelet domain operator [12]. Both investigate the power law behavior of specific statistical properties in a time series. They are, however, very complex to calculate, resulting in the Hurst exponent not being able to be definitively calculated, only estimated. Moreover, these models produce conflicting results, with it also not being clear which produces the most accurate estimations [12].

2.2.2 Trace-Based Traffic Replaying

Instead of using models to generate traffic as similar to real network traffic as possible, one can use tools [23] to capture live network traffic on a packet-level basis, and replay it back to a test network in order to investigate performance. This is what is referred to as trace-based traffic replaying [11]. Intuitively, using captured traffic should result in a higher result accuracy when investigating performance. The reasoning behind this is that since we do not have to first empirically study data network traffic, then use mathematical tools to generate traffic *similar* to it, we do not have to care about the quality of the replayed data. It is, after all, *real* data that has been sent over a network.

However, because this data is produced from state machines, keeping track of the state of TCP connections is of utmost importance in order to obtain usable results [11]. Results obtained from data not captured in a stateful manner will not be a good representation unless the congestion situation of both networks are exactly the same [3]. Therefore, so-called direct replay tools [1, 10, 24] whose aim is to simply inject captured IP packets into a test network, are not a good approach to performance investigation. Since they do not take the semantics of the networks into account, they give rise to problems such as ghost packets, which severely worsen the

testing results [3, 11].

To avoid the issues from the unidirectional strategy of the direct replay tools, another form of replay tools take the approach combining model- and trace-based methods into what is called *interactive replay* [3, 11, 15]. By instead using bidirectional traffic transformation techniques, i.e., partitioning input traffic into an internal and external part according to their direction, they are able to map captured traffic into target network conditions. For instance, before sending a packet, the system first extracts state information (sequence number, acknowledgement number, data length, etc.) from its header. Then it compares the information to the current state of its TCP connection in order to determine whether the packet is allowed to be sent. Should it fail, the packet will be buffered. Once passed, the state of its connection is updated. This traffic volume control strategy allows for recreating congestion scenarios that would likely happen in a real network [3]. Figure 2.3 illustrates an example of this scenario.

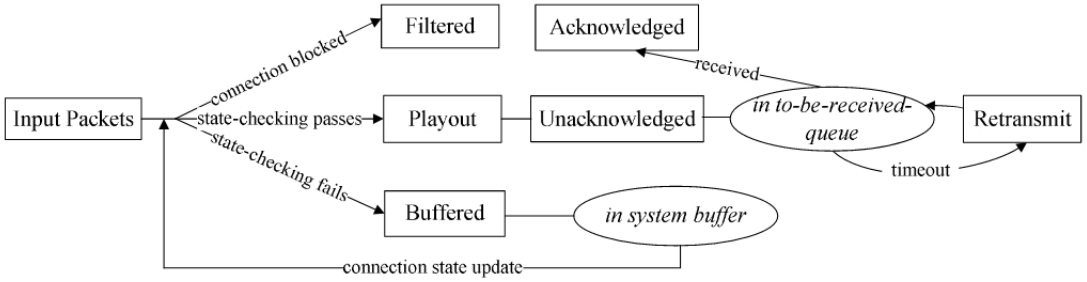

Figure 2.3: Example of a traffic volume control strategy for interactive replay tools [3].

One should keep in mind that although many of these tools may include the stateful properties needed to correctly emulate traffic flow on a node-to-node network, the number of actual nodes on a scaled-down simulated network is much less than an actual live wireless network. Not taking multi-node interactions into account properly might therefore also negatively affect the final testing results [15].

2.3 4G and 5G architecture

This project uses a simulator intended for 5G traffic. 5G itself is a successor to 4G, with more advanced hardware and different architectural design. Therefore, in order to understand 5G, it makes sense to first underlying concept of 4G technology. This section explains, on a basic and relevant level for this thesis, how 4G and 5G work.

2.3.1 4G architecture

Many concepts regarding 4G architecture in this section are directly taken from the book *EPC and 4G packet networks: driving the mobile broadband revolution* [19]. As seen in Figure 2.4, the basic architecture of a 4G network is divided mainly into three parts:

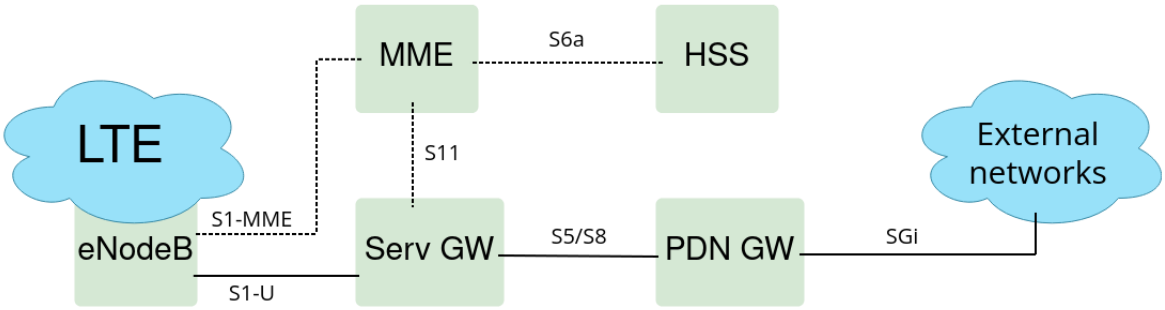

Figure 2.4: Basic EPC architecture for LTE [19].

- **LTE Network** - A network consisting of a User Equipment (UE) and at least one base station, also called an Evolved Node B (eNodeB).
- **Evolved Packet Core (EPC)** - Several network components used mainly to forward traffic back and forth between the UE and its destination, usually a server on the Internet.
- **External IP networks** - devices connected to the Internet.

When a user wants to establish a wireless connection, it first connects to the closest base station available, the eNodeB. The base station then forwards the user traffic to the EPC, which itself consists of four parts:

- **Mobility Management Entity (MME)** - Handles LTE-related control plane signaling, that is traffic from UE to eNodeB. It is also responsible for mobility and security functions, as well as session management.
- **Serving Gateway (SGW)** - As the names suggests, it serves the data to different components depending on the direction of the traffic. If the traffic is down-link, the data will be forwarded to the UE. For up-link, the data is forwarded to the PGW.
- **Packet Gateway (PGW)** - Establishes mainly a connection with the SGW and the external IP network. It also includes functionality such as IP address allocation, packet filtering and maintaining statistical data for charging and accounting.
- **Home Subscriber Server (HSS)** - Used mainly for subscriber information such as billing plans. This component is ignored throughout this thesis as it is not deemed relevant.

For all the components to communicate with each other, several interfaces are used:

- **S1-MME** - Connects the UE with the MME. It should be noted that this interface is used mainly for signaling events and not payload data. That is, the messages sent are used to establish as coordinated connection between different components of the EPC.
- **S1-U** - Where S1-MME is used to coordinate communication between components, the S1-U interface is used to send actual payload data in either the up- or down-link direction between the UE and the SGW. The General Packet

Radio Service Tunneling Protocol (GTP) is used to transfer the payload data between the LTE and EPC network.

- **S11** - Connects the MME and SGW. Handles important functions such as creating, deleting and modifying sessions toward the external IP network.
- **S5/S8** - Connects the SGW and PGW. S5 transfers payload data, whereas S8 transfers signaling events. Both are based on GTP, which ensure connectivity should a user move through space and connect to different base stations (roam).
- **SGi** - Connects the PGW with the external IP network. From the network's point of view, the PGW is seen as a normal IP router.
- **S6a** - Connects the MME with the HSS to handle subscription related information.

2.3.2 5G architecture

As mentioned, with 5G being the successor to 4G, it will have more advanced hardware and different architectural design. This section goes into further detail explaining the differences between both architectures.

Shown in Figure 2.3.1, the core functionality in an 4G architecture is located in the EPC. The two main gateways, SGW and PGW, are sometimes clumped together into what is called an Evolved Packet Gateway (EPG). Further, these two gateways are also split into two components: a User Plane (UP) component and a Control Plane (CP) component. The UP is responsible for inspecting and transferring payload data, while the CP handles signaling events. They both use the above-mentioned interfaces (S5/S8) for their corresponding goals. This splitting into UP and CP components of the EPG, is an essential part of the future 5G architecture and is referred to as Control- and User Plane Separation (CUPS) [9, 14].

Figure 2.5: Reconstruction of EPG into CUPS architecture [9].

Figure 2.5 how the separation of the EPG for an 5G architecture will look like. As mentioned, the core idea is to separate the user- and control plane into different components. The main advantage is that it allows the CP and UP to scale independent of each other without affecting existing functionality [9]. Further, it reduces expenditure and energy consumption of base stations, as well as providing more simplified network management with the help of software [14].

Two new interfaces are introduced for the communication between components in the CUPS architecture: Sx_a and Sx_b. Sx_a is used for the SGW, while Sx_b is used

for the PGW. Both follow the Packet Forwarding Control Plane (PFCP) protocol, whose purpose is to deliver UDP signaling events between the UP and CP [4].

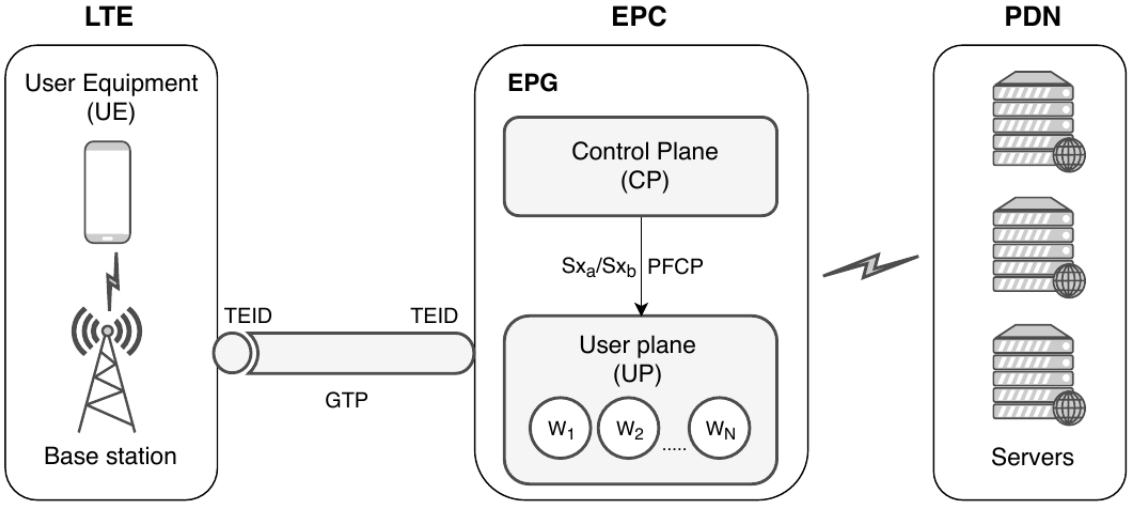

Figure 2.6: Illustration of a simulated 5G network [9].

Figure 2.6 illustrates a possible 5G network that utilized the new CUPS architecture. While very similar to a 4G network in its structure, there are a few notable differences. To start with, when the UE connects to a base station, the station assigns a Tunnel Endpoint Identifier (TEID) to the UE's data before connecting to the UP. An ARP request is then sent to the UP in order to map a MAC address to the IP addresses of the transport layer. Connected to the UP, the data is then transferred using GTP, which in turn is forwarded to servers on the Internet, so called PDNs. It should be noted that it is possible for the data to travel through several UP components before arriving at its intended destination.

The UP component is run on virtual machines and containers deployed on several platforms. A single component in turn has several worker threads, each mapped to a separate CPU core on the UP, allowing for parallel processing of data traffic. The CP in turn sends routing information to these parallel processes using PFCP. The down-link version of this flow works similarly. However, since the UE might have changed position due to roaming, MME is utilized as seen in the 4G architecture.

2.4 Ericload

Ericload is the name given at Ericsson to one of their simulators, and is in its essence a tool used for load testing the UP by sending and receiving network packets to and from it. A *load* is defined as different approaches of generating and using numerous packets, while a *load test* is thus simply monitoring and analyzing the performance of a system (UP in our case) under these different loads. In *5G User Plane Load Simulator*, three different load tests of generating model-based traffic were introduced and analyzed: steady-rate, step-wise, and Poisson process [9]. Poisson processes are the most complicated of these and are described in section 2.2.1. The other two are a

variation of generating and sending a fixed amount of packets during a certain time span, as well as being able to incrementally increase this maximum amount during different time intervals. All three of these load tests fall into the same category: simulated packets from a fixed distribution, and many current design decisions of Ericload have therefore been formed around this. In this thesis, we explore the feasibility and scalability of trace-based simulation. To compare the two approaches, we use as baseline the simplest and most efficient simulated packet approach, namely the steady-rate one. Further, in the above-mentioned thesis, the goal was primarily to perform detailed load tests and calculate the efficacy of the UP. Traffic modelling was of secondary thought. In this thesis, we instead focus on the latter. Hence, testing for scalability by only scaling the number of PDNs was deemed sufficient, since the final number of flows end up being the same as when scaling the number of users too. Refer to *5G User Plane Load Simulator* for comparing scaling in the number of PDNs and scaling in the number of sources, i.e., number of flows versus number of users [9].

2.4.1 Setup

Ericload sets up its own working environment by processing, and applying, parameter values found in the configuration file (see Listing 1). Other than technical specification such as routing info and endpoint addresses, one specifies several parameters like number of groups, the amount of user clients per groups and flows (a network connection where up- and downlink traffic is possible) per user in a group. With this information, Ericload calculates the number of UEs and PDNs with IP connectivity needed to perform the simulation. Packets to and from these units are always encapsulated inside GTP, which allows us to choose IP addresses arbitrarily. Further, each flow is associated with a load type, where users in the same group are assumed to have the same load type. Details about the load types are also defined inside the configuration file, with the primary ones being how many packets to send in both directions and the time interval of the load type. With this, Ericload calculates the total amount of packets to generate and send during its entire run time. Further, Ericload uses data from the same configuration file to establish the overall simulation-ready network. Here, DPDK is used in order to bypass the kernel and ultimately boost the performance of the simulation [5]. ARP requests are used to establish a connection with the devices connected to the UP, and a PFCP session is established with the UP. This session allows Ericload to set up user sessions with the UEs and PDNs. Tunnel Endpoint Identifiers (TEID) are also used here by the UP to differentiate between the devices and know how to correctly route the traffic. Finally, Ericload uses a set value in the configuration file as the total number of seconds a simulation should be run for, before Ericload exits its entire process.

2.4.2 Execution

Ericload is run inside a docker container, where work is divided into two primary threads. Both threads are used for processing incoming packets and log output data, while only one is responsible for creating and sending packets. During run time, a

local timer that represents the total time passed since the start of the simulation is continuously updated. This timer is available for both threads. For instance, the thread responsible for creating and sending packets refers to this timer value when deciding whether it is allowed to perform any action at its current point in time. Packet sending is done in bursts of 32 packets. If the simulator cannot send packets quick enough, it stores away the leftover packets and resends them in the future in order to maintain the set flow rate. Packet creation depending on direction is more or less the same, with the caveat being that uplink packets need to specifically be encapsulated inside a GTP message with the correct TEID since they are going through a GTP tunnel. Otherwise, correct source and destination addresses are set to the corresponding local UE and PDN addresses. The payload of all the packets is completely arbitrary but is set to a fixed size and appended with a timestamp which is later used to calculate the delay of the packet. The receiving process is similar in that it also is done in 32 packet bursts, although these are successively repeated until all packets are received. Around every five seconds, Ericload prints statics to the output terminal used, as well as writes statistics to a CSV file on disk which include information about traffic quantity, rate and latency. The sampling amount used for the CSV is set in the configuration file mention above and is defined as samples per second. The simulator continues running until the cut-off time from the configuration file is reached. Once this happens, it stops the simulation and performs a clean exit.

Using Ericload with the model-based load types described has shown that the strength of the UP is to handle large number of flows simultaneously, rather than giving higher rates to fewer flows [9].

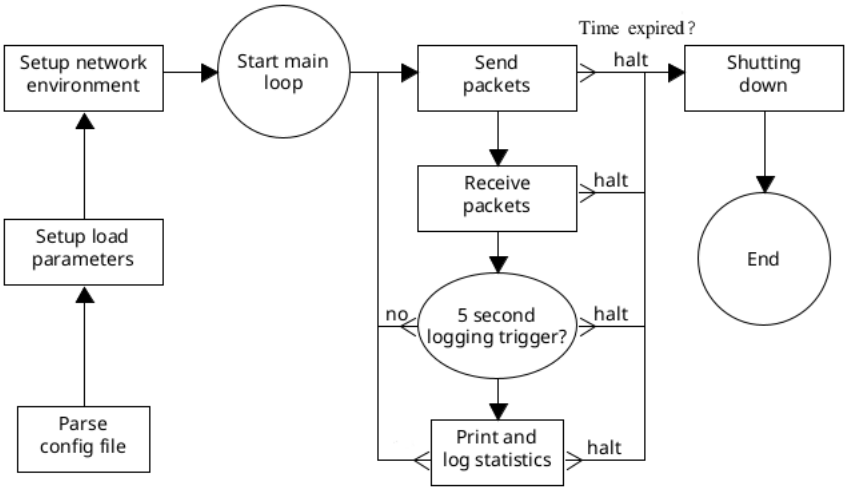

Figure 2.7: Process flow of Ericload from start to end of execution.

3
Implementation

All code extensions and improvements introduced to Ericload are added in a modular way as to not interfere with the existing code. Many core solutions like setting up proper connections to the UP using PFCP, or processing incoming packets from the UP, have not been changed and remain compatible with the new code. The main addition is how traffic is generated. As mentioned briefly in section 2.4, many design choices in the current code were made with model-based traffic generation in mind. Thus, current packet generation functionality cannot be reused in order to create trace-based traffic and many new solutions are introduced around this.

This chapter discusses the implementation of this traffic replay functionality. It starts by introducing a time-lapse of the steps from reading a `pcap` file, to having the reconstructed packets of the file received backed from the UP. It then explains in detail what happens during these steps and argues for the choices made.

3.1 Time-lapse overview

Replaying traffic in Ericload is essentially divided into two phases. The first phase prepares a buffer array of packets from a `pcap` file for the simulator to use, and a reference table with IP addresses. This is shown as the first step in Table 3.1. The contents of this array are never modified and stored in memory. We do this in order to skip reading from disk every time we want to generate packets from the `pcap` file, which would otherwise slow down the simulation. In the second phase the simulator is actually running its main loop where it generates, sends and receives packets based off of local timestamps. It iterates through the stored array and compares the packet timestamps with its local ones. If a packet is allowed to be sent over the network, it copies this packet to another local packet array, where it simultaneously changes the packet's content using the mentioned reference table. It then sends all packets found in its local array to the UP. In the next and final step, it checks for incoming packets from the UP. These should be the ones sent in step two, which have traveled across the network back to the simulator to process. Statistical calculations are done for every packet received.

The simulator then goes back to the second step and repeats this process until a specified cut-off time is reached, shutting down the simulator. All these steps are explained in more detail in the upcoming subsections. Note that the initial setup and shutdown phases of the simulator are omitted in this time-lapse, since nothing

new has been added implementation wise.

Step	Functionality used	Output
1. Reading and processing a `pcap` file.	Functions from *libpcap* library used to temporary load the `pcap` file into memory and process its contents	Hash map M containing the mappings from packet addresses to local simulator addresses. Packet buffer array P filled with identical copies of the original packets. Both are stored in memory.
2. Sending packets.	Iterating through the packets from the array P and deciding whether to send them out on the network by comparing their timestamp with the current timestamp of the simulator.	Partial, or full array O of packets ready to be sent over the network. Every added packet is copied from P to O. The content of the packets in O are changed based on the mappings in M. The current timestamp is also injected into their payloads.
3. Receiving packets.	Inspecting all incoming packets and extracting the timestamps found in their payloads. These timestamps are used for statistical calculations.	CSV file containing statistical raw data about the traffic such as total packet quantity, total data quantity and latency.

Table 3.1: Time-lapse description of the simulator when generating, sending and receiving replay packets.

3.2 Designing groups and processing `pcap` files

Unlike the model-based approach of traffic generation, trace-based methods have the advantage of already knowing the final representation of the traffic sought to mimic. Models set small initial parameters that in *future* run time can bloom into complex traffic patterns. Traffic captures, on the other hand, rely on as much *past* historical details of traffic as possible. Both methods approach to the time domain can almost be seen as each other's inverse, and thus greatly affects the design approach of creating packets for a simulator.

As mentioned in section 2.4, Ericload generates packets based on initial parameters for the traffic type of a particular flow group. For instance, when using step-wise traffic flow, one can set up- and downlink rates, number of users in a group, along

with a time interval t. The simulator uses this time interval to, at any time t_n, calculate how many packets need to be generated in either direction and to whom. Then later, at starting time, Ericload calculates the necessary amount of UEs and PDNs needed, as well as their network properties like local addresses to complete the whole simulation.

There are some issues with this approach from the perspective of replayed traffic. First, setting initial traffic rates does not make sense since all the rates can be extracted from the trace itself. Further, these rates can vary greatly depending on what point in time of the trace we are currently looking at, and are also impossible to predict beforehand given that there are unlimited ways for network flows to exist. Next there is the issue of randomly picking a packet p_n from a time t_n and determining attributes like which direction (up- or downlink) it is supposed to be sent, in order to maintain traffic states. A reference table is required here with network addresses where one can map the source and destination addresses of the packet p_n with the local UE and PDN addresses of the current flow group. Lastly, constructing packets is very costly if accessing the `pcap` file from disk is done every time. Accessing physical disks during simulation should therefore happen scarcely.

To solve these issues, the simulator makes use of the functions from Listing 5 to create a buffer array consisting of identical packet copies found in the original `pcap` file. This array becomes a one-to-one representation of the original `pcap` file. It is then saved to memory, which circumvents the need of rereading the `pcap` file from disk in the future. To correctly construct this array, a hash map is constructed as a reference table between the local addresses used by the simulator and the addresses found in the `pcap` file. Each address found is treated as a unique key in the hash map, and it points to a value containing its local simulator address.

3.3 Reference table

Let us use the `pcap` file from Figure 4.1 as an example to show how the reference table for the simulator addresses is created. When picking the first packet p_0, the initial hash map is empty. Since the source address s_0 of the packet does not exist as a key in the hash map yet, we treat it as a user trying to establish a connection with a server and thus as a packet in the uplink direction. It should be noted that this approach does not necessarily result in the correct directional mapping every time, and may in fact cause the reverse translation to happen. The reason for this is that if p_0 for example is captured in the middle of a UDP traffic exchange, simply inspecting the packet does not help conclude which address initiated the connection. However, this is merely an esthetical issue. The simulator itself does not care, nor gets affected by the directional setup. A key/value pair is inserted into the hash map, with s_0 as the key pointing to a value UE_0 containing the fixed local UE address given by the config file. Next we do the same procedure but use the address d_0 as a key. This time, the value placeholder which the key d_0 points to the start address of flow group's local PDN address. This is calculated using the following

equation:

$$\begin{aligned} PDN_i &= PDN_{start} + PDN_{total} \\ PDN_{total} &= PDN_{total} + 1 \end{aligned} \tag{3.1}$$

Where PDN_{start} is the starting PDN address of the flow group found in the config file, and PDN_{total} is the total number of PDN addresses used, where the value gets incremented by one each time a new address is added. We have now a reference table containing an entry point for the first packet in our capture file. Next, we inspect the second packet from the capture file, which shows how addresses are extended. This packet contains a new flow with a unique, unknown, source address s_1 but an already seen destination address d_1 pointing to UE_0 in our hash map. Therefore, we treat this flow as the PDN sending a packet to the UE and therefore map the source address s_1 with a new PDN address PDN_1. In the third packet both the source and destination have known mappings to UE_0 and PDN_0 and the hash map will thus not be updated with any new values. We continue these operations until all the packets from the capture file have had their addresses correctly mapped to our reference table. This hash map is then stored in memory for later use. The total number of local PDN addresses used PDN_{total}, is also stored in memory. The final hash map constructed for this scenario is shown in Listing 3.

```
1 PCAP_TRACE_HASH_MAP
2 {
3     10.48.0.2  -> { 16.0.0.1 },  # UE_0
4     60.0.0.1   -> { 192.0.0.0 }, # PDN_0
5     106.0.0.3  -> { 192.0.0.1 }  # PDN_1
6 }
```

Listing 3: Abstract representation how the `pcap` trace in Figure 4.1 translates into a hash map.

3.4 Creating and sending packets

For the flow group to properly make use of the constructed packet array, a few more variables are set beforehand. First, a variable *repetitions* is used to determine how many times the complete flow of the packet buffer array will sequentially be repeated. Its value is set manually in the configuration file for the simulator. Accompanying this, an extra variable $t_{initial}$, holding the initial time stamp of the first packet, is also set. Second, the variable $instances_{total}$ is set that represents the total number of parallel instances the whole packet array uses. An instance is defined as the correct replay of all the packets inside the packet array from beginning to end, using local addresses unique to the given instance and no other. That is, the second instance of our packet array would reuse the addresses from our hash map, but shift all of them to new, unused ones. Scaling this way treats all copies of the local flows within a `pcap` file as unique, even though their structures are identical. An *instance* variable initialized to zero is also set to keep track of the current instance. Lastly,

we also introduce an *index* variable with an initial value of zero, representing the index of the current packet being processed.

During the actual execution phase of the simulation, the simulator sends packets in bursts, where each burst can contain a maximum of 32 packets at a time [9]. The simulator is also able to keep track how much time t_{passed} has passed since the simulator started. Using this, we determine if a packet is allowed to be sent by comparing t_{passed} with the difference:

$$\Delta t = t_{p(index)} - t_{initial} \tag{3.2}$$

where $t_{p(index)}$ is the timestamp of the current packet $p(index)$. If $\Delta t \leq t_{passed}$ we are allowed to send the packet and proceed to allocate a simulator packet where we copy the contents from the packet of the buffer array to this simulator packet. In this copy we first determine which addresses are UE and PDN respectively, then change their address to match the current *instance* with the help of this formula:

$$PDN = PDN_{start} + (PDN_{total} \cdot instance) \tag{3.3}$$

Here *instance* is used as a scaling factor, guaranteeing no instance shares the same local addresses as another.

Next, the first 64 bits of the data field of the packet are overwritten with t_{passed} in order to later compare and calculate the transit delay of the packet. This corrupts the original payload but causes no overall issues since we are not interested in the actual value of the payload. Finally, the proper headers are encapsulated depending on which direction the packet is headed. For uplink packets, we encapsulate with a GTP and Ethernet header, while a downlink packet only receives an encapsulated Ethernet header. The packet is then ready to be used by Ericload and is thus added to the burst array. Lastly, we increment *instance* by one and check if it matches the value of $instances_{total}$. If true, we reset *instance* to zero and increment *index* by one, allowing us to process the next packet in the array. If false, we know that there are future instances left that need to process the same packet again, and we thus only increment *instance* by one.

There are two scenarios in which we are not allowed to send packets due to timing constraints. The first is when the above $\Delta t > t_{passed}$. This means that the packet timestamp is ahead of our simulator timestamp and thus not allowed to be processed yet. Here we simply keep doing nothing until the comparison does not hold anymore. The second scenario is when *index* is out of bounds since all the packets have been sent, but *repetitions* still holds a value greater than zero. This means that all the timestamps in the packets need to be shifted to accommodate for the time passed. We manage this by taking the time difference of the last and first packets in the array, and appending this value to the timestamp of all elements. Then we decrement *repetitions* by one, reset *instance* and *index* to zero, and exit. Thus, we have completed a full replay of the original `pcap` trace and reset it for another rerun.

3.5 Receiving packets and finishing execution

Receiving packets is also done in 32 packets bursts, much like sending. However, different from sending, receiving packets is repeated in successive bursts until all current packets received from the UP have been handled. When a packet is received, it gets decapsulated to extract the timestamp contained in the data field. This timestamp is calculated against the current timestamp and logged as the transit delay of the packet in a CSV file. The simulator keeps running until the set simulation duration is met, where it then proceeds to correctly exit the loop and the whole simulation in general. Even though we can calculate the minimum amount of time needed to replay the traffic from all `pcap` files by looking at the timestamps and all scaling variables, there is nothing predicting the exact execution time in general. Further, since replaying traffic is intended to be done in parallel with model-based traffic which goes on infinitely, manual set times are preferred.

4

Evaluation

This chapter introduces different evaluation approaches used throughout the thesis and how these are applied to properly interpret results.

4.1 Metrics

During simulation, Ericload calculates several metrics over a given sampling step and write them to a CSV file. The frequency of these samples are set by a value in the configuration file representative of the number of samples per second. Each sample step contains information about the outgoing and incoming traffic at the current step, as well as latency measurements. The final CSV file is fed into a simple python script, where the output is represented as graphs for further analysis.

4.1.1 Total amount of packets and data

Every time Ericload sends or receives packets, it stores information about the total amount of packets processed. It uses two different arrays to accomplish this, one for outgoing (sending) traffic and one for incoming (receiving). The elements in the array are integer values that represent either the current total amount of packets, or the current total amount of bytes. These elements are further grouped by the interfaces used during the simulation: uplink or downlink. This alone grants a total of eight data points to be used for statistical calculations. All the statistical variables in Ericload are accumulative over time. Statistics for any sample point s_n is therefore defined as the difference between the total variable data at time $t(n)$ and the total variable data at time $t(n-1)$ where $n > 0$. It is assumed that all variables are set to zero at $t(0)$.

Table 4.1 shows an example where data has been gathered over the span of one second. Since samples are configured as samples per second, and there are a total of five sampling points showing, the columns at any point n therefore shows a sampling at every 200th millisecond. The total outgoing uplink packets of the simulator at 600 milliseconds would in this case happen at sample point s_3 and simply be calculated

as:

$$\begin{aligned} s_n &= t(n) - t(n-1) \\ s_3 &= t(3) - t(2) \\ s_3 &= 6 - 4 \\ s_3 &= 2 \end{aligned} \tag{4.1}$$

This difference in outgoing uplink packets happens to be the same no matter what sampling point n we choose in the table. Should we therefore choose to plot this on a graph, we would see a flat graph showing a steady rate of two packets per sampling point.

t(n)	0	1	2	3	4	5
Downlink Packets Out	0	1	2	3	4	5
Downlink Bytes Out	0	10	20	30	40	50
Uplink Packets Out	0	2	4	6	8	10
Uplink Bytes Out	0	20	40	60	80	100
Downlink Packets In	0	2	4	6	8	10
Downlink Bytes In	0	20	40	60	80	100
Uplink Packets In	0	1	2	3	4	5
Uplink Bytes In	0	10	20	30	40	50

Table 4.1: Hypothetical accumulation of statistical variable data over the span of one second.

4.1.2 Latency

Latency is defined as the elapsed time from when a packet was sent from the UP, to when the same packet has been returned and processed by the UP again. The latency of a packet p is measured by comparing the timestamp t_r when receiving the packet, with the timestamp t_s found in the data field of the same packet. t_s was the current timestamp during the data injection. Both timestamps come from inspecting the total time passed of the current simulation, represented as t_{passed} in Section 3.4. The latency calculation is then expressed with the following equation:

$$Latency(p) = t_r(p) - t_s(p) \tag{4.2}$$

Latency, like all other variables, is an accumulative data point and is therefore calculated like any other data point for a sample shown in Eq. (4.1). However, the CSV file will contain information about the average directional (incoming or outgoing) latency for a specific sample point. This is simply calculated as the total latency at that point over the total amount of packets sent of the same point, of a given direction.

4.1.3 Throughput

As shown in Section 4.1.1, since both the total amount of packets and the total amount of bytes are measured at every sampling point, the throughput at every

sampling point is already given. Note that the throughput, like latency, can be measured for any combination of the direction of the traffic and the interface it is being sent on. The average throughput of the entire simulation can also easily be calculated by taking the average of the sum throughput across all sampling points:

$$AvgThroughput = \frac{k \sum_{i=1}^{n} Throughput(s_i)}{n} \tag{4.3}$$

Where n is the total amount of sampling points and k the amount of samples per second.

4.2 Replay verification

This section presents different approaches to confirm that Ericload indeed can replay packet capture files on the most basic level.

4.2.1 Correct packet creation

It is expected of the simulator to reconstruct all the packets found from the inputted `pcap` file, and to correctly change some parameters of the packets in order to be able to send it over the network. This is confirmed by having the simulator write its own `pcap` file to disk and append every packet created during run time to it. This file can then be compared with the original to check whether the correct packet manipulations have been done. It also allows us to determine if the simulator has sent the packets in the same order as the original file, which is also a strict criterion for a successful replay. It should be noted that writing `pcap` files to disk in this way is only a temporary debugging solution to verify the traffic, and not something that is done during each run of the simulator.

4.2.2 Traffic repetition

Another function of the simulator is that it must be able to correctly repeat the traffic of a `pcap` file for any number of time given. Correct in this case means that it must abide by the time constraints found in the file. For example, if the `pcap` file used contains 100 packets sent over the span of a second and is set to repeat 10 times, the simulator is expected to produce traffic for a minimum of 10 seconds with around 100 packets sent each second. The key takeaway here is that the time domain must be respected during traffic replay as well, regardless how fast the simulator is capable of processing the replay trace.

4.2.3 Scaling

Scaling in the replay sense is defined as being able to resend the same packets of a `pcap` file in parallel, but introduced as new unique flows. Meaning that if the first packet p_0 of the `pcap` file is given the local destination address PDN_0 when reconstructed by the simulator, scaling would instead give a new packet the local addresses PDN_1. PDN_1 is only used in the flow group the packets are constructed

for, and not as local addresses for any other packet p_n in other flows groups, as shown in Eq. 3.3. By comparison to the scenario in section 4.2.2, scaling this file by a factor of 10 would result in a minimum of 10 seconds of traffic with around 1000 packets sent each second. The new addresses introduced will range from PDN_0 to PDN_9.

4.2.4 `pcap` traces used for verification

Two very small `pcap` traces will be used in order to confirm that Ericload correctly replays traffic. The reason for this is that Ericload does a write-to-disk call for every single packet found, which would severely impact performance if larger traces were to be used instead. Both traces will later be shown as a sending trace and as a receiving trace. The sending trace will consist of packets to be sent to the UP by Ericload. These packets are what Ericload constructs and logs during runtime right before sending the packet out. The received trace, instead, consists of packets received by Ericload from the UP. It is expected that both the sending and receiving trace match as closely as possible. This however not a guarantee since the receiving trace depends on every packet traveling across the network and back to the UP and simulator. Delays might affect the ordering of the packets, depending on how quickly they are sent and received. Also, since received packets are currently processed by two threads in Ericload, these might compete for the write-to-disk call and reorder the packets due to their timestamp being different. It is not expected that the sending trace have these problems since it is processed only by one thread, and since it abides by stricter timing constraints from the original `pcap` trace it is trying to replay. It is expected however that the amount of packets in the traces match one by one, due to their smaller size.

UDP trace

The first trace is a UDP trace shown in Figure 4.1 consisting of ten packets, each 134 bytes in size. This trace includes three unique addresses. It is expected that the source address from the first packet be converted to an UE_0 address, and the destination address to an PDN_0 address by Ericload. The second packet introduces the third unique address, sending UDP data to the source address of the first packet. Here, we expect Ericload to convert this third address to PDN_1, since the destination address of the second packet is already mapped to UE_0. Finally, this trace repeated once, making it a total of 20 packets to be sent and expected to be received back.

No.	Time	Source	Sport	Destination	Dport	Protocol	Length
3	0.000000	10.48.0.2	55001	60.0.0.1	80	UDP	134
4	0.000001	106.0.0.3	80	10.48.0.2	55001	UDP	134
5	0.000003	10.48.0.2	55001	60.0.0.1	80	UDP	134
6	0.000001	106.0.0.3	80	10.48.0.2	55001	UDP	134
7	0.000003	10.48.0.2	55001	60.0.0.1	80	UDP	134
8	0.000001	106.0.0.3	80	10.48.0.2	55001	UDP	134
9	0.000003	10.48.0.2	55001	60.0.0.1	80	UDP	134
10	0.000001	106.0.0.3	80	10.48.0.2	55001	UDP	134
11	0.000003	10.48.0.2	55001	60.0.0.1	80	UDP	134
12	0.000001	106.0.0.3	80	10.48.0.2	55001	UDP	134

Figure 4.1: Wireshark output showing a small UDP trace. Filtered for relevant packets. Timestamps are shown as time passed between each displayed packet.

TCP trace

The first trace is a TCP trace shown in Figure 4.2 consisting of 20 packets in total with varying sizes. Here only two unique addresses are found, therefore it is expected that the source address from the first packet be converted to an UE_0 address, and the destination address to an PDN_0 address by Ericload. This trace will be scaled once and will introduce a second PDN address, PDN_1, which is expected to be mapped to the same destination address as PDN_0. Due to the scaling, it is also expected that Ericload correctly takes turn when sending the packets while still keeping the correct order. In total, 40 packets are expected to be shown in the traces due to the scaling factor.

No.	Time	Source	Sport	Destination	Dport	Protocol	Length
1	0.000000	192.168.4.1	62497	119.235.235.110	9418	TCP	74
2	0.321877	119.235.235.110	9418	192.168.4.1	62497	TCP	66
3	0.008368	192.168.4.1	62497	119.235.235.110	9418	TCP	56
4	0.000015	192.168.4.1	62497	119.235.235.110	9418	TCP	134
5	0.326269	119.235.235.110	9418	192.168.4.1	62497	TCP	54
6	0.007078	119.235.235.110	9418	192.168.4.1	62497	TCP	140
7	0.001590	119.235.235.110	9418	192.168.4.1	62497	TCP	1414
8	0.006786	192.168.4.1	62497	119.235.235.110	9418	TCP	56
9	0.002563	192.168.4.1	62497	119.235.235.110	9418	TCP	56
10	0.316771	119.235.235.110	9418	192.168.4.1	62497	TCP	874
11	28.587107	192.168.4.1	62497	119.235.235.110	9418	TCP	56
12	0.141542	192.168.4.1	62497	119.235.235.110	9418	TCP	135
13	0.315115	119.235.235.110	9418	192.168.4.1	62497	TCP	54
14	0.006169	119.235.235.110	9418	192.168.4.1	62497	TCP	264
15	0.005298	192.168.4.1	62497	119.235.235.110	9418	TCP	56
16	0.091501	192.168.4.1	62497	119.235.235.110	9418	TCP	135
17	0.356405	119.235.235.110	9418	192.168.4.1	62497	TCP	54
18	4.934943	192.168.4.1	62497	119.235.235.110	9418	TCP	56
19	0.322772	119.235.235.110	9418	192.168.4.1	62497	TCP	54
20	0.005157	192.168.4.1	62497	119.235.235.110	9418	TCP	56

Figure 4.2: Wireshark output showing a small TCP trace. Filtered for relevant packets. Timestamps are shown as time passed between each displayed packet.

4.3 Performance

The below sections describe how we are expected to interpret and compare the results from the simulator when successfully replaying traffic. The expectation is that trace-based traffic does not affect the maximum rate of the simulator any more than model-based. Further, any strengths of the UP, such as handling many users

without performance impact [9], are also expected to hold.Nevertheless, it is still of value to perform the same measurements and compare these to previous ones. All these different tests, model- and trace-based, will be conducted for 60 seconds in total. Lastly, the latency for all tests involving replaying traffic will be presented as the median over all the averages in the sampling point. The reason for this is to account for any latency spikes that may appear during simulation because of the bursty nature of the traffic, that would normally yield skewed mean latency results.

4.3.1 Maximum rate using model-based traffic

To find at which packet per second rate the UP is able to perform sufficiently, a model-based approach will be used by generating steady rate traffic for 60 seconds. Further, an arbitrary fixed packet size of 1400 bytes will be set to measure throughput performance. This value does not include the added sizes due to packet encapsulation. The tests performed will be separated into three categories: downlink, uplink and a combined up- and downlink test. All three categories will measure three scenarios each using one, 10 and 100 flows. Measurements will be performed on the throughput, median delay and the ratio of packet loss. Packet loss ratio is defined as one minus the amount of incoming packets received from the UP to Ericload, over the amount of outgoing packets sent from Ericload to the UP. This ratio will be the main measurement used to find a stable performing rate for the UP, with 2% set as the maximum amount of acceptable packet loss.

4.3.2 `pcap` traces used for performance evaluation

Two separate `pcap` traces will be used in order to evaluate the performance of Ericload and the UP. One trace is used for sending mainly TCP traffic, while the other is used for UDP traffic. Both traces will be used in different testing scenarios to compare how well Ericload and the UP can handle replaying traffic. Testing will be done using either the traces alone, in combination with each other, or in combination with model based traffic produced by Ericload in the form of steady rate UDP traffic. Figure 4.3 shows the packet size distribution of the traces as well as their total amount of packets. Further detail about the traces themselves will be explained below.

`60mbTCP.pcap`

The TCP traced used is a recording of a 60Mb file downloaded over a local Wi-Fi network. The recording took place until the file transfer was completed and took around two seconds with a little over 60k packets being sent. A majority, 68% percent, of the packets are 1514 bytes and contain the actual payload of the file being downloaded. The remaining packets around 54 bytes in size are the acknowledgements sent from the host to the server. The full file transfer took around two seconds. This trace is always repeated 28 times during every test case it is used for, in order to be present as much as possible during the 60-second simulation, but still shy from the mark to leave a bit of room for all the packets to be processed before

the simulation ends.

youtubeLarge.pcap

The main UDP trace is a recording of viewing an hour-long YouTube video for around 26 seconds, totaling in around 40k packets. The reason behind this choice was to leave the video on long enough for the packets to arrive in different sized bursts, spread out at different time intervals. This recording will always be replayed twice in order to be present as much as possible during the simulation. 89% of the packets recorded in the trace are 1250 bytes in size. The largest spike in packets sent happens around 8.6 seconds with 2221 packets being sent over the span of 100ms. If any latency issues arise, it is expected to happen around this period and the 35-second mark, since the trace is repeated.

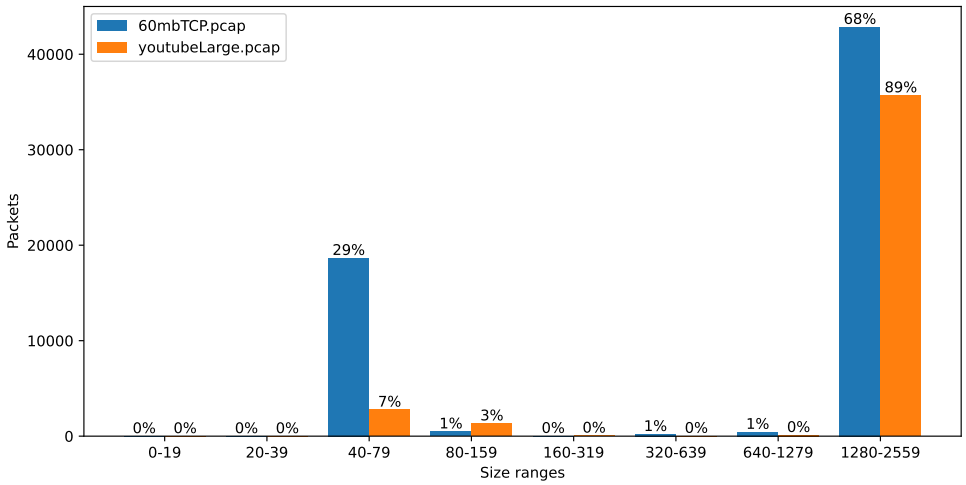

Figure 4.3: Packet size distribution shown of the two `pcap` traces used to evaluate Ericload and UP performance.

4.4 Inter-arrival times and long-range dependency

As mentioned in Section 2.2.1, network traffic comes in fractal-like bursts. Therefore, even when replaying the same traffic in the simulator, we should expect this characteristic to still hold. Further, since long-range dependency is preserved irrespective of scaling in space in time [12, 13], we should also expect long-range effects to be present when correctly repeating and scaling the traffic ourselves using the simulator. Theoretical calculations will be performed on the `60mbTCP.pcap` to present Ericloads potential effects on LRD. First, the inter-arrival times of the packets from the trace will be gathered in a list simply by calculating the timestamp difference of each packet. This list will then be plotted as a histogram of the inter-arrival times. Due to the properties of network traffic, we expect to see an exponential distribution in the times, with higher amounts closer to zero seconds inter-arrival times. The same list will then be used to performed calculation on

the LRD properties using Eq. 2.2. This will be plotted as a graph, where we expect to see a trend toward infinity in order for the data to showcase LRD properties.

The calculations will be performed in four scenarios: using the original trace as it is, scaling the trace 100 times, repeating the trace 100 times sequentially, and repeating and scaling the trace 100 each. Repetition is replaying the same trace again once it has finished replaying in its entirety. Since the inter-arrival times remain the same, this is the same as copying the above-mentioned list by a set factor and appending these smaller lists into a bigger one to perform calculations on. The `60mbTCP.pcap` trace used is a recording of downloading a 60mb file from start to finish, and since the repetitions represents repeating this scenario 100 times over, we should expect to see no direct impact on LRD. Scaling on the other hand is seen as introducing more of the same, already present flows, to the traffic. The effects of this type of scaling is expected to occur immediately for every single packet sent from the trace, since this is essentially sending a copy of the same packet at the same point in time. For our calculations, the scaling will be represented as adding more packets to the list with zero seconds inter-arrival times. Here we should see a negative impact in both plots. The exponential distribution should see a larger initial spike, and LRD should start converging to a specific value, indicating that the traffic no longer holds the typically seen properties of network traffic.

5
Results

This section presents the results obtained during the thesis. The first section will present a confirmation that the new replay functionality of Ericload works as intended. The later sections will present the results obtained by performance testing this replay functionality together with model-based data. The chapter will then conclude by presenting theoretical analysis done on the inter-arrival times of packets from a specified trace.

5.1 Replay verification

This section presents the results when running both the `pcap` traces mentioned in Section 4.2. Both tests have two traces each: one created by Ericload when reconstructing packets before sending them out to the UP, and one for packets received by the UP. It is important to keep in mind that the timestamps of the packets are presented as the *time elapsed between each shown packet*, and not as the usual recorded time of capture.

5.1.1 UDP trace

The UDP trace used was set to be repeated once, expected to show a total of 20 replayed packets. Figure 5.1 shows the trace constructed by Ericload during runtime when sending packets to the UP. The first two packets show the PFCP connection Ericload establishes with the UP before sending out actual traffic. The rest of the packets shown are the replayed packets from the original trace. The packets have different protocols depending on their direction, where packets in the uplink direction (UE to PDN) are wrapped with an GTP header, whereas packets in the downlink direction have not received this extra wrapping. Similarly, Figure 5.2 shows the incoming packets to Ericload as received by the UP. Here, one extra fragmented packet shows up in the beginning of the PFCP establishment. The rest are translated packets from the original trace, as identified by their source and destination addresses. One thing to note here is that this trace has had several irrelevant packages filtered out, hence why the numbering of the packets seem odd. Finally, the visible packet protocols for this trace is the reverse of Figure 5.1. This is due to the packets now traveling in the opposite direction, i.e., uplink packets being regarded as downlink packets and vice versa.

No.	Time	Source	Sport	Destination	Dport	Protocol	Length
1	0.000000	8.20.0.0	8805	108.2.0.1	8805	PFCP	81
2	1.152470	8.20.0.0	8805	108.2.0.1	8805	PFCP	267
3	2.248439	16.0.0.1	55001	194.0.0.0	80	GTP <UDP>	178
4	0.000047	194.0.0.1	80	16.0.0.1	55001	UDP	138
5	0.000023	16.0.0.1	55001	194.0.0.0	80	GTP <UDP>	178
6	0.000027	194.0.0.1	80	16.0.0.1	55001	UDP	138
7	0.000022	16.0.0.1	55001	194.0.0.0	80	GTP <UDP>	178
8	0.000028	194.0.0.1	80	16.0.0.1	55001	UDP	138
9	0.000029	16.0.0.1	55001	194.0.0.0	80	GTP <UDP>	178
10	0.000034	194.0.0.1	80	16.0.0.1	55001	UDP	138
11	0.000022	16.0.0.1	55001	194.0.0.0	80	GTP <UDP>	178
12	0.000027	194.0.0.1	80	16.0.0.1	55001	UDP	138
13	0.000023	16.0.0.1	55001	194.0.0.0	80	GTP <UDP>	178
14	0.000027	194.0.0.1	80	16.0.0.1	55001	UDP	138
15	0.000022	16.0.0.1	55001	194.0.0.0	80	GTP <UDP>	178
16	0.000031	194.0.0.1	80	16.0.0.1	55001	UDP	138
17	0.000022	16.0.0.1	55001	194.0.0.0	80	GTP <UDP>	178
18	0.000027	194.0.0.1	80	16.0.0.1	55001	UDP	138
19	0.000022	16.0.0.1	55001	194.0.0.0	80	GTP <UDP>	178
20	0.000027	194.0.0.1	80	16.0.0.1	55001	UDP	138
21	0.000022	16.0.0.1	55001	194.0.0.0	80	GTP <UDP>	178
22	0.000028	194.0.0.1	80	16.0.0.1	55001	UDP	138

Figure 5.1: Wireshark output showing packets created by Ericload the moment before being sent to the UP. Filtered for relevant UDP and PFCP packets. Timestamps are shown as time passed between each displayed packet.

No.	Time	Source	Sport	Destination	Dport	Protocol	Length
21	0.000000	108.2.0.1		8.20.0.0		IPv4	1518
22	0.000341	108.2.0.1	8805	8.20.0.0	8805	PFCP	1380
24	0.158454	108.2.0.1	8805	8.20.0.0	8805	PFCP	161
27	2.049734	16.0.0.1	55001	194.0.0.0	80	UDP	138
28	0.000043	16.0.0.1	55001	194.0.0.0	80	UDP	138
29	0.000000	16.0.0.1	55001	194.0.0.0	80	UDP	138
30	0.000000	16.0.0.1	55001	194.0.0.0	80	UDP	138
31	0.000000	16.0.0.1	55001	194.0.0.0	80	UDP	138
32	0.000000	16.0.0.1	55001	194.0.0.0	80	UDP	138
33	0.000000	16.0.0.1	55001	194.0.0.0	80	UDP	138
34	0.000054	194.0.0.1	80	16.0.0.1	55001	GTP <UDP>	174
35	0.000000	194.0.0.1	80	16.0.0.1	55001	GTP <UDP>	174
36	0.000085	194.0.0.1	80	16.0.0.1	55001	GTP <UDP>	174
37	0.000000	194.0.0.1	80	16.0.0.1	55001	GTP <UDP>	174
38	0.000000	194.0.0.1	80	16.0.0.1	55001	GTP <UDP>	174
39	0.000000	194.0.0.1	80	16.0.0.1	55001	GTP <UDP>	174
40	0.000000	194.0.0.1	80	16.0.0.1	55001	GTP <UDP>	174
41	0.000000	194.0.0.1	80	16.0.0.1	55001	GTP <UDP>	174
42	0.000000	194.0.0.1	80	16.0.0.1	55001	GTP <UDP>	174
43	0.000000	194.0.0.1	80	16.0.0.1	55001	GTP <UDP>	174
44	0.000003	16.0.0.1	55001	194.0.0.0	80	UDP	138
45	0.000000	16.0.0.1	55001	194.0.0.0	80	UDP	138
46	0.000050	16.0.0.1	55001	194.0.0.0	80	UDP	138

Figure 5.2: Wireshark output showing captured UP traffic by Ericload. Filtered for relevant UDP and PFCP packets. Timestamps are shown as time passed between each displayed packet.

5.1.2 TCP trace

The TCP trace used was set to be scaled by a factor of two, expected to show a total of 40 replayed packets. Figure 5.3 shows the trace constructed by Ericload during runtime when sending packets to the UP. The first two packets show the PFCP connection Ericload establishes with the UP before sending out actual traffic. We note that after these packets, all the 40 expected packets do indeed show up. Further, an added flow between UE_0 (local address 16.0.0.1) and PDN_1 (local address

194.0.0.1) is visible. This comes from introducing the scaling factor, which for this replay doubles the amount of packets produced by Ericload compared to the original trace. These extra packets are injected right after the packet of the first flow (UE_0 to PDN_0 in this case) has been constructed. For example, we notice that packet number six is an extra, immediate, injection of a packet to be sent from the UE to PDN before the simulator continues on with replaying the original trace. The receiving trace from Figure 5.4 shows, like in the above UDP example, incoming packet to Ericload received from the UP. We also noted the fragmented PFCP packet in the beginning here before the rest of the remaining trace is presented. In further similarity with the receiving UDP trace of Figure 5.2, the skewed identification number of the packets are due to filtering out irrelevant packets. Finally, in both figures, we notice the different added protocol wrappings of the packets depending on their direction of travel.

No.	Time	Source	Sport	Destination	Dport	Protocol	Length
1	0.000000	8.20.0.0	8805	108.2.0.1	8805	PFCP	81
2	1.147836	8.20.0.0	8805	108.2.0.1	8805	PFCP	267
3	2.207626	16.0.0.1	62497	194.0.0.0	9418	GTP <TCP>	118
4	0.000045	16.0.0.1	62497	194.0.0.1	9418	GTP <TCP>	118
5	0.274589	194.0.0.0	9418	16.0.0.1	62497	TCP	70
6	0.000021	194.0.0.1	9418	16.0.0.1	62497	TCP	70
7	0.008347	16.0.0.1	62497	194.0.0.0	9418	GTP <TCP>	100
8	0.000026	16.0.0.1	62497	194.0.0.1	9418	GTP <TCP>	100
9	0.000018	16.0.0.1	62497	194.0.0.0	9418	GTP <TCP>	178
10	0.000029	16.0.0.1	62497	194.0.0.1	9418	GTP <TCP>	178
11	0.326211	194.0.0.0	9418	16.0.0.1	62497	TCP	58
12	0.000015	194.0.0.1	9418	16.0.0.1	62497	TCP	58
13	0.007063	194.0.0.0	9418	16.0.0.1	62497	TCP	144
14	0.000028	194.0.0.1	9418	16.0.0.1	62497	TCP	144
15	0.001562	194.0.0.0	9418	16.0.0.1	62497	TCP	1418
16	0.000219	194.0.0.1	9418	16.0.0.1	62497	TCP	1418
17	0.006567	16.0.0.1	62497	194.0.0.0	9418	GTP <TCP>	100
18	0.000023	16.0.0.1	62497	194.0.0.1	9418	GTP <TCP>	100
19	0.002540	16.0.0.1	62497	194.0.0.0	9418	GTP <TCP>	100
20	0.000022	16.0.0.1	62497	194.0.0.1	9418	GTP <TCP>	100
21	0.316749	194.0.0.0	9418	16.0.0.1	62497	TCP	878
22	0.000136	194.0.0.1	9418	16.0.0.1	62497	TCP	878
23	28.586971	16.0.0.1	62497	194.0.0.0	9418	GTP <TCP>	100
24	0.000042	16.0.0.1	62497	194.0.0.1	9418	GTP <TCP>	100
25	0.141500	16.0.0.1	62497	194.0.0.0	9418	GTP <TCP>	179
26	0.000037	16.0.0.1	62497	194.0.0.1	9418	GTP <TCP>	179
27	0.315078	194.0.0.0	9418	16.0.0.1	62497	TCP	58
28	0.000020	194.0.0.1	9418	16.0.0.1	62497	TCP	58
29	0.006149	194.0.0.0	9418	16.0.0.1	62497	TCP	268
30	0.000047	194.0.0.1	9418	16.0.0.1	62497	TCP	268
31	0.005251	16.0.0.1	62497	194.0.0.0	9418	GTP <TCP>	100
32	0.000023	16.0.0.1	62497	194.0.0.1	9418	GTP <TCP>	100
33	0.091478	16.0.0.1	62497	194.0.0.0	9418	GTP <TCP>	179
34	0.000035	16.0.0.1	62497	194.0.0.1	9418	GTP <TCP>	179
35	0.356370	194.0.0.0	9418	16.0.0.1	62497	TCP	58
36	0.000018	194.0.0.1	9418	16.0.0.1	62497	TCP	58
37	4.934925	16.0.0.1	62497	194.0.0.0	9418	GTP <TCP>	100
38	0.000031	16.0.0.1	62497	194.0.0.1	9418	GTP <TCP>	100
39	0.322741	194.0.0.0	9418	16.0.0.1	62497	TCP	58
40	0.000015	194.0.0.1	9418	16.0.0.1	62497	TCP	58
41	0.005142	16.0.0.1	62497	194.0.0.0	9418	GTP <TCP>	100
42	0.000022	16.0.0.1	62497	194.0.0.1	9418	GTP <TCP>	100

Figure 5.3: Wireshark output showing packets created by Ericload the moment before being sent to the UP. Filtered for relevant TCP and PFCP packets. Timestamps are shown as time passed between each displayed packet.

No.	Time	Source	Sport	Destination	Dport	Protocol	Length
22	0.000000	108.2.0.1		8.20.0.0		IPv4	1518
23	0.000303	108.2.0.1	8805	8.20.0.0	8805	PFCP	1380
24	0.189544	108.2.0.1	8805	8.20.0.0	8805	PFCP	161
42	2.048732	16.0.0.1	62497	194.0.0.0	9418	TCP	78
43	0.000033	16.0.0.1	62497	194.0.0.1	9418	TCP	78
45	0.275242	194.0.0.1	9418	16.0.0.1	62497	GTP <TCP>	106
46	0.000033	194.0.0.0	9418	16.0.0.1	62497	GTP <TCP>	106
47	0.008371	16.0.0.1	62497	194.0.0.0	9418	TCP	60
48	0.000011	16.0.0.1	62497	194.0.0.1	9418	TCP	60
49	0.000004	16.0.0.1	62497	194.0.0.0	9418	TCP	138
50	0.000018	16.0.0.1	62497	194.0.0.1	9418	TCP	138
51	0.326302	194.0.0.0	9418	16.0.0.1	62497	GTP <TCP>	94
52	0.000032	194.0.0.1	9418	16.0.0.1	62497	GTP <TCP>	94
53	0.006981	194.0.0.0	9418	16.0.0.1	62497	GTP <TCP>	180
54	0.000030	194.0.0.1	9418	16.0.0.1	62497	GTP <TCP>	180
55	0.001529	194.0.0.0	9418	16.0.0.1	62497	GTP <TCP>	1454
56	0.000032	194.0.0.1	9418	16.0.0.1	62497	GTP <TCP>	1454
57	0.006712	16.0.0.1	62497	194.0.0.0	9418	TCP	60
58	0.000032	16.0.0.1	62497	194.0.0.1	9418	TCP	60
59	0.002450	16.0.0.1	62497	194.0.0.0	9418	TCP	60
60	0.000032	16.0.0.1	62497	194.0.0.1	9418	TCP	60
61	0.316801	194.0.0.0	9418	16.0.0.1	62497	GTP <TCP>	914
62	0.000031	194.0.0.1	9418	16.0.0.1	62497	GTP <TCP>	914
171	28.587125	16.0.0.1	62497	194.0.0.0	9418	TCP	60
172	0.000025	16.0.0.1	62497	194.0.0.1	9418	TCP	60
173	0.141482	16.0.0.1	62497	194.0.0.0	9418	TCP	139
174	0.000033	16.0.0.1	62497	194.0.0.1	9418	TCP	139
192	0.315083	194.0.0.0	9418	16.0.0.1	62497	GTP <TCP>	94
193	0.000007	194.0.0.1	9418	16.0.0.1	62497	GTP <TCP>	94
195	0.006121	194.0.0.1	9418	16.0.0.1	62497	GTP <TCP>	304
196	0.000032	194.0.0.0	9418	16.0.0.1	62497	GTP <TCP>	304
197	0.005277	16.0.0.1	62497	194.0.0.0	9418	TCP	60
198	0.000034	16.0.0.1	62497	194.0.0.1	9418	TCP	60
199	0.091579	16.0.0.1	62497	194.0.0.0	9418	TCP	139
200	0.000034	16.0.0.1	62497	194.0.0.1	9418	TCP	139
201	0.356345	194.0.0.0	9418	16.0.0.1	62497	GTP <TCP>	94
202	0.000032	194.0.0.1	9418	16.0.0.1	62497	GTP <TCP>	94
225	4.936167	16.0.0.1	62497	194.0.0.1	9418	TCP	60
226	0.000386	16.0.0.1	62497	194.0.0.0	9418	TCP	60
227	0.321113	194.0.0.0	9418	16.0.0.1	62497	GTP <TCP>	94
228	0.000032	194.0.0.1	9418	16.0.0.1	62497	GTP <TCP>	94
229	0.005114	16.0.0.1	62497	194.0.0.1	9418	TCP	60
230	0.000008	16.0.0.1	62497	194.0.0.0	9418	TCP	60

Figure 5.4: Wireshark output showing captured UP traffic by Ericload. Filtered for relevant TCP and PFCP packets. Timestamps are shown as time passed between each displayed packet.

5.2 Performance

This section presents results from measurements done on the UP using Ericload. First, we will go through finding an optimal rate of the UP by using steady rate traffic. Secondly, results are shown from only replaying the `pcap` files presented in Section 4.3.2 and how the UP handles this. Finally, the results from combining both trace-based and model-based traffic will be shown.

5.2.1 UP performance

Rate (pps)	1 flow			10 flows			100 flows		
	Ratio	Gbps	Delay	Ratio	Gbps	Delay	Ratio	Gbps	Delay
100 000	0.0	1.16	0.35	0.02	1.14	0.27	0.02	1.14	0.26
110 000	0.0	1.28	0.5	0.02	1.26	0.35	0.02	1.26	0.3
120 000	0.08	1.28	97.43	0.02	1.37	0.28	0.02	1.37	0.3
130 000	0.11	1.29	147.98	0.02	1.49	0.31	0.02	1.49	0.31
140 000	0.1	1.4	124.92	0.02	1.6	0.33	0.02	1.6	0.34
150 000	0.15	1.48	127.26	0.02	1.71	0.35	0.02	1.72	0.4
160 000	0.21	1.47	126.91	0.02	1.83	0.39	0.02	1.82	0.87
170 000	0.27	1.45	123.78	0.02	1.93	0.49	0.02	1.94	0.59
180 000	0.34	1.33	124.97	0.02	2.04	0.64	0.02	2.04	0.91
190 000	0.33	1.47	123.91	0.03	2.14	0.9	0.03	2.16	1.27
200 000	0.36	1.5	122.11	0.04	2.23	1.27	0.04	2.23	2.1
Uplink and Downlink traffic									
100 000	0.0	1.1	0.38	0.02	1.13	0.36	0.02	1.13	0.34
110 000	0.0	1.21	0.56	0.02	1.24	0.39	0.02	1.24	0.33
120 000	0.03	1.28	19.16	0.02	1.36	0.32	0.02	1.35	0.37
130 000	0.04	1.43	58.52	0.02	1.47	0.35	0.02	1.47	0.38
140 000	0.08	1.48	117.42	0.02	1.58	0.33	0.02	1.58	0.37
150 000	0.14	1.49	120.07	0.02	1.69	0.5	0.02	1.69	0.54
160 000	0.19	1.49	108.16	0.02	1.8	0.39	0.02	1.81	0.42
170 000	0.23	1.5	114.47	0.02	1.91	0.51	0.02	1.91	0.74
180 000	0.3	1.44	116.66	0.03	2.01	0.71	0.03	2.01	0.95
190 000	0.31	1.52	114.88	0.03	2.11	0.97	0.04	2.09	1.81
200 000	0.35	1.49	120.06	0.06	2.16	2.35	0.06	2.15	2.66
Uplink traffic									
100 000	0.0	1.18	0.41	0.02	1.15	0.31	0.02	1.16	0.26
110 000	0.0	1.29	1.0	0.02	1.27	0.26	0.02	1.27	0.23
120 000	0.0	1.41	12.85	0.02	1.39	0.25	0.02	1.39	0.25
130 000	0.03	1.48	111.0	0.02	1.51	0.26	0.02	1.51	0.29
140 000	0.1	1.41	121.08	0.02	1.62	0.26	0.02	1.62	0.34
150 000	0.16	1.42	126.33	0.02	1.74	0.31	0.02	1.74	0.37
160 000	0.21	1.48	124.02	0.02	1.84	0.37	0.02	1.85	0.57
170 000	0.25	1.51	125.82	0.03	1.95	0.5	0.02	1.96	1.19
180 000	0.3	1.49	121.7	0.04	2.03	0.93	0.04	2.07	1.02
190 000	0.33	1.51	122.84	0.04	2.16	0.95	0.05	2.18	1.32
200 000	0.36	1.51	120.85	0.05	2.23	1.42	0.05	2.28	2.21
Downlink traffic									

Table 5.1: Loss ratio, incoming throughput and median millisecond delay of Ericload sending traffic to the UP for 60 seconds at different packets per second rates, flows and directional combinations.

To find the optimal performance of the UP in use, multiple steady rate tests were conducted as described in Section 4.3.1 that lasted 60 seconds each. A starting rate of 100k packets per second was chosen where the value was incremented by 10k at each new simulation, up until a max rate of 200k packets per second was hit. Table 5.1 shows the results of all these measurements, divided up into three main categories based on the direction of the traffic used. From this table we notice that the packet loss rate is below 2% on a single flow at around 110-120k packets per second, pushing around 1.2Gbps incoming throughput on average, before severely starting to drop in performance. The median delay also increases fast once the packet loss rate has been exceeded. Noticeable performance improvements appear once more flows are introduced. Here we see peaks of around 170k packets per second with a 2Gbps incoming throughput before performance degradation begins. The median delay is also lower compared to single flow traffic, but also begins to increase once the 2% packet loss rate is exceeded. Steady rate simulations using 10 flows seem to perform slightly better overall than 100 flows, and this flow amount was therefore used in later tests for model-based traffic when combining them with trace-based traffic.

5.2.2 Replay performance

Scale	Peak out	Peak in	Ratio	Gbps	Delay
1	5317	4874	0.0	0.26	0.71
2	10646	10310	0.0	0.52	0.71
3	15913	14903	0.05	0.74	0.37
4	21256	16725	0.2	0.8	0.5
5	26615	15684	0.39	0.77	0.67
`60mbTCP.pcap`					
1	2113	2421	0.0	0.01	16.7
2	4172	3131	0.0	0.03	1.03
3	6339	4721	0.0	0.04	0.55
4	8452	6422	0.0	0.05	0.47
5	10430	8269	0.01	0.06	0.79
6	12678	9336	0.01	0.08	0.82
7	14546	10760	0.04	0.09	1.25
8	16904	12600	0.06	0.1	1.3
9	19017	13349	0.09	0.1	1.53
10	22160	13247	0.11	0.12	1.86
`youtubeLarge.pcap`					

Table 5.2: Performance of the `pcap` files presented in Section 4.3.2. The peak values are the highest total amount of packets measured at a 100ms interval. Ratio represents the loss ratio, and Delay is the median millisecond delay of the whole simulation.

The first replay test cases were conducted on both `pcap` files presented in Section 4.3.2, where each file was replayed alone at incremental scaling values until roughly the same packet per second rates found in Table 5.1 were reached. For the UDP based capture file, this limit was around a scaling factor of 10, and five for the TCP based one. Table 5.2 shows the results of these measurements. Although a bit harder to conclude from the TCP trace, packet loss starts exceeding 2% where the peak of total outgoing packets sent from Ericload reaches around 12,5k using 100ms interval samples. Theoretically, scaling this to a whole second would yield 125k, which would be on par with the results seen in Table 5.1. Figure 5.5 and Figure 5.6 show detailed results from the first row of Table 5.2 for each file. Note that the peak amounts do not match the result from the table when accounting for the time intervals, due to the figures showing samples every second whereas the data from the tables are at 10 samples per second instead. Latency numbers seem to roughly coincide with the model-based traffic numbers, although slightly higher numbers appear in some instances, particularly for the replayed UDP trace. Here we also see an abnormally high median latency on the first test case compared to all others.

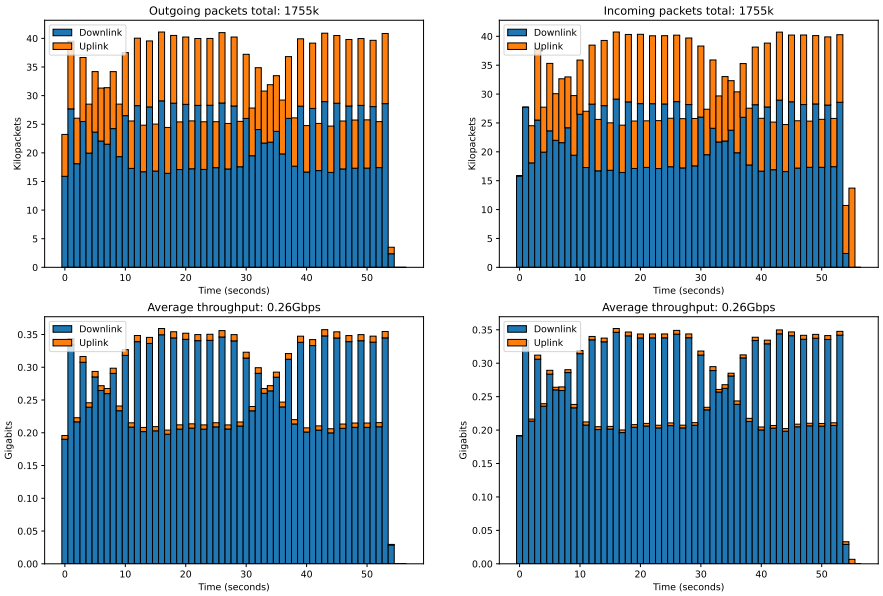

Figure 5.5: Measurements from replaying `60mbTCP.pcap` where the trace is repeated 28 times.

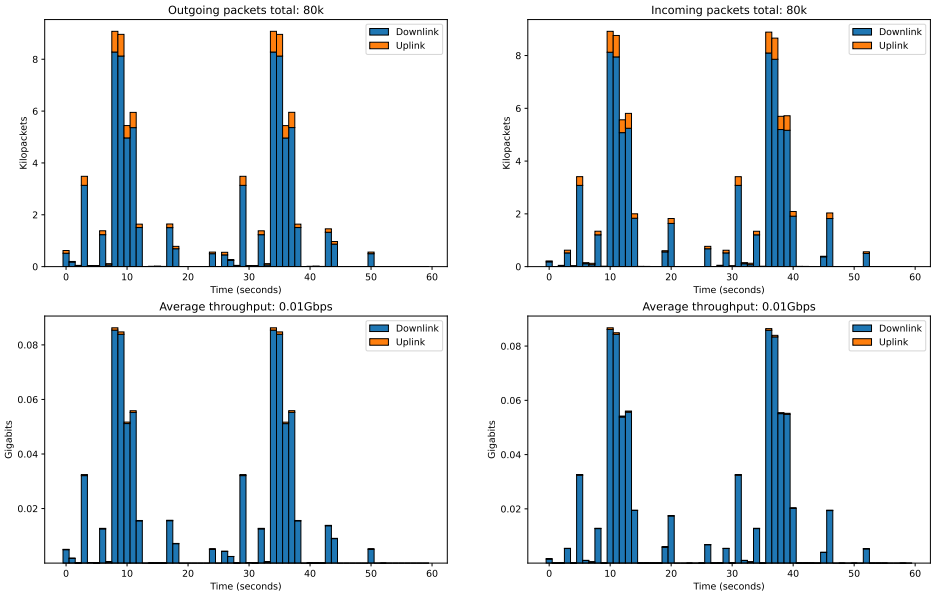

Figure 5.6: Measurements from replaying `youtubeLarge.pcap` where the trace is repeated twice.

TCP	1 scale			2 scale			3 scale		
UDP	Ratio	Gbps	Delay	Ratio	Gbps	Delay	Ratio	Gbps	Delay
1 scale	0.0	0.26	27.49	0.0	0.51	8.01	0.08	0.69	5.33
2 scale	0.0	0.28	27.11	0.0	0.52	8.68	0.06	0.72	5.94
3 scale	0.0	0.29	27.0	0.01	0.53	9.52	0.08	0.75	6.83
4 scale	0.0	0.3	27.18	0.01	0.56	10.44	0.07	0.74	6.98
5 scale	0.01	0.32	28.6	0.02	0.57	10.87	0.09	0.76	7.76
6 scale	0.01	0.33	31.12	0.03	0.56	11.25	0.09	0.74	8.13
7 scale	0.02	0.33	29.25	0.04	0.56	11.37	0.10	0.74	8.16
8 scale	0.04	0.34	29.19	0.05	0.59	12.06	0.11	0.78	8.74
9 scale	0.05	0.34	28.94	0.07	0.57	11.8	0.13	0.78	8.79
10 scale	0.07	0.35	28.1	0.08	0.57	11.62	0.13	0.75	8.48

Table 5.3: Loss ratio, throughput and median millisecond delay of Ericload replaying traffic from `pcap` files, in different combinations, to the UP for 60 seconds.

Table 5.3 shows test cases performed using different combinations of the two `pcap` files. For every column, the scale of `60mbTCP.pcap` increases by one, whereas for every row, the scale of `youtubeLarge.pcap` increases by one. The threshold of 2% packet loss is reached twice, one when the TCP trace has a scale factor of one and the UDP trace a scale factor of seven, and the other time when the TCP trace has a scale factor of two and the UDP trace a scale factor of five. Taking into account the outgoing peaks from Table 5.2 and combining these would not yield a correct representation of peaks from the combined replays, since their peaks might be spread

out. Upon closer inspection of the CSV file produced by Ericload for the two test cases mentioned above, however, we noticed peak values of around 19k outgoing packets and 20k outgoing packets respectively, when using a sampling frequency of 10 samples per second. For the median latency values, we see unusually high values compared to the test cases where traces were not combined.

5.2.3 Trace-based and model-based performance

The final test cases performed were a combination of all `pcap` test cases shown previously with the added model-based steady rates of 25k, 50k, and 100k UDP packets respectively, all using 10 flows. The only exception was the removal of the third scaling of the TCP `pcap` trace, since performing tests with this combination yielded a high packet loss already from the start. All the results can be seen in Table 5.4, Table 5.5 and Table 5.6. From the results, we see that the addition of steady rate on top of the already replayed `pcap` traffic, most of the time, does not affect the overall performance of the UP. The loss ratio increases by only a very small margin, around 1%-2%, in most cases. The biggest deviation to this is when a 100k packets per second of additional steady rate is applied to the test cases, where `60mbTCP.pcap` is scaled by a factor of two. Here we notice an increase in packet loss by 10% already in the first test case. When looking at the median delay, we notice two things. Firstly, adding only additional steady rate traffic in the uplink direction impacts the delay less than all other cases where downlink, or a combination of up- and downlink traffic, is introduced. Seeing the heavy downlink rates that are produced from both the `pcap` replays, when looking at Figure 5.5 and Figure 5.6, we can conclude that this results in a more evenly distributed directional traffic for the test cases. Secondly, this increase in delay is not being affected in any of the test cases where only `youtubeLarge.pcap` is being replayed together with the steady rate traffic.

Overall, it can be concluded that the UP handles the combination of model- and trace-based traffic well, with the best performance produced when trying to evenly distribute the amount of traffic flowing in both the uplink and downlink direction.

Scenario	Up- and Downlink			Uplink			Downlink		
	Ratio	Gbps	Delay	Ratio	Gbps	Delay	Ratio	Gbps	Delay
TCP1	0.01	0.55	34.72	0.01	0.53	2.01	0.01	0.54	69.66
TCP2	0.01	0.78	15.82	0.01	0.77	1.24	0.01	0.79	30.46
UDP1	0.02	0.3	0.22	0.02	0.3	0.2	0.0	0.3	0.2
UDP2	0.02	0.31	0.22	0.02	0.31	0.23	0.02	0.31	0.2
UDP3	0.0	0.33	0.26	0.02	0.32	0.24	0.02	0.33	0.24
UDP4	0.02	0.34	0.3	0.01	0.33	0.29	0.02	0.34	0.3
UDP5	0.02	0.35	0.42	0.02	0.35	0.4	0.02	0.35	0.4
UDP6	0.02	0.36	0.47	0.02	0.36	0.45	0.02	0.36	0.5
UDP7	0.03	0.37	0.58	0.03	0.37	0.53	0.03	0.37	0.63
UDP8	0.04	0.38	0.76	0.05	0.37	0.77	0.04	0.38	0.82
UDP9	0.02	0.39	1.02	0.05	0.38	0.94	0.06	0.39	1.08
UDP10	0.07	0.39	1.13	0.08	0.39	1.16	0.07	0.39	1.28
T1 × U1	0.01	0.55	16.15	0.01	0.54	2.67	0.01	0.55	29.71
T1 × U2	0.01	0.56	15.71	0.0	0.57	3.05	0.01	0.57	28.59
T1 × U3	0.01	0.57	15.37	0.01	0.57	3.38	0.0	0.59	27.38
T1 × U4	0.02	0.58	15.39	0.01	0.58	3.69	0.01	0.59	26.94
T1 × U5	0.01	0.6	15.1	0.02	0.59	3.92	0.01	0.6	26.49
T1 × U6	0.04	0.62	15.52	0.02	0.6	4.23	0.02	0.61	26.86
T1 × U7	0.03	0.61	15.19	0.03	0.61	4.35	0.03	0.61	26.49
T1 × U8	0.04	0.62	15.06	0.05	0.61	4.42	0.04	0.62	26.22
T1 × U9	0.05	0.62	15.0	0.05	0.62	4.62	0.05	0.63	25.78
T1 × U10	0.07	0.62	15.03	0.06	0.63	4.62	0.07	0.63	25.77
T2 × U1	0.01	0.79	35.77	0.01	0.79	5.22	0.01	0.8	68.1
T2 × U2	0.01	0.81	33.46	0.01	0.8	5.79	0.01	0.81	63.21
T2 × U3	0.01	0.82	31.61	0.01	0.81	6.1	0.09	0.74	58.86
T2 × U4	0.02	0.82	30.59	0.02	0.82	6.47	0.02	0.83	55.95
T2 × U5	0.01	0.85	29.13	0.03	0.82	6.7	0.09	0.77	53.44
T2 × U6	0.04	0.83	29.46	0.04	0.82	7.26	0.04	0.84	53.61
T2 × U7	0.05	0.84	28.32	0.05	0.83	7.32	0.05	0.84	51.87
T2 × U8	0.06	0.84	27.89	0.06	0.83	7.39	0.06	0.85	50.78
T2 × U9	0.07	0.84	27.45	0.07	0.84	7.48	0.05	0.88	49.35
T2 × U10	0.08	0.84	27.32	0.09	0.83	7.73	0.08	0.85	48.59

Table 5.4: Loss ratio, throughput and median millisecond delay of Ericload replaying traffic from `pcap` files, in different combinations, to the UP for 60 seconds. This includes an added steady rate of 25 000 packets per second UDP traffic using 10 flows.

Scenario	Up- and Downlink			Uplink			Downlink		
	Ratio	Gbps	Delay	Ratio	Gbps	Delay	Ratio	Gbps	Delay
TCP1	0.01	0.82	23.33	0.01	0.81	0.56	0.01	0.83	49.13
TCP2	0.02	1.06	13.66	0.02	1.05	0.51	0.02	1.07	27.72
UDP1	0.02	0.58	0.21	0.02	0.58	0.25	0.02	0.59	0.19
UDP2	0.02	0.6	0.23	0.02	0.59	0.25	0.02	0.6	0.23
UDP3	0.02	0.61	0.27	0.02	0.6	0.28	0.02	0.62	0.25
UDP4	0.02	0.62	0.3	0.02	0.62	0.32	0.02	0.63	0.29
UDP5	0.02	0.63	0.39	0.02	0.63	0.43	0.02	0.64	0.4
UDP6	0.0	0.64	0.52	0.0	0.64	0.51	0.03	0.65	0.47
UDP7	0.04	0.65	0.59	0.0	0.64	0.59	0.03	0.66	0.63
UDP8	0.04	0.66	0.77	0.04	0.65	0.82	0.08	0.66	0.75
UDP9	0.06	0.66	0.92	0.06	0.65	0.89	0.01	0.68	0.97
UDP10	0.04	0.67	1.11	0.08	0.65	1.07	0.06	0.67	1.12
T1 × U1	0.04	0.84	23.14	0.01	0.83	1.76	0.01	0.84	47.55
T1 × U2	0.01	0.85	22.09	0.01	0.84	2.11	0.01	0.86	44.43
T1 × U3	0.01	0.86	21.22	0.01	0.85	2.35	0.01	0.87	42.33
T1 × U4	0.02	0.87	20.56	0.02	0.86	2.79	0.02	0.88	40.5
T1 × U5	0.02	0.88	20.09	0.02	0.87	2.75	0.02	0.89	39.28
T1 × U6	0.03	0.88	20.12	0.0	0.89	3.19	0.03	0.9	39.22
T1 × U7	0.04	0.89	19.51	0.04	0.88	3.15	0.01	0.92	38.32
T1 × U8	0.05	0.89	19.37	0.05	0.88	3.21	0.05	0.9	37.73
T1 × U9	0.06	0.89	19.15	0.09	0.9	3.39	0.06	0.91	36.7
T1 × U10	0.08	0.89	19.22	0.05	0.9	3.53	0.07	0.91	36.2
T2 × U1	0.03	1.06	13.82	0.03	1.05	1.24	0.03	1.08	27.34
T2 × U2	0.03	1.08	13.43	0.03	1.06	1.5	0.03	1.09	26.19
T2 × U3	0.02	1.11	13.25	0.04	1.07	1.75	0.03	1.1	25.59
T2 × U4	0.04	1.09	12.98	0.01	1.11	1.92	0.02	1.13	25.13
T2 × U5	0.02	1.13	12.84	0.04	1.09	2.04	0.04	1.11	24.52
T2 × U6	0.03	1.13	12.91	0.05	1.1	2.23	0.04	1.12	24.38
T2 × U7	0.06	1.11	12.84	0.06	1.09	2.59	0.06	1.12	24.14
T2 × U8	0.07	1.1	12.92	0.05	1.12	2.43	0.07	1.12	24.0
T2 × U9	0.08	1.11	12.95	0.07	1.1	2.48	0.07	1.12	23.77
T2 × U10	0.08	1.11	12.63	0.1	1.09	2.71	0.1	1.11	24.24

Table 5.5: Loss ratio, throughput and median millisecond delay of Ericload replaying traffic from `pcap` files, in different combinations, to the UP for 60 seconds. This includes an added steady rate of 50 000 packets per second UDP traffic using 10 flows.

Scenario	Up- and Downlink			Uplink			Downlink		
	Ratio	Gbps	Delay	Ratio	Gbps	Delay	Ratio	Gbps	Delay
TCP1	0.02	1.39	11.74	0.02	1.37	0.37	0.02	1.4	25.48
TCP2	0.09	1.53	10.09	0.12	1.49	0.86	0.09	1.55	20.88
UDP1	0.02	1.15	0.26	0.02	1.14	0.31	0.02	1.17	0.24
UDP2	0.02	1.17	0.28	0.02	1.15	0.53	0.02	1.18	0.22
UDP3	0.02	1.18	0.33	0.02	1.17	0.36	0.02	1.19	0.3
UDP4	0.02	1.19	0.4	0.02	1.18	0.43	0.02	1.2	0.33
UDP5	0.0	1.21	0.47	0.03	1.18	0.6	0.03	1.21	0.44
UDP6	0.03	1.2	0.55	0.03	1.19	0.71	0.0	1.23	0.52
UDP7	0.0	1.21	0.62	0.04	1.19	0.69	0.04	1.22	0.6
UDP8	0.05	1.21	0.96	0.05	1.19	0.81	0.05	1.22	0.73
UDP9	0.06	1.21	0.87	0.07	1.19	1.0	0.06	1.23	0.98
UDP10	0.07	1.21	1.02	0.08	1.19	1.18	0.07	1.22	1.05
T1 × U1	0.02	1.4	11.69	0.02	1.38	0.65	0.0	1.44	24.91
T1 × U2	0.02	1.41	11.42	0.02	1.4	0.81	0.02	1.43	24.13
T1 × U3	0.02	1.42	11.11	0.02	1.4	0.94	0.02	1.44	23.24
T1 × U4	0.03	1.43	10.94	0.03	1.41	1.09	0.03	1.44	22.42
T1 × U5	0.0	1.45	10.87	0.01	1.42	1.29	0.04	1.45	22.17
T1 × U6	0.04	1.43	10.88	0.05	1.4	1.33	0.04	1.45	22.3
T1 × U7	0.05	1.43	10.74	0.06	1.4	1.43	0.05	1.45	21.9
T1 × U8	0.06	1.43	10.69	0.04	1.41	1.53	0.06	1.45	21.63
T1 × U9	0.07	1.43	10.7	0.08	1.4	1.61	0.07	1.44	21.53
T1 × U10	0.08	1.43	10.74	0.09	1.4	1.68	0.08	1.45	21.29
T2 × U1	0.09	1.54	10.18	0.13	1.47	1.34	0.09	1.56	20.69
T2 × U2	0.1	1.54	10.09	0.13	1.49	1.34	0.1	1.56	20.18
T2 × U3	0.11	1.53	9.94	0.14	1.49	1.46	0.1	1.57	19.61
T2 × U4	0.11	1.55	9.8	0.14	1.5	1.53	0.12	1.55	19.94
T2 × U5	0.12	1.55	9.92	0.15	1.49	1.74	0.12	1.56	19.35
T2 × U6	0.13	1.54	10.04	0.16	1.48	1.81	0.12	1.57	19.43
T2 × U7	0.13	1.55	9.91	0.14	1.51	1.85	0.16	1.49	20.86
T2 × U8	0.12	1.57	9.84	0.17	1.48	1.89	0.14	1.56	19.21
T2 × U9	0.13	1.56	9.98	0.18	1.48	2.07	0.12	1.59	18.99
T2 × U10	0.13	1.54	9.97	0.19	1.48	2.08	0.15	1.57	19.0

Table 5.6: Loss ratio, throughput and median millisecond delay of Ericload replaying traffic from `pcap` files, in different combinations, to the UP for 60 seconds. This includes an added steady rate of 100 000 packets per second UDP traffic using 10 flows.

5.3 Inter-arrival times and long-range dependency

Theoretical calculations were performed on the inter-arrival times of the `60mbTCP.pcap` file to show how scaling and repetition affects properties like LRD. Figure 5.7 shows the results from this. These are separated into two columns: the left shows the histogram och the inter-arrival times of packets, while the right plots the result of

using Eq. 2.2 to calculate the LRD.

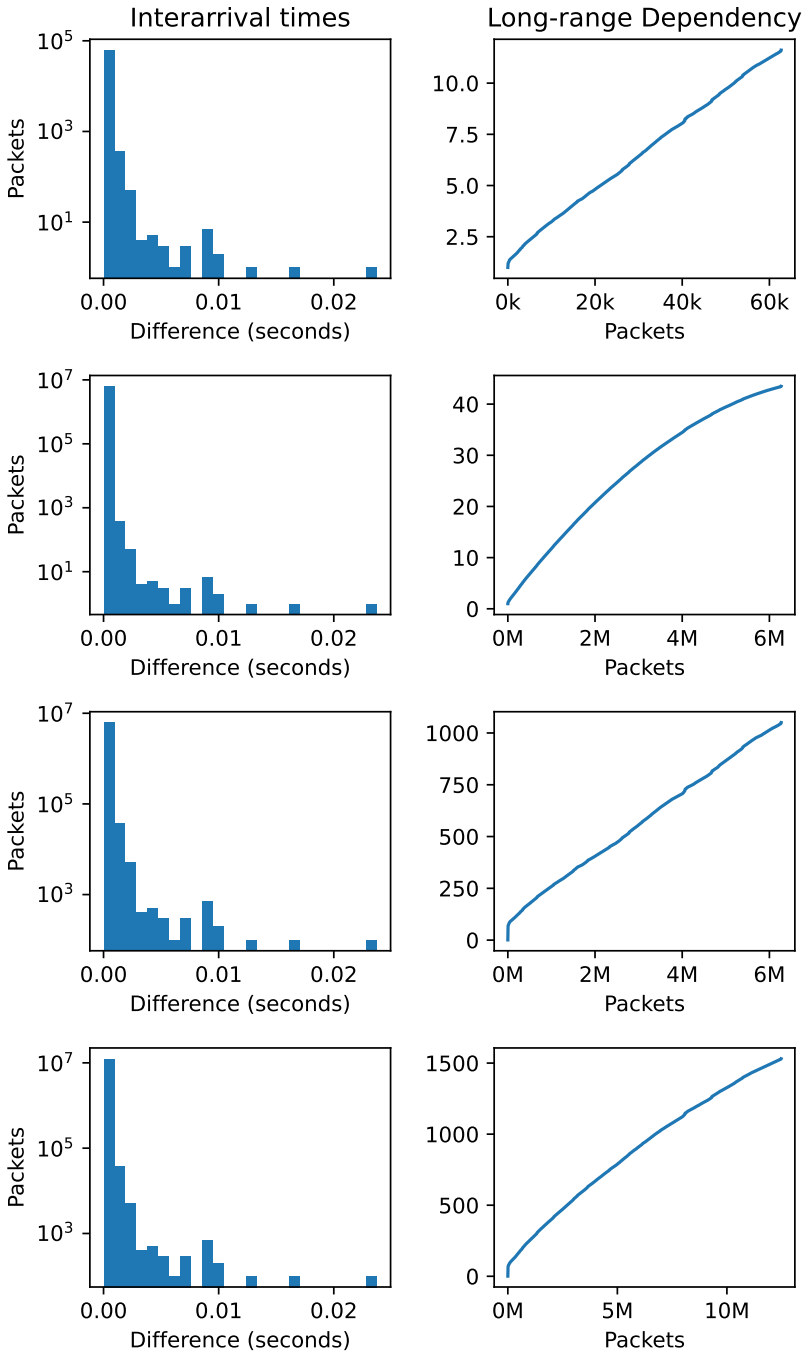

Figure 5.7: Histogram of inter-arrival times and LRD calculation of `60mbTCP.pcap` with the combinations: no repetitions or scaling, scaled 100 times, repeated 100 times, and repeated and scaled 100 times each.

The top two figures shows calculations performed on the original file, as is. We note the exponential distribution in the histogram when counting and plotting the inter-arrival times as well as the trend toward infinity when plotting the LRD. The second rows shows the results from scaling the trace by a factor of 100. Since scaling using this approach in practice results primarily in sending extra packets instantly, we notice the immediate effects of faking the data this way: the histogram is more heavily skewed toward packets with zero seconds inter-arrival time, and the LRD is beginning to diverge away from infinity, showing that the data no longer holds the time-dependent properties of typical network data. In the third row the calculations were performed as if the original file was repeated 100 times sequentially. Here we note the similarities with the top results, indicating that repeating a trace this way is a realistic form of scaling data. The final row shows the the results of combining both the scaling and repetition. Again we see the negative effects scaling has on traffic properties, both in the skewed histogram and the convergence bend in LRD. This time the effects are not as severe, likely due to the repetitions being introduced.

6
Discussion

This chapter presents an analysis of the results found in Chapter 5.

6.1 Replaying traffic

Looking at the results from Section 5.1, we can conclude that Ericload successfully manages to perform trace-based traffic replays on a basic level. We see that it not only is able to scale in the time domain by repeating the same input trace any given amount of times, but it is also capable of scaling in the number of flows a trace will be used. This latter case stems from it being able to correctly translate the source and destination addresses found in the trace to local ones on the network, by utilizing a reference table of addresses it creates during runtime, as described back in Section 3.3. Further, it is correctly able to abide by the time constraints in the `pcap` files it is trying to reproduce, as shown in Figure 5.1 and Figure 5.3. This produces a correctly ordered packet trace that is sent to the UP, as well ensuring correct time delays between packets. The slight exception to this is when looking at the results of the UDP trace from Figure 5.1 and Figure 5.2. In the former figure, we see an increase in the time between each packet by a factor of around 20 when comparing it to the original trace found in Figure 4.1. An explanation for this might be in the already low time numbers between the packets in the original trace. Having a one to three microseconds delay between each packet is unrealistic and might suggest that this trace might have been recorded on a local network where speeds exceed those found in the real world. These short time difference might also help explain why the packets came out of order in the latter figure shown, instead of the expected ping-like scenario between the UE and the PDN. The packets are not only separated into groups of what direction on the network they were sent, but also have a zero-second time difference between each other.

When looking at the trace produced for the replayed TCP trace found in Figure 5.4, the above-mentioned timing problems are not present. The major reason for this is the more realistic time between the packets in this trace, compared to the UDP trace. Not only is the trace in the completely expected order, but the difference in time between packets is similar to its original trace. The only exception to this is in some scaled packets, for example in the last two packets of the trace. Here the order should be swapped when compared to the send trace from Figure 5.3. However, due to the low time difference between packets when scaling in the amount of flows, and the added time delay it takes for the packets to be received back, it is not

surprising that the order of the packets might appear reversed in some cases in the final output trace. This slight anomaly is of lesser importance than maintaining the original trace order, however, and we can still conclude that the simulator still behaves as expected.

6.2 Performance

When looking at the results from Section 5.2, we can first conclude that the tests when finding a steady performance value of the UP, coincide with previous results found that the strengths of the UP is handling multiple flows in parallel rather than performing well on a single flow [9]. Further, the UP shows similar performance output when using trace-based generated traffic compared to model-based ones. As expected, packet loss starts being noticeable when Ericload is trying to push roughly the same amount of packets in both approaches, around 120k packets per second in these instances. In combining both `pcap` traces, we noticed higher output spikes when measuring the results using 10 sample points per second. These would at points reach as high as 20k, which scaling this to a whole second would yield an output of 200k packets per second. This could also hint to the aforementioned UP advantage of better handling multiple flows in parallel.

As for the occasional odd delay results, most notably when trying to replay the UDP trace with a scale factor of one, these results are harder to explain. Plotting the sampled delay points of this trace as presented in Figure 6.1, we can see that the delay increases happen around the same time as the high output spikes occur in Figure 5.6. This was somewhat to be expected. Why this test case in particular yielded higher delays compared to the other test cases, however, remains unclear. We see similar high delay spikes when inspecting the factor two scaled TCP trace shown in Figure 6.2. Here, on the other hand, the only increase occurs right at the beginning of the simulation, where it later evens out to a more stable rate. Why only at this point in time and no other, given that this trace is always repeated 28 times, also remains unclear. A combination of these two factors might explain the higher delays when combining both traces, as shown in Table 5.3. Here we see an extreme increase by a factor of 50 at some points. Upon plotting the delays of one of these results as shown in Figure 6.3, we see a very high initial latency, and two smaller bumps consistent with the increases from the UDP trace. We can thus conclude that both traces probably affect the overall delay during the test cases.

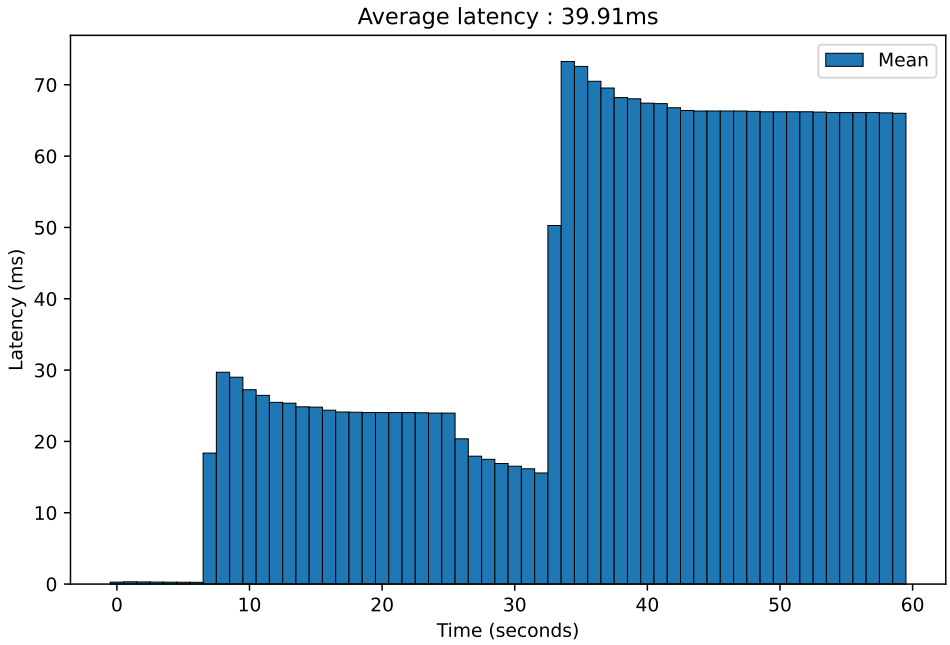

Figure 6.1: Measurements from replaying `youtubeLarge.pcap` where the latency rises during the matching high output peaks from Figure 5.6

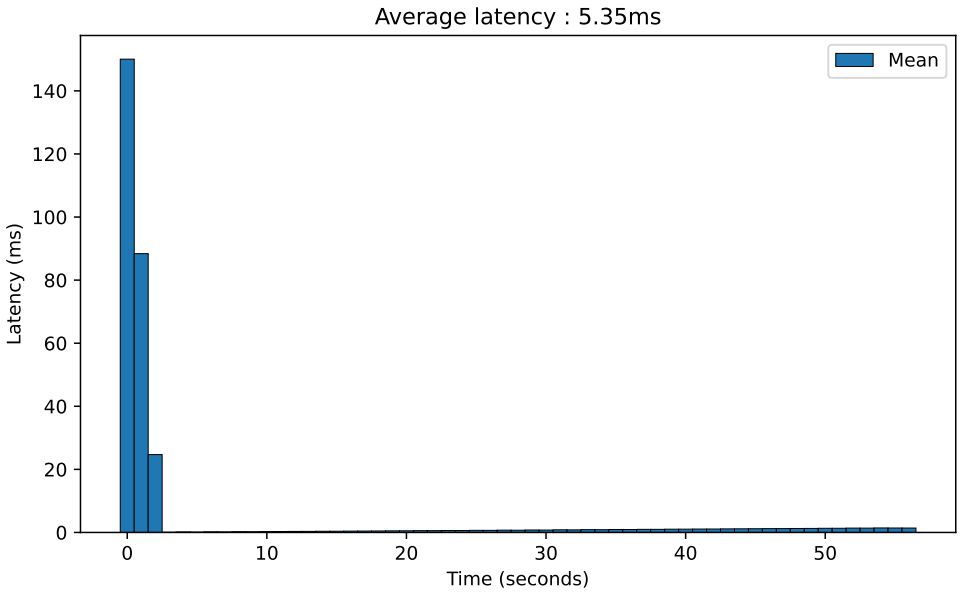

Figure 6.2: Measurements from replaying `60mbTCP.pcap` at a scale factor of two, with unusual high latency numbers in the beginning.

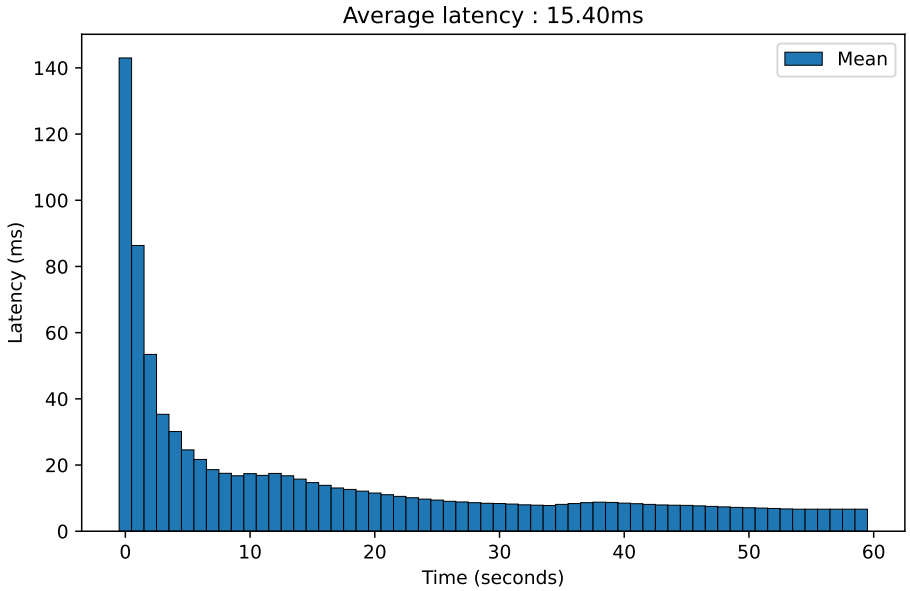

Figure 6.3: Measurements from replaying a combination of `60mbTCP.pcap` and `youtubeLarge.pcap`, where both traces are scaled by a factor of two.

For the final test cases where model-based and trace-based traffic was combined, we can conclude that Ericload is able to perform well while both of these traffic generation approached are used simultaneously. This further confirms the previously mentioned strengths of the UP. The unusual low delay numbers, when combining `youtubeLarge.pcap` with model-based traffic, might be explained by the amount of packets generated by both approaches. The total amount of packets generated by replaying this `pcap` file is 80k packets for the whole 60-second simulation. At the largest scale, this averages around 13k packets per second, which would effectively be drowned out by the model-based traffic. We might therefore conclude that these test cases with the UDP trace only, is not sufficient to comment on the overall performance due to the skewed numbers in outgoing packets generated.

There is no good explanation for the better delay numbers when only introducing model-based uplink traffic. One might theorize that the UP is better suited for handling traffic when it is more evenly spread in both directions, but then we should have also noticed similar results when doing performance measurements using model-based traffic only, as shown in Table 5.1. This is not the case, however.

The final thought regarding these tests is that the introduction of basic trace-based traffic generation to Ericload has been successful, even when this is used in combination with model-based traffic. With the heaviest extra load of an extra 100k UDP packets per second, the UP touches an overall 150k incoming packets per second throughput in some instances. Most notably when replaying using only the TCP `pcap` trace, and when this trace is combined with the UDP `pcap` trace. Figure 6.4 and Figure 6.5 illustrate this more clearly.

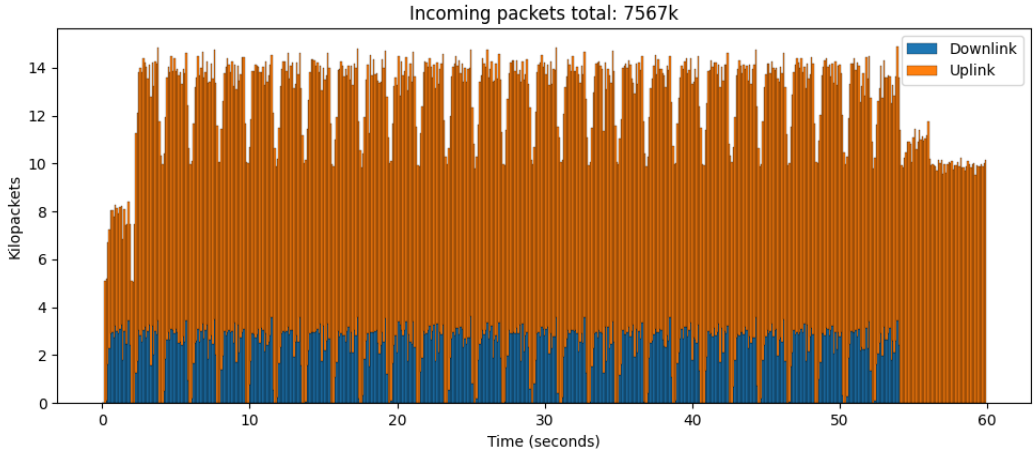

Figure 6.4: Measurements from replaying `60mbTCP.pcap` of scale factor one, with an added 100k uplink UDP traffic. The sampling frequency shown here is 10 samples per second.

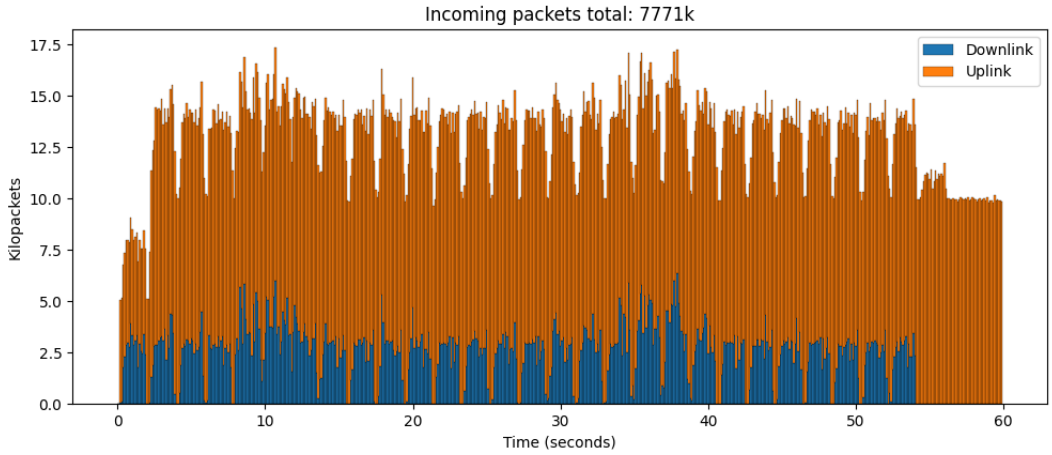

Figure 6.5: Measurements from replaying `60mbTCP.pcap` of scale one and `youtubeLarge.pcap` of scale three, with an added 100k uplink UDP traffic. The sampling frequency shown here is 10 samples per second.

6.3 Inter-arrival times and long-range dependency

Looking at Figure 5.7, we notice the expected behavior repetition and scaling has on traffic property as discussed in Section 4.4. The original file shows the exponential distribution of inter-arrival times trace-based traffic has, as well as LRD by not seemingly trying to converge to a specific value. Performing the same calculations as if repeating the original trace 100 times over, we notice no significant impact on the characteristics of either plots, compared to the original. This affirms that repeating traces this way still keeps the self-similar nature of the traffic. When scaling by artificially introducing more zero-second inter-arrival times, however, we see these properties no longer hold. As expected, the histogram distribution is skewed more

heavily toward these instant packets, which has the effect of the traffic losing its self-similar properties it showed before. This is most notable when only introducing scaling by itself. When combining it with the repetitions, this negative effect is not as large, most likely due to the error muddling out as the total packet amount increases with the repetitions. The negative effect scaling has could potentially be corrected for by not introducing the additional packets as having zero seconds inter-arrival times, but instead randomly drawing the time that a packet would be sent according to the original time and a random offset.

7
Conclusion

During this project, Ericload has been enhanced with the capability of replaying packet traces on a basic level to the User Plane (UP). It accomplishes this by first reading the traces from disk, loading them into buffer memory and sending these packets out to the network. To decide whether a packet is allowed to be sent out during a given time, Ericload compares its internal timestamps to the timestamp of the packets found in their original trace. Should enough simulation time have passed to match the elapsed time of any packet p_i compared to the timestamp of the first packet p_0 from its corresponding trace, Ericload will send out the packet to the network. It uses this method primarily to mimic both the order and time delays found in the original trace. Furthermore, Ericload is capable of scaling the traffic from a `pcap` trace both in time and in the number of parallel flows introduced that sends identical trace-based traffic. The time scaling is solved by having the full trace, i.e., all packets, replayed before being allowed to replay from the beginning of the trace again. Scaling in the number of flows is solved by having an arbitrary packet at time t_n being resent immediately, but with its packet information accommodated for the flow it belongs to. The two modes of scaling gives Ericload the capability of controlling the amount of trace-based traffic as a property of continuous time and the amount of packets generated at a given point in time.

Performing load tests using different combination of trace-based and model-based traffic generated by Ericload to the UP, further confirmed that the simulator is capable of replaying traffic on a basic level. At times, we noticed high latency values for trace-based traffic compared to sending model-based traffic only. These seemed to correlate with the burst spikes found in the traces themselves, which would affect the latency of the overall simulation. The exact reason for this still remains unclear. However, the overall performance of the simulator and the UP was not affected by these unusual latency numbers. Also, theoretical calculations performed on the inter-arrival times of one of the traces used, confirmed the known self-similar properties found in real life network traffic. It also suggests a more appropriate flow scaling to accommodate the negative impact instantly copied and sent packets can have on the self-similar properties of the final traffic output.

With all this, we conclude that the simulator is capable of generating scalable trace-based traffic for the UP. Whose performance is on par with the already implemented model-based traffic, when either running trace-based traffic alone or in combination with model-based traffic. Future work in this area might yield more interests results if focused on the following topics:

- **Improved replay state machine** - Replaying the traces correctly relies heavily on abiding by the time constraints of the timestamps of each packet. A complex state machine could improve correctness by taking into account more properties like current network conditions or more detailed state information of packets.
- **Improved network** - Scaling in flows was limited by only using one static UE address. While this did not seem to affect the overall performance, it does not entirely mimic a properly scaled traffic flow. A solution to incorporate more UE addresses would possibly provide better analysis on, scaled, trace-based traffic.
- **Improved traffic analysis** - Ericload can currently only create its own trace, of the traffic it is generating, inside the simulation itself and completely restricts any capture performed from the outside, which does not allow detailed traces of larger replays to be produced without severely impacting performance. A solution to this might yield better and more interesting results when analysis of the final output of the replayed traffic.

Bibliography

[1] Yu-chung Cheng et al. "Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying". In: *In USENIX Annual Technical Conference*. 2004, pp. 87–98.

[2] Siyoung Choi et al. "5G K-SimNet: End-to-End Performance Evaluation of 5G Cellular Systems". In: *2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC)*. 2019, pp. 1–6. DOI: 10.1109/CCNC.2019.8651686.

[3] Weibo Chu et al. "Model-based real-time volume control for interactive network traffic replay". In: *2012 IEEE Network Operations and Management Symposium*. 2012, pp. 163–170. DOI: 10.1109/NOMS.2012.6211895.

[4] *Control and User Plane Separation of EPC nodes (CUPS)*. URL: https://www.3gpp.org/news-events/3gpp-news/1882-cups. Accessed: Friday 1st July, 2022.

[5] *Data Plane Development Kit (DPDK)*. URL: https://www.dpdk.org/. Accessed: Friday 1st July, 2022.

[6] *Docker*. URL: https://www.docker.com/. Accessed: Friday 1st July, 2022.

[7] Tomas Dominguez-Bolano et al. "The GTEC 5G link-level simulator". In: *2016 1st International Workshop on Link- and System Level Simulations (IWSLS)*. 2016, pp. 1–6. DOI: 10.1109/IWSLS.2016.7801585.

[8] Marcin Dryjanski. *5G Core Network Functions*. 2018. URL: https://www.linkedin.com/pulse/5g-core-network-functions-marcin-dryjanski. Accessed: Friday 1st July, 2022.

[9] Olof Düsterdieck and Tasdikul Huda. *5G User Plane Load Simulator*. Master's Thesis. Gothenburg, Sweden, 2019.

[10] Wu-chang Feng et al. "TCPivo: a high-performance packet replay engine". In: (Jan. 2003), pp. 57–64. DOI: 10.1145/944773.944783.

[11] Seung-Sun Hong and S. Felix Wu. "On Interactive Internet Traffic Replay". In: *Recent Advances in Intrusion Detection*. Ed. by Alfonso Valdes and Diego Zamboni. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 247–264. ISBN: 978-3-540-31779-1.

[12] T. Karagiannis, M. Molle, and M. Faloutsos. "Long-range dependence ten years of Internet traffic modeling". In: *IEEE Internet Computing* 8 (2004), pp. 57–64.

[13] Will E. Leland et al. "On the self-similar nature of ethernet traffic." In: *Computer Communication Review*. Vol. 25. 1. Bellcore, 1995, pp. 202–213. URL: https://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-0029179676&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[14] K. Liang et al. “Performance Analysis of Cellular Radio Access Networks Relying on Control- and User-Plane Separation.” In: *IEEE Transactions on Vehicular Technology, Vehicular Technology, IEEE Transactions on, IEEE Trans. Veh. Technol* 68.7 (2019), pp. 7241–7245. ISSN: 0018-9545. URL: https://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.8721118&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[15] Hongri Liu et al. “An Interactive Traffic Replay Method in a Scaled-Down Environment”. In: *IEEE Access* 7 (Oct. 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2947062.

[16] Marco Mezzavilla et al. “5G MmWave Module for the Ns-3 Network Simulator”. In: MSWiM ’15. Cancun, Mexico: Association for Computing Machinery, 2015, pp. 283–290. ISBN: 9781450337625. DOI: 10.1145/2811587.2811619. URL: https://doi.org/10.1145/2811587.2811619.

[17] J Susan Milton and Jesse C Arnold. *Schaum’s Outline of Introduction to Probability Statistics: Principles Applications for Engineering the Computing Sciences*. McGraw-Hill Higher Education, 1994.

[18] Martin Müller et al. “Flexible multi-node simulation of cellular mobile communications: the Vienna 5G System Level Simulator”. In: *EURASIP Journal on Wireless Communications and Networking* 2018 (Sept. 2018). DOI: 10.1186/s13638-018-1238-7.

[19] Magnus Olsson and Catherine Mulligan. *EPC and 4G packet networks: driving the mobile broadband revolution*. Academic Press, 2012.

[20] Kihong Park and W. Willinger. “Self-Similar Network Traffic and Performance Evaluation”. In: 2000.

[21] V. Paxson and S. Floyd. “Wide area traffic: the failure of Poisson modeling”. In: *IEEE/ACM Transactions on Networking* 3.3 (1995), pp. 226–244. DOI: 10.1109/90.392383.

[22] Stefan Pratschner et al. “Versatile mobile communications simulation: the Vienna 5G Link Level Simulator”. In: *EURASIP Journal on Wireless Communications and Networking* 2018 (Sept. 2018). DOI: 10.1186/s13638-018-1239-6.

[23] *tcpdump*. URL: https://www.tcpdump.org/. Accessed: Friday 1st July, 2022.

[24] *tcpreplay*. URL: https://tcpreplay.appneta.com/. Accessed: Friday 1st July, 2022.

[25] *YAML*. URL: https://www.yaml.org/. Accessed: Friday 1st July, 2022.

A

Appendix 1

```c
/**
 *  Main function used to read a pcap file from disk.
 *
 *  fname:  string containing the full path of a pcap file.
 *  errbuf: string of the error messages generated during
 *          function call.
 *
 *  returns: pointer to a pcap_t, the handle used for reading the
 *           captured packets. A failed reading fills an error
 *           messages into errbuf.
 **/
pcap_t* pcap_open_offline(const char* fname, char* errbuf);

/**
 *  Compiles a filter string and returns a filter program. Example
 *  of a filter string could be "port 80 or 22" which would filter
 *  out all packets except those sent on port 80 and port 22.
 *
 *  p:        handle for reading packets from packet capture.
 *  fp:       pseudo-machine-language compiled from filter string.
 *  str:      string representation of the filter to apply.
 *  optimize: controls whether optimization is performed on result.
 *  netmask:  IPv4 netmask of the network.
 *
 *  returns: success or failure of the function call.
 **/
int pcap_compile(pcap_t* p, struct bpf_program* fp, const char* str,
                 int optimize, bpf_u_int32 netmask);
```

Listing 4: Relevant functions used from the *libpcap* library.

```
/**
 *  Applies a filter program to a processed pcap file and filters
 *  out unwanted packets.
 *
 *  p:  handle for reading packets from packet capture.
 *  fp: pseudo-machine-language compiled from a filter string.
 *
 *  returns: success or failure of the function call.
 **/
int pcap_setfilter(pcap_t* p, struct bpf_program* fp);

/**
 *  Packet processing loop which process packets provided by the
 *  handle and executes the provided callback function and its
 *  arguments on the packets. Future calls of this function will
 *  make it continue where it left off if any packets remain
 *  since the previous call.
 *
 *  p:        handle for reading packets from packet capture.
 *  cnt:      max number of packets to be processed.
 *            0 or -1 represents infinity.
 *  callback: callback function applied on every processed packet.
 *  user:     user arguments provided to the callback function.
 *
 *  returns: total number of processed packets or an error value.
 **/
int pcap_dispatch(pcap_t* p, int cnt, pcap_handler callback,
                  u_char* user);

/**
 *  Function used to process packets in a step-wise manner.
 *  One packet is processed from the handle per function call.
 *
 *  p:          handle for reading packets from packet capture.
 *  pkt_header: points to the 'header' part of the processed packet
 *              containing timestamps.
 *  pkt_data:   points to the 'data' part of the processed packet
 *              containing actual data.
 *
 *  returns: success or failure of the function call.
 **/
int pcap_next_ex(pcap_t* p, struct pcap_pkthdr** pkt_header,
                 const u_char** pkt_data);
```

Listing 5: Continuation of Listing 4.