KTH Computer Science
and Communication

An Evaluation of Software-Based Traffic Generators using Docker

SAI MAN WONG

Master's Thesis in Computer Science at
School of Computer Science and Communication, KTH

Supervisor: Alexander Kozlov
Examiner: Joakim Gustafson

TRITA xxx yyyy-nn

Abstract

The Information and Communication Technology (ICT) industry and network researchers use traffic generator tools to a large extent to test their systems. The industry uses reliable and rigid hardware-based platform tools for high-performance network testing. The research community commonly uses software-based tools in, for example, experiments because of economic and flexibility aspects. As a result, it is possible to run these tools on different systems and hardware. In this thesis, we examine the software traffic generators Iperf, Mausezahn, Ostinato in a closed loop physical and virtual environment to evaluate the applicability of the tools and find sources of inaccuracy for a given traffic profile. For each network tool, we measure the throughput from 64- to 4096-byte in packet sizes. Also, we encapsulate each tool with container technology using Docker to reach a more reproducible and portable research. Our results show that the CPU primarily limits the throughput for small packet sizes, and saturates the 1000 Mbps link for larger packet sizes. Finally, we suggest using these tools for simpler and automated network tests.

Referat

En Utvärdering utav Mjukvarubaserade Trafikgeneratorer med Docker

IT-branschen och nätverksforskare använder sig av trafikgeneratorer till stor del för att testa sina system. Industrin använder sig av stabila och pålitliga hårdvaruplattformar för högpresterande nätverkstester. Forskare brukar använda mjukvarubaserade verktyg i till exempel experiment på grund av ekonomiska och flexibilitet skäl. Det är därför möjligt att använda dessa verktyg på olika system och hårdvaror. I denna avhandling undersöker vi mjukvarutrafikgeneratorerna Iperf, Mausezahn, Ostinato i en isolerad fysisk och virtuell miljö, det vill säga för att utvärdera användbarheten av verktygen och hitta felkällor för en given trafikprofil. För varje nätverksverktyg mäter vi genomströmningen från 64 till 4096 byte i paketstorlekar. Dessutom paketerar vi varje verktyg med molnteknologin Docker för att nå ett mer reproducerbart och portabelt arbete. Våra resultat visar att processorn begränsar genomströmningen för små paketstorlekar och saturerar 1000 Mbpslänken för större paketstorlekar. Slutligen föreslår vi att man kan använda dessa verktyg för enklare och automatiserade nätverkstester.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

APP Application software

ARP Address Resolution Protocol

CLI Command Line Interface

CNCF Cloud Native Computing Foundation

CPU Central Processing Unit

DevOps Development and Systems Operation

DPDK Data Plane Development Kit

Gbps Gigabit per second

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

ICT Information and Communication Technology

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

ISP Internet Service Provider

IT Information Technology

LKM Loadable Kernel Module

LXC Linux Containers

MAC Media Accecs Protocol

Mbps Megabit per second

NAT Network Address Translation

NIC Network Interface Card

OCI Open Container Initiative

OS Operating System

RHEL Red Hat Enterprise Linux

SCTP Stream Control Transmission Protocol

SSH Secure Shell

TCP Transport Control Protocol

UDP User Datagram Protocol

VLAN Virtual Local Area Network

VM Virtual Machine

Chapter 1

Introduction

Information and Communication Technology (ICT) companies, for example, cloud service providers and mobile network operators provide reliable network products, services or solutions to handle network traffic on a large scale. These companies rely on proprietary and hardware-based network testing tools to test their products before deployment. That is, to generate realistic network traffic, which is then injected into a server to verify its behavior. Because of high demand to send and receive information with high speed and low latency, it is essential to test the ICT solutions thoroughly.

1.1 Problem Statement

In contrast to ICT enterprises, the network research community develops and uses open-source and software-based network testing tools. Thus, researchers use software-based network testing tools [7–9] in experiments because of its flexibility and for economic reasons [2, 3]. However, the generated network traffic from these tools is not as reliable as the one from the hardware-based platform, because of the underlying hardware and software, such as Network Interface Card (NIC) and Operating System (OS). Use of the software-based tools without an awareness of these variables, can produce inaccurate results between the generated and requested network traffic.

This paper examines the software-based network traffic generators Iperf, Mausezahn, and Ostinato. The purpose is to test the chosen tools concerning accuracy for different network profile, and the efficiency with lightweight hardware and software typical for academic environments. Distinctively to other similar studies, this project uses the container technology Docker to encapsulate the tools for automated tests and to achieve a higher degree of a reproducibility [10–12].

1.2 Limitation

Our study only investigates software-based traffic generators that primarily operate in user space and uses the Linux networking stack.

1.3 Sustainability, Ethics, and Societal Aspects

In modern life, the Internet is an integral part of the human environment, where stability, security, and efficiency are the essential aspects. This project has no direct and significant impact on sustainability in general. Except, the power consumption of laptops with the purpose to gather data.

From an ethical standpoint, we documented our steps throughout our thesis, provided the code in the appendices, and in a public repository https://github.com/saimanwong/mastersthesis. That is, to contribute to reproducible research and transparency. Hence, other people are encouraged to try to replicate and achieve similar results. However, and most likely, the results can vary because of the underlying hardware and software.

Since this project only uses open source software, it is just morally right to make everything public. Also, the network tools used in this project are purposely for private and controlled labs or virtual environments. Thus, it is inadvisable to use these tools on public networks.

Chapter 2

Background

2.1 Traffic Generator

The more the Information Technology (IT) infrastructures and networks grow, the higher the demand there is to test and validate its behavior as more devices connect. The network providers and researchers use traffic generator tools to a large extent for experiments, performance testing, verification and validation [2, 3]. As pointed out in the recent studies the network testing tools are either software- or hardware-based platforms [13–16].

2.1.1 Why Software-Based Traffic Generator

The network research community commonly uses or develops, open-source and software-based networking tools. In contrast, network equipment and solution providers often use proprietary and hardware-based ones, for example, Spirent [17] and Ixia [18]. These are proprietary and specialized software and hardware for network testing. Flexibility, accuracy, and cost are the factors for this general division between hardware- and software-based platforms. That is, 1) software-based networking tools are more flexible and cheaper than the hardware-based platform, on the other hand, 2) hardware-based tools generate more accurate and realistic network traffic than software-based tools at higher rates.

Botta, Dainotti, and Pescapé [2] identified that the flexibility of software-based network tools narrows down to three points. First, the ease to deploy these tools in a distributed fashion. Second, the freedom to make changes to fit a specific research purpose. Third, it can run on top of a variety of OSs and its networking stack. However, a hardware-based platform is more rigor and stable for network testing, because of its specialization.

The traffic's accuracy comes down to how well a network provider can fulfill the customer's requirements, for example, a mobile operator or an Internet Service Provider (ISP). These profiles are often rigor and detailed data sheets that describe, for ex-

ample, a specified speed or correctness within an error interval. Software-based tools often do not meet these requirements. That is, without knowledge of underlying hardware and software, there is a high chance to produce inaccurate results. Thus, Botta, Dainotti, and Pescapé [2] examined four software traffic generators and tried to raise awareness within the network research community to assess traffic generators critically.

2.1.2 Metrics and Types

There is a vast amount of software-based traffic generators with different purposes, for example, [7–9] are three lists of traffic generators to only mention a few. Therefore, Botta et al. [2], Molnár et al. [3] have attempted to categorize most frequently used traffic generators in the papers "Do you trust your software-based traffic generator" and "How to validate traffic generators?" respectively. Both found that the most frequently used traffic generators in literature are packet-level and maximum throughput traffic generators, which we marked with an asterisk in Table 2.1. Moreover, conventional metrics are byte throughput, packet size, and inter-departure/packet time distribution.

Table 2.1. Summary of Traffic Generators Types [2, 3]

Replay Engines	Replay network traffic back to specified NIC from a file which contains prerecorded traffic, usually a pcap-file.
(*) Maximum Throughput Generators	Generate maximum of network traffic with the purpose to test overall network performance, for example, over a link.
Model-Based Generators	Generate network traffic based on stochastic models.
High-Level and Auto-Configurable Generators	Generate traffic from realistic network models and change the parameters accordingly.
Special Scenario Generators	Generate network traffic with a specific characteristic, for example, video streaming traffic.
Application-level Traffic generators	Generate network traffic of network applications, for example, the traffic behavior between servers and clients.
Flow-Level Traffic generators	Generate packets in a particular order that resembles a particular characteristic from source to destination, for example, Internet traffic.
(*) Packet-Level Traffic Generators	Generate and craft packets, usually, from layer 2 and up to 7.

On a surface level, it is also possible to divide software-based traffic generators into three general categories: network software tools that run in user space/kernel space or circumvent the default kernel via framework to send and capture traffic.

User Space

Userspace traffic generators often use the library libpcap for Unix-like systems or WinPcap for Windows system [19]. That is, a library to access the default network stack that primarily uses system calls, such as socket API, to capture and inject packets. For instance, the network tools like Iperf [20], Mausezahn [21], Ostinato [22] and Tcpdump [23].

Kernel Space

These tools are primarily developed as a Loadable Kernel Module (LKM) and run close to the physical hardware, such as the NIC. Thus, they introduce little processing overhead compared to userspace tools. There are fewer context switches between user and kernel space, and as a consequence, such tools generate synthetic network traffic more efficient, for example, Brute [24] and Pktgen [13].

External Framework

The Linux network stack is complex and designed for general purpose. However, it is not explicitly designed for packet processing at higher rates, especially for small packets. Gallenmüller, Emmerich, Wohlfart, Raumer, and Carle [25] examined the software frameworks to circumvent the standard Linux network stack, that is, the frameworks netmap [26], DPDK [27] and PF_RING [28]. In comparison to the Linux network stack, the authors concluded: "The performance increase comes from processing in batches, preallocated buffers, and avoiding costly interrupts". For example, MoonGen [29] and TRex [30] are two traffic generators built on DPDK.

2.1.3 Known Bottlenecks

For packet processing application that uses the Linux network stack, the common bottlenecks are the processor, memory, software design and cache size [25, 31–33]. Most of the applications can reach 1 Gbps on the default network stack. Above this rate, for example, at the rate of 10 Gbps, noticeable limits appear.

Firstly, the CPU limits the tool to craft more complex packets. For example, the traffic generator uses x cycles to process a single packet, which may lead the CPU to go on full workload for a more substantial number of packets to process. Secondly, `sk_buff`, a data structure that stores information about a packet, is large and complex [34]. It becomes costly to allocate and deallocate memory for the packets at high rates. Thirdly, software design comes down to, for example, the packet queue gets stuck in a spinlock. As a result, CPU cycles go to waste due to the wait. Finally, the CPU cache size influences cache hits, miss and CPU cycles. For example, with a larger CPU cache, the chances are higher to get cache hits and minimize CPU idle for packet processing.

2.2 Virtualization

MIT and IBM introduced the concept of virtualization in the 1960s [35]. Now, in the 21st century, cloud computing has become one of the mainstream technology, and virtualization is the core of it [36]. That is a technology to partly or entirely separate software services from the physical hardware.

Virtualization technology enables multiple and isolated instances of a guest OS to share the same hardware resources [37]. An instance of a guest OS is therefore often called a Virtual Machine (VM) or virtual server. Thus, today it is common that more than ten instances run on a single physical server, but each of these operates as its virtual machine. For example, a physical server runs Ubuntu as the host OS. On top of the hardware and Ubuntu, virtualization makes it then possible to run various guest OSs upon it seamlessly, such as Windows and other Unix-like OSs.

Virtualization is commonly related to servers and categorized into three types. The server virtualization types are 1) full virtualization, 2) paravirtualization and 3) OS virtualization [38]. The two former types use a hypervisor, and the latter does not, see Figure 2.1. A hypervisor is a layer between the underlying hardware and virtual machines. Its purpose is to manage and allocate hardware resources to the virtual machines. There are two types of hypervisors, type 1 and type 2.

In short, type 1 hypervisor integrates a layer in the hardware system as firmware. This type is also called a bare metal hypervisor because it is directly on top of the hardware. It provides high performance, but also high complexity as it requires modification of the OS. Finally, a type 2 hypervisor runs, as software, on the host OS to achieve virtualization. This approach is flexible but introduces high overhead compared to type. Both of these types can run multiple and entire OSs as virtual machines. In contrast, OS virtualization is a lighter version of virtualization, does not run entire OSs and require no hypervisor.

2.2.1 Operating System Virtualization (Containers)

An operating system virtualization approach enables isolation of instances in user space without a hypervisor. Instead, this type uses system calls and other system Application Programming Interface (API) primarily to access the kernel and its hardware resources [38]. An instance of such is called a container; hence, this approach is also often referred to as container-based virtualization or container technology.

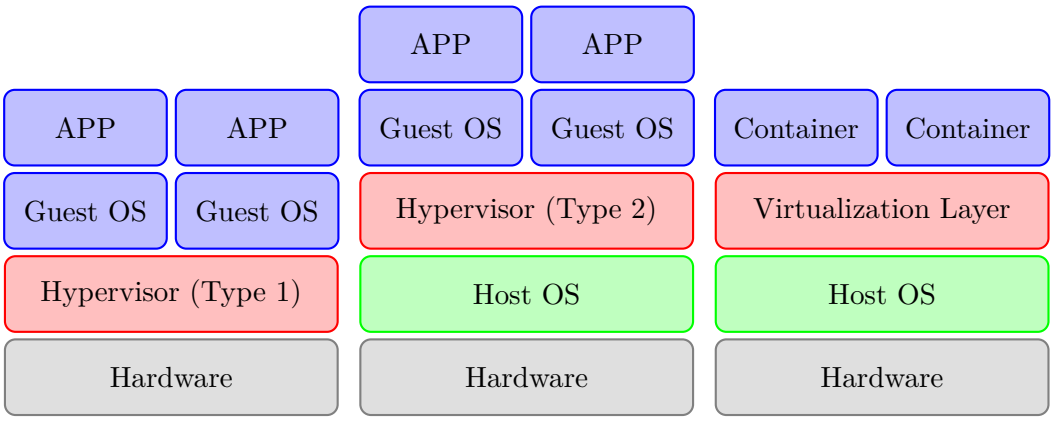

Figure 2.1. Hypervisor-Based (Type 1 and 2) and Container-Based Virtualization

Containers use the same resources as the host OS kernel to achieve isolated virtual environments, that is, in contrast to virtual machines that use a hypervisor. This approach uses kernel features typically to run separated containers [39, 40]. Thus, containers must support the host OS and its kernel to run, for example, a Linux distribution, such as CentOS, Ubuntu or Red Hat Enterprise Linux (RHEL).

Joy [41] identified four reasons container technology has been and is gaining popularity among developers, IT architects and operation persons.

"Portable Deployments" allows encapsulation of applications, and can then a developer can deploy these on many different systems.

"Fast application delivery" facilitates the product pipeline because the applications never leave the containers throughout the development, testing, and deployment stages. Also, a large part of this process, if not entire, can be automated with containers.

"Scale and deploy with ease" allow straightforward transfer of containers between the desktop, dedicated server or cloud environments. Also, it is easy to run and stop hundreds of containers.

"Higher workloads with greater density" allow more guest virtual environments (containers) to run on the host, that is, does not run entire OS as a virtual machine with a hypervisor, and thus, container technology introduces little to no overhead.

In parallel with cloud computing, OS virtualization or container technology evolved into open-source projects that are part of the Linux Foundation.

2.2.2 Docker

The concept of containers grew incrementally from 1979 till at the time of writing [42–46]. It began with unsecured and partly isolated virtual environments. In 2015, leaders within the container industry started the project Open Container Initiative (OCI) as part of the Linux Foundation to establish a standard for containers, that is, the OCI specification [46, 47]. Among these industry leaders, Docker is one of the driving forces behind this project. Figure 2.2 shows the popularity of Docker in recent years compared to other container technology approaches.

Docker is an open-source software platform built on Moby [48, 49]. It is a platform that enables development, provisioning, and deployment of software insecure and isolated containers. Docker's first iteration of the platform built upon Linux Containers (LXC) to manage containers. However, several iterations later, LXC was replaced with their library called libcontainer [50]. Finally, they donated libcontainer as runc to OCI, and containerd to Cloud Native Computing Foundation (CNCF), both are Linux Foundation projects [51, 52].

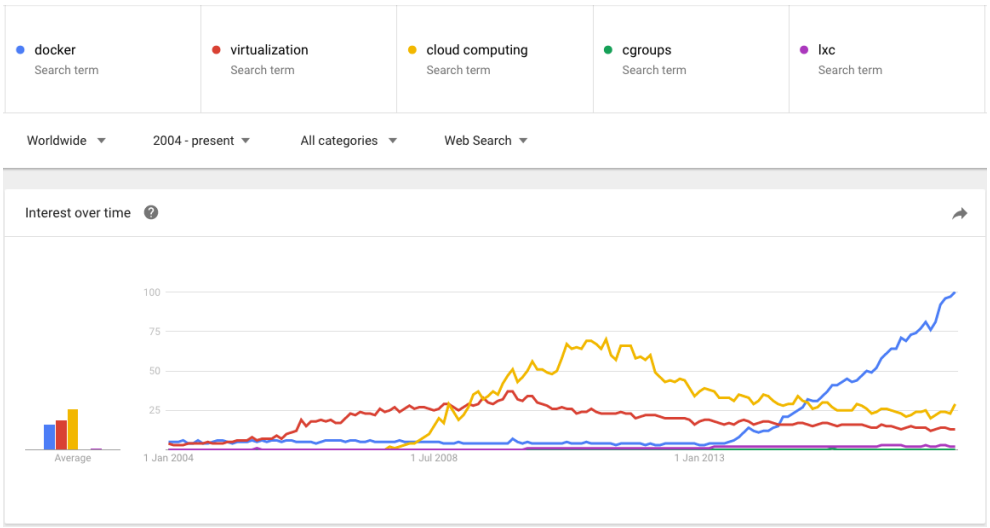

Figure 2.2. Google Trends of Container Technology (May 28, 2017)

Figure 2.3 shows a high-level view of Docker's architecture That is, the user uses (1) Command Line Interface (CLI)-commands to interact with (2) Docker engine to manage Docker images, containers, network configurations, orchestration and more. (3) containerd spins up container based on OCI container industry standard, such as (4) runc. However, the fundamentals of container lie primarily around the Linux kernel features cgroups and namespaces. In other words, cgroups "limits how much you can use" and namespaces "limits what you can see (and therefore use)" [53].

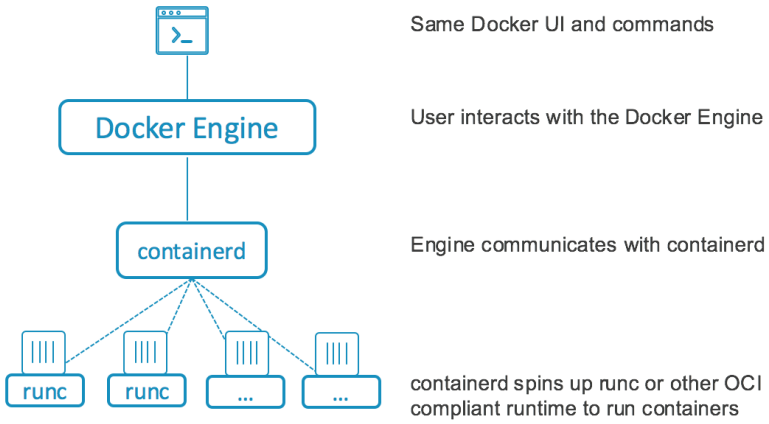

Figure 2.3. Overview of Docker Architecture [1]

Docker is feature rich in different ways to manage containers. The following simplifies and describes a few Docker fundamentals. That is, there are more advanced features that require specific CLI-flags to achieve a particular purpose.

Docker Image

Docker image is like a template, for example, a class in object-oriented programming. An image contains different layers of the file system, for example, layer 1) Ubuntu base image and layer 2) updated packages. Then, on top of all other layers, there is finally only a read-only file system layer. It is possible to inspect the different layers of a Docker image with:

```
$ docker history <IMAGE_NAME:TAG>
```

There are two ways to create a Docker image. Either use already running container:

```
$ docker commit <RUNNING_CONTAINER> <IMAGE_NAME:TAG>
```

alternatively, build an image from a human-readable and portable file called Dockerfile:

```
$ docker build -t <IMAGE_NAME:TAG> <PATH_TO_DOCKERFILE_DIR>
```

Docker Container

Docker container is a running instance of an image, for example, an object in object-oriented programming. Thus, once a container is running, it can then use the same hardware resources as the host kernel, such as file system, memory, CPU, network, user and more. For example:

```
$ docker run <IMAGE_NAME:TAG>
```

In our experiments, we followed the container technology standards to package the software in light-weight and portable containers. It enabled us to move the software between environments with the dependencies intact more easily. Also, the method to containerize the software facilitated the automation process of the tests.

Chapter 3

Related Work

In past research on traffic generators [2, 4–6, 13], were conducted experiments in a closed-loop environment between two hosts connected via a link. All these experiments included throughput test on traffic generators, in either packet per second or bit rate. As mentioned in Section 2.1.2, the network research community frequently uses the metrics byte throughput and packet size. We try to focus on the following metrics in our project. In [4–6], the authors conducted similar throughput experiments with userspace tools, and we used their method as inspiration for this project.

In [4], the authors generated TCP traffic over a 100 Mbps link, packet sizes ranging from 128 to 1408 bytes, and used the tools Iperf, Netperf, D-ITG and IP Traffic. The authors in [5, 6] generated both TCP and UDP traffic over 10 and 40 Gbps links respectively, packet sizes between 64 and 8950 bytes, and with Iperf, PackETH, Ostinato, and D-ITG. The results are similar in all these experiments, in the sense that the throughput increases for larger packet sizes because small packet sizes introduce higher overhead and cause lower performance. Table 3.1 summarizes the highest throughput from these experiments.

In latterly mentioned experiments, we encountered two observations that could make it harder for us to reproduce. First, they used network tools with either an architecture to both send and capture traffic or only send. Therefore, it is unclear what method they used to monitor the traffic. Second, we could not find which version of the network tools they used. To avoid ambiguity and to make our project more reproducible, we packaged the tools into different Docker containers, which makes them easy to deploy on various of systems. In our study, we try to evaluate the limits of userspace traffic generators used in Docker container environment. Finally, [54, 55] showed that Docker containers are feasible for data-intensive applications, and reach close to native CPU and memory performance, because these share the same resources as the host OS.

Table 3.1. Maximal Throughput Summary of [4–6]

Tools	**[4] TCP**	**[5] TCP**	**[6] TCP**	**[5] UDP**	**[6] UDP**
D-ITG	83.8 Mbps	6520 Mbps	16.2 Gbps	7900 Mbps	26.7 Gbps
Iperf	**93.1 Mbps**	5400 Mbps	-	1.05 Mbps	-
Ostinato	-	**9000 Mbps**	38.8 Gbps	9890 Mbps	36.9 Gbps
PackETH	-	7810 Mbps	**39.8 Gbps**	**9980 Mbps**	**39.8 Gbps**
IP Traffic	76.7 Mbps	-	-	-	-
Netperf	89.9 Mbps	-	-	-	-
Iperf multithread	-	9540 Mbps	-	12.6 Mbps	-
D-ITG multithread	-	9820 Mbps	39.9 Gbps	8450 Mbps	39.8 Gbps

Alternatively to our approach and as examined in Section 2.1.2, other network testing tools operate in the kernel space, or use external libraries to circumvent the host kernel and directly access commodity NIC to achieve faster packet processing. In [13], the authors developed the kernel module and traffic generator called Pktgen. They ran throughput, latency and packet delay variation experiments between two machines to compare their tool to the userspace tools Iperf and Netperf; as well as Pktgen-DPDK and Netmap, which use an external framework to process packets, such as DPDK. In the maximal throughput experiment for UDP traffic at 64 bytes packet size, former tools achieved between 150 and 350 Mbps, Pktgen generated a throughput of 4200 Mbps, and both the latter types saturated the link at 7600 Mbps. MoonGen is another high-speed and DPDK-based traffic generator. The creators of MoonGen generated UDP traffic at minimal packet size and also saturated a 10 Gbps link at a lower clock frequency than Pktgen-DPDK [14]. We compare and summarize our method to other related studies in Table 3.2.

All the related studies used the similar experimental methodology to examine software generators. That is, to conduct tests in an isolated environment of two machines to avoid external influences. Naturally, traffic generators built closer to the hardware perform significantly better than userspace tools. However, the hardware and software become more adaptable and cheaper; we decided to further examine the performance of userspace tools and their use cases. Also, to enable DevOps[1] practices with container technology to more efficiently automate tests in a standardized manner.

[1]Development and Systems Operation (DevOps) – It is a software development philosophy, which a large number of automation and monitoring tools arose. Of which, Docker is one of the leading tools on the market. The author in [11] describes the DevOps methodology as: "The approach is characterized by scripting, rather than documenting, a description of the necessary dependencies for software to run, usually from the Operating System (OS) on up."

Table 3.2. Comparison and Summary Table of Related Work

	Traffic Generator	**Metrics** frequently used in literature indentified by [3]	**Lab Environment**
2017 Our Work	User Space – Iperf – Mausezahn – Ostinato	– Byte Throughput – Packet Size Distribution	Physical Lab – Intel Core i5-2540M CPU – 8 GB Memory – 1000 Mbps NIC – Ubuntu Server 16.04.2 – Linux kernel version 4.4.0 – Docker version 17.05.0-ce Virtual Lab Host – Intel Core i5-5257U CPU – 8 GB Memory – macOS 10.12.5 – Oracle VM VirtualBox 5.1.22 Guest – 1 CPU – 2 GB Memory – Ubuntu server 16.04.2 – Linux kernel version 4.4.0 – Docker version 17.05.0-ce
2016 [13]	User Space – Netperf – Iperf Kernel Space – Pktgen External Framework – Netmap – Pktgen-DPDK	– Byte Throughput – Packet Size Distribution – Inter Packet Time Distribution	Physical Lab – Intel Xeon CPU – 3 GB Memory – 10 Gbps NIC – Ubuntu 13.10 – Linux kernel version 3.18.0
2015 [14]	External Framework – MoonGen – Pktgen-DPDK	– Byte Throughput – Packet Size Distribution – Inter Packet Time Distribution	Physical/Virtual Lab – Intel Xeon CPU – 1/10/40 Gbps NIC – Debian – Linux kernel version 3.7 – Open vSwitch 2.0.0
2014 [6]	User Space – D-ITG – Ostinato – PackETH	– Byte Throughput – Packet Size Distribution	Physical Lab – Intel Xeon CPU – 40 Gbps NIC – CentOS 6.5 – Linux kernel version 2.6.32
2014 [5]	User Space – D-ITG – Iperf – Ostinato – PackETH	– Byte Throughput – Packet Size Distribution	Physical Lab – Intel Xeon CPU – 64 GB Memory – 10 Gbps NIC – CentOS 6.2 – Linux kernel version 2.6.32
2011 [4]	User Space – D-ITG – IP-Traffic – Iperf – Netperf	– Byte Throughput – Packet Size Distribution	Physical Lab – Intel Pentium 4 CPU – 1 GB Memory – 100 Mbps NIC – Windows Server 2003
2010 [2]	User Space – D-ITG – MGEN – RUDE/CRUDE – TG	– Byte Throughput – Packet Size Distribution – Inter Packet Time Distribution	Physical Lab – Intel Pentium 4 CPU – 1 Gbps NIC – Linux kernel version 2.6.15

Chapter 4

Experiment

4.1 Reproducible Research with Docker

In academia, one criterion for a credible research is that other people should be able to reproduce it. The ability to reproduce, or replicate, findings in computer science is especially crucial as systems, technology and tools become more complex, which results in that new challenges arises for reproducibility. Therefore, a credible research often includes a level of rigor and transparency.

Sandve, Nekrutenko, Taylor, and Hovig [56] suggest some fundamental rules that can help a researcher reach a higher level of reproducibility. These rules are, for example, to document, record, automate and version control the process. Also, present the scripts, code, and results for transparency. Thus, research is reproducible if another person can execute the same documented steps and obtain a similar or identical result as the original author [10, 56].

Experimental findings in computer science often rely on code, algorithms or other software tools. Boettiger [11] identified common challenges within computational research are 1) "Dependency Hell", 2) "Imprecise documentation", 3) "Code rot" and 4) "Barriers to adoption and reuse in existing solutions". The author examines how Docker tackles these challenges.

Firstly and in contrast to VMs, Docker images are light-weight and share the same kernel as the host OS. Thus, it is possible to run containers with its dependencies and software intact. Secondly, a Dockerfile is a human-readable documentation that summarizes the dependencies, software, code and more. A researcher can, therefore, make modifications to the Dockerfile and build an own image from it. Thirdly, it is possible to save docker images and export containers into portable binaries. Finally, Docker has features to simplify the development and deployment process on different platforms, such as to move between local machines and cloud.

4.2 Lab Environment

Figure 4.1 shows that the lab consisted of two laptops (servers) with similar specifications, such as the Intel(R) Core(TM) processor i5-2540M at 2.60 GHz, 8 GB memory, Intel 82579LM Gigabit Network Connection adapter, Ubuntu Server 16.04.2 LTS (Xenial Xerus) operating system and Linux kernel version 4.4.0.

To avoid external influences and network traffic, a 1000 Mbps Ethernet cable connected these two laptops. Finally, we set up a Raspberry Pi 3 Model B as a wireless router to be able to SSH into the two servers and wrote a script to automate the process.

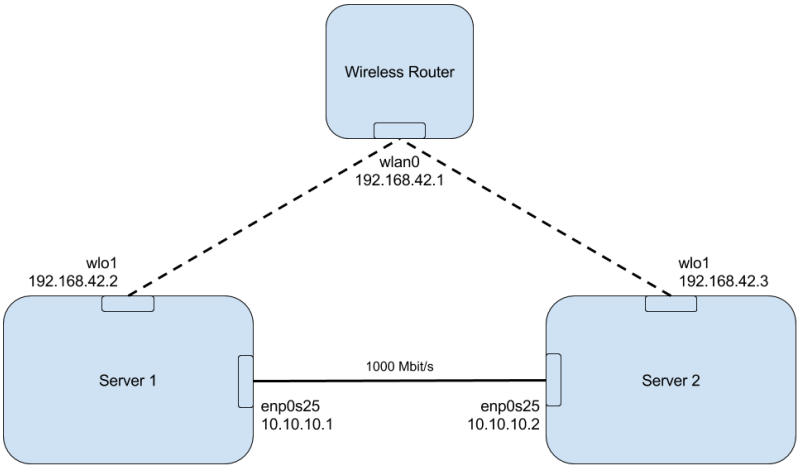

Figure 4.1. Overview of Physical Lab

Figure 4.2 shows a virtual lab in Oracle VM VirtualBox (5.1.22) on a host with Intel(R) Core(TM) processor i5-5257U at 2.7 GHz, 8 GB memory, and macOS Sierra 10.12.5 operating system. There are two VMs, or guest OSs, on top of the host with similar settings, such as one processor, 2 GB base memory, Ubuntu Server 16.04.2 LTS (Xenial Xerus) operating system and Linux kernel version 4.4.0.

Also, each VM has two virtio network interfaces. First one connects to the host in Network Address Translation (NAT)-mode to manage the VMs via SSH. Moreover, the second interface connects the VMs in "internal networking"-mode.

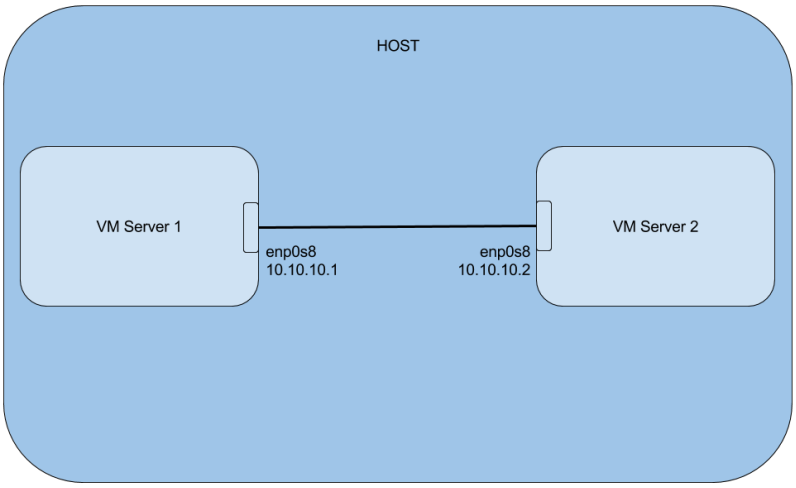

Figure 4.2. Overview of Virtual Lab

4.3 Tools

We ran Docker (17.05.0-ce) on both of the servers to manage containers. Then, used open source software, such as Iperf (2.0.9), Mausezahn (0.6.3) and Ostinato (0.8-1) to generate network traffic on the first server. On the second server, we ran Tcpdump (4.9.0) to capture packets and Capinfos (2.2.6) to analyze captured packets. Also, these tools were packaged or containerized, into five different Docker images. For a more detailed look into the implementation, please see Appendix B or `https://github.com/saimanwong/mastersthesis`.

4.3.1 Iperf

Iperf is available on Windows, Linux and BSD [20, 57]. This tool focuses on throughput testing rather than to craft packets. Thus, Iperf supports only layer 3 and 4, such as TCP, UDP, SCTP, IPv4, and IPv6. Iperf uses a client-server architecture to analyze network traffic. However, Iperf 3 does not support a serverless client to send UDP traffic [58]. Thus, we selected Iperf 2 for this project's purpose, and it can also run multiple threads.

The Iperf include the image `saimanwong/iperf`, as shown in Appendix B.1. It only required CLI-command to spin up this container for traffic generation.

4.3.2 Mausezahn

Mausezahn is only available for Linux and BSD [21, 59]. It is a tool to craft packets, generate and analyze network traffic. Also, it supports protocols from layer 2 to 7. Finally, it is possible to run Mausezahn in either "direct" or "interactive" mode. The former craft packets direct via the CLI with multiple parameters. The latter can create arbitrary streams of packets with its CLI.

The Mausezahn include the image `saimanwong/mausezahn`, as shown in Appendix B.2. It only required CLI-command in "direct mode" to spin up this container for traffic generation.

4.3.3 Ostinato

Ostinato is compatible with Windows, Linux, BSD and macOS [22, 60]. This tool is similar to Mausezahn as it can craft, generate and analyze packets. Also, Ostinato supports protocols from layer 2 to 7, for example, Ethernet/802.3, VLAN, ARP, IPv4, IPv6, TCP, UDP and HTTP to only mention a few. Its architecture consists of controller(s) and agent(s). That is, it is possible to use either a GUI or Python API as a controller to manage the agent and generate streams of packets from a single or several machines at the same time.

The Ostinato solution consists of the images `saimanwong/ostinato-drone` and `saimanwong/ostinato-python-api`, as shown in Appendix B.3 and Appendix B.4 respectively. The first image creates a container with an Ostinato agent that waits for instructions to generate packets. Finally, the second image spins up a controller container to communicate with the agent via a Python script.

4.3.4 Tcpdump and Capinfos

Tcpdump uses the library libpcap, thus, works on Linux, BSD, and macOS [23]. Additionally, there is also a port called WinPcap [61]. Nevertheless, the usage of Tcpdump is to capture and analyze packets on a specified network interface. These captured packets can then either be printed out on the terminal or saved to a file, usually ".pcap". We then use the Wireshark's CLI-tool Capinfos to interpret the pcap-file in the form of statistics, such as the bit rate [9].

Tcpdump and Capinfos are combined into the image `saimanwong/tcpdump-capinfos`, as shown in Appendix B.5. It is similar to the latter section, that is, to execute a CLI-command to capture and analyze packets.

4.4 Data Collection

Similar to [4–6], we ran UDP throughput experiments with packet sizes that vary from 64 bytes to 4096 bytes. That is, server 1 (source) sends packets over the link, and server 2 (sink) captures the packets. Finally, for each packet size, we generate and capture packets for 10 seconds to get the throughput in Mbps. We repeated it100 times to get a sample mean and standard deviation to understand the sparseness.

We evaluated Iperf, Mausezahn, and Ostinato in two different throughput experiments:

- Experiment 1 – Physical Hardware (Figure 4.1)
- Experiment 2 – Virtual Hardware (Figure 4.2)

Finally, Equation (4.1) shows the theoretical limit of throughput over a link [62]. S represents packet size, and λ the packet rate. Each Ethernet frame includes a 7-byte preamble, a 1-byte start of frame delimiter and a 12-byte interframe gap.

$$D_b = (S + 7B + 1B + 12B) * 8\frac{Bit}{B} * \lambda \tag{4.1}$$

4.4.1 Settings

In Iperf, it is only required to specify the source and destination Internet Protocol version 4 (IPv4) address since it tests either TCP or UDP throughput. Table 4.1 shows the rest of the parameters for Iperf.

In Ostinato and Mausezahn, we build streams of packets up to transport layer (layer 4), such as Media Accecs Protocol (MAC), Ethernet II, IPv4 and User Datagram Protocol (UDP) respectively. Thus, it is required to specify the MAC and IPv4 addresses for both the destination and source. Also, to select the NIC to transmit packets. Table 4.3 and Table 4.2 presents the remaining settings of Ostinato and Mausezahn respectively.

Table 4.1. Experiment Parameters – Iperf

Number of Packets	Infinite
Bandwidth	1000 Mbps
Threads	1
Protocol	UDP
Packet Size	64 - 4096
Experiment Time	10 seconds × 100 iterations

Table 4.2. Experiment Parameters – Mausezahn

Number of Packets	Infinite
Protocol	UDP
Packet Size	64 - 4096
Experiment Time	10 seconds × 100 iterations

Table 4.3. Experiment Parameters – Ostinato

Number of Bursts	1 000 000
Packets per Burst	10
Bursts per Second	50 000
Protocol	UDP
Packet Size	64 - 4096
Experiment Time	10 seconds × 100 iterations

Chapter 5

Results

We used the parameter settings in Section 4.4.1 to send UDP traffic with various packet sizes between two hosts in a physical and virtual environment. In both environments presented in Section 4.2, the first host (source) generates and sends the traffic to the second host (sink) which captures it. For each packet size varied between 64 and 4096 bytes, we ran 100 simulations and each run for 10 seconds. The following couple of sections present the performance of the traffic generators concerning throughput.

5.1 Physical Hardware

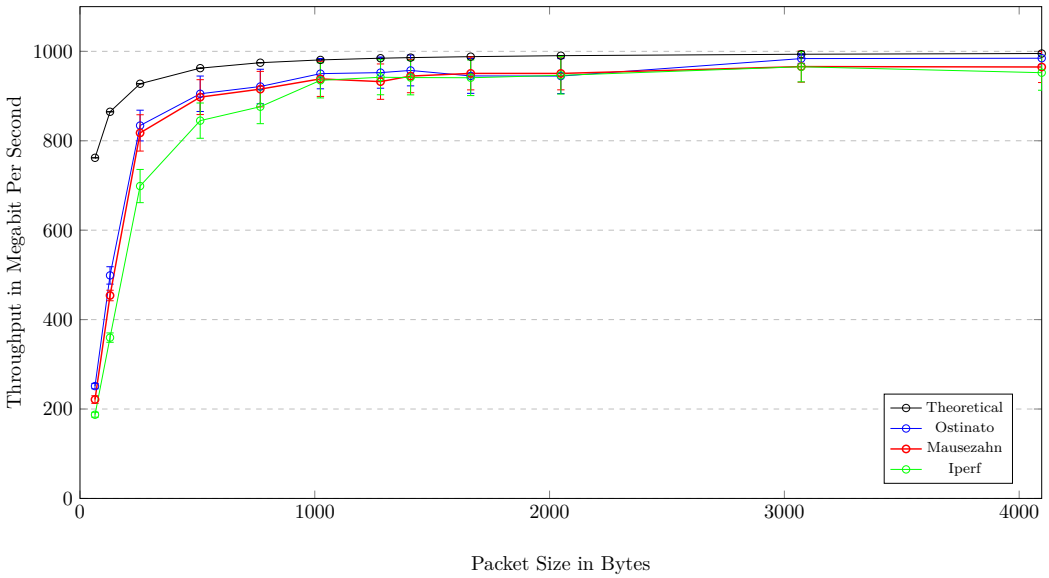

Figure 5.1. Throughput Graph Summary in Physical Environment

As shown in Figure 5.1, Ostinato reached the highest throughput of 984.61±14.54 Mbps at packet size 4096 byte. Mausezahn and Iperf reached their maximum throughput of 965.94±23.47 Mbps and 965.55±48.67 Mbps respectively, both at packet size 3072 byte. Iperf had a slow start from 64 to 768 bytes in packet size. That is, in contrast to Ostinato and Mausezahn that achieved similar throughput from 256 to 3072 bytes in packet size. Additionally at packet sizes 3072 and 4096 bytes, Ostinato has a throughput sparseness (standard deviation) that is more than half compared to Mausezahn and Iperf.

5.2 Virtual Hardware

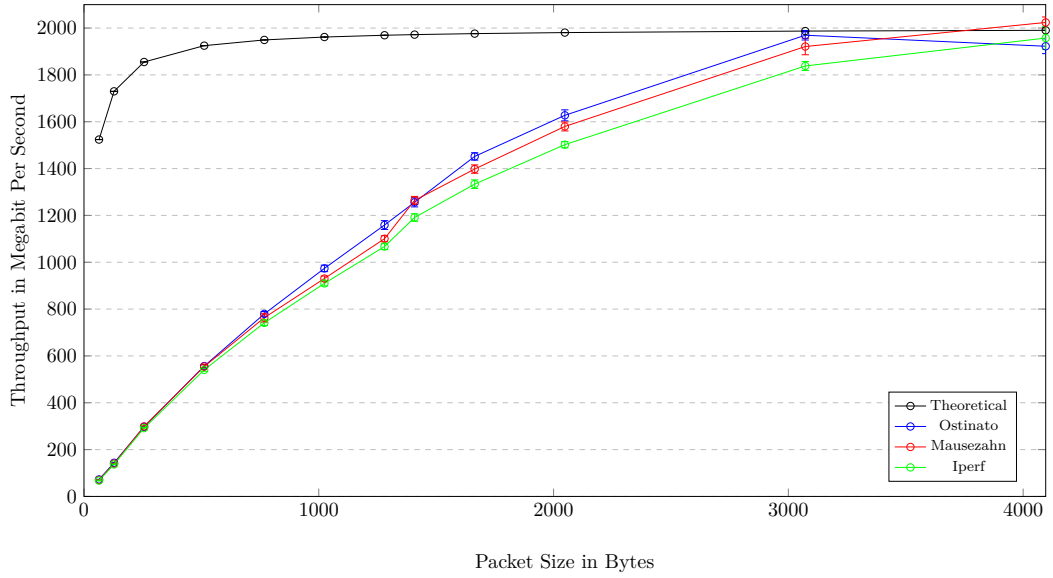

Figure 5.2. Throughput Graph Summary in Virtual Environment

Figure 5.2 illustrates the results from the virtual lab. Mausezahn reached the highest throughput of 2023.85±23.47 Mbps at 4096 bytes in packet size. Ostinato and Iperf achieved 1969.38±20.42 and 1957.25±48.67 Mbps maximum throughput at packet sizes 3072 and 4096 bytes respectively. All the network tools keep relatively and similar throughput rate. Except, Ostinato at packet size 3072 byte where it diverges from its maximum throughput.

Chapter 6

Discussion

The network research community uses software-based traffic generators for a variety of purposes because these are often open sourced and flexible to handle. In this project, we examined open source traffic generators that primarily operate in the user space and use the Linux default network stack. The authors in [4–6] examined these type of generators, concerning throughput with varied packet sizes, over 100 Mbps, 10 Gbps, and 40 Gbps links. We presented a summary of these past research experiments in Chapter 3 and Table 3.1. We conducted similar benchmark experiments and achieved traffic characteristics comparable to theirs, but over a 1000 Mbps link. We also used container technology, Docker, to package this experiment and make it simpler to replicate.

6.1 Performance Evaluation

As expected from previous studies, our results in Chapter 5 (Figure 5.1 and Figure 5.2) show that the throughput increased with the packet size. All the traffic generators yield lower throughput on smaller packet sizes because the CPU is put on high workload when it has to generate these smaller packets. It is an expensive operation to copy data between user and kernel space on top of the Linux default network stack [31]. Mainly, the experimental results are approximately between 12% and 75% below the theoretical throughput for 64 to 512 bytes in packet size, as shown in Figure 5.1. Thus, the CPU reaches its full capacity before it saturates the link.

In both labs, we noticed in broad strokes that the standard deviation also increases with the packet size, but there are exceptions. For example, Ostinato generated the highest throughput and lowest standard deviation in the physical lab. It yielded 985.61±14.54 Mbps at 4096 bytes packet size. That is, in contrast to Mausezahn and Iperf that generated 965.94±34.83 and 965.55±33.77 Mbps, respectively, both at 3072 bytes packet size. We send multiple packets at once with Ostinato in burst mode instead of single. It might explain why Ostinato generated the highest

throughput more consistently than Mausezahn and Iperf.

Our results in the physical lab reach consistent and similar traffic characteristics as the previous research, [4–6] individually. That is, the traffic generators directly use the host's NIC to saturate the link for larger packet size, and CPU limits throughput for smaller packet sizes. However, the results in the lab environment with virtual hardware differed from the physical one, as the tools generated over 1900 Mbps throughput. The primary reason is that we use "internal networking"-mode [63]. That is, packets flow via a virtual network switch instead of the host and its NIC. Thus, the packet processing and maximum throughput depended entirely on the host's CPU thereof, the higher performance.

We also encountered another unanticipated result when we experimented with Iperf. In [6], the authors examined the throughput of Iperf with multiple threads. Their UDP traffic did not saturate the 10 Gbps link ranged. That is, it generated from 1.05 Mbps to 12.6 Mbps, with one respectively twelve threads. However, our results indicate that Iperf with one thread keeps up with the throughput of both Ostinato and Mausezahn. We investigated it and found that Iperf 2.0.5 produced unexpected throughput on various packet sizes, similar to the latter mentioned authors' findings. Thus, we used Iperf 2.0.9, a newer version, with one thread in this project.

Altogether, we wanted to examine the traffic generators in two different environments. One might argue that the lab with physical hardware yielded more realistic results compared to the virtual environment because we only used the real equipment. On the other side, sometimes resources are limited, then virtual environments on a single host might be a more feasible option to experiment within. However, in the end, it comes down to picking the most suitable tool for the job. It sounds simple, but it is more difficult to apply in practice, because there are a large number of traffic generators with different purposes, and we will further discuss this in Section 6.2.2.

6.2 Experiment Evaluation

We used the closed-loop approach to experiment with the traffic generators. It is a standard approach to test the behavior of both traffic generator and underlying hardware. That is, it consists of two directly connected hosts in a controlled and isolated environment to minimize external influences. Besides, we packaged the software into standardized images/Dockerfile(s), which facilitated the work for us when we were to, for example, tune the parameters and move between different environments. This approach had an insignificant impact on performance as the containers had direct and privileged access to the kernels resources.

6.2.1 Reproducible Research

As for reproducible research, there is a challenge to replicate the exact results, because these network tools heavily depend on the underlying system. However, we use container technology to achieve a reproducible research partly. That is, we wrote Dockerfile(s) that the user can build an image from, or make modifications to fit a specific purpose. Furthermore, we provide a bash-script to automatically spin up containers of these tools on two remote servers, either physical or virtual one. The only requirement is that the servers have Docker installed on these servers.

6.2.2 Validation – Traffic Generators and Metrics

Our literature review in Section 2.1 delves into why researchers use software-based traffic generators, the different types, metrics, and challenges with them. However, in this section, we try to discuss our procedure, to raise awareness and maybe help other people to choose a suitable network tool or several tools. Additionally, we specifically discuss around and delve into [2, 3], where the authors critically assessed a large number of traffic generators from different sources.

First off, we expect a traffic generator to send traffic with specific characteristics. We can interpret these characteristics as different kind of requirements. When we talk about accuracy, it is about how well the generated synthetic traffic matches with the specified requirements and error margins. Thus, it is equally as important to consider which metric(s) to experiment around, as it is to pick the network tool that can fulfill these.

Since there are a large number of software-based traffic generators, the authors in [2, 3] questioned the methods to validate the accuracy and reliability of these tools in the network community, notably, the traffic generators that primarily operate in user space on top of any arbitrary OS and network stack. That is, given the tool's generality and flexibility, it requires from the researcher(s) a greater understanding of the underlying system to produce accurate results, which is often a challenge that gets minimal attention in the literature.

In regards to our findings, we used the metrics byte throughput and packet size, a similar method to [4–6] to try to experiment with two recently active network tools (Ostinato and Iperf) and an inactive one (Mausezahn). We showed that these tools saturate the link at packet sizes above 4096 bytes. If the only purpose is to test end-to-end system performance regarding maximum throughput, then these tools can be a reasonable choice. However, our and [4–6] studies do not examine other parameters, such as packet delay, loss, and fragmentation. As in the previously mentioned papers, we want to suggest to carefully consider which network tool is the most suitable to user's requirements.

6.2.3 Recommendations

The tools we evaluated are flexible to use in a broad variety of environments but can yield different results depending on the circumstance. Thus, we recommend applying these tools in a smaller scale and low-risk environments as complements for hardware-based traffic generators. For example, for personal experiments and small networks with more straightforward traffic characteristics.

We recommend Ostinato and Mausezahn for users that want to test and create any arbitrary packets from layer 2 and up. In this case, the former is a more flexible choice as it is available cross-platform, has a GUI and Python-API with automation capabilities. For general bandwidth test, we recommend Iperf because the creators developed it for this specific purpose. Finally, we recommend a more profound research into network tools that use libraries specialized for fast packet processing on commodity hardware.

Chapter 7

Conclusion

Our study evaluates the performance of the network tools Iperf, Mausezahn, and Ostinato. We use the metrics throughput and varying packet sizes to measure the performance of these in closed-loop environments with both physical and virtual hardware, as summarized in Table 3.2. The tools operate in the user space and use the host OS' default network stack to craft and send traffic. Given the tools generality, the user can efficiently conduct smaller tests and deploy them on many various systems. Also, due to its generality, the tools depend heavily on the underlying system to generate traffic with specific characteristics. Thus, the responsibility lies with the user to grasp both a practical and theoretical understanding of the tool and the underlying system.

In agreement with the previous work, we emphasize here the importance to identify a specific purpose and choose the most suitable metrics and tools accordingly. Our results show that the CPU and NIC limit the throughput produced from the userspace tools for different packet sizes. The results remain consistent alongside previous studies. Besides, we run the userspace tools on top of a virtual switch and hardware and show that only the CPU limits the tool's throughput performance. Conclusively, the tools are useful for smaller end-to-end system tests. Especially suitable, in combination with container technology to achieve higher reproducibility and automation capabilities in both research and industry.

In the future, it would be interesting to examine network tools that use high-speed packet processing libraries, for example, Data Plane Development Kit (DPDK) on commodity hardware. Also, to further investigate software tools that facilitate network virtualization, for example, Software-Defined Networking (SDN) technologies.

Appendix A

Experiment Results in Tables

A.1 Physical Environment

Table A.1. Physical Environment – Theoretical Throughput Table Results

Packet Size in Bytes	Throughput in Megabits Per Second	Standard Deviation
64	761.90	0.00
128	864.86	0.00
256	927.54	0.00
512	962.41	0.00
768	974.62	0.00
1024	980.84	0.00
1280	984.62	0.00
1408	985.99	0.00
1664	988.12	0.00
2048	990.33	0.00
3072	993.53	0.00
4096	995.14	0.00

Table A.2. Physical Environment – Iperf Throughput Table Results

Packet Size in Bytes	Throughput in Megabits Per Second	Standard Deviation
64	187.23	5.54
128	359.73	10.45
256	698.72	37.14
512	845.04	39.44
768	876.17	37.94
1024	935.15	39.32
1280	942.06	39.14
1408	941.39	38.69
1664	940.96	39.87
2048	946.33	41.84
3072	965.55	33.77
4096	952.23	39.24

Table A.3. Physical Environment – Mausezahn Throughput Table Results

Packet Size in Bytes	Throughput in Megabits Per Second	Standard Deviation
64	221.19	8.52
128	453.94	11.79
256	817.58	40.60
512	897.62	39.01
768	915.49	39.69
1024	938.41	39.24
1280	932.27	39.57
1408	944.72	37.23
1664	950.64	36.81
2048	950.56	36.63
3072	965.94	34.83
4096	965.01	34.77

Table A.4. Physical Environment – Ostinato Throughput Table Results

Packet Size in Bytes	Throughput in Megabits Per Second	Standard Deviation
64	251.28	6.01
128	498.84	19.46
256	834.04	34.35
512	905.11	39.66
768	921.35	38.60
1024	950.11	33.86
1280	952.42	34.91
1408	957.28	34.47
1664	945.04	38.99
2048	944.69	39.12
3072	983.79	10.64
4096	984.61	14.54

A.2 Virtual Environment

Table A.5. Virtual Environment – Theoretical Throughput Table Results

Packet Size in Bytes	Throughput in Megabits Per Second	Standard Deviation
64	1523.81	0.00
128	1729.73	0.00
256	1855.07	0.00
512	1924.81	0.00
768	1949.24	0.00
1024	1961.69	0.00
1280	1969.23	0.00
1408	1971.99	0.00
1664	1976.25	0.00
2048	1980.66	0.00
3072	1987.06	0.00
4096	1990.28	0.00

Table A.6. Virtual Environment – Iperf Throughput Table Results

Packet Size in Bytes	Throughput in Megabits Per Second	Standard Deviation
64	67.77	1.93
128	136.76	1.42
256	291.75	3.16
512	540.45	6.15
768	741.21	11.60
1024	909.71	11.11
1280	1067.07	12.54
1408	1191.10	16.23
1664	1333.50	18.33
2048	1502.06	13.25
3072	1838.08	19.19
4096	1957.25	48.67

Table A.7. Virtual Environment – Mausezahn Throughput Table Results

Packet Size in Bytes	Throughput in Megabits Per Second	Standard Deviation
64	69.98	0.77
128	140.26	1.28
256	299.09	3.49
512	554.53	7.65
768	763.11	18.82
1024	930.44	10.76
1280	1100.44	13.06
1408	1263.94	17.14
1664	1397.84	17.84
2048	1579.88	19.11
3072	1921.08	35.08
4096	2023.85	23.47

Table A.8. Virtual Environment – Ostinato Throughput Table Results

Packet Size in Bytes	Throughput in Megabits Per Second	Standard Deviation
64	72.59	0.71
128	143.65	1.66
256	297.05	2.53
512	556.55	6.21
768	779.45	10.11
1024	973.74	13.85
1280	1158.89	18.88
1408	1255.85	19.51
1664	1451.81	15.73
2048	1627.30	23.47
3072	1969.38	20.42
4096	1922.30	31.51

Appendix B

Code Listings

B.1 saimanwong/iperf

B.1.1 Dockerfile

```
# docker build -t saimanwong/iperf .
# docker run --rm -d \
# --name host-iperf \
# --privileged \
# --network host \
# saimanwong/iperf \
# <PACKET_SIZE> \
# <BANDWIDTH_IN_MBPS> \
# <THREADS> \
# <SRC_IP> \
# <DST_IP>

FROM ubuntu:xenial
LABEL maintainer "Sai Man Wong <smwong@kth.se>"

WORKDIR /

# Change software repository according to your geographical location
# COPY china.sources.list /etc/apt/sources.list

COPY docker-entrypoint.sh /

RUN apt-get update && apt-get install -y \
    wget --no-install-recommends && \
    wget --no-check-certificate -O /usr/bin/iperf https://iperf.fr/download/ubuntu/
        ↪ iperf_2.0.9 && \
    chmod +x /usr/bin/iperf && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["./docker-entrypoint.sh"]
```

B.1.2 `docker-entrypoint.sh`

```
#!/bin/bash
FRAME_LEN=$(($1-42))
BW=$2
THREADS=$3
SRC_IP=$4
DST_IP=$5

iperf -l $FRAME_LEN \
    -B $SRC_IP \
    -c $DST_IP \
    -t 1000 \
    -b ${BW}m \
    -P $THREADS \
    -u
```

B.2 saimanwong/mausezahn

B.2.1 Dockerfile

```
# docker build -t saimanwong/mausezahn .
# docker run --rm -d \
# --name host-mausezahn \
# --privileged \
# --network host \
# saimanwong/mausezahn \
# <SRC_INTERFACE> \
# 0 \
# <PACKET_SIZE> \
# <SRC_MAC> \
# <DST_MAC> \
# <SRC_IP> \
# <DST_IP> \
# udp

FROM ubuntu:xenial
LABEL maintainer "Sai Man Wong <smwong@kth.se>"

WORKDIR /

# Change software repository according to your geographical location
# COPY china.sources.list /etc/apt/sources.list

COPY docker-entrypoint.sh /mausezahn/docker-entrypoint.sh


RUN apt-get update && apt-get install -y git && \
    apt-get install -y --no-install-recommends \
    gcc ccache flex bison libnl-3-dev \
    libnl-genl-3-dev libnl-route-3-dev libgeoip-dev \
    libnetfilter-conntrack-dev libncurses5-dev liburcu-dev \
    libnacl-dev libpcap-dev zlib1g-dev libcli-dev libnet1-dev && \
    git clone https://github.com/netsniff-ng/netsniff-ng && \
    rm -rf /var/lib/apt/lists/*

WORKDIR /netsniff-ng
RUN ./configure && make mausezahn && mv mausezahn/* /mausezahn

WORKDIR /mausezahn
RUN rm -rf /netsniff-ng && chmod +x docker-entrypoint.sh

ENTRYPOINT ["./docker-entrypoint.sh"]
```

B.2.2 `docker-entrypoint.sh`

```
#!/bin/bash
IFACE=$1
COUNT=$2
FRAME_LEN=$(($3-42))
SRC_MAC=$4
DST_MAC=$5
SRC_IP=$6
DST_IP=$7
PROTOCOL=$8

./mausezahn $IFACE \
    -c $COUNT \
    -p $FRAME_LEN \
    -a $SRC_MAC \
    -b $DST_MAC \
    -A $SRC_IP \
    -B $DST_IP \
    -t $PROTOCOL
```

B.3 saimanwong/ostinato-drone

B.3.1 Dockerfile

```
# docker build -t saimanwong/ostinato-drone .
# docker run --rm -d \
# --name host-drone \
# --privileged \
# --network host \
# saimanwong/ostinato-drone

FROM ubuntu:xenial
LABEL maintainer "Sai Man Wong <smwong@kth.se>"

WORKDIR /

# Change software repository according to your geographical location
# COPY china.sources.list /etc/apt/sources.list

RUN apt-get update && apt-get install -y --no-install-recommends \
    gdebi-core \
    wget && \
    wget http://security.ubuntu.com/ubuntu/pool/main/p/protobuf/libprotobuf10_3
        ↪ .0.0-9ubuntu2_amd64.deb && \
    wget http://cz.archive.ubuntu.com/ubuntu/pool/universe/o/ostinato/ostinato_0.8-1
        ↪ build1_amd64.deb && \
    echo y | gdebi libprotobuf10_3.0.0-9ubuntu2_amd64.deb && \
    echo y | gdebi ostinato_0.8-1build1_amd64.deb && \
    apt-get purge -y wget gdebi-core && \
    rm -rf /var/lib/apt/lists/* \
    libprotobuf10_3.0.0-9ubuntu2_amd64.deb \
    ostinato_0.8-1build1_amd64.deb

ENTRYPOINT [ "drone" ]
```

B.4 saimanwong/ostinato-python-api

B.4.1 Dockerfile

```
# docker build -t saimanwong/ostinato-python-api .
# docker run --rm -d \
# --name host-python \
# --network host \
# saimanwong/ostinato-python-api \
# <DRONE_IP> \
# <SRC_INTERFACE> \
# <SRC_MAC> \
# <DST_MAC> \
# <SRC_IP> \
# <DST_IP> \
# <PACKET_SIZE> \
# <NUM_BURST> \
# <PACKET_PER_BURST> \
# <BURST_PER_SECOND>

FROM ubuntu:xenial
LABEL maintainer "Sai Man Wong <smwong@kth.se>"

WORKDIR /

# Change software repository according to your geographical location
# COPY china.sources.list /etc/apt/sources.list

COPY docker-entrypoint.py /

RUN apt-get update && apt-get install -y --no-install-recommends \
    python \
    python-pip \
    wget && \
    pip install --upgrade pip && \
    pip install setuptools && \
    wget https://pypi.python.org/packages/fb/e3/72
        ↪ a1f19cd8b6d8cf77233a59ed434d0881b35e34bc074458291f2ddfe305/python-
        ↪ ostinato-0.8.tar.gz && \
    pip install python-ostinato-0.8.tar.gz && \
    apt-get purge -y python-pip && \
    rm -rf /var/lib/apt/lists/* \
    python-ostinato-0.8.tar.gz

ENTRYPOINT ["./docker-entrypoint.py"]
```

B.4.2 docker-entrypoint.py

```
#! /usr/bin/env python

import os
import time
```

```
5  import sys
6  import json
7  import binascii
8  import socket
9  import signal
10 from pprint import pprint
11
12
13 from ostinato.core import ost_pb, DroneProxy
14 from ostinato.protocols.mac_pb2 import mac
15 from ostinato.protocols.ip4_pb2 import ip4, Ip4
16 from ostinato.protocols.udp_pb2 import udp
17
18 print(sys.argv)
19 host_name = sys.argv[1]
20 iface = sys.argv[2]
21 mac_src = "0x" + sys.argv[3].replace(":", "")
22 mac_dst = "0x" + sys.argv[4].replace(":", "")
23 ip_src = "0x" + binascii.hexlify(socket.inet_aton(sys.argv[5])).upper()
24 ip_dst = "0x" + binascii.hexlify(socket.inet_aton(sys.argv[6])).upper()
25 frame_len = int(sys.argv[7])
26 num_bursts = int(sys.argv[8])
27 packets_per_burst = int(sys.argv[9])
28 bursts_per_sec = int(sys.argv[10])
29
30 def setup_stream(id):
31     stream_id = ost_pb.StreamIdList()
32     stream_id.port_id.CopyFrom(tx_port.port_id[0])
33     stream_id.stream_id.add().id = 1
34     drone.addStream(stream_id)
35     return stream_id
36
37 def stream_config():
38     stream_cfg = ost_pb.StreamConfigList()
39     stream_cfg.port_id.CopyFrom(tx_port.port_id[0])
40     return stream_cfg
41
42 def stream(stream_cfg):
43     s = stream_cfg.stream.add()
44     s.stream_id.id = 1
45     s.core.is_enabled = True
46     s.control.unit = 1
47     s.control.num_bursts = num_bursts
48     s.control.packets_per_burst = packets_per_burst
49     s.control.bursts_per_sec = bursts_per_sec
50
51     s.core.frame_len = frame_len + 4
52     return s
53
54 drone = DroneProxy(host_name)
55 drone.connect()
56
57 port_id_list = drone.getPortIdList()
58 port_config_list = drone.getPortConfig(port_id_list)
```

```
59
60  print('Port List')
61  print('---------')
62  for port in port_config_list.port:
63      print('%d.%s (%s)' % (port.port_id.id, port.name, port.description))
64      if iface == port.name:
65          print("IFACE: " + iface)
66          iface = port.port_id.id
67
68  tx_port = ost_pb.PortIdList()
69  tx_port.port_id.add().id = int(iface)
70
71  stream_id = setup_stream(1)
72  stream_cfg = stream_config()
73  s = stream(stream_cfg)
74
75  p = s.protocol.add()
76  p.protocol_id.id = ost_pb.Protocol.kMacFieldNumber
77  p.Extensions[mac].src_mac = int(mac_src, 16)
78  p.Extensions[mac].dst_mac = int(mac_dst, 16)
79  p = s.protocol.add()
80  p.protocol_id.id = ost_pb.Protocol.kEth2FieldNumber
81
82  p = s.protocol.add()
83  p.protocol_id.id = ost_pb.Protocol.kIp4FieldNumber
84  ip = p.Extensions[ip4]
85  ip.src_ip = int(ip_src, 16)
86  ip.dst_ip = int(ip_dst, 16)
87
88  p = s.protocol.add()
89  p.protocol_id.id = ost_pb.Protocol.kUdpFieldNumber
90
91  s.protocol.add().protocol_id.id = ost_pb.Protocol.kPayloadFieldNumber
92
93  drone.modifyStream(stream_cfg)
94
95  drone.clearStats(tx_port)
96
97  drone.startTransmit(tx_port)
98
99  # wait for transmit to finish
100 try:
101     time.sleep(1000)
102 except KeyboardInterrupt:
103     drone.stopTransmit(tx_port)
104     drone.stopCapture(tx_port)
105
106     stats = drone.getStats(tx_port)
107
108     print(stats)
109
110     drone.deleteStream(stream_id)
111     drone.disconnect()
```

B.5 saimanwong/tcpdump-capinfos

B.5.1 Dockerfile

```
# docker build -t saimanwong/tcpdump-capinfos .
# docker run --rm \
# --name host-tcpdump \
# --privileged \
# --network host \
# -v /tmp:/tmp \
# saimanwong/tcpdump-capinfos \
# <CAPTURE_DURATION_SEC> \
# <INTERFACE> \
# udp

FROM ubuntu:xenial
LABEL maintainer "Sai Man Wong <smwong@kth.se>"

WORKDIR /

COPY china.sources.list /etc/apt/sources.list
COPY docker-entrypoint.sh /

RUN apt-get update && apt-get install -y \
    tcpdump \
    wireshark-common \
    --no-install-recommends && \
    mv /usr/sbin/tcpdump /usr/bin/tcpdump && \
    rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["./docker-entrypoint.sh"]
```

B.5.2 docker-entrypoint.sh

```
#!/bin/bash
DURATION=$1
IFACE=$2
PROTOCOL=$3

tcpdump -B 16110 -G $DURATION -W 1 -i $IFACE -w /tmp/tmp.pcap $PROTOCOL > /dev/null
    ↪ 2>&1
```

B.6 Scripts

B.6.1 `run_experiment.sh`

```
#!/bin/bash

# ./send_receive_packets.sh
# <ENV | host, vm> \
# <TG_NAME | ostinato, mausezahn, iperf> \
# <ITERATION> \
# <FRAME_LEN> \
# <CAPTURE DURATION> \
# <LOG_DIR>

packet_size=(64 128 256 512 768 1024 1280 1408 1664 2048 3072 4096)

for i in "${packet_size[@]}"
do
    echo ./send_receive_packets.sh host ostinato 100 $i 10 ./data/data_host
    ./send_receive_packets.sh host ostinato 100 $i 10 ./data/data_host
    echo ./send_receive_packets.sh host mausezahn 100 $i 10 ./data/data_host
    ./send_receive_packets.sh host mausezahn 100 $i 10 ./data/data_host
    echo ./send_receive_packets.sh host iperf 100 $i 10 ./data/data_host
    ./send_receive_packets.sh host iperf 100 $i 10 ./data/data_host
done
```

B.6.2 `send_receive_packets.sh`

```
#!/bin/bash

# PACKET
ENV=$1 # host, vm
TG_NAME=$2 # ostinato, mausezahn, iperf
ITER=$3
FRAME_LEN=$4
CAP_DUR=$5
DATA_DIR=$6

# Host
# SERVER1=192.168.42.3
# SERVER1_PORT=22
# SERVER1_TX=enp0s25
#
# SERVER2=192.168.42.2
# SERVER2_PORT=22
# SERVER2_RX=enp0s25

# VM settings
SERVER1=localhost
SERVER1_PORT=2223
SERVER1_TX=enp0s8

```

```
25 SERVER2=localhost
26 SERVER2_PORT=2224
27 SERVER2_RX=enp0s8
28
29 mkdir -p ${DATA_DIR}
30
31 SERVER1_IP=$(ssh -p ${SERVER1_PORT} root@${SERVER1} /sbin/ifconfig ${SERVER1_TX} |
       ↪ grep 'inet addr:' | cut -d: -f2 | awk '{print $1}')
32 SERVER1_MAC=$(ssh -p ${SERVER1_PORT} root@${SERVER1} /sbin/ifconfig ${SERVER1_TX} |
       ↪  grep 'HWaddr' | awk '{print $5}')
33
34 SERVER2_IP=$(ssh -p ${SERVER2_PORT} root@${SERVER2} /sbin/ifconfig ${SERVER2_RX} |
       ↪ grep 'inet addr:' | cut -d: -f2 | awk '{print $1}')
35 SERVER2_MAC=$(ssh -p ${SERVER2_PORT} root@${SERVER2} /sbin/ifconfig ${SERVER2_RX} |
       ↪  grep 'HWaddr' | awk '{print $5}')
36
37 function tcpdump () {
38     sleep 5
39     echo "[TCPDUMP@SERVER2] CAPTURING"
40     ssh -p ${SERVER2_PORT} root@${SERVER2} "docker run --rm \
41         --name host-tcpdump \
42         --privileged \
43         --network host \
44         -v /tmp:/tmp \
45         saimanwong/tcpdump-capinfos \
46         $CAP_DUR \
47         $SERVER2_RX \
48         udp"
49     echo "[TCPDUMP@SERVER2] CAPTURE DONE"
50
51     capinfos host
52 }
53
54 function capinfos () {
55     echo "[CAPINFOS@SERVER1] ANALYZING CAPTURE"
56     ssh -p ${SERVER2_PORT} root@${SERVER2} "docker run --rm \
57         -v /tmp:/tmp \
58         --entrypoint capinfos \
59         saimanwong/tcpdump-capinfos \
60         /tmp/tmp.pcap" | tee /dev/stderr | \
61         grep "Data bit rate:" | \
62         awk '{print $4,$5}' >> ${DATA_DIR}/${ENV}_${TG_NAME}_${FRAME_LEN}_${ITER}.
               ↪ dat 2>&1
63     echo "[CAPINFOS@SERVER1] ANALYSIS DONE"
64 }
65
66 function ostinato () {
67     echo "[OSTINATO DRONE@SERVER1] STARTING"
68     ssh -p $SERVER1_PORT root@${SERVER1} "docker run --rm -d \
69         --name host-drone \
70         --privileged \
71         --network host \
72         saimanwong/ostinato-drone" > /dev/null 2>&1
73     echo "[OSTINATO DRONE@SERVER1] RUNNING"
```

```
74
75      sleep 10
76
77      echo "[OSTINATO PYTHON-API@SERVER1] STARTING"
78      ssh -p $SERVER1_PORT root@${SERVER1} "docker run --rm -d \
79          --name host-python \
80          --network host \
81          saimanwong/ostinato-python-api \
82          $SERVER1 \
83          $SERVER1_TX \
84          $SERVER1_MAC \
85          $SERVER2_MAC \
86          $SERVER1_IP \
87          $SERVER2_IP \
88          $FRAME_LEN \
89          1000000 \
90          10 \
91          50000" > /dev/null 2>&1
92      echo "[OSTINATO PYTHON-API@SERVER1] RUNNING"
93
94      tcpdump
95
96      # CLEANUP
97      echo CLEAN UP STARTED
98      ssh -p ${SERVER1_PORT} root@${SERVER1} "docker rm -f host-drone host-python"
99      echo CLEAN UP DONE
100 }
101 function mausezahn {
102     echo "[MAUSEZAHN@SERVER1] STARTING"
103     ssh -p $SERVER1_PORT root@${SERVER1} "docker run --rm -d \
104         --name host-mausezahn \
105         --privileged \
106         --network host \
107         saimanwong/mausezahn \
108         $SERVER1_TX \
109         0 \
110         $FRAME_LEN \
111         $SERVER1_MAC \
112         $SERVER2_MAC \
113         $SERVER1_IP \
114         $SERVER2_IP \
115         udp" > /dev/null 2>&1
116     echo "[MAUSEZAHN@SERVER1] RUNNING"
117
118     tcpdump
119
120     # CLEANUP
121     echo CLEAN UP STARTED
122     ssh -p ${SERVER1_PORT} root@${SERVER1} "docker rm -f host-mausezahn"
123     echo CLEAN UP DONE
124 }
125
126 function iperf () {
127     echo "[IPERF@SERVER1] STARTING"
```

```
128     ssh -p $SERVER1_PORT root@${SERVER1} "docker run --rm -d \
129         --name host-iperf \
130         --privileged \
131         --network host \
132         saimanwong/iperf \
133         $FRAME_LEN \
134         10000 \
135         1 \
136         $SERVER1_IP \
137         $SERVER2_IP" > /dev/null 2>&1
138     echo "[IPERF@SERVER1] RUNNING"
139 
140     tcpdump
141 
142     # CLEANUP
143     echo CLEAN UP STARTED
144     ssh -p ${SERVER1_PORT} root@${SERVER1} "docker rm -f host-iperf"
145     echo CLEAN UP DONE
146 }
147 
148 COUNTER=0
149 while [ $COUNTER -lt $ITER ]; do
150     ${TG_NAME}
151     let COUNTER=COUNTER+1
152     sleep 5
153 done
```

B.6.3 raw_data_to_latex.py

```
1  import numpy
2  import os
3  import re
4  import sys
5  
6  directory_src = sys.argv[1] # Directory of raw data
7  directory_dst = sys.argv[2] # Directory of latex data
8  
9  _nsre = re.compile('([0-9]+)')
10 def natural_sort_key(s):
11         return [int(text) if text.isdigit() else text.lower()
12                             for text in re.split(_nsre, s)]
13 
14 lst = []
15 p = []
16 
17 for filename in os.listdir(directory_src):
18     p.append(filename)
19 
20 p.sort(key=natural_sort_key)
21 
22 for filename in p:
23     if filename.endswith(".dat"):
```

```
24          path = os.path.join(directory_src, filename)
25          path_array = path.replace(directory_src + "/","").replace(".dat","").split("
                ↪ _")
26
27          with open (path) as f:
28              for line in f:
29                  temp = line.split()
30                  if temp[1] == "kbps":
31                      lst.append(float(temp[0])/1000)
32                  else:
33                      lst.append(float(temp[0]))
34
35
36          temp_path = directory_dst + "/" + path_array[0] + "_" + path_array[1] + ".
                ↪ dat"
37          text = path_array[2] + " " + str(numpy.mean(lst)) + " " + str(numpy.std(lst)
                ↪ ) + '\n'
38          f = open(temp_path, 'a')
39          f.write(text)
40          f.close
41          lst = []
42          continue
43      else:
44          continue
```

B.6.4 calculate_theoretical_throughput.py

```
1  import sys
2
3  link_speed = float(sys.argv[1]) # In Mbit
4  packet_size = float(sys.argv[2]) # In Bytes
5
6  PREAMBLE = 7.0
7  START_OF_FRAME_DELIMITER = 1.0
8  INTERFRAME_GAP = 12.0
9
10 packet_and_overhead = packet_size + PREAMBLE + START_OF_FRAME_DELIMITER +
       ↪ INTERFRAME_GAP
11 packets_per_sec = float( (link_speed) / (packet_and_overhead*8) )
12
13 print(sys.argv[2] + " " + str(round(packets_per_sec*packet_size*8,2)) + " 0")
```

Bibliography

[1] Docker 1.11: The first runtime built on containerd and based on OCI technology - Docker Blog. [Online]. Available: https://blog.docker.com/2016/04/docker-engine-1-11-runc/ [Accessed: 23-May-2017]

[2] A. Botta, A. Dainotti, and A. Pescapé, "Do you trust your software-based traffic generator?" *IEEE Communications Magazine*, vol. 48, no. 9, 2010.

[3] S. Molnár, P. Megyesi, and G. Szabo, "How to validate traffic generators?" in *Communications Workshops (ICC), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1340–1344.

[4] S. S. Kolahi, S. Narayan, D. D. Nguyen, and Y. Sunarto, "Performance monitoring of various network traffic generators," in *Computer Modelling and Simulation (UKSim), 2011 UkSim 13th International Conference on*. IEEE, 2011, pp. 501–506.

[5] S. Srivastava, S. Anmulwar, A. Sapkal, T. Batra, A. Gupta, and V. Kumar, "Evaluation of traffic generators over a 40Gbps link," in *Computer Aided System Engineering (APCASE), 2014 Asia-Pacific Conference on*. IEEE, 2014, pp. 43–47.

[6] S. Srivastava, S. Anmulwar, A. Sapkal, T. Batra, A. K. Gupta, and V. Kumar, "Comparative study of various traffic generator tools," in *Engineering and Computational Sciences (RAECS), 2014 Recent Advances in*. IEEE, 2014, pp. 1–6.

[7] Other Internet Traffic Generators. University of Napoli Federico II. [Online]. Available: http://www.grid.unina.it/software/ITG/link.php [Accessed: 21-May-2017]

[8] Wikipedia contributors. Packet generator. Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Packet_generator [Accessed: 25-May-2017]

[9] Wireshark wiki contributors. Tools. Wireshark. [Online]. Available: https://wiki.wireshark.org/Tools [Accessed: 23-June-2017]

[10] S. R. Piccolo and M. B. Frampton, "Tools and techniques for computational reproducibility," *GigaScience*, vol. 5, no. 1, p. 30, 2016.

[11] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[12] R. Chamberlain and J. Schommer, "Using Docker to Support Reproducible Research," 7 2014, [Online]. Available: https://doi.org/10.6084/m9.figshare.1101910.v1 and https://figshare.com/articles/Using_Docker_to_Support_Reproducible_Research/1101910.

[13] D. Turull, P. Sjödin, and R. Olsson, "Pktgen: Measuring performance on high speed networks," *Computer Communications*, vol. 82, pp. 39–48, 2016.

[14] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: a scriptable high-speed packet generator," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*. ACM, 2015, pp. 275–287.

[15] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. McKeown, N. Feamster, B. Felderman, M. Blott *et al.*, "Osnt: Open source network tester," *IEEE Network*, vol. 28, no. 5, pp. 6–12, 2014.

[16] M. Ghobadi, G. Salmon, Y. Ganjali, M. Labrecque, and J. G. Steffan, "Caliper: Precise and responsive traffic generator," in *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on*. IEEE, 2012, pp. 25–32.

[17] Network, devices & services testing - Spirent. Spirent Communications. [Online]. Available: https://www.spirent.com/ [Accessed: 25-May-2017]

[18] Ixia Makes Applications Stronger. Ixia. [Online]. Available: https://www.ixiacom.com/ [Accessed: 25-May-2017]

[19] T. Carstens and G. Harris. Programming with pcap. TCPDUMP/LIBPCAP. [Online]. Available: http://www.tcpdump.org/pcap.html [Accessed: 21-June-2017]

[20] iPerf - The TCP, UDP and SCTP network bandwidth measurement tool. The Iperf team. [Online]. Available: https://iperf.fr/ [Accessed: 21-June-2017]

[21] netsniff ng. netsniff-ng: A Swiss army knife for your daily Linux network plumbing. Github repository. [Online]. Available: https://github.com/netsniff-ng/netsniff-ng [Accessed: 21-June-2017]

[22] P. Srivats. Ostinato Network Traffic Generator. [Online]. Available: http://ostinato.org/ [Accessed: 21-June-2017]

[23] TCPDUMP/LIBPCAP public repository. TCPDUMP/LIBPCAP. [Online]. Available: http://www.tcpdump.org/index.html [Accessed: 23-June-2017]

[24] N. Bonelli, S. Giordano, G. Procissi, and R. Secchi, "Brute: A high performance and extensible traffic generator," in *Proc. of SPECTS*, 2005, pp. 839–845.

[25] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO," in *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*. IEEE, 2015, pp. 29–38.

[26] R. Luigi. netmap - the fast packet I/O framework. UniversitÃă di Pisa. [Online]. Available: http://info.iet.unipi.it/~luigi/netmap/ [Accessed: 25-May-2017]

[27] DPDK. Linux Foundation Project. [Online]. Available: http://dpdk.org/ [Accessed: 25-May-2017]

[28] PF_RING. ntop. [Online]. Available: http://www.ntop.org/products/packet-capture/pf_ring/ [Accessed: 25-May-2017]

[29] emmericp. MoonGen: MoonGen is a fully scriptable high-speed packet generator built on DPDK and LuaJIT. Github repository. [Online]. Available: https://github.com/emmericp/MoonGen [Accessed: 27-June-2017]

[30] TRex. Cisco. [Online]. Available: https://trex-tgn.cisco.com/ [Accessed: 27-June-2017]

[31] D. Raumer, F. Wohlfart, D. Scholz, P. Emmerich, and G. Carle, "Performance exploration of software-based packet processing systems," *Leistungs-, Zuverlässigkeits-und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen*, vol. 8, 2015.

[32] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Assessing soft-and hardware bottlenecks in PC-based packet forwarding systems," *ICN 2015*, p. 90, 2015.

[33] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle, "Comparing and improving current packet capturing solutions based on commodity hardware," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 206–217.

[34] Linux Foundation wiki contributors. networking:sk_buff. The Linux Foundation. [Online]. Available: https://wiki.linuxfoundation.org/networking/sk_buff [Accessed: 2-July-2017]

[35] J. Daniels, "Server virtualization architecture and implementation," *Crossroads*, vol. 16, no. 1, pp. 8–12, 2009.

[36] S. Srinivasan, *Cloud computing basics*. Springer, 2014.

[37] O. Cherkaoui, R. Menon, and H. Geng, "Virtualization, cloud, sdn, and sddc in data centers," *Data Center Handbook*, pp. 389–400, 2014.

[38] E. Bauer and R. Adams, *Reliability and availability of cloud computing*. John Wiley & Sons, 2012.

[39] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. IEEE, 2013, pp. 233–240.

[40] K. Michael. namespaces(7) - Linux manual page. [Online]. Available: http://man7.org/linux/man-pages/man7/namespaces.7.html [Accessed: 20-May-2017]

[41] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*. IEEE, 2015, pp. 342–346.

[42] G. Imesh. (2016, July) The Evolution of Linux Containers and Their Future. DZone Cloud Zone. [Online]. Available: https://dzone.com/articles/evolution-of-linux-containers-future [Accessed: 21-May-2017]

[43] Moments in Container History. Pivotal. [Online]. Available: https://content.pivotal.io/infographics/moments-in-container-history [Accessed: 21-May-2017]

[44] O. Rani. (2016, May) A Brief History of Containers: From 1970s chroot to Docker 2016. Aqua Security. [Online]. Available: http://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016 [Accessed: 21-May-2017]

[45] thilred. (2015, August) The History of Containers. Red Hat Enterprise Linux Blog. [Online]. Available: http://rhelblog.redhat.com/2015/08/28/the-history-of-containers/ [Accessed: 21-May-2017]

[46] Open Container Initiative. About. The Linux Foundation. [Online]. Available: https://www.opencontainers.org/about [Accessed: 21-May-2017]

[47] Projects. The Linux Foundation. [Online]. Available: https://www.linuxfoundation.org/projects [Accessed: 23-May-2017]

[48] moby. moby: Moby Project - a collaborative project for the container ecosystem to assemble container-based systems. Github repository. [Online]. Available: https://github.com/moby/moby [Accessed: 23-May-2017]

[49] What is Docker? Docker. [Online]. Available: https://www.docker.com/what-docker [Accessed: 23-May-2017]

[50] docker. libcontainer: PROJECT MOVED TO RUNC. Github repository. [Online]. Available: https://github.com/docker/libcontainer [Accessed: 23-May-2017]

[51] opencontainers. runc: CLI tool for spawning and running containers according to the OCI specification. Github repository. [Online]. Available: https://github.com/opencontainers/runc [Accessed: 25-May-2017]

[52] containerd. containerd: An open and reliable container runtime. Github repository. [Online]. Available: https://github.com/containerd/containerd [Accessed: 25-May-2017]

[53] J. Petazzoni. Anatomy of a Container: Namespaces, cgroups & Some Filesystem Magic. [Online]. Available: https://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon [Accessed: 8-July-2017]

[54] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 171–172.

[55] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, and N. Thoai, "Using docker in high performance computing applications," in *Communications and Electronics (ICCE), 2016 IEEE Sixth International Conference on*. IEEE, 2016, pp. 52–57.

[56] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig, "Ten simple rules for reproducible computational research," *PLoS computational biology*, vol. 9, no. 10, p. e1003285, 2013.

[57] rjmcmahon. iperf2. SourceForge project. [Online]. Available: https://sourceforge.net/projects/iperf2/ [Accessed: 21-June-2017]

[58] esnet. iperf: "Serverless client" UDP test · Issue #194 · esnet/iperf. [Online]. Available: https://github.com/esnet/iperf/issues/194 [Accessed: 21-June-2017]

[59] K. Dustin. mausezahn - a fast versatile packet generator with Cisco-cli. Ubuntu Manpage Repository. [Online]. Available: http://manpages.ubuntu.com/manpages/xenial/man8/mausezahn.8.html [Accessed: 21-June-2017]

[60] pstavirs. ostinato: Ostinato - Packet/Traffic Generator and Analyzer. Github repository. [Online]. Available: https://github.com/pstavirs/ostinato [Accessed: 21-June-2017]

[61] Winpcap. Riverbed Technology. [Online]. Available: https://www.winpcap.org/ [Accessed: 23-June-2017]

[62] T. Meyer, F. Wohlfart, D. Raumer, B. E. Wolfinger, and G. Carle, "Measurement and simulation of high-performance packet processing in software routers," *Leistungs-, Zuverlässigkeits-und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen*, vol. 7, 2013.

[63] 6.6. Internal networking. Oracle Corporation. [Online]. Available: https://www.virtualbox.org/manual/ch06.html#network_internal [Accessed: 30-June-2017]