ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Networked Systems
ETH Zürich — seit 2015

High-speed Traffic Generation

Semester Thesis

Author: Leonardo Rodoni

Tutor: Tobias Bühler

Supervisor: Prof. Dr. Laurent Vanbever

March 2020 to June 2020

Abstract

In recent years, the overall Internet traffic has been increasing considerably, along with a constantly rising number of connected devices, but mostly due to the usage paradigm shifting towards larger utilization of video streaming services and conference calls. Commodity hardware manufacturers managed to keep up with this trend, as nowadays standard servers can support multiple 40Gbps and 100Gbps Ethernet Interfaces. This paves the way for the development of applications and devices which run at very high bandwiths, such as routers and switches but also firewalls, proxies or packet-inspection tools. These appliances then require extensive testing before deployment, and this is a typical use-case where a high-speed traffic generator comes in handy. This thesis analyses and compares software-based solutions for Internet traffic generation that can cope with these extremely high bandwidths. We will introduce the Data Plane Development Kit (DPDK), which is the technology enabler for these tools to be able to process Internet packets at such high speeds. One of the traffic generators that we tested, Moongen, manages to saturate a 10Gbps glass-fiber link with 60 byte sized Ethernet packets by utilizing a single CPU core, which would be unimaginable even to think of for a standard Linux program.

Contents

Chapter 1

Introduction

A network traffic generator is a specialized tool used to create Internet packets and inject them into a network. Traffic generation tools aim to mimic real Internet traffic, thus in industry and among the research community they are employed to benchmark the performance of newly developed devices and applications. Packet generators are implemented either as software programs or over hardware platforms, with the latter being faster and more precise. However, hardware solutions are usually very expensive and limited in configurability and range of offered features. Moreover, modern software traffic generators are programmed on top of fast IO frameworks, such as DPDK, which enables them to reach very high packet rates with commodity NICs on standard Linux servers, hence without the need to buy costly additional specialized hardware. Processing and generating high amounts of traffic is very resource intensive, especially for small sized packets, and with the standard Linux Kernel's networking stack it was previously impossible to saturate 100Gbps links with software running on a single server. Nowadays, DPDK-based programs can cope with these speeds with standard server CPUs.

Over the recent years the Internet has undergone a huge rise in global traffic, and this trend is set to continue. One of the most influential reasons for this increase is video streaming, which makes up 60% of the global Internet traffic. To handle these enormous volumes, the hardware industry is constantly improving the packet processing technology. Nowadays servers can support multiple 40Gbps and 100Gbps Ethernet interfaces, and the Ethernet Alliance's 2020 Technology Roadmap[1] expects even higher speeds up to 1.6Tbps to become standard in the years to come. This opens the way for the development of new devices and applications that support these extremely high bandwidths. New appliances then require extensive testing before deployment, and this is where high-speed traffic generators come into play.

In our work we have thoroughly explored, tested and finally compared three modern software DPDK-based packet generators: Moongen, Warp17 and T-Rex. These tools, thanks to DPDK, are able to saturate multiple high-speed links up to 200Gbps when running on a single server. We have been able to completely fill a 10Gbps link with small 60 byte sized UDP packets with Moongen using two CPU cores and with T-Rex using 4 cores. Warp17 instead, is a stateful tool, focused on generating and managing the state of TCP flows at high speeds. Warp17 is capable of handling tens of millions of TCP sessions per second, which would not be feasible with standard Linux software.

This thesis aims to provide the reader with sufficient insight and information to be able to choose among these tools which one suits best for his use case. The rest of the report is organized as follows: Chapter 2 provides the reader with the required background knowledge in the topic of Internet traffic generation. Chapter 3 offers an in-depth presentation of Moongen, Warp17 and

[1] `https://ethernetalliance.org/technology/2020-roadmap/`

T-Rex by explaining some of the main features and illustrating performance results based on our tests. Chapter 4 presents the more relevant and interesting results of our work by laying out a broad comparison of the tools. As previously mentioned, we aim at providing the reader with sufficient information to choose a traffic generator, thus we try whenever possible to offer insight on the programs utilization and functionalities based on our personal experience. Table 4.1 summarizes some of our results and serves as a quick reference for comparison. Finally, Chapter 5 briefly summarizes the thesis and illustrates possible ideas for continuation of our work in the future.

Chapter 2

Background and Related Work

This chapter aims to provide the reader with the required background knowledge in the topic of Internet traffic generation. In Section 2.1, we introduce traffic generators, along with the basic ideas behind them and their more common use cases. Then, in Section 2.1.1 the attention will shift towards modern software traffic generation tools, on which this work is based. In this regard, we introduce the Data Plane Development Kit (DPDK [8]), upon which most of the modern high-speed traffic generators are built. Finally, in Section 2.2 we present some interesting papers describing related work comparing high-speed traffic generators.

2.1 Traffic Generators

Traffic generation systems, or packet generators, are specialized tools used to inject Internet packets in the network in a controlled way [4]. Network engineers and researchers rely on traffic generators to analyse the performance of a device or a newly developed application [1], which may also be referred to as "Device Under Test" (DUT) respectively "System Under Test" (SUT). A packet generator allows to evaluate the performance of the DUT or SUT by generating and sending packets through specific ports. It also provides a way to output important statistics for each interface, usually as CLI output, such as bandwidth (transmitting and receiving rates), latency, lost packet counts, and a bunch of other statistics depending on the tool [3]. This statistics then enable the user to generate a benchmark of his device or application, define its limits in terms of workload and identify some possible design flaws.

Validating a new apparatus under different loads to determine its performance in terms of throughput (maximum bandwidth at which none of the frames are dropped by the device [25]) and latency is a crucial step in the production life-cycle [1]. Traffic generators provide users with a way to mimic actual network traffic at high rates, with which a new network can be stress-tested. This may help identify any vulnerable areas or devices, that if subject to high load operation may experience packet loss or drop connections, giving users a clear understanding of what their network can manage, where the areas of concern are, and what points of the network are at a higher risk of becoming congested [7].

For these reasons, one of the main goals when it comes to traffic generators development is for the tool to be able to replicate realistic Internet traffic patterns, so that the performance of the network can be evaluated in an accurate manner. Moreover, as in the past few years the networks have become extremely diverse in terms of protocols, applications, and connected devices, it is increasingly important that traffic generators allow for high flexibility [4]. A packet generator should enable each user to efficiently produce specific traffic patterns that match the required network conditions where the DUT will be put into operation.

Packet generators can be implemented as software but also over hardware platform, e.g. on an FPGA [1, 5]. The latter are faster and more accurate, as the packet processing functionality is directly implemented on hardware, but they are typically proprietary devices and thus closed-source and very expensive. Hardware based solutions for traffic generation are also in many cases inflexible, as providing high configurability in a hardware based product results in high costs [5]. Software-based traffic generators, on the other hand, have lower performance and accuracy, but provide a much higher degree of configurability and more importantly are usually open-source [1]. Moreover, in recent years, the rise of fast IO frameworks (such as DPDK [8]) is making software packet generators even more attractive, as they can reach very high packet rates on commodity NICs [1]. For these reasons, in our work we have only considered software based packet generators.

There are many software traffic generators and they can be distinguished in two different classes, depending on the network stack that they utilize: traditional software packet generators and modern software packet generators. Traditional packet generators, such as trafgen [27] and Packet-Sender [21], rely on the network stack implemented in the operating system's kernel. Using the standard kernel IO interface allows for high compatibility, as the tool can simply use all the features implemented in the kernel's network stack, and all protocols already supported by the OS do not need to be re-implemented [1]. However, the OS network stack is mainly optimized for stability and compatibility with all other network stack implementations. It is therefore not specifically designed to generate high amounts of traffic at high speeds, and this results in strongly reduced performance for the tool when it comes to generate traffic at more that 1Gbps (especially with small packets).

In recent years the available bandwidth for commodity hardware has been increasing rapidly, and nowadays normal servers can support multiple 40Gbps and 100Gbps Ethernet interfaces. As of early 2016, core router platforms from Cisco, Juniper and other major manufacturers support 400Gbps full duplex data rates per slot [26]. In 2019, 200Gbps Ethernet cards have been standardized by the IEEE, and the Ethernet Alliance's 2020 technology roadmap [12] expects speeds of 800Gbps and 1.6Tbps to become standard possibly between 2023 and 2025 [26]. At these huge bandwidths, traditional traffic generators are simply of no use, and thus modern high-speed software traffic generators were developed.

The following section presents an overview of existing modern traffic generation tools, illustrating the basic concepts and ideas behind them and introduces the DPDK framework.

2.1.1 Modern Software Traffic Generators

Modern software traffic generators bypass the entire OS network stack and use another framework, or IO API[1], to access the interfaces [1]. This way the tools can be optimized for high speeds and low latency, as now the overhead on packet processing from the OS kernel's network stack has been removed. The Linux kernel's network stack is interrupt-driven, and thus creates a high number of context switches when a lot of packets need to be processed. This context switches are costly in terms of performance, especially at high bandwidth rates. By using these special frameworks that are polling based, such as DPDK [8] or PF_RING ZC [22], most of the expensive context switches can be avoided, and higher computing efficiency and packet throughput can be achieved. Polling refers to continually sampling the status of an external device as a synchronous activity [24]. In our work we will thoroughly test three DPDK-based traffic generation tools: Moongen [19], WARP17 [35] and Cisco's T-Rex [28].

DPDK, or the Data Plane Development Kit, is an open-source software project managed by the Linux Foundation which provides a set of data plane libraries and NIC polling mode drivers

[1] Input/Output Application-Programming-Interface

for offloading packet processing from the operating system's kernel to processes running in the user space [11]. This concept is illustrated in Figure 2.1.

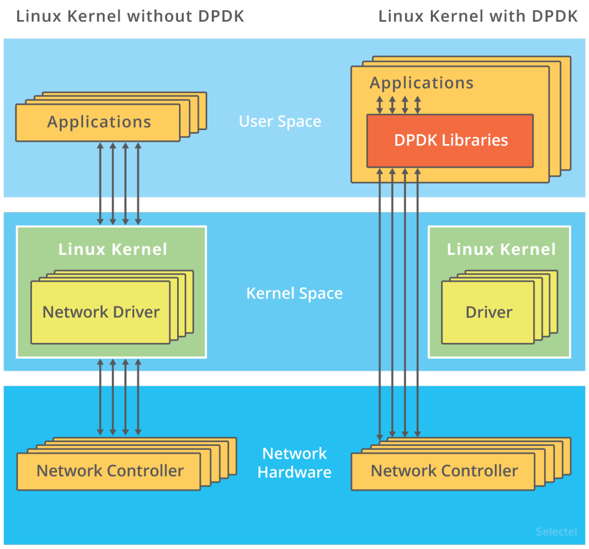


Figure 2.1: Difference between applications that use the Linux kernel only and application that integrate the DPDK framework to access NICs. In the latter case, the kernel doesn't step in at all during interactions between the application and the network interfaces: these are performed via the DPDK libraries and drivers. (image source: [9])

The data plane consists basically of a routing table, and its job is to make decisions on the actions to be taken when packets arrive at an inbound interface, i.e. drop or forward the packets [3]. The DPDK framework operates by creating an Environment Abstraction Layer (EAL) that contains a set of libraries for specific hardware and software environments. This abstraction layer hides the specifics of the environment, such as huge-pages setup and multi-thread support, and provides a standard programming interface to libraries and other hardware or operating system elements [11]. Developers can then link to the library to create software applications built on DPDK.

As briefly mentioned above, DPDK implements a low overhead run-to-completion model (scheduling model in which each given task must run until it finishes) by accessing interfaces via different drivers than the ones used by the OS kernel and with a set of libraries. The DPDK drivers and libraries are optimized for polling, thus eliminating the performance overhead of interrupt processing [11], and busy waiting, which allows for precise timing and thus increases packet transmission accuracy. Busy-waiting, or spinning, is a technique in which a process repeatedly checks to see if a certain condition is true [6], e.g. if an interface just received packets that need to be processed. The libraries further implement a memory manager, which pre-allocates huge-pages so that memory access is as fast as possible during program execution. A queue manager, a buffer manager and other packet processing helper libraries are also implemented [11]. DPDK spawns threads that handle the receiving and processing of packets from the assigned queues [10] and they do this on a time loop, therefore anything interrupting these threads (including Linux kernel tasks) might cause packet loss. For this reason when possible it should run on dedicated CPU cores for better performance.

DPDK is the most widely used high-speed IO API, mainly because of its high compatibility, as it supports all major CPU architectures and NICs from multiple vendors [8]. Besides Intel, which is the creator of the project, many companies contribute to the DPDK development. Intel, Mellanox and Broadcom also periodically test their NICs with the newest DPDK versions and publish performance reports[2]. This highlights the fact that interest on DPDK is growing among the telecommunication industry. In fact it has many possible applications, apart from traffic generators, and in some cases (e.g. firewalls, deep-packet-inspection or caching) it can replace traditional hardware solutions in the Internet Backbone, where extremely fast packet processing is the main requirement [10].

2.2 Related Work

The goal of this work is to test, analyse, evaluate and compare three high-speed DPDK-based software traffic generation tools. In this section we present similar studies that have been previously performed on this subject.

In particular, Soumya Mahalakshmi et al. [3] in 2016 have tested and compared four DPDK-based traffic generators: T-Rex, Pktgen [23], MoonGen and Ostinato [20]. The paper places particular emphasis on explaining the essential functionalities and architectural details of T-Rex, then provides a basic analysis of the other tools. The main goal is to provide an efficient framework and present methodology insights for fast packet processing in data plane applications. In this regard, the four software packages have been tested and compared. One of the main results presented in the paper is that MoonGen and Pktgen have the best efficiency and compatibility. In our work we have tested Moongen against two other modern traffic generators, and we also observed that when it comes to generating stateless Traffic, Moongen is indeed the more efficient tool.

Another relevant study was conducted by Emmeric et al. [1] in 2017. Their work investigates the properties and performs a comparative analysis of several high-performance software packet generators. The paper aims to "highlight the actual limitations in high-performance software packet generation, thus helping the research community to build better tools". Among the tools that have been tested are MoonGen and Pktgen, both DPDK-based, but also tools based on other high-speed IO APIs and some traditional packet generators. They focus on implications on precision when a given traffic pattern needs to be reliably generated at high rates. This is extremely relevant because if the generated pattern is not accurate, it can influence the observed performance of the System Under Test. They discovered that modern traffic generators easily satisfy performance requirements in terms of bandwidth (tests were conducted on a 10Gbps link), however precision problems start to arise for packet rates above 1Mpps (million packets per second). In our work, we also observed that with DPDK based tools it's fairly easy to fully saturate a 10Gbps link, and in most cases it's already possible by utilizing one or two CPU cores. Additionally, they inspected the latency measurement feature of modern packet generators, which thanks to the fast IO frameworks can be performed via hardware timestamping. This feature was discovered enabling better accuracy and reliability than any other pre-existing software-based solution.

In addition to the papers mentioned above, other studies exist, that consider traditional software packet generation tools operating at high-speed, such as Srivastava et al. [4] (2014), which compares the performance of three Linux-kernel-based packet generators over a 10Gbps link.

[2] `https://core.dpdk.org/perf-reports/`

Chapter 3

Analysis of Traffic Generators

This chapter offers an in-depth presentation of the three modern software-based packet generators that we consider in our work: Moongen (Section 3.1), Warp17 (Section 3.2) and T-Rex (Section 3.3). The tools have been thoroughly tested on a server equipped with two 10Gbps Ethernet interfaces, back-to-back connected with a glass fiber link. The hardware specifics of the server and the software versions that we deployed for testing are summarized in Appendix A.

The chapter is structured in the following manner: for each traffic generator we first give an introduction, describing how it works and what the developer's claims are in terms of performance and possible use cases. Then we explain in detail some of the important functionalities that in our opinion are relevant to better understand the tool capabilities. To this regard we already present relevant results from our tests. We also attempt to offer insights and useful intuition regarding the program use based on our experience.

3.1 Moongen

Moongen [19] is a flexible high-speed packet generator, that runs on top of the DPDK framework. It's claimed to "every so often" be able to saturate 10Gbps Ethernet links with minimum-sized packets [2] using a single core. Furthermore, thanks to the linear multi-threading scalability of DPDK, extremely high bandwidths and packet rates can be achieved by simply utilizing more cores. This is possible mainly thanks to the DPDK feature that allows to allocate multiple queues per NIC. Each configured queue is then statically assigned to a single CPU core. Since cores operate on independent queues, losses due to concurrent operations are avoided and performance scales up linearly with the number of cores. The tool developers have been able to test Moongen up to speeds of 178.5Mpps at 120Gbps [2].

Moongen is controlled through its API, and not by configuration files like for most software packet generators. Warp17 and T-Rex on the other hand, are both controlled by configuration files. The API makes DPDK packet processing capabilities available to user-controlled Lua scripts. Lua[1] is a lightweight, high-level programming and scripting language, primarily designed for embedded use in applications [18] and suitable for high-speed packet processing tasks at high packet rates [2]. This approach enables flexibility, as users can modify the Lua scripts to be called upon program execution (called "userscript") in a way that best suits their needs. In the userscript we can define more threads and initialize more queues, in order to scale up the available processing power. It's also possible to craft personalized packets to be sent or perform packet processing on the received packets [2].

[1] `https://www.lua.org/home.html`

In order to test and evaluate functionalities of Moongen in an efficient way we have prepared some bash scripts to automate parsing of certain arguments at program start-up. Some parameters like bandwidth, packet size and execution time can directly be passed as CLI arguments when running the tool. However, others like the number of CPU cores, whether to include latency measurement, or whether to send packet in full-duplex mode or in simplex mode, can only be changed by actively modifying the userscript. This is a downside of allowing for high flexibility, but thanks to the complete examples provided in the Moongen's Github repository we have found it quite easy and straightforward to adjust the userscript to our needs. We also think that for the purposes of stress-testing a device or application with Moongen, these parameters need to be decided only once at the beginning (given bandwidth and traffic pattern requirements), but then don't need to be changed anymore. The scripts we prepared are available in our Gitlab repository[2], to allow for quick replication of our tests.

3.1.1 Link Saturation

In this section, we evaluate the performance of Moongen in terms of maximal achieved bandwidth and packet rates on our testing setup. According to the developer's claims, the program should be able to saturate a 10Gbps link with 64 byte sized packets on a single CPU core [2]. We have found this claim to be true, by setting up the tool to generate one-directional traffic of raw Ethernet packets from one interface to the other via the 10Gbps glass fiber link on our server. A single thread, running on a single CPU, is able to fully saturate the link with 64 byte raw Ethernet packets, at a rate of 14.88Mpps.

At this point, it's important to make a consideration on the achieved bandwidth when the link is near full utilization. In fact the real bandwidth (measured on the sent Ethernet packets) is smaller than the bandwidth observed on the link, and this is particularly relevant for small-sized packets. Moongen clearly shows this in the real-time live CLI statistics by outputting two values: real bandwidth (based on packets sent by the tool) and "bandwidth with framing", which represents the actual bandwidth observed on the link. The graph in Figure 3.1 plots the maximal real bandwidth, or "bandwidth without framing", versus different packet sizes, and it helps to clarify this concept.

Framing, or frame synchronization, is the process of identification of frame alignment signals, which are sequences of bits or symbols that have the purpose of indicating to the receiver the beginning and end of the data within the stream of symbols or bits it receives [14, 13]. Then the receiver can extract only the data bits to be decoded or re-transmitted if necessary. As illustrated in Figure 3.1, to reach a bandwidth of 10Gbps with small packets, we need to send a high number of packets, thus in total there will be a lot more framing bits that need to be pushed in the link. This explains why for 64 byte sized packets we can only reach up to 7.6Gbps of "real" bandwidth with a fully saturated 10Gbps link.

As a next step, we have tested Moongen with streams of UDP packets instead of raw Ethernet packets. This caused some overhead, as expected, and the tool was no longer able to saturate the link with a single CPU core. Figure 3.2 shows the respective measurement results. This overhead comes from the real-time processing of the UDP header, which is implemented in the userscript to be called upon Moongen startup.

[2] `https://gitlab.ethz.ch/nsg/student-projects/sa-2020-06_traffic_generation/-/tree/master/code/MoonGen`

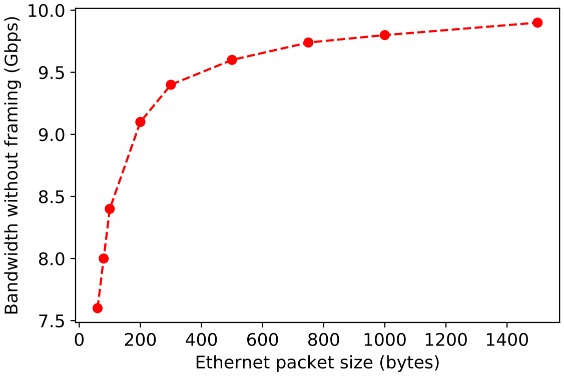


Figure 3.1: The “real” bandwidth, or “bandwidth without framing”, which is measured in terms of packets actually being sent by Moongen to the sending interface, is plotted here versus different packet sizes. During all these measurements the link is always completely filled, meaning that the “bandwidth with framing”, or link bandwidth, is reaching 10Gbps for all packet sizes. This means that the plotted bandwidth is the maximal achievable real bandwidth on a 10Gbps link. We immediately notice that for small packets a lot of the link bandwidth is taken up by framing bytes.

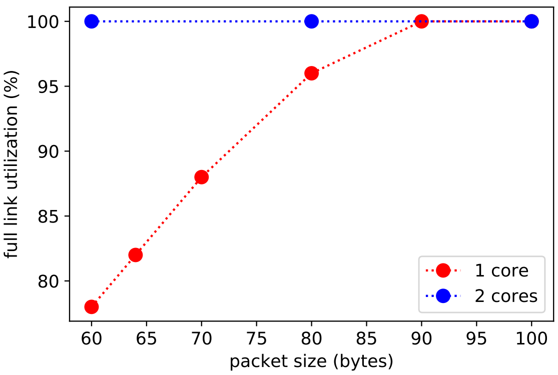


Figure 3.2: The link utilization, or “bandwidth with framing” as referred to above, for a stream of UDP packets is plotted here versus different packet sizes. With 2 CPU cores, Moongen manages to saturate the 10Gbps link also with small sized packets. Otherwise with a single core, it’s able to fill up the link only with packets bigger than 90 bytes. For 60 byte sized packets (the minimum packet size that Moongen can generate), the observed link bandwidth is 7.8Gbps.

3.1.2 Supported Protocols and other Testing Setups

Moongen supports in theory all existing protocols, as they can be manually implemented in the Lua scripts. The tool currently provides example scripts for UDP, TCP, ICMP, IPsec, and ARP traffic [2]. However, even though it can send TCP packets and any other manually crafted packets of existing stateful protocols, it is a purely stateless tool, as no TCP stack is implemented. It could be employed to simulate a TCP syn-flood attack (an example script about this is provided in the official Github repository), but it won't acknowledge any incoming packets nor keep any state for outgoing ones. It's worth mentioning that it's in theory possible to implement a TCP stack in Lua, however the performance would likely be bad, or at least much worse than an implementation in C such as the one of Warp17.

Moongen can also be deployed on a single DPDK-binded interface (directly by only declaring a single port as CLI argument). This can be useful for example when one needs to generate a traffic flow between two different physical servers, or to stress-test an application running on the OS kernel. In the latter case, the tool could be used to generate a stream of packets at high rates directed towards the application, to see how well it operates under heavy load conditions or under a possible DOS attack.

Given that Moongen does't have a TCP stack, deploying it on the receiving interface only doesn't make a lot of sense, as it will only detect but not react to any incoming packets. One possible use case for this setup (other than just a packet counter) is to employ the program as a packet capturing tool. The next section explains and illustrates the performance of Moongen's pcap functionality, which we have thoroughly tested in our work.

3.1.3 Packet Capture

Another interesting feature integrated in Moongen is the ability of capturing packets (or "dumping" packets). Like all other packet processing related operations in Moongen, this functionality needs to be implemented in the userscript that is called upon program execution. In order to integrate it with a standard UDP stream generation between the two interfaces, we had to include a dumper function (which was available as an example script in Moongen's Github repository), and we added some additional threads exclusively dedicated to capturing packets. This approach was fairly complicated and not so versatile, as in practice we had to merge two scripts into one to build the userscript. However this seems a fair solution, since due to how DPDK works, there cannot be two separate instances trying to access the NICs at the same time. In fact it's not possible to run two separate program tasks that access the same DPDK-binded interface, because DPDK needs to reset the NIC's queues and register each time at startup.

Figure 3.3 shows the evolution of the percentage of captured packets for different one-directional UDP packet streams. We have setup 4 threads to generate the UDP traffic and 2 threads to capture packets at the receiving interface. All the scripts are available in our Gitlab repository and can be used to replicate our tests. In order to compute the percentage of captured packets we have used the summarizing statistics of Moongen, which are outputted at the end of program execution. The total number of packets seen at the receiving interface is provided, and each dumper thread also logs the total amount of successfully captured packets. Each data point is an average over 5 different measurements, in each one of which we captured between 20 and 70 million packets (approximately 2-3 seconds of capture at 7Gbps, 5-6 seconds at 5Gbps and 9-10 seconds of at 2Gbps). We did this to ensure we had a fair and reliable measurement, because every time the percentage of captured packets slightly changed, especially with high bandwidths like 7Gbps.

In general we expected some performance loss also with respect to the ability to generate packet

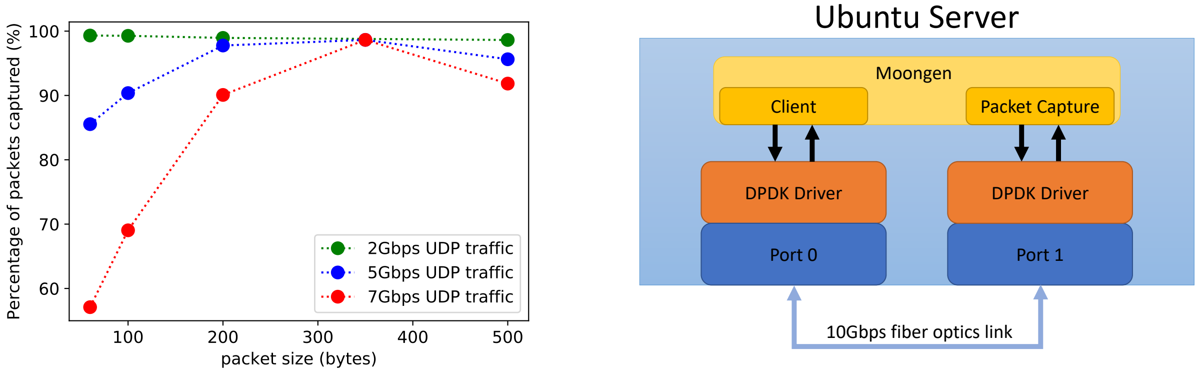


Figure 3.3: The percentage of captured packets by Moongen's pcap tool versus UDP streams with different packet sizes and bandwidths is plotted here (left). The UDP traffic bandwidth stands for the "real" bandwidth, measured on sent packets ("bandwidth without framing"). Packets are sent in a one-directional manner from one interface to the other. We can see that for 2Gbps UDP streams the dumper threads manage to capture all packets even for small 64 byte sized packets. For higher bandwidths, the tool starts to struggle with small packet sizes, but anyway it still manages to capture 85% of the traffic for a stream of 64 byte sized packets at 5Gbps. The diagram on the right side illustrates the testing setup.

streams at high rates, since now there are a lot more lines of code in the userscript, but this resulted to be true only at very high bandwidths. We were able to reliably reach bandwidths of 7Gbps even for small packet sizes, which means that for 64 byte sized packets the link is almost saturated (we have seen before that for 64 byte sized packets, the link is already fully saturated at 7.6Gbps of "real" bandwidth due to framing).

A last consideration on Figure 3.3 concerns the performance loss that we observe when capturing 500 byte sized packets. We initially speculated that this was because it takes more effort to write larger packets to a pcap file. However, Moongen provides an option to set a "snap-length" in bytes, which makes sure that we capture only the first part of each packet (e.g. only the headers) and the rest is discarded. We tried this option, but we obtained the same performance drop for 500 bytes packets. The issue might still come from a difficulty to process larger packets, but we weren't able to precisely determine the cause.

As a next step we compared Moongen's pcap tool with the classical t-shark[3] packet capturing software, which runs on the Linux OS networking stack. In order to achieve that we run Moongen as a client only instance generating a flow of UDP packets directed towards an interface binded to our server's OS kernel. This setup is illustrated in Figure 3.4 (right).

Figure 3.4 (left) plots the percentage of packets captured by t-shark versus UDP packet flows of different sizes. With this setup we weren't able to reliably generate traffic at 7Gbps, due to resource limitations. What happened is probably that t-shark was trying to use more threads, when stressed with a high amount of traffic, and these threads then entered in competition with Moongen. Because of this, in our tests, Moongen managed to generate 7Gbps traffic only for the first few seconds of execution, but then when t-shark started requesting more resources, Moongen's

[3] `https://www.wireshark.org/docs/man-pages/tshark.html`

performance reduced and it was no longer able to generate 7Gbps traffic anymore. Therefore we only have measurements for 2Gbps and 5Gbps traffic. This time, to compute the percentage of captured packets we used Moongen's statistics for the total number of sent packets, and the t-shark statistic for the amount of successfully captured packets.

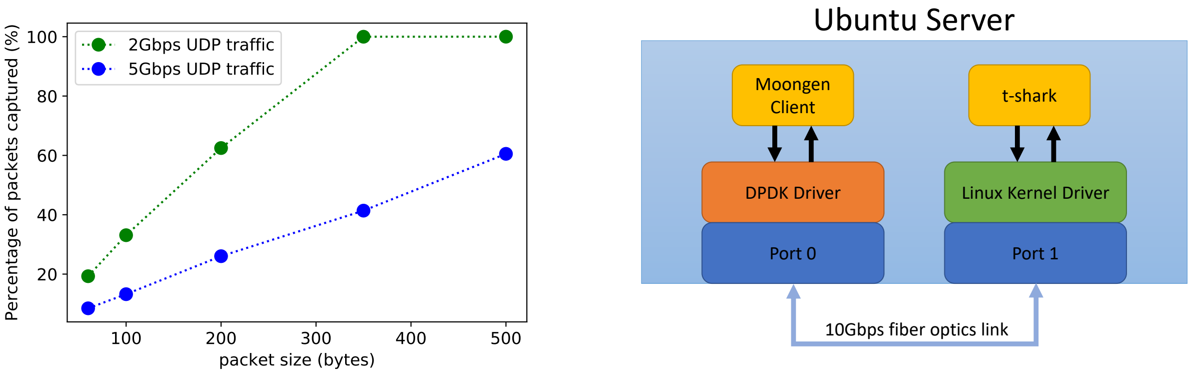


Figure 3.4: The percentage of packets captured by t-shark versus UDP streams with different packet sizes and bandwidths is plotted here (left). Again, packets are sent in a one-directional manner from one DPDK-binded interface to the other kernel-binded interface. We observe that t-shark manages to capture all the packets for a 2Gbps UDP stream with packet size bigger than 350 bytes. The diagram on the right side illustrates the testing setup.

From Figure 3.4 it is immediately clear that the performance of this capturing method is significantly lower than our previous test with the Moongen pcap feature. However, this drop in performance doesn't come from t-shark as one could expect. In fact, after running t-shark with a proper buffer size, no packet drop at all was signaled by the tool. With our setup, a 2GB buffer was sufficient for t-shark to capture all the received packets without dropping them. Nonetheless, a significantly high packet loss is detected: for a 64 byte sized packets stream at 5Gbps, only 8% of the packets arrive at t-shark! This means that most packets are previously dropped at the interface, before even reaching the t-shark socket. The reason behind this is probably due to the receiving interface being kernel-binded, thus much slower than a DPDK-binded interface, because of its interrupt-driven operational mode. When stressed with a very high packet rate the interface simply cannot keep up processing all the packets and some are dropped.

Also for this test, like we did with Moongen's DPDK-pcap tool, we captured approximately for 9-10 seconds at 2Gbps and for 5-6 seconds at 5Gbps. It's important to mention that, if we were to capture for a longer time period, a 2GB t-shark buffer may not sufficient to accommodate all packets. The fact that we had to increase the buffer size to avoid packet loss shows that t-shark processes packets slower than they enter the buffer, thus the amount of packets that need to be temporarily stored in it will constantly increase as long as there's traffic incoming, eventually filling the buffer up no matter its size.

3.1.4 Packet Timestamping and Latency Measurements

Besides the throughput, another essential performance characteristic of a system is latency. For developers it's crucial to determine the latency of different protocols on their new application or device, because it influences user performance considerably. Moongen can perform extremely precise latency measurements (up to a 64ns precision) by making use of modern NIC's hardware support for synchronization across networks [2]. This feature is however restricted to Ethernet and UDP packets.

In order to measure the latency of a protocol, Moongen periodically sends Precision Time Protocol (PTP) packets along with a UDP or Ethernet packet stream. These PTP packets need to have specific activated flags, and serve as a signal to the interface that a timestamp measurement needs to be performed for that packet. The NIC then, upon reception of a PTP packet, saves the timestamp in a hardware register, which needs to be read back by Moongen to retrieve the accurate timestamp. Most NICs in general write the time value in the register as late as possible in the transmit path and very early in the receive path, so that the measurement is more precise. This is also valid for the NICs that we utilized in our server [17]. Since the hardware clocks of the receiving and sending interface are different, Moongen synchronizes them by reading and subtracting their current clock time, before computing the latency. Since the clocks need to be synchronized there can't be more than one packet simultaneously on the fly, hence the PTP messages are configured to be sent periodically up to a maximum rate of 1Pkt/RTT. They can also be crafted in a way that they aren't distinguishable from the normal UDP or Ethernet packets in the stream [2]. This means that the latency measurement is effectively made on random packet samples in the traffic stream, thus highly reliable. The values are then averaged and collected in histograms. For our testing setup we have measured on average a 250-350ns latency for a UDP packet stream.

This hardware timestamping feature is very precise, however its use cases cannot be extended much. Due to the way timestamps are retrieved, which is by reading back a value from a register on the NIC, the maximal throughput of timestamped packets is limited. For this reason, timestamping of packets by the dumper function is still implemented using software clocks.

We compared the precision of software timestamps for packets captured by t-shark with packets captured by Moongen's dumper tool. The comparison was performed on 100Mbps traffic, so that both tools captured 100% of the sent packets with few CPU resources. Based on our observations, t-shark appears to have a much more precise timestamping mechanism, going up to nanosecond precision, whereas Moongen's dumper only records timestamps up to microsecond precision. However, this probably only depends on the timing function called on the Lua scripts.

3.2 WARP17

Warp17 [35], or simply Warp, is a stateful traffic generator, suited for generating high volumes of session-based traffic at very high session setup rates and data send rates. It runs on top of the DPDK framework, hence it includes a full DPDK-based dedicated TCP stack. The reason why this TCP stack was developed, is that the Linux kernel implementation cannot setup and manage a high number of flows at high rates. Thanks to the DPDK-based TCP stack, Warp17 can handle up to 17 million TCP sessions per second on a single server with 2 back-to-back connected interfaces and 34 threads [35]. An NGINX server running on a similar setup with 32 threads cannot handle more than 20k sessions per second, due to kernel software interrupts and thread processing (Linux Kernel is the bottleneck and not NGINX) [34]. Even though Warp's focus resides mainly on stateful traffic, it supports UDP streams as well.

Warp17 works by having users initialize test cases and parameters via a CLI. Server and client ports need to be initialized, along with many specific settings for each declared test case: TCP options such as windows-size or retransmission timeout, setup and teardown rates, data sending rate per connection, packet sizes, MTU, etc. It's also possible to define all tests and parameters in a configuration file to be passed as an argument when running the tool. Warp will then execute all the lines in the file right away.

To speed up the testing process, analogously as we did for Moongen, we prepared some scripts to automate the change of parameters in the configuration files. Otherwise, every slight modification in the testing setup, such as increasing the packet size, connection setup rate or tweaking any other parameter would require a manual change. All the scripts are available in our Gitlab repository[4].

3.2.1 Memory Settings

Another important aspect of Warp17 is how it allocates memory to be used by its threads. On Linux machines, by default the CPU allocates RAM to processes by chunks (or pages) of 4KB. If a process requires a lot of RAM, then it needs to request many 4KB sized pages. For this reason, memory access can suffer from an overhead due to having the program retrieve data stored in various locations in the RAM. The CPU needs to remember the address of each page allocated to each process, and if a process has many pages, then it takes time to find where the memory is mapped [15]. Warp17, instead, requires the memory to be already split up in 1GB sized chunks. The Linux kernel integrates a feature that makes it possible for the operating system to support memory pages greater than the default 4KB size up to a maximum of 1GB, the so called Hugepages. Using larger pages can greatly increase performance by reducing the amount of system resources required to access page table entries [16]. Hugepages are especially convenient, since they allow Warp to manage many TCP session states that are stored in memory in the most efficient way possible. To allocate 32GB of RAM with 1GB Hugepages, the CPU simply needs to store 32 addresses for Warp's processes. Each single TCP session state that needs to be stored in memory requires 1KB of RAM, and an allocated TCB (Thread Control Block) to locate it. With 32GB of memory, the CPU can allocate 32 million TCBs for Warp17, which means that the tool can then efficiently manage up to 16 million live TCP connections (16 million clients and 16 million servers) [35].

3.2.2 Link Saturation

This section evaluates the performance of Warp17 in terms of maximal achieved bandwidth with respect to session data send rates on our testing setup. Since Warp17's purpose is to generate stateful traffic, we have attempted to saturate the 10Gbps link in our server with TCP traffic. For the sake of comparison with the other traffic generators, we have also tested Warp's performance in generating UDP streams, and the results will be presented in Chapter 4.

In order to saturate the link, we defined a test case with a TCP client listening on one port and another test case with a TCP server listening on the other port. We further defined a maximum number of 200k TCP sessions with 10k setup rate and infinite data send rate. This way upon start of the tests, Warp17 brings up 10k sessions per second up to a maximum of 200k and immediately starts sending data packets from each session at the maximum possible rate, though only limited by allocated system resources. We repeated the tests for multiple data packet sizes, also increasing the number of utilized cores. Since the tool generates stateful traffic, we expected quite a performance overhead compared to the Link Saturation testing we performed with Moongen (Section 3.1.1). In fact for each TCP session Warp needs to store and manage a state in memory, acknowledge all received packets, and retransmit some of them whenever required. The results we obtained are illustrated in Figures 3.5 and 3.6.

[4] `https://gitlab.ethz.ch/nsg/student-projects/sa-2020-06_traffic_generation/-/tree/master/code/Warp17`

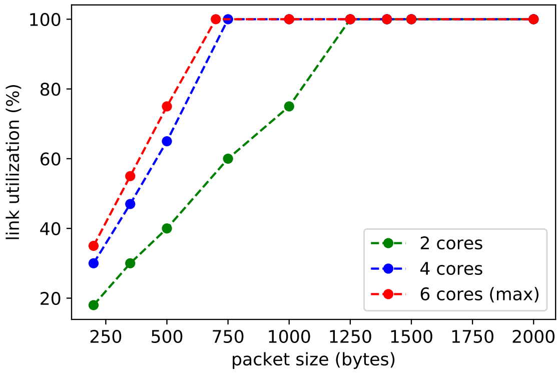


Figure 3.5: This figure plots the percentage utilization of the 10Gbps link when traversed by Warp17 generated TCP traffic. After the TCP sessions are established, data packets of different sizes are sent only from the client to the server as one-directional traffic flow. The tests have been repeated by launching the tool with 2, 4, respectively 6 cores dedicated to packet processing. Since Warp17 always allocates 2 CPU cores for CLI management [35], in our testing setup the maximum number of cores that can perform packet processing is 6.

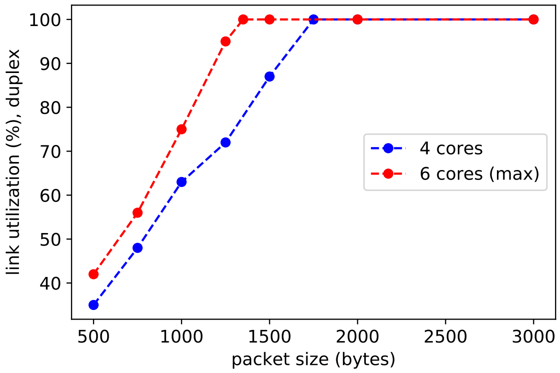


Figure 3.6: This figure plots the percentage utilization of the full duplex 10Gbps link when traversed by Warp17 generated TCP traffic. In this testing setup, data packets are sent by both client and server at the maximum possible rate. This time, the tests have been carried out by launching the tool with 4 and 6 cores dedicated to packet processing, since only 2 cores yielded a very low performance.

From Figure 3.5 we denote that Warp manages to saturate the 10Gbps link with 750 byte sized or bigger packets by using 4 cores. It's also interesting to observe that the performance increase if 6 cores are allocated instead of 4 is very little for our testing environment. On the other hand, if only 2 cores are used, the tool manages to saturate the link only starting from 1250 byte sized packets.

For the duplex tests in Figure 3.6, values are given in terms of utilization of the full duplex link, which we have computed by averaging the percentage utilization of the link in both directions. Warp's live UI displays transmitting and receiving percentage utilization at both client and server side. In general we observed that the tool is able to precisely balance system resources, such that the link is always utilized by the same percentage in both directions, when duplex traffic with equally sized data packets is generated. Regarding the performance, in this case we need at least 1350 byte sized data packets with 6 cores to be able to saturate the link.

3.2.3 TCP Stack

This section presents results of tests that we have performed on Warp's TCP stack implementation. One of the goals was to verify that all the main TCP features were implemented and correctly functioning. We were also interested in checking whether Warp's TCP implementation could work against another TCP implementation, such as the Linux kernel's stack. In order to achieve these goals, we have tested Warp17 as client against a small and simple python TCP server (also available in our Gitlab repository), which makes use of the Linux TCP stack. Warp17 can very easily be deployed as client only, by declaring one client test-case for a single interface.

Warp17 is implemented in a way that request (data packets send by client) and response (data packets sent back by server) sizes need to be declared in both client and server test-cases. After each request sent to the server, if the client doesn't receive back a response with the expected size, then it won't sent the next request. For this reason, if a Warp17 client interacts with a TCP server than upon reception of a request sends back packet of an unexpected size, Warp17 client will acknowledge again the previously received packet (stating that it received a wrong packet), and will stop sending requests. To circumvent this problem we had to code the python TCP server such that upon reception of a request, it replied back with the rightfully sized packet. Another solution, though only applicable to one-directional traffic, is to set the response size to 0 bytes when defining the test-case parameter. This way Warp17 doesn't expect a response from the server and continues to send requests.

In order to check the correctness of the sessions, we inspected the packets by capturing with t-shark at the server interface. In general we observed that Warp17 as a client-only is perfectly able to interact with another TCP server implementation. The TCP sessions are correctly established, acknowledgements from the python server are successfully received, and sessions are also tore down in the proper way (when specified in the test-case). One thing we observed is that when previously terminated session are reopened, the exact same packet flow as before is sent again (all new packets have equal sequence and acknowledgement numbers). In fact Wireshark identifies these resumed sessions as TCP retransmission, signaling a potential replay attack. However this is not a problem in terms of correctness of the TCP state machine implementation.

As a next step, we looked into the performance of Warp17's packet retransmission. Already when trying to setup 100 sessions with the python server, some retransmissions occurred. This can be explained by the fact that Warp's timer is set at a default value of 50ms. Due to the Linux kernel overhead, upon reception of 100 SYN packets in a short time span, the python server might take longer than 50ms to respond. Nevertheless, all 100 sessions were successfully established, which suggests us that Warp's TCP retransmission mechanism works correctly.

3.2.4 Periodical Output of Statistics

This section introduces a way to periodically log specific output statistics into a file, to make test data also available after the tool stops. Warp provides a User Interface, which can be called with a simple CLI command, and outputs live statistic on link utilization, established connections, data send rate, etc. However, for real-life testing purposes one would like to be able to run long tests without needing to periodically check the UI to collect data.

Warp17 allows users to define some conditions to determine whether tests were successful or not (e.g. if a certain number of connection was successfully established). This can be useful, but after running a long test it would be better to also have more statistics from the test during execution. To solve this, there are some python scripts in the Github repository, created as an attempt to automate Warp's operation. Some of these scripts apparently provide a way to generate a periodic output log of statistic. However, we found these scripts poorly explained and not documented, and we haven't been able to successfully run them on our server setup.

For the reasons explained above, we have integrated in our testing scripts[5] the optional feature to collect statistics by periodically writing Warp's CLI output in a text file. This is possible since Warp allows for different statistics and operational information to be manually dumped from the CLI (as an alternative of running the live UI). The script can be modified such that it outputs any statistics available among the CLI commands, and the scraping period can also be passed as an argument.

3.2.5 Internet-Mix Traffic

An important feature of a traffic generator, when it is to be employed for real-life testing, is that it should be able to generate realistic traffic patterns. This is possible with Warp, by declaring multiple test cases with different parameters in the configuration file. To illustrate how it can be done, we prepared a configuration file with 8 client test cases and 3 server test cases. Together, the clients establish 2.6 million TCP sessions, some of which going up and down periodically. Various flows of different data packet sizes and rates are then sent from the established sessions. We also attempted to integrate some UDP test cases but for some reason these tests did not work when merged with the TCP tests. The configuration file is available in our Gitlab repository and can further be expanded as needed.

In conclusion, we found this approach a bit complicated since each test case along with its parameters needs to be manually defined in the configuration file. If one would like to generate many test cases, such as 50 or more, in order to e.g. realistically replicate the traffic going through an Internet-core router, this approach would require a lot of time. A possible solution would be to create a script that would automate the writing of test cases in configuration files.

3.3 T-Rex

T-Rex [28] is an open source, low-cost, traffic generator fuelled by DPDK. It can scale up to 200Gbps with one server. It has three operational modes: stateful, stateless, and advanced stateful mode. Throughout our work we have concentrated mostly on stateful mode, but we also performed a few tests on stateless mode. Therefore in this chapter we principally focus on explaining the functionalities and presenting testing results on stateful T-Rex. Nevertheless for completeness we also provide a description and some insight on the other two modes, as they both have very interesting features.

We have tested T-Rex on a server with two back-to-back connected interfaces, like we did for Moongen and Warp17, but the tool can be operated also as a client only or server only instance. This can be achieved by tweaking the main T-Rex configuration file and setting a dummy port (port

[5] `https://gitlab.ethz.ch/nsg/student-projects/sa-2020-06_traffic_generation/-/blob/master/code/Warp17/tests/test1_tcp_sessions.sh`

not connected to an interface on the local machine) for either client or server side. This feature is particularly useful when the DUT, e.g. a link or a router, resides between two different physical servers. Or else it could be simply deployed as a client only traffic generator for stress-testing an application built upon the Linux Kernel stack.

In order to perform all our tests, we have created many configuration files, some taken or adapted from T-Rex's examples and some specifically written for our pcap files. All the yaml configuration files and pcaps we used can be accessed in our Gitlab repository[6], as to allow for quick replication of all tests.

3.3.1 Stateful Mode

T-Rex in stateful mode, or stateful T-Rex, is a traffic generator based on pre-processing and smart replay of real traffic templates [28]. More precisely it's a flow generator, as it makes use of one or multiple pcap files containing TCP or UDP flows, which it will then replay multiple times with new ip addresses and ports for client and server [30]. It's important to note that even though it generates stateful traffic, T-Rex only replicates pcap files with manually crafted or captured stateful flows, but it does not implement a full TCP/IP stack. Even though it can in principle work as a standalone client or server, the tool won't be able to interact with another TCP implementation. A TCP stack is implemented in the advanced stateful T-Rex mode, which we will introduce in Section 3.3.3.

The pcap file location and and the address pool range from which T-Rex takes the ip addresses to replicate the flows are both specified in a yaml configuration file, which is given as an argument when running the tool. In a pcap file there can be only one single flow, but many pcaps can be declared in the same configuration file. Along with each pcap, the respective cps (connections-per-seconds) rates needs to be specified, and these rates determine the total bandwidth that the test will generate. It's best practice to normalize the bandwidth of the file, i.e. tweaking the cps parameters such that it generates e.g. 1Gbps of traffic. T-Rex allows for an argument "-m" to be specified when running the program, which acts as a multiplication factor for the yaml file's bandwidth.

Since stateful T-Rex works with pcap files, this implies that the user has great flexibility in terms of what kind of traffic he wants to generate. A lot of examples are provided in the official package, such as single test cases for emulating VoIP calls, simple HTTP browser requests or Video Streaming flows. Moreover, T-Rex also provides Internet-Mix traffic examples, created with capture files from SFR[7] (French Internet Provider), that emulate real traffic going through core Internet routers. More information about this aspect is given in Section 3.3.1.2. In our work we have also often manually generated or extracted new pcap files to use for testing, as we'll explain throughout the next sections of this Chapter.

A limitation of this mode is that only TCP or UDP packets can be contained in the pcap files. Moreover, if a pcap contains a UDP packet flow, which is a stateless flow, T-Rex in stateful mode will simply take the pcap and replicate it up to a specified number of times specified by the cps parameter in the yaml file, as it does for TCP packets. This is called mimicking stateless flows, but in practice T-Rex is generating UDP streams each time with new IP addresses and ports (and even storing state information in memory), thus actually treating them like stateful flows. More flexibility with respect to these issues is given by the stateless mode, where UDP traffic can be

[6] `https://gitlab.ethz.ch/nsg/student-projects/sa-2020-06_traffic_generation/-/tree/master/code/TRex`

[7] `https://www.sfr.fr/`

generated as a stateless stream of packets, and where all kinds of protocols can be integrated by manually crafting packets with Scapy[8].

A final aspect worth mentioning regards the tool's statistic output. T-Rex provides a live UI where link and port statistics are displayed live, but it doesn't give a way to periodically drop statistics into a file during program execution. This could be achieved with a similar approach as we did for Warp17 (Section 3.2.4). However, in the T-Rex distribution there are programs specifically built to simulate T-Rex's execution on a yaml file and produce a verbose statistics output or if requested even a whole pcap file containing the traffic that would be generated by running the main T-Rex tool.

3.3.1.1 Link Saturation

This section illustrated the results of our attempts to saturate the 10Gbps link on our testing setup with T-Rex traffic. Given that we are considering a tool specifically built to generate stateful traffic, we have first tried to saturate the link with TCP flows. To allow for a comparison with Warp17, we have captured some Warp-generated TCP flows with various data packet sizes to replay with T-Rex. It's important to bear in mind that this doesn't allow for a direct comparison, since T-Rex simply replicates the flows in the pcap files with a fix number of captured data packets. We captured Warp17 flows with 1 TCP session sending at 10Mbps rate for an average of 10s, generating pcap files with 200 to 600 packets. Warp17 on the other hand would send many more data packets until the TCP session is closed. However, we think this should give an idea on the performance of T-Rex handling stateful flows. The test results are plotted in Figure 3.7 for a one-directional flow of data packets.

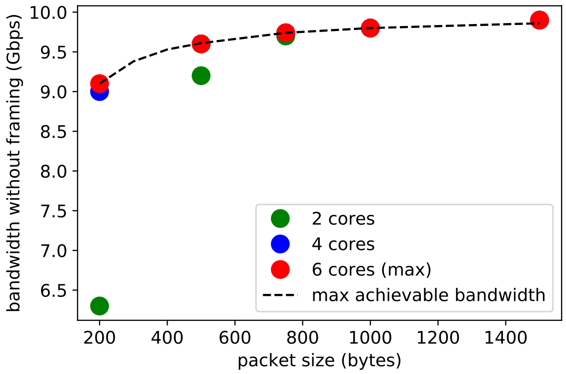


Figure 3.7: The bandwidth without framing (refer to Section 3.1.1 to understand framing) is plotted here for one-directional TCP flows with various data packet sizes. The black dotted line represents the maximum achievable bandwidth without framing on the 10Gbps link.

From Figure 3.7, we see that the tool can almost saturate the link with one-directional traffic for data packet sizes bigger than 200 bytes with 4 cores, and it can do it easily with 6 cores. Utilizing only 2 cores, T-Rex is able to generate traffic at 6.3Gbps for 200 bytes packets, which is around 70% full link utilization. With 2 cores, it manages to saturate the link for packets bigger than 750bytes. Comparing with the Warp17 results, it's immediately clear that T-Rex performs much

[8] https://scapy.net/

more efficiently. However, this is mainly due to the fact that it doesn't have any overhead coming from TCP stack operations. Warp17 in fact needs to manage a real TCP state for each session, which takes time in terms of CPU operations.

As previously mentioned, stateful T-Rex can mimic stateless UDP traffic streams. In the interest of performing a comparison with the other traffic generators, we have also tested stateful T-Rex's performance in generating UDP streams, and the results will be presented in Chapter 4.

3.3.1.2 Internet-Mix Traffic

In this section we illustrate how stateful T-Rex can be employed for generation of realistic traffic patterns at high speeds. As previously mentioned, the T-Rex package comes with various examples, and some of them are specifically constructed to generate Internet-Mix traffic, which is meant to be a realistic reproduction of real traffic traversing core Internet routers. The pcaps utilized by these examples are extracted from traces provided to T-Rex by SFR, a French Internet Provider. In order to construct the configuration files, the T-Rex developers first analysed the SFR traffic to determine the percentage of appearance of specific flows. They found that on average 70% of the total bandwidth is made of TCP packets and the other 30% are UDP. The average packet size is 580 bytes, and each flow has on average 50 packets. Furthermore they observed that around 30% of the traffic belongs to Video Streaming, 20% to HTTP Browsing and the rest to other HTTP traffic or other protocols [32]. Based on this information, they constructed a yaml file that replicates the SFR distribution, and normalized the file's bandwidth at 1Gbps. In our work we have tested some of these Internet-Mix SFR examples, and we were able to saturate the 10Gbps link using 2 cores. We have found this to be a very efficient and simple way to generate realistic traffic patterns at high bandwidths.

As a next step we have decided to personally construct our yaml and pcap files to generate realistic traffic. In order to do that we have extracted some TCP flows from a CAIDA[9] trace containing 100 million packets. Since in the trace only the header of packets is available (the payload has been snapped away to save space), we also had to manually add a random payload, which we could easily do with Scapy. All the python files we used to find and extract the flows are available in our Gitlab repository[10]. Most of the flows in the CAIDA trace are unidirectional, meaning that the other direction passes through another router, and very few of them were complete. Also, most of them are very small flows (few packets). Based on these observations, we then constructed a yaml file with the new pcaps, and tweaked the cps parameters accordingly. It's important to note that this was a rough estimate, far from a precise calculation of the percentage occurrence of flows. This said, with our approach we aim at providing some ideas and hints on how people can build their own personalized yaml file for real-life traffic generation. Our scripts can be extended on many aspects, such as building a more precise mapping of the traffic going through the CAIDA router, or integrating support for UDP packets and other protocols.

A problem that we encountered while writing the yaml configuration file, was that T-Rex provides no way to delay the start of replication of certain pcap files. Through the parameters, it's only possible to specify the rate at which the pcap should be replied, but not the time instant when the tool should start replying it. Upon T-Rex execution, the tool starts generating traffic from all pcaps at the same time. This could be an issue if someone would like to run a long test with variations on the generated traffic over time. We have found no solution to address this issue with T-Rex, thus we suggest to use Warp17 if this feature is particularly required.

[9] `https://www.caida.org/home/`

[10] `https://gitlab.ethz.ch/nsg/student-projects/sa-2020-06_traffic_generation/-/tree/master/code/TRex/caida`

3.3.2 Stateless Mode

T-Rex in stateless mode, or stateless T-Rex, focuses on generating streams of packets without storing any state in memory. The developers claim it can generate about 10-22 million packets per second per core. Furthermore, it supports multiple traffic profiles per interface, which in turn can contain multiple parallel streams (up to 10k). The traffic profiles need to be defined in a python script, which will then be called upon T-Rex's execution, and contains the declaration of streams, bandwidth rates, and other options such as whether to collect statistics, change the packet on the fly, or send bursts of packets instead of a continuous flow. Stateless T-Rex is extremely flexible in terms of packets and protocols that can be generated. Packets can be either imported from a pcap file as in the stateful mode, or they can be generated directly with Scapy in the python file. Thus, any protocol can be easily implemented, and malformed or incomplete packets can be included in a stream as well [33]. Stateless T-Rex enables L2/L3 testing, mostly relevant to test a switch or a router, but it doesn't provide the ability to emulate a L7 application.

In order to try out the tool, we have attempted to saturate the 10Gbps link on our server with stateless T-Rex traffic. We have created some python scripts, adapted from T-Rex's examples, importing UDP packets from pcaps that we already had available since used for previous testing. The scripts generate a one-directional UDP stream of packets with varying size. The results are presented in Figure 3.8.

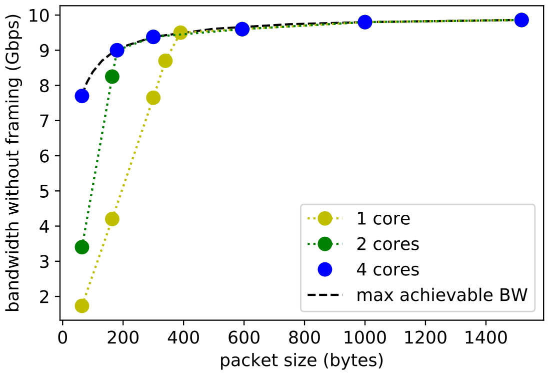


Figure 3.8: The bandwidth without framing is plotted for a one-directional UDP stream with varying packet size. The black dotted line represents the maximal achievable bandwidth without framing for the 10Gbps link.

From Figure 3.8 we observe that the tool is able to saturate the link with small sized packets only with 4 or more cores. With one core, stateless T-Rex is able to saturate the link only with UDP streams with 400 byte sized packets or bigger. We can directly compare this results with the testing performed on Moongen, and we immediately see that for small packets Moongen is much more efficient. In fact, it could saturate the 10Gbps link with a single core and 90 byte sized packets and already with 2 cores and 60 bytes packets. The reason for this may be that T-Rex has a lot more libraries and dependencies, which can cause an overhead in packet processing. Also, as it can be seen in the Appendix A, the version of T-Rex that we tested utilizes a newer DPDK version, which has been found being slower than the older versions, due to increased overhead [1].

In Chapter 4 the performance of this mode will be also compared with the other packet generators that we have analysed in our work.

3.3.3 Advanced Stateful Mode

T-Rex in advanced stateful mode (ASTF) is a packet generator which supports a user space TCP stack for emulating L7 protocols, such as HTTP [29]. The TCP stack is implemented on top of DPDK for better performance, as it was the case for Warp17. The developers claim it can reach up to 200Gbps of bandwidth for realistic flows, i.e. with around 600 bytes of average size and 1500 bytes maximal size. Some possible use cases are testing a router, a firewall or a deep-packet-inspection (DPI) engine. Furthermore, like the stateless mode, it runs via python scripts and thus allows for high flexibility in terms of packet processing and integration of functionalities. It also seems to have a nice automation support, which would be a great advantage if compared to Warp17.

In our work we haven't tested T-Rex in this mode, and it could be interesting as a future work. It's important to note, however, that this mode is still under heavy development. As stated in the official documentation[11], the tool is under constant improvement and many features have not yet been integrated.

[11] https://trex-tgn.cisco.com/trex/doc/trex_astf.html

Chapter 4

Results

This Chapter presents the more relevant and interesting results of our work. For this purpose, we lay out a broad comparison of the three modern software traffic generator tools that we analysed and thoroughly introduced in Chapter 3. We aim at providing the reader with sufficient information to choose the traffic generator which best suits his use scenario, thus we try whenever possible to offer insight on the programs utilization and functionalities based on our personal experience. In Section 4.1 we compare the performance of the tools when it comes to generating stateless traffic, whereas in Section 4.2 we provide a general comparison of various features and aspects with the help of Table 4.1. Finally, in Section 4.3 we give insight on which tool suits best for some common deployment scenarios.

4.1 Stateless Traffic Generation

This Section analyses and compares the capability of the programs to generate stateless traffic. All the packet generators that we have considered in our work are able originate UDP streams, which is a useful feature to stress-test devices or applications. These tools can be used to benchmark the performance of a newly developed application or device when subject to heavy traffic loads, by flooding it with packet streams at high rates.

Figure 4.1 provides a comparison of the performance of the three programs when generating UDP streams with a single CPU core. We measure performance in terms of percentage filling of the 10Gbps back-to-back link on our server. Warp17 and Moongen directly output the link utilization as statistics in their live UI during program execution. T-Rex, on the other hand, only displays the bandwidth without framing (real bandwidth measured based on packets generated by the tool). Therefore, in order to compute the link utilization percentage, we have divided this value by the maximal achievable bandwidth without framing on the 10Gbps link for all packet sizes.

From Figure 4.1 we observe that Moongen has clearly the best performance. In fact it is able to saturate the link with 1 core and packet sizes of 90 bytes or bigger. With 64 byte sized packets it manages to fill 80% of the 10Gbps link. All the other tools can only fill the link up to 20% or less with packets of 64 bytes using 1 core. The two T-Rex modes, stateful and stateless, perform similarly (with stateless mode having a slightly better performance), and they manage to saturate the link with 350-400 byte sized packets. Warp17 has the lowest performance, and can saturate the link with UDP flows only with 800 byte sized or bigger packets.

So far things are more or less as expected, with the two purely stateless tools having the best performance. The overhead of Warp17 and stateful T-Rex can be explained by the fact that stateless traffic generation is not the primary goal of the tool. UDP packets are sent using the exact same

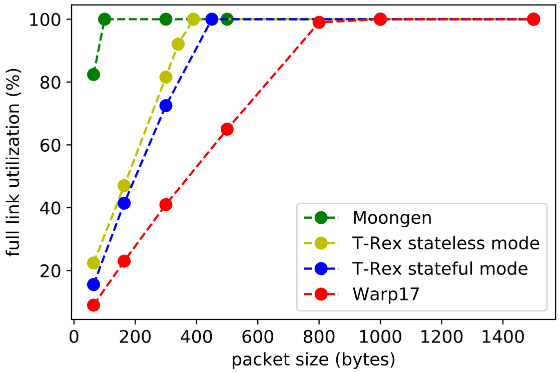


Figure 4.1: The link utilization of the 10Gbps link on our server when traversed by one-directional UDP streams is plotted here for various UDP packet sizes. The measurements are performed for Moongen, Warp17, T-Rex in stateful mode and T-Rex in stateless mode, and for all the tools we have allocated a single CPU core (or hardware thread) to packet processing.

infrastructure with which TCP packets are sent, even with memory hugepages allocated.

The higher efficiency of Moongen with respect to the other tools might be related to its simplicity, as more libraries and dependencies always come with an overhead on performance. Furthermore, stateless T-Rex has many more directly configurable parameters and options for the traffic streams, and this could explain the efficiency loss with respect to Moongen. In fact, to add more options or features in Moongen, we would need to extend the Lua userscript, whereas for our testing we have used a very simple script, which has a low overhead. A somewhat strange aspect is that we would have expected stateless T-Rex to have a better performance, or at least show a more significant improvement compared to stateful T-Rex, since it doesn't need to allocate a state for each flow in memory anymore. Instead, the efficiency of the two T-Rex modes when generating UDP streams is very similar.

Another aspect that could influence the performance of the packet generators is the DPDK version integrated with it. In their paper, Emmeric et al. [1] have found the later DPDK versions to be slightly slower than the old ones due to increased overhead. Moongen uses an older version of DPDK than T-Rex, which uses a more recent one, and this could add up to the reasons why Moongen is more efficient. All the software versions and hardware specifics from our testing setup are summarized in Appendix A for reference. Given these considerations we conclude that Moongen is the more efficient tool to perform stress-testing of devices or applications with stateless traffic.

4.2 General Comparison

Table 4.1 summarizes and compares some of the most important aspects of the three traffic generators that we analysed throughout our work. It aims to serve as a quick reference to compare the tools and help readers in choosing the best suited one for their benchmark use-case. In section 4.3, we will also provide some real-life examples, along with suggestion on which packet generator to deploy for each specific use-case.

Feature	**Moongen**	**Warp17**	**T-Rex**
License	MIT License (open-source, free software).	BSD 3-Clause License (open-source with some clauses).	Apache 2.0 License (open-source, but with limitations).
Maximal BW	120Gbps	Not specified.	200Gbps
Maximal Packet Rate	>20Mpps per core.	>20Mpps per core.	10-30Mpps per core.
Stateful Traffic Generation	No.	Yes.	Yes in stateful and ASTF mode.
TCP stack	No.	Yes.	Only in ASTF mode.
Memory Hugepages	Not required.	Required (automated hugepages-generating script available).	Used, but the tool handles allocation automatically.
DPDK Latency Measurement	Yes.	Yes.	Yes.
DPDK Packet Capture	Yes.	No.	Only in Stateless Mode.
Protocols available	All existing protocols can be defined via Lua scripts. Already implemented: Ethernet, UDP, TCP, ICMP, IPsec, and ARP.	TCP, HTTP, UDP.	TCP and UDP in stateful mode. All existing protocols in stateless and ASTF mode (via Scapy or pcap file).
Traffic Patterns	In theory everything, via Lua scripts. Already implemented: raw Ethernet and UDP streams, TCP SYN flood, Poisson bursts...	TCP flows, HTTP flows, UDP streams. Can also create traffic mix by merging test-cases and tweaking parameters.	Everything, just need to generate a pcap file with the required pattern.
Configurable Options	Any desired option (defined along with protocol in Lua script).	MTU, many TCP stack settings, some IPv4 and VLAN options.	Any option, as packets are built with Scapy or taken from pcap file.
Scalability	Yes, additional threads need to be manually declared in Lua script.	Yes, by passing core-mask with -c parameter	Yes, by passing number of cores with -c parameter
Project Status (June 2020)	Currently not in development, and not maintained.	Currently not in development, but maintained (not very often though).	Currently in active development and constantly maintained.

Table 4.1: Comparison of packet generators (1/2).

Feature	**Moongen**	**Warp17**	**T-Rex**
Pros	• Simple tool, easy to use. • Very efficient in terms of CPU utilization for packet generation. • Extremely flexible thanks to Lua-script based operation.	• TCP full stack with many configurable parameters. • Can be deployed as client-only (standalone DPDK-based TCP client).	• Can replicate any existing traffic stream, via pcap file or Scapy. • Very well documented and constantly maintained. • Powerful Python automation.
Cons	• No automation, thus for most modifications and change of parameters, e.g. scaling up the number of cores, manual modification of Lua scripts is required. (argument parsing to Lua can be implemented, though).	• Documentation sometimes poorly explained. • Extention of yaml file to generate more traffic patterns complicated. • Limited modification of traffic patterns allowed (packet generation process cannot be modified).	• Not so easy to use, many operational modes and long documentation. • Many configuration parameters that can be overwhelming.

Table 4.1: Comparison of packet generators (2/2).

4.3 Common Deployment Scenarios

Traffic generators are used to investigate the performance of network infrastructure, devices or applications. To ensure platform resilience and resistance to heavy load condition and cyber attacks, every newly developed software of hardware appliance should be subjected to rigorous stress tests before deployment [31]. In general all the three tools that we analysed in our work can be used to benchmark middle-box devices, e.g. a router, a switch, or a firewall, that can be connected to two Interfaces on the same server in a back-to-back manner. This setup is illustrated in Figure 4.2. Depending on the type of device that we want to benchmark, and which feature we specifically want to test, a stateful or stateless solution may be selected.

A common use-case, especially relevant for security applications, would be generating traffic which simulates cyber attack behaviour, such as a DDoS (Distribute Denial of System) attack. For this specific scenario, a stateless packet generator is more than enough, as the tool doesn't need to interact with the DUT, but only flood it with DDoS traffic. Moongen provides examples to generate UDP and TCP-SYN floods, which makes it a nice option for this use-case, also given the fact that it is simple and easy to use. However, if we would like to generate a wider variety of DDoS attacks based on other protocols, such as ICMP or DNS floods, Moongen is not a good solution, as it would require these protocols to be implemented from scratch in the Lua scripts. Therefore, in the latter case we suggest using T-Rex (either in stateful or stateless mode), as it comes with many pcap files with multiple protocols, which can be amplified to generate the attack traffic.

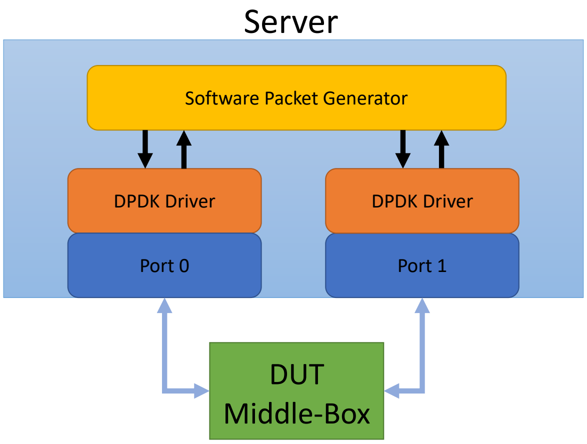


Figure 4.2: Middle-Box testing setup.

Another common use scenario would be stress-testing a device or software application that alters the connection, such as a proxy, reverse-proxy, firewall or DPI engine. To this purpose, we would require the tool to generate "clean" traffic directed through the DUT. Furthermore, the packet generator should be able to maintain the established connections and adapt the traffic depending on how the device modifies it (e.g. if a firewall blocks some packets, it should retransmit them). For this scenario, we need a traffic generator provided with a full TCP stack, such as Warp17 or T-Rex in ASTF mode. To be more specific, Warp17 could be deployed to generate legit traffic at high rates, from a specifically defined IP address range that a firewall should not filter. This would allow us to measure the performance of the firewall when subject to heavy load condition.

Another example where a packet generator could be utilized is for stress-testing an application's TCP implementation. Again, we would require a traffic generator provided with a TCP stack, which we would run as a client-only instance. The tool would then direct the traffic towards the application, which can be running on a port on the same server, or also on another physical server.

One final scenario where a traffic generator can be utilized, is for measuring the latency of protocols through a device or application. Latency is a very important performance characteristic, as it has a great impact on user performance. All the three tools that we have considered in our work implement a hardware-based approach for measuring latency via DPDK. This approach is thoroughly explained for Moongen in Section 3.1.4. In our work, however, we haven't tested this feature on Warp17 and T-Rex, thus we cannot provide a comparison with respect to precision and reliability of the measurement. This is illustrated as a possible future development of our work in Chapter 5.

Chapter 5

Summary and Outlook

In our work, we have first thoroughly analysed then compared three high-speed DPDK-based software traffic generators. We found these tools to be very efficient in terms of packet processing, mainly thanks to the DPDK libraries and drivers on which they are built upon. Moreover, a simple and stateless tool such as Moongen achieves the best performance in terms of maximal throughput, as it manages to saturate a 10Gbps glass fiber link with 60 byte sized Ethernet packets. This shows that a traffic generator's performance is also greatly affected by the amount of packet processing operations it needs to perform, options to configure and libraries to access. We also provided the reader with an extensive table that compares some of the main features and aspects of the three tools, and aims to serve as a guideline for choosing the ideal traffic generator to deploy. In the next Section, we lay out some possible ideas for continuation of our work.

5.1 Future Work

A first possibility would be to scale up the testing to a server with 100Gbps Interfaces. All the tools should in theory be able to fill up 100Gbps links without any major problems, since the packet processing capability of DPDK scales linearly with the number of allocated cores. We have already prepared a cheat-sheet, available on our Gitlab repository[1], with some ready-to-run commands to perform the link saturation testing with UDP packets on a 100Gbps link. This could serve as a starting point, then one could extend the testing to stateful traffic generation and latency measurements.

Another idea for future work, could be to conduct a deeper analysis of Warp17's TCP stack performance at high speeds. In fact, in our work, we have performed an in-depth comparison all the tools when generating stateless traffic, but we have only compared Warp17's TCP flows generation capability against stateful T-Rex. As we mentioned in Chapter 3, this is not a fair comparison, since T-Rex in stateful mode doesn't have a TCP stack implemented, thus has much less packet processing overhead. For this reason, it could be interesting to also try out T-Rex in advanced-stateful mode, where a TCP stack is implemented on top of DPDK. This would allow for a fairer comparison of Warp's TCP stack implementation.

A final proposition for continuation of our work could be to confront the latency measurement feature of the traffic generators. In Chapter 4, we mention that all three tools implement a hardware-based approach for measuring latency via DPDK. It would be interesting to see if there

[1] https://gitlab.ethz.ch/nsg/student-projects/sa-2020-06_traffic_generation/-/blob/master/code/100gbps_cheatsheet.txt

are any differences in precision, accuracy and reliability of the latency measurement, even though it is in theory implemented using the same procedure.

Bibliography

[1] Emmerich, P., Gallenmüller, S., Antichi, G., Moore, A. W., and Carle, G. Mind the gap - a comparison of software packet generators. In *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (2017).

[2] Emmerich, P., Wohlfart, F., Raumer, D., and Carle, G. Moongen: A scriptable high-speed packet generator. *CoRR abs/1410.3322* (2014).

[3] Soumya Mahalakshmi A, Amulya B S, and Moharir, M. A study of tools to develop a traffic generator for l4 – l7 layers. In *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)* (2016).

[4] Srivastava, S., Anmulwar, S., Sapkal, A. M., Batra, T., Gupta, A. K., and Kumar, V. Comparative study of various traffic generator tools. In *2014 Recent Advances in Engineering and Computational Sciences (RAECS)* (2014).

[5] Tockhorn, A., Danielis, P., and Timmermann, D. A configurable FPGA-based traffic generator for high-performance tests of packet processing systems.

[6] Busy-waiting Wikipedia page. `https://en.wikipedia.org/wiki/Busy_waiting`. [Accessed May-2020].

[7] Best network traffic generator and simulator stress test tools. `https://www.dnsstuff.com/network-traffic-generator-software`. [Accessed May-2020].

[8] DPDK official website. `https://www.dpdk.org/about`. [Accessed May-2020].

[9] Introduction to DPDK: Architecture and principles. `https://blog.selectel.com/introduction-dpdk-architecture-principles/`. [Accessed May-2020].

[10] Routing freak: Intel-DPDK. `https://routingfreak.wordpress.com/tag/intel-dpdk/`. [Accessed June-2020].

[11] DPDK Wikipedia page. `https://en.wikipedia.org/wiki/Data_Plane_Development_Kit`. [Accessed May-2020].

[12] Ethernet Alliance's 2020 Technology Roadmap. `https://ethernetalliance.org/technology/2020-roadmap/`. [Accessed May-2020].

[13] Frame Wikipedia page. `https://en.wikipedia.org/wiki/Frame_(networking)`. [Accessed June-2020].

[14] Lua Frame synchronization Wikipedia page. `https://en.wikipedia.org/wiki/Frame_synchronization`. [Accessed June-2020].

[15] Debian Website on Hugepages. https://wiki.debian.org/Hugepages. [Accessed June-2020].

[16] Oracle Docs about Hugepages. https://docs.oracle.com/database/121/UNXAR/appi_vlm.htm#UNXAR391. [Accessed June-2020].

[17] Intel 82599 10 Gigabit Ethernet Controller Datasheet. https://cdrdv2.intel.com/v1/dl/getContent/331520. [Accessed June-2020].

[18] Lua Wikipedia page. https://en.wikipedia.org/wiki/Lua_(programming_language). [Accessed June-2020].

[19] Moongen's github page. https://github.com/emmericp/MoonGen. [Accessed May-2020].

[20] Ostinato Packet Generator. https://ostinato.org/. [Accessed June-2020].

[21] Packetsender github page. https://github.com/dannagle/PacketSender. [Accessed May-2020].

[22] PF_RING ZC official website. https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/. [Accessed May-2020].

[23] Ostinato Pktgen documentation. https://pktgen-dpdk.readthedocs.io/en/latest/. [Accessed June-2020].

[24] Polling (Computer science) Wikipedia page. https://en.wikipedia.org/wiki/Polling_(computer_science). [Accessed May-2020].

[25] RFC 1242. https://tools.ietf.org/html/rfc1242. [Accessed May-2020].

[26] Terabit Ethernet Wikipedia page. https://en.wikipedia.org/wiki/Terabit_Ethernet. [Accessed May-2020].

[27] Traf-gen Linux manual page. http://man7.org/linux/man-pages/man8/trafgen.8.html. [Accessed May-2020].

[28] T-Rex's official webpage by Cisco. https://trex-tgn.cisco.com/. [Accessed May-2020].

[29] T-rex Advanced Stateful mode Documentation. https://trex-tgn.cisco.com/trex/doc/trex_astf.html. [Accessed June-2020].

[30] T-Rex's official documentation. https://trex-tgn.cisco.com/trex/doc/trex_manual.html. [Accessed June-2020].

[31] Imperva T-Rex vs Avalance Guide. https://www.imperva.com/blog/trex-traffic-generator-software/. [Accessed June-2020].

[32] T-rex Stateful mode Presentation Slides. https://trex-tgn.cisco.com/trex/doc/trex_preso.html#slide-11. [Accessed June-2020].

[33] T-rex Stateless mode Documentation. https://trex-tgn.cisco.com/trex/doc/trex_stateless.html. [Accessed June-2020].

[34] Advanced Stateful T-Rex mode versus NGINX. https://trex-tgn.cisco.com/trex/doc/trex_astf_vs_nginx.html. [Accessed June-2020].

[35] WARP17's github page. https://github.com/Juniper/warp17. [Accessed May-2020].

Appendix A

Hardware and Software Specifics

The configuration of the server on which all the testing was performed:

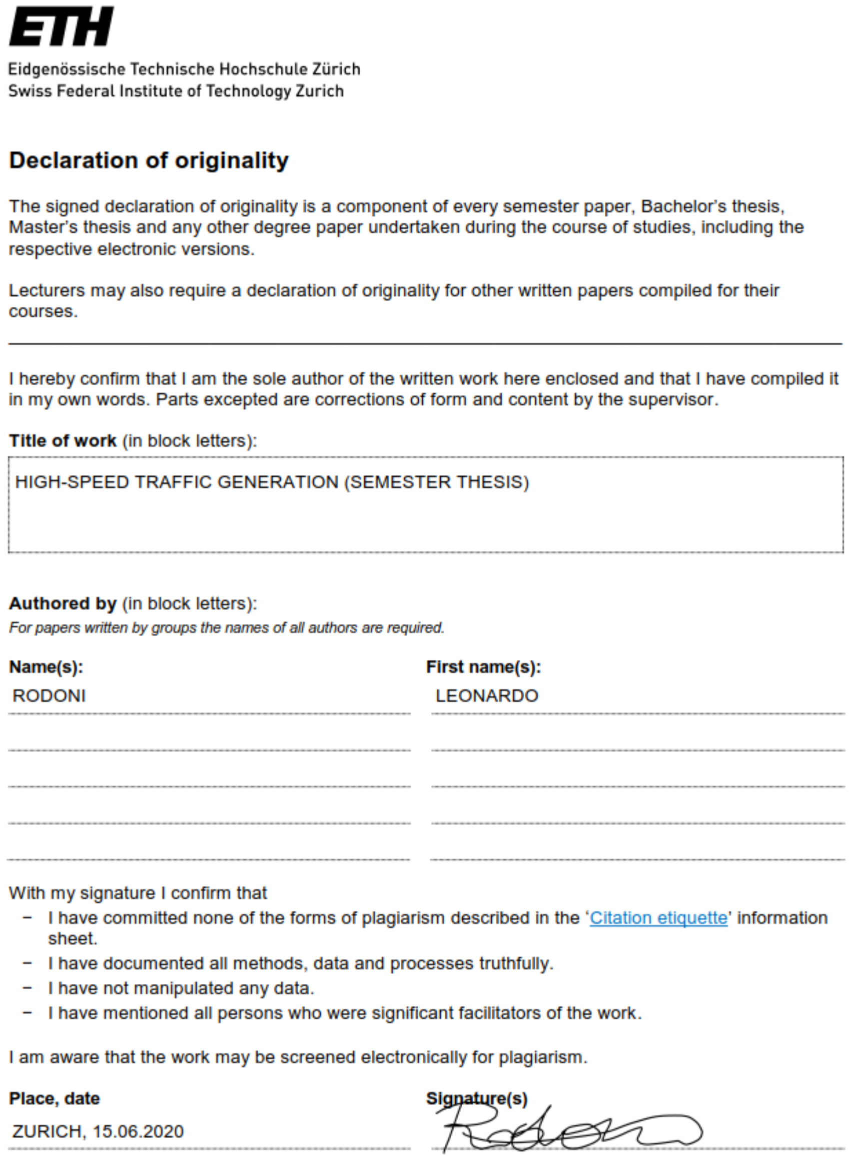
- CPU Intel Xeon E5620 @ 2.40GHz
- 36GB of RAM
- 2 X 10GBps Intel 82599ES SFI/SFP+ Ethernet Interfaces
- Ubuntu 16.04.6 LTS

The software and DPDK versions that were deployed:

- Moongen: latest master branch with DPDK 17.08
- Warp17: dev/common branch with DPDK 17.11.6
- T-Rex: version v2.80 with DPDK 20.02.0

Appendix B

Declaration of Originality



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

HIGH-SPEED TRAFFIC GENERATION (SEMESTER THESIS)

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
RODONI	LEONARDO

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZURICH, 15.06.2020	