Universiteit van Amsterdam
System and Network Engineering

Research Project 2

Session based high bandwidth throughput testing

Bram ter Borch
bram.terborch@os3.nl

29 August 2017

Abstract

To maximize and ensure service availability, system and network administrators need to know what the limits of the infrastructure supporting a service are. That infrastructure often has to support multiple different services for users and systems, the combined application demand placed on the infrastructure can result in parts of that infrastructure reaching its (practical) limit. Generating packets in order to test the full capacity of a link is not hard to accomplish. Especially for UDP traffic since it does not provide session synchronization nor congestion control and does not offer retransmission of data. TCP, due to its design and used techniques, prevent the capacity of a network link or path from being overloaded.

This research however focuses on the testing of limits pertaining to the packets per second rate and the number of concurrent sessions. This research investigates the suitability of various load testing tools - with and without the use of the Data Plane Development Kit - and apply these tools to various end-to-end network topologies and hardware systems that connect a client and a destination. The result of this research is a set of proposed tests, making use of the most suitable tools, to test an infrastructure up to application level. By using these proposed tests the report demonstrates that the weakest links in the path can be identified and its load tolerance limits can be found.

Table of Contents

1 Introduction **2**
1.1 Terminology and abbreviations 3

2 Scope **5**
2.1 The reliability of TCP 5
2.2 Assumptions and constraints 6
2.3 Research question 8

3 Related work and available tooling **9**
3.1 Related work 9
3.2 Tools 10

4 Experiments **12**
4.1 Standards and best practices 14
4.2 Kernel based tools 14
4.3 Tools using the Data Plane Development Kit 15

5 Methodology **20**
5.1 Tests 20
5.2 Real world scenario 22

6 Results **25**
6.1 Infrastructure 25
6.2 Monitoring 26
6.3 Data Plane Development Kit based tools 27

7 Conclusion **33**
7.1 Suitability of the Data Plane Development Kit 34
7.2 Future Work 34

Appendix A **39**

Appendix B **40**
B.1 Software 40
B.2 benchmark test 41
B.3 scripts used during the tests 51

Chapter 1

Introduction

This paper addresses and describes the path towards obtaining knowledge on the limitations of hardware in the path towards a service, whilst accounting the full path from client to application which operate in Open Systems Interconnect (OSI) layer 7. The research considers and explains the importance of the end-to-end user experience.

Before an IT department offers new systems or applications to the users, system and network administrators need to know the limitations of the hardware in the path towards an application in order to set thresholds for monitoring alerts. These limitations can be caused by any of the devices in the path towards an application. Routers and switches are capable of forwarding packets at line rate, making sure the server running an application receives all the data destined for the application without unnecessary delays. The OS and the running application should be able to handle the inbound traffic from the Network Interface Card (NIC). To provide more bandwidth to an application, for example link aggregation can used to bundle multiple physical links to one logical link.

Kernel based open source tooling capable of generating data to saturate links well beyond 100Gb/s using UDP traffic, are available. Tools like iPerf[1], hping[3] and BoNeSi[4] are kernel based and can be used to perform session based throughput tests up to OSI layer 3. Utilizing TCP in order to guarantee data delivery can result in capacity problems for both network infrastructures and applications.

Figure 1.1 provides an example infrastructure and a representation of testing at different OSI layers. Testing at each and every layer can reveal different limitations in the infrastructure.

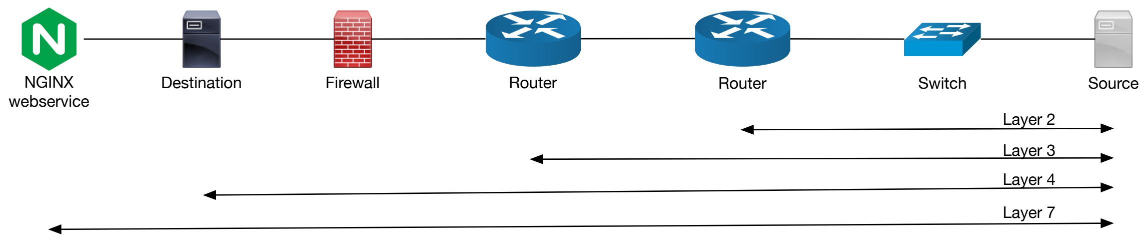


Figure 1.1: Representation of the mentioned OSI layers in a network diagram.

A path towards an application could contain stateful devices like a firewall or a load balancer. Stateful devices keep track of sessions, which evidently costs resources. These stateful devices in a path towards an application can become a bottleneck when their resources run out and can cause unavailability of services. Alerting before resources run out is needed to keep a network and the services available. Therefore, end-to-end performance tests are needed from a client to a service in order to find the limitations in the infrastructure providing the service.

Institutes like Nikhef (a tier 1 location for the large Hadron Collider (LHC) Computing Grid) need to transfer large datasets from CERN. CERN produces around 30 petabytes annually[10], which makes high-capacity links critical for Nikhef's research purposes. The Nikhef network is designed around a high-capacity core that contains only stateless switching and routing devices. Distributed storage systems are directly connected thereto (akin to Data Transfer Nodes (DTN) in a ScienceDMZ[23]). Nikhef is constantly upgrading infrastructure devices to meet the increasing demands of bandwidth. Besides capacity, data integrity is important, and therefore preferably TCP should be used for data streams between Nikhef and CERN. Engineers at Nikhef have a need to find the limitations of the infrastructure up to OSI layer 7, before they put the hardware into production.

Testing the infrastructure using dedicated commercial equipment for every new service is costly. Budget constraints do not allow for the procurement of special commercial hardware or (expensive) tooling. Currently the engineers of Nikhef rely on the hardware specification from the vendors.

The use of high capacity links and a need for session based application layer testing requires a different approach for traffic generation over network paths.

The Data Plane Development Kit[5] (DPDK) introduced by Intel offers a different approach for traffic generation, it does so without using the kernel network stack. Through DPDK, Linux userland applications are able to bypass the kernel and communicate with the network hardware directly. Memory, processors and interfaces have to be dedicated to DPDK. Applications are then built on top of DPDK, utilizing DPDK's functionality to bypass the kernel (and the copying of memory regions inherent in such use). MoonGen[8], pktgen[6], and WARP[9] are designed based on different ideas offering the ability to test up to layer 7 of the OSI model.

1.1 Terminology and abbreviations

The terminology used in this paper is based on RFC1242[28], with the most relevant terms listed in table 1.1 for convenience. A list of acronyms used in this paper can be found in appendix A.

Term	**Explanation**
Constant Load	Fixed length frames at a fixed interval time.
Data link frame size	The number of octets in the frame from the first octet following the preamble to the end of the Frame check Sequence (FCS), if present, or to the last octet of the data if there is no FCS.
Inter Frame Gap	The delay from the end of a data link frame, to the start of the preamble of the next data link frame.
Overload behavior	When demand exceeds available system resources.
Throughput	The maximum rate at which none of the offered frames, are dropped by the device.

Table 1.1: Useful terminology

Chapter 2

Scope

The availability of high capacity links going up to 100Gb/s require infrastructure providers to maintain high capacity core networks in order to provide the increasing need for bandwidth. An institute like Nikhef transfers a large volume of data and requires a reliable data transfer between source and destination.

In order to guarantee system uptime and service availability, the limitations of the hardware and services need to be known. In order to find the limitations, testing up to the application layer (layer 7) using open source software is preferred. The engineers at Nikhef never had the ability to perform high bandwidth applications tests at OSI layer 7.

This chapter presents the scope for the problem stated in chapter 1. The scope is applicable for corporate networks in order to set the requirements for a set of tests during the experimental phase in chapter 4. The scope and the problem lead to the research question presented at the end of this chapter.

2.1 The reliability of TCP

When sending data over UDP, data gets generated and transported to the destination without any form of transport reliability. If the destination UDP port is open the data will be forwarded to the application listening on that specific port. When the destination does not listen on that specific UDP port the data will be dropped. The protocol does not provide feedback for any of the actions taken at the receivers side (except for ICMP error messages).

TCP, in contrast to UDP, guarantees the delivery of the data as long as the session is established between the end nodes. The session needs to be created before data can be exchanged. A three way handshake between source and destination is performed in order to synchronize TCP settings. Guaranteed delivery is realized by acknowledging received data and resending unacknowledged data. Other techniques like flow control, congestion control and fast retransmission of packets ensure that data is delivered in time and offered to the higher layer protocol in the correct order. These techniques all require resource reservation at the client and the server, and at stateful network devices along the path between client and server.

The techniques implemented in TCP require buffers to store data until the data is acknowledged by the receiving end. The buffer sizes are reserved by the Operating System (OS) per session and are negotiated during the three way handshake. When more new sessions are created than the number of sessions that get closed due to a time-out or by terminating the session, eventually server resources will be depleted.

When it comes to layer 7 protocols like HTTP, even more resources need to be reserved. HTTP sessions, HTTP state and application state need to be saved. HTTP requests need to be processed, a response needs to be generated and sent over the session to the user. Normally a web server will cache files that are requested for a specific time using up memory. Hosting a dynamic web page requires the web server to generate the page on request which makes the CPU utilization higher than just hosting static web content.

2.2 Assumptions and constraints

In order to execute some experiments and interpret their results, specific technical constraints in the real world operating environment and several key implementation parameters are important for this research. The technical constraints are not specific for Nikhef. The research is looking at a set of constraints that is applicable in corporate networks in order to specify requirements for this research.

Overloading
All the built in techniques used by TCP ensure an IP path will not be overloaded. In order to find the weakest link in the path from a client to a service, data should be sent using the expected maximum capacity of the devices in the path. When client and server are connected with 40Gb/s interfaces one should sent data using the maximum capacity of the links. This could result in overloading a device in the path, which immediately shows that the intermediate device is the weakest link.

When data can be sent at the link's full capacity, there might be other hardware limitation (maximum amount of packets per second or a maximum amount of session per second) in the path to the destination. For network and systems administrators it is important to know these limitation for monitoring purposes.

Throughput
The increasing need for bandwidth requires high capacity links between clients and destination in order to minimize transfer times. A link's throughput is the most significant characteristic for generating load. This research considers links with a capacity of 40Gb/s and up, as high bandwidth links. Therefor considered tools need to be able generate at least 40Gb/s of throughput, this is the first requirement.

Frame size
Ethernet is used during this research, therefore all references to frame sizes are based on Ethernet standards[12] with IP and TCP headers included. Due to collision detection the minimum payload inside an Ethernet frame is 46 bytes. The Ethernet frame and the payload combined have a minimum size of 64 bytes, this does not include a 4 byte VLAN tag. A packet is always preceded by an 8 byte preamble. A 12 byte Inter Frame Gap

between frames is used to separate the frames. This makes a total of at least 88 bytes of data from the beginning of a packet to the beginning of a new packet. A representation of an Ethernet frame that is used in the report is given in figure 2.1.

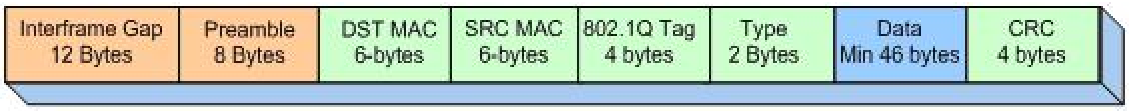


Figure 2.1: Representation of an Ethernet frame.

Packet size
According to Murray et all.[36] in 2012, 99% of the traffic inside the corporate networks used during their research had an MTU size of maximum 1500 bytes. When using Jumbo frames[26] (best practice is an MTU of 9000 bytes towards clients[17]) the amount of overhead is less because more data can fit in one packet. More data inside a packet could result in less packets. Jumbo packets are helpful when large amounts of data need to be transfered between 2 nodes. Jumbo packets are not used during the project because according to Murray et all. less than one percent of transferred data has an MTU larger than 1500 bytes. Exceptions can be made during a test to see if hardware limits can be reached.

Packets per second
When a link has a capacity of 40Gb/s and packets have a minimum size of 88bytes (which includes the inter frame gap, the preamble, the data link frame and a VLAN tag) a maximum of 56.8 million packets per second (Mpps) can be transferred over the link in one direction. When using a 100Gb/s line the theoretical maximum is 142 Mpps. The research is focusing on 40Gb/s links and therefore the second requirement is that a tool needs to be able to generate 56.8 million packets per second in order to be considered for the proposed tests.

Sessions
A TCP session is a unique tuple of source IP, destination IP, source port and destination port. An established session may be used to send more than one packet and can transfer data bidirectional. The amount of sessions per second is a determining factor for the availability of services behind stateful devices. A stateful firewall, for example, needs to keep track of the states of the sessions from source to destination. When new sessions to a server are opened, the firewall has to process them according to the rule base. When a session is approved, most firewall vendors move it to fast-path processing. This is a table with accepted sessions, allowing traffic in the same session to be handled in hardware. This means that only the first packet of a new session is handled in the slow-path and the limitations of a firewall can be found in the amount of new sessions per second. The amount of sessions the tool is capable of generating is the third requirement this research focuses on during the experimental phase.

Application specific traffic
According to Sandvine[21], a global communications solutions service provider that published bi-annual traffic baseline reports, around 70% of the traffic on the Internet is streaming audio and video. Dynamic Adaptive Streaming over HTTP (DASH)[37] is used to stream video and audio over HTTP. Netflix uses DASH to deliver content to the users. Next to DASH, streaming video services like YouTube are also accessible over HTTP. Although the traffic inside the Nikhef network is not HTTP based, HTTP is the protocol that is the most used in the Internet. This makes HTTP a suited protocol to use for layer 7 tests during the research.

End-to-end testing
Performing tests by generating traffic between two connected links at OSI layer 2 or 3 does not test the limitations of an application. The application and the kernel should be tested using OSI layers above layer 4. When a server running an application is connected with a 40Gb/s link, it does not mean that the application can process 40Gb/s. End-to-end testing is a term used in this report to refer to tests that are application based. A client sending application layer protocol requests and the server replying with application layer protocol responses.

2.3 Research question

The problem statement and the specifications lead to the following research question.

What are the requirements to perform high bandwidth session based throughput testing and how can this be applied to end-to-end application level testing?

Chapter 3

Related work and available tooling

As described in chapter 2, the goal of this research is the study for the usability of traffic generation tools in order to find the limits of an end-to-end path between a client and a destination, possibly going up to the application layer. Techniques used by TCP to prevent links being overloaded is a well researched topic. When looking into limitations of an infrastructure using session based protocols, there is a limited amount of related papers. In the field of end-to-end performance testing focusing on a minimum bandwidth of 40Gb/s this research provides a beginning. Therefore this chapter will focus on different tools that can be used for bandwidth generation up to layer 7 of the OSI model.

3.1 Related work

The amount of related work for end-to-end testing is limited when using session based protocols and trying to generate throughput over 40Gb/s. There is a prior research performed by Emmerich et al. for generating throughput over 40Gb/s using DPDK and a research is performed looking into application based throughput testing using DPDK by Malakshmi et al.

Emmerich et al.[32] published a paper about MoonGen in 2015, MoonGen is capable of generating 120Gb/s and 178.5 Mpps (over multiple 10Gb ethernet interfaces using twelve 2 GHz CPU cores) according to the developers. Exceeding the throughput criteria and being session-based, it thus meets the requirements for this study. Therefore, MoonGen will be used for OSI layer 2, 3 and 4 testing.

In 2016 research was performed by Malakshmi et al.[35] on different DPDK applications with the purpose of creating a tool for layer 4 to layer 7 application testing. The result of Malakshmi's research is a tool called T-REX. Their project goal is to generate stateful traffic up to 10Gb/s. However, the main T-REX functionality is Cisco-proprietary and requires a Cisco device to run. The public (free) version is limited in functionality to an extent that it is not applicable to this research.

3.2 Tools

A lot of tools are available for bandwidth testing. The tools in table 3.1 each satisfy one or more of the requirements mentioned in chapter 2. Specifically, we consider these tools in order to assess the suitability of these tools for session based bandwidth testing at high volumes.

iPerf3, hping and BoNeSi depend on kernel drivers and the kernel TCP stack. pktgen, MoonGen and WARP use drivers and the TCP stack provided by DPDK.

Tool	**Session based**	**TCP stack and drivers**
iPerf3[1]	Yes	Kernel
hping[3]	Yes	Kernel
BoNeSi[4]	Yes	Kernel
pktgen DPDK[6]	Yes	DPDK
MoonGen[8]	Yes	DPDK
WARP[9]	Yes	DPDK

Table 3.1: Packet generation tools

3.2.1 iPerf3

IPerf3 is a client-server based tool that allows packet generation. It needs a client and server to generate traffic and it needs tweaking of kernel parameter to generate traffic over 40Gb/s. Efforts have been made to make it available for DPDK[33]. Unfortunately these efforts did not have the success the author was hoping for. A small test is performed to see if the kernel based version of iPerf3 can be used to generate high bandwidth session based traffic streams. The test and the results can be found in chapter 4.

3.2.2 hping

Hping was started in 2006. It is a command-line oriented TCP/IP packet assembler. Hping is capable of sending crafted packets to a destination using spoofed IP addresses if necessary. ICMP, UDP, TCP and raw IP modes are supported. Random source addresses can be used to sent requests to simulate a DDoS attack. Tests will be performed to see if hping is able to generate the bandwidth or packet and session per second needed for this research.

3.2.3 BoNeSi

BoNeSi is 'the DDoS botnet simulator' according to its developers. BoNeSi supports ICMP, UDP and TCP (HTTP) flooding attacks from a defined botnet size. Source addresses can be specified in a text file which is used as input. This makes it possible to send crafted packets masquerading the original source of attack.

3.2.4 Data Plane Development Kit

The Data Plane Development Kit[5] (DPDK) was developed by Intel to provide the ability to generate traffic from user space, bypassing the kernel and directly talking to the network hardware. In order to make DPDK run, supported NICs[11] needs to be used. Applications can be created to run on top of DPDK. Pktgen, MoonGen and WARP are three applications that are written on top of DPDK and should thus be able to generate traffic in high volumes.

Pktgen

Pktgen for DPDK is available since May 2013. The developers from DPDK provide Pktgen from the DPDK download page. This makes it a good option for a reference experiment to assess the difference of DPDK compared to kernel-based tests.

MoonGen

MoonGen was initially released in October 2014. It is designed to generate packets at high speed using a minimum amount of resources from the source. According to the developers it is more efficient than Pktgen[32]. A 10Gb/s link can be filled using only one core. MoonGen builds on libmoon[7] by extending it with features for packet generators such as software rate control and software timestamping.

WARP

Juniper WARP was released in May 2016. It allows users to execute performance testing up to layer 7, where however currently only HTTP version 1.1 as a layer 7 protocol is supported and only IPv4 is supported at layer 3. A server equipped with two Intel Xeon E5-2660 v3 processor, 128Gb RAM and two 40 Gb Ethernet interfaces, is suggested to be able to generate around 2 million sessions per second between client and server. The performance graphs included in the packet source provide the idea that WARP can generate high throughput and a high amount of sessions per second up to the application layer.

Chapter 4

Experiments

In order to find the limits and the usability of the tools in table 3.1 for high bandwidth session based throughput testing, the tools are subject to various experiments. The results the research is looking for:

- The capability of generating 40Gb/s of throughput.
- The maximum amount of packets per second.
- The maximum amount of sessions per second.

The kernel based tools are tested on top of two different kernels. FreeBSD 11.0 and Ubuntu 16.04 are used to see if the kernel has any influence on one of the three items stated above. The reason for using two kernels is that a preliminary test showed major differences in generating UDP packets per second, these differences could also apply for TCP packet generation. The DPDK tools are tested on top of the Ubuntu kernel since FreeBSD does not support DPDK. All experiments described in this chapter are executed in a test environment at Nikhef. The visualization of the test environment is shown in figure 4.1.

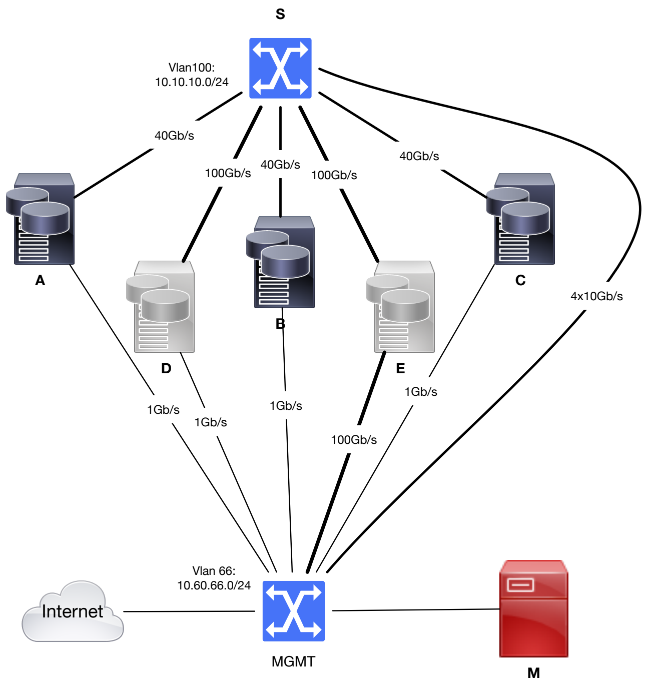


Figure 4.1: Environment used for experiments at Nikhef.

The figure shows five server systems, of which three identical servers (A, B and C in table 4.1) are used to perform the tests, which results are described in this paper. During the experiments 2 extra machines (D and E in table 4.1), both containing 100Gb/s Mellanox NIC cards, have been introduced into the network to test links with a capacity up to 100Gb/s to verify if reached limits are resource or hardware based. All servers were connected to a Juniper QFX10002 (device S in Fig. 4.1), which provides 32 40Gb/s QSFP ports, of which some are configured as a single 100Gb/s interface[22].

Machine	**A & B & C**	**D**	**E**
Cores / Threads	2 / 8	14 / 56	128 Threads
CPU	Intel(R) Xeon(R) CPU E3-1230 v5 @ 3.40GHz	Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz	POWER8E
Memory	4x 16GB @ 2133 Mhz	24x 8GB @ 1067Mhz	2x 64GB
NIC	Intel XL710 40Gb/s	Mellanox ConnectX-4 100Gb/s	Mellanox ConnectX-4 100Gb/s

Table 4.1: Specifications from the machines used for experiments

Device A is always acting as the destination for traffic unless stated otherwise. Depending on the tests the source can be machine B, C or B and C together. This depends on the capability of the tools tested at that moment. For testing beyond 40Gb/s, one of the 100Gb/s machines (D or E) can be used. Machine M is a Simple Network Management Protocol (SNMP) collector. This SNMP collector queries the test servers every 10 seconds for status. Packets per second and bits per second are retrieved from the devices under test. Switch S is also added to the collector as a data source. SNMP is active on the management interface of the devices. The high bandwidth interfaces are connected to switch S in order to perform throughput tests. The SNMP collector polls the servers through their management interfaces to make sure the SNMP data is not a part of the collected measurements during the executed tests.

4.1 Standards and best practices

Multiple RFCs have been written that provide guidelines for throughput testing. This research refers to RFCs that contain the information needed for the research. Some terms from the mentioned RFCs are updated in a new RFC, but these new RFCs do not contain the important information used by this research. The terminology from RFC 1242[28] will be used throughout this paper and techniques from RFC 2544[29] have been taken into account during this research. However, the aforementioned practices and benchmarks of RFC 2544 and 6349[31] are centered on traffic generation that overloads network devices, which is detrimental to user network experience in production (shared) networks, as was discussed in RFC 6815[30].

A quick list with guidelines from these RFCs is as follows:

- Throughput tests should have a minimum runtime of 60 seconds
- The environment cannot be a production environment
- Devices Under Test (DUT) will possibly be overloaded
- Tests should be done 3 times and the average should be taken from the results.

4.2 Kernel based tools

All the kernel based tools from table 3.1 are tested running on top of Ubuntu and FreeBSD. All the tools were used to find its limits for bandwidth generation, maximum amount of packets per second (pps) and maximum amount of sessions per second.

iPerf3

Machine A is set up as a server running on default TCP port 5201, machine B is setup as a client connecting to the server. Tests are executed from the client side on a FreeBSD and Ubuntu server and the performance differences were minimal. The initial tests were performed sending 46 byte of data, which concluded in a minimal Ethernet frame of 88 bytes as described in chapter 2.2 . The practical maximum amount of 42Mpps was not reached until 6 threads were utilized to generate traffic (the practical maximum amount of packets is explained in section 4.3.1). The link capacity was filled using 16 threads and a 1500 bytes of data in an Ethernet frame. iPerf3 starts a TCP session per running thread. This makes it less useful for this project since we explained already that maximizing bandwidth utilization is not enough to reach the performance limitations.

Hping
Hping is used to sent spoofed traffic to a server. It does not need a server side application running on server A. Therefore only the client side was tested on top of FreeBSD and Ubuntu. Both kernels were capable of sending a maximum of 13Gb/s of SYN packets towards a server padded up to 9000 bytes. Although it has features to craft packets and probably has the capacity to overload certain small environments, the maximum bandwidth is far under the limits this research is looking for.

BoNeSi
The maximum output that was produced with BoNeSi was 300Mb/s and 500Kpps on an Ubuntu kernel. The FreeBSD kernel was capable of generating the same values. The reached limits are not sufficient for this project and therefore BoNeSi will not be used for further experiments.

4.3 Tools using the Data Plane Development Kit

The Data Plane Development Kit (DPDK) enables fast packet generation and transportation inside a system. Tools are available to generate raw IP packets. A recent tool is introduced that offers HTTP v1.1 packet generation. The tools tested during these experiments are: pktgen, MoonGen and WARP.

4.3.1 pktgen

For this experiment server B is the source which is generating the traffic. Servers A and B are identical as seen in table 4.1. Pktgen is not a client-server based application. While server A is idle, B will generate traffic in the form of TCP messages padded to the desired size where only the ACK flag is set. This means that pktgen is sending TCP packets but it is not setting up sessions to the destination. To find the hardware limitations one can send small packets of 64 bytes at a high rate which should show the hardwares capabilities of generating packets or one should generate larger packets of 1500 bytes (not considering jumbo frames) to fill up link capacity.

When sending small packets to the destination the maximum amount of packets per second peaked at 42Mpps where the expected amount of packets is 56Mpps unidirectional. Additional configuration was necessary in order to reach the expected amount of packets. The DPDK website offers a guide[15] to setup the system in order to get the maximum performance out of the Intel XL710 40Gb/s card. Flashing a new firmware version into the card was the first step. After following the DPDK guide the result remained the same and according to conclusions drawn out of a report from Chelsio[34] the PCI express bus (V3.0, 8.0GT/s, 8 lane) is capable of transferring 70Mpps bidirectional. Unidirectional it can reach a maximum of 42Mpps. Both the Chelsio[24] and the Intel[25] card use the PCIe v3.0 (8.0 GT/s) 8 lane interface to connect to the main board. The limitation caused by the PCI express bus gives the research a new practical maximum amount of 42 million pps that can be transferred by a host in the experimentation environment.

Ramping up the packet size, starting at 64 bytes and adding 16 bytes per test round up to a maximum of 1024 bytes resulted in finding the optimal packet size of 400 bytes. This packet size is the optimal packet size since it generates a bandwidth usage of 39.8Gb/s and a total of 11Mpps.

4.3.2 WARP

In order to get to know the capabilities of DPDK in combination with WARP on top of the hardware in the experimentation environment a benchmark was run on server B. A machine running WARP as a client needs a service to respond to SYN packets, otherwise sessions are not opened and there will not be any traffic flowing between client and server. Appendix B.2 displays the benchmark script used for this test. The benchmark script was provides in the application source. For this benchmark the server B will be acting as the client and as the destination for the traffic, both client and destination are using a dedicated NIC. An extra Intel XL710 card is added to machine B since the XL710 card (which is a NIC providing two ports) can not utilize both ports in one slot at their full capacity[16]. The second port on the card is designed as a fail-over interface. Port A from card one and two are connected to switch S. Traffic is generated from port A of NIC 1 to port A of NIC 2. All of the machine's CPUs and memory will be dedicated to this test. The benchmark first does the tests for raw TCP, where after it will perform the tests using HTTP.

The client side of the benchmark attempted to open 4 million session although not all of the sessions succeed. The amount of sessions per second, packets per second and the used bandwidth is registered along with the time spent until all sessions are tried. When a TCP session is opened between client and server, the tests will request a file or send a raw TCP packet. The raw TCP reply is a TCP packet, for the HTTP file request a padded 200-OK message is returned. The goal for this test is to find the capabilities of the server while it is running WARP therefore the guidelines for bandwidth testing regarding runtime from RFC2544 are not applicable.

The first test is raw TCP. As seen in figure 4.2 the client attempts to open about one million sessions per second. When the packet size goes up the amount of sessions per second drops slightly. The succeeded sessions use up some of the link's total capacity as is displayed in figure 4.3. This means that the sessions that succeed are capable of filling 50 % of the link capacity.

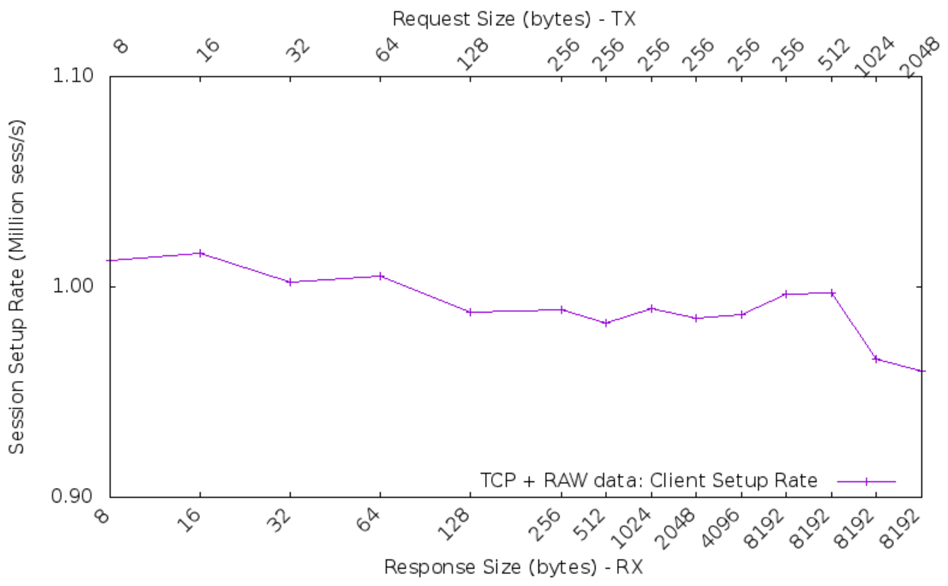


Figure 4.2: Amount of requested session per second from the client to the server for raw TCP (amount of requested sessions versus the request(Rx) and respond(Tx) size)

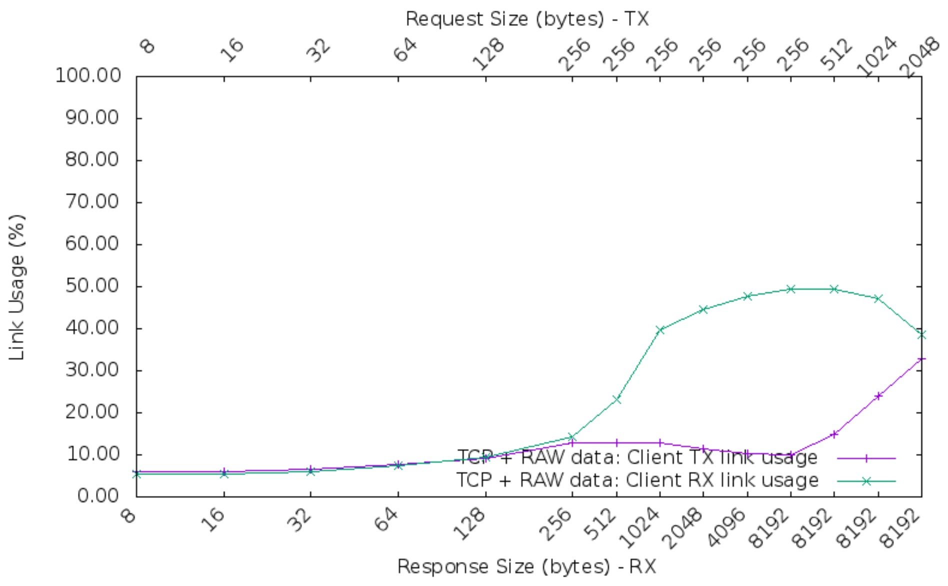


Figure 4.3: Link usage for raw TCP (percentage of link usage versus the request(Rx) and response(Tx) size)

The second test contains a HTTP file request. The server responds with a 200-(OK) message using a configured file size. The server is able to generate around one million sessions per second as seen in figure 4.4 but not all of the sessions were answered in time.

Figure 4.5 shows the bandwidth usage for HTTP traffic for the established sessions is at most 50%.

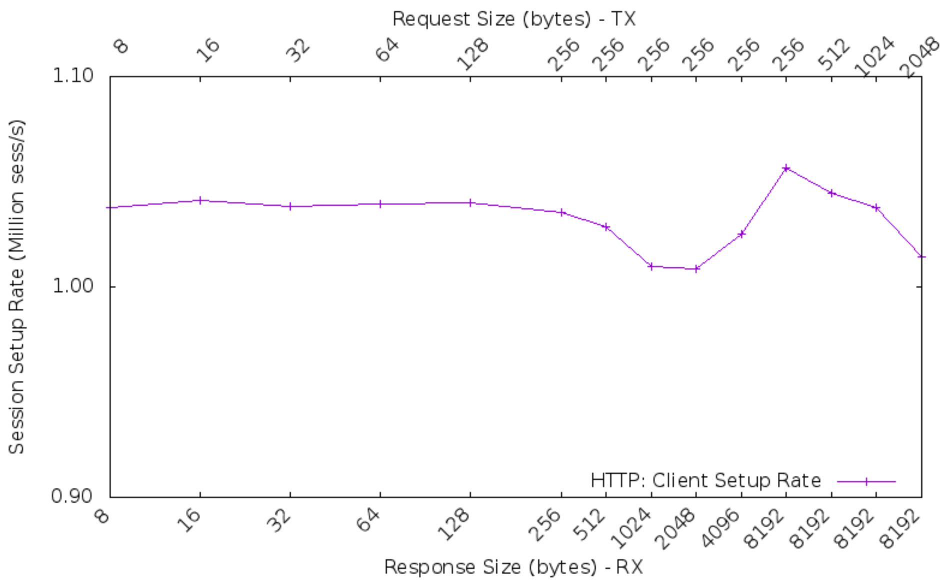


Figure 4.4: Amount of requested sessions per second from the client to the server for HTTP (amount of requested sessions versus the request(Rx) and respond(Tx) size)

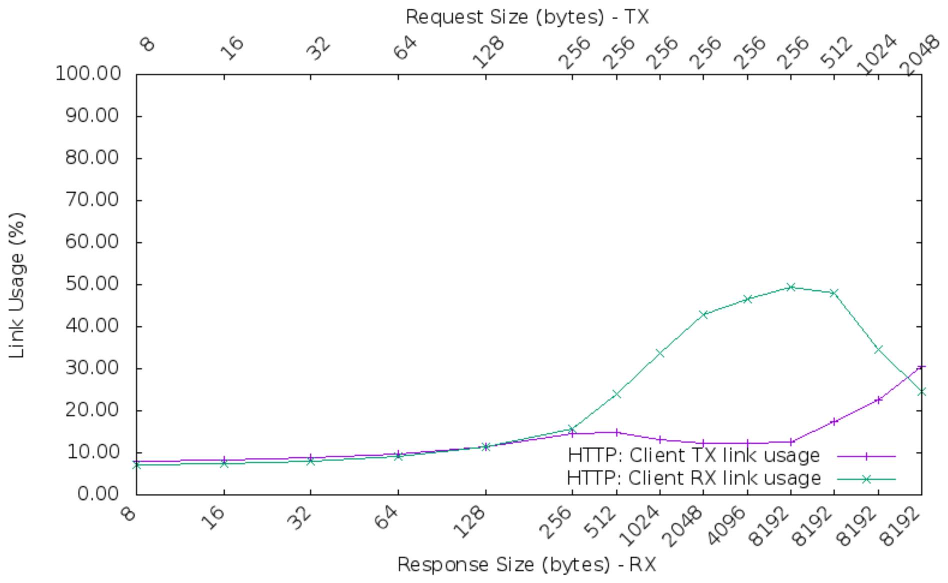


Figure 4.5: link usage for established sessions at HTTP (percentage of link usage versus the request(Rx) and response(Tx) size)

From the figures we can see WARP is capable of generating 1 million sessions per second for both raw TCP and HTTP traffic. It can fill up half of the link capacity when large packets are generated.

WARP is capable of generating 1 million sessions per second for raw TCP and HTTP traffic towards a destination, when the packet size increases it can generate 20Gb/s of application layer data. For stateful devices in a path to a server, the amount of sessions can be problematic. Overload behavior is expected from stateful devices in a path when WARP is used for limitation testing. This makes WARP useful for application level testing purposes.

4.3.3 MoonGen

MoonGen offers benchmark scripts to determine the machines capabilities. The benchmark script used for the tests can be found in appendix B.2 and is also provided with the application source. A single machine (B) is connected to the switch using two 40Gb/s ports at 2 separate cards (similar to the setup of the WARP test), both connected to switch S (one NIC for the server part and the other for the clients part of the benchmark). Sending UDP traffic with a size of 1500 bytes resulted in a maximum of 24 Gb/s. When the smallest possible Ethernet frames of 64 bytes are sent over the line a maximum of 15Mpps is reached. When TCP is used on the same machine (B) connected to the switch using two interfaces, a maximum of 10Gb/s was reached. These low values did not make MoonGen interesting enough since Pktgen is capable of reaching hardware limits.

Chapter 5

Methodology

From the results of the preliminary conducted experiments in chapter 4, two of the DPDK tools reached hardware limits and have provided the possibility to go up to application layer testing. to answer the underlying research question ”how can this be applied to end-to-end application level testing”, the research produced some tests using the most adequate tools from the experimental phase. Executing the proposed tests can pinpoint the limits of an infrastructure up to application level. The product of this research is presented in this chapter in the form of a set of tests using open source applications which enables network and system engineers to perform tests up to the application layer.

5.1 Tests

Table 5.1 displays four tests, all serving a specific purpose. Test 1 is performed to find the hardware limits of the NIC and the host that generates the traffic. When these limits are known, one can choose to use more sources to generate traffic if necessary. Test 2 is to find the limits of the switching and routing hardware in the path between the client and the server. Test 3 will find the client and server limits up to layer 4. To handle TCP sessions, resources need to be allocated at client, server and other stateful devices in the path from client to server if present. Test 4 will find the limits of an application running on top of a kernel. This test should be performed as an end-to-end test.

Tests 1 and 2 will show possible limitation at OSI layer 2 and 3. Tests 3 and 4 are meant to test the entire path from a client to a server over a production network, that might include a stateful firewall, aggregated links and redundant systems. The tests will stress the weakest link in the infrastructure towards the server. It should be noted that monitoring is a necessary part of the test in order to determine which device in the path is the 'weakest' and to find the threshold for a device before it starts to fail or exhibit other non-characteristic behavior.

5.1.1 T1 Bandwidth Generation

In paragraph 2.2 it is stated that a 40Gb/s card needs to be able to handle 56 million packets per second with a size of 88 bytes in a unidirectional stream of 40Gb/s. In chapter 4 it is explained that the new practical maximum of 42 million packets per second can be reached during this research due to the limitations of the PCI express bus. Using pktgen on top of DPDK, Ethernet frames with a minimum size of 64 bytes containing TCP traffic (without inter frame gap, preamble and VLAN tag) can be created. To reach the

NR	Tests	DUT	Goal
T1	Traffic generation	Client	The goal is to see if the client is capable of filling up the link and to reach the maximum amount of pps
T2	Throughput	Switch/router	Generate the maximum amount of data in both ways to make sure the hardware is able to forward at line rate
T3	TCP based	Client/Server	Get the limitations of the systems regarding memory and CPU usage
T4	Application	Server and intermediate devices	The clients will try to overload the server with requests at application level

Table 5.1: Performance tests

40Gb/s, larger packets have to be created. When the packet size is set to 1500 bytes, 3.3 Mpps saturates the link. When 9000 byte packets are used only 555 Kpps are needed to saturate the link. Using pktgen on top of DPDK allow numbers like these to be reached for TCP traffic. When DPDK is used, the kernel network stack does not communicate with the devices anymore since the interfaces are claimed by DPDK. Therefore traffic statistics cannot be read from the kernel and DPDK does not offer an straight forward method of reading these statistics either. Monitoring on the switch ports connecting the servers is needed. For this test the clients should try to generate the maximum amount of packets per second using Ethernet frames of 64 bytes. Section 4.3.1 explained that an optimum of maximum throughput and 11 Mpps can be sent by the client using a frame size of 400 bytes. This needs to be done during the second phase of the test to see if the required maximum bandwidth can be generated.

5.1.2 T2 Throughput

The backplane of a switch should be able to forward traffic from one port to another at line rate. This should also be possible when the traffic is routed from one VLAN to the other. Routers should be able to route packets at line rate. To test this a client and a server need to be connected to two different ports of the router in a different segment or VLAN and the links maximum capacity should be filled. Frames of 400 bytes or more should be generated from the client to the server and vice versa, preferably the maximum link capacity. Monitoring the ports of the switch using SNMP should show the same input rate on port 1 compared to the output rate of port 2.

5.1.3 T3 TCP Based

To determine the limitations of the server hardware when TCP is used, we have shown that the kernel has to be bypassed to reach the requirements this research was looking for. The combination of DPDK and WARP are capable of finding the limits of the hardware under load as seen in the experiments in chapter 4. The benchmark that was run on a single machine, running both as client and server, shows the ability to generate 1 million session requests per second from the client side, which generated a maximum of 20Gb/s of raw TCP traffic from the server to the client. Two useful tests can be derived from the results at the experimentation section: a raw TCP test between two different machines, and a test between client and server using the HTTP v1.1 protocol. WARP is capable of sending HTTP GET requests and it is capable of responding with a 200-OK message. WARP offers the possibility to set a session per second limitation for the clients. By ramping up the amount of sessions the load tolerance limits can be found. Monitoring of the amount of successful and failed session has to be done on both the client and server. When the API is used, detailed results can be retrieved as shown during the experimentation phase. Otherwise the maximum amount of sessions per second has to be throttled by the client.

5.1.4 T4 Application

End-to-end testing means testing from a client to a server application running on top of an OS. Running a web server offering files to clients, using HTTP request to retrieve files (since HTTP is the protocol that is used to distribute audio and video streaming) is a representative test. HTTP GET requests can be generated by WARP at a high rate (1 million session per second). The client will use WARP to generate HTTP GET requests for a web page hosted by the destination server. The web server can provide some files of different sizes. WARP has to send a request for a certain file using a request size determined by the user (the content of the requests will be padded to generate the configured frame size). Resources of the client, the web server and stateful devices on the path towards the server will be claimed opening up the sessions. By sending a million requests per second the machines will be experiencing a Denial of Service (DoS) like attack.

5.2 Real world scenario

The tests mentioned in section 5.1 need to be performed in a real world scenario to see if they can find limitation in the design of the environment. Therefore the infrastructure of a company was used to test this method, keeping in mind the guidance from section 4.1 on the use of production networks. Figure 5.1 shows the simplified infrastructure for the test between client and server. It is displaying the main equipment in the path from the client to the destination, including aggregated links, an upstream Internet provider, a stateful firewall and a data center layer using an overlay technique. The tests are performed during a maintenance window at the side of the destination, downtime for the tested part of the infrastructure was allowed during this window. All tests were generating and sending traffic for 90 seconds, with a gap of 30 seconds to let buffers be depleted before the next step of the test started in the same category. The monitoring system used at company x collects the links usage every minute. A 2 minute interval between the tests is 2 collection cycles and should provide the correct feedback. The server is connected to

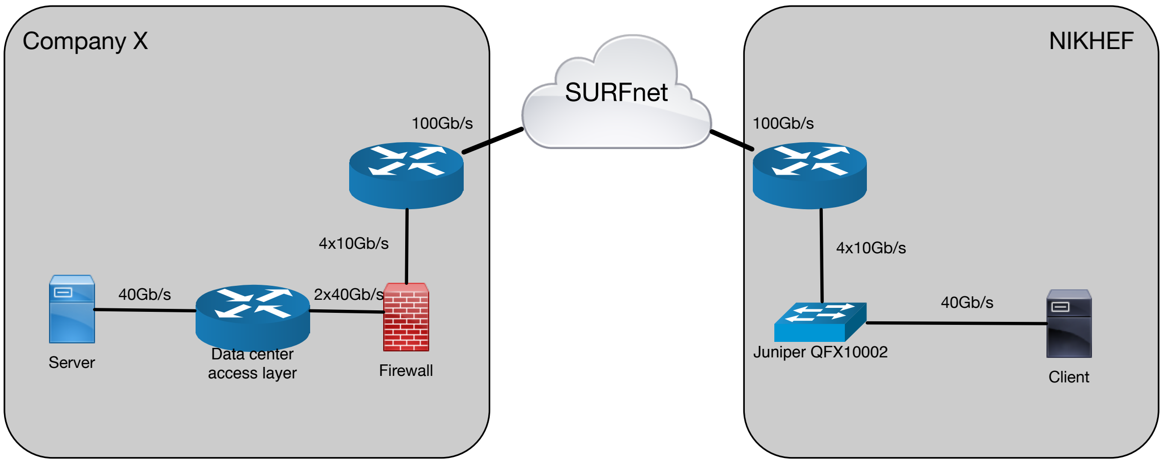


Figure 5.1: Simplified Infrastructure of the real world scenario for the real world test. The server on the left-hand side is the HTTP server.

a Data Center Interconnect (DCI) switch where an overlay technique (Ethernet Virtual Private Network (EVPN)) is used to guarantee service uptime.

5.2.1 T1 Bandwidth Generation

Using pktgen on top of DPDK with an Ethernet frame size of 64 bytes generated a maximum of 42 Mpps towards the server during the experimental phase. Pktgen only sends from one source to one destination, also using just one source and destination port. Appendix B.3 contains the script used for this test. The plan was to ramp up the frame size from 64 bytes to 400 bytes. The link from the switch to the router at the client side is an aggregated link built up from 4x 10Gb/s interfaces. Hashing algorithms[14] decide what physical interface of the aggregated link is used per stream. A switch's default hashing algorithm uses layer 2 addresses as input in this case causing all the traffic to end up in a single 10Gb/s link towards the destination. Even when a different hashing algorithm was used the link would have been the same since source and destination address and port are the same in this test. To reach the network of company x, the data had to flow over the production network at Nikhef. Hashing algorithms could not be altered during the tests since hashing algorithms affect every aggregated link on the device. Using extra sources to generate traffic, using different IP addresses could produce a higher amount of throughput because the hashing algorithm would choose a different link per server (depending on the chosen algorithm).

5.2.2 T2 Throughput

Due to the hashing settings that could not be changed without affecting Nikhef's production environment, the result of the second test is predictable and will be 10Gb/s (capacity of the single link used). Using four servers as a traffic source could generate a total throughput of 40Gb/s. Since the test environment had only 2 servers left (one was acting as the destination in the data center of company x), the link limitations could never be reached.

5.2.3 T3 TCP Based

Sending raw TCP sessions between 2 servers using WARP with different packet sizes should provide a good representation of the capabilities of the servers and the intermediate devices in the path from source to destination. The server should be running WARP, acting as a raw TCP server listening on 100 ports, and sending responses of a specified size when requests come in. The client should be configured to use 40.000 ports for the requests, targeting the 100 ports at the server side. This makes a total of 4 million possible flows between client and server. Request and response sizes can be equal at every run. Chosen TCP packet sizes are, 64, 256, 512, 1024 and 2048 bytes since utilization of the link is only 10% of the 40Gb/s connection when a TCP packet size of 256 bytes is used, since the test is looking for limitations, some load was necessary. Packets with a size of 2048 bytes take up 50% of the links capacity. The source and destination ports were changing at every session, therefore we expected to generate traffic up to a link capacity of 20 Gb/s.

Appendix B.3 displays the script that was used for this test. WARP was started using 8 CPU's and using all the memory. Server A from the experimental phase was used as the server side. Server B was used as the client generating the requests.

5.2.4 T4 Application

A web server needs to be set up offering a couple of different files. The files must be hosted in memory (on a RAM disk) to make sure disk IO will not become the bottleneck for the tests. Caching was disabled to make sure the performance of the web service was tested. Using WARP at the client side requesting the files from the web server. NGINX[2] was used as the web service for this test. The performance tuning page from NGINX was used to set the correct settings for the tests[20]. The amount of maximum available sockets was set to 50.000. This test will show the limits of the infrastructure towards the destination or from the server running the web application. WARP can be used as a web server responding with an HTTP 200-OK message, padded to enlarge the Ethernet frame to match the configured size. Testing application layer protocols using WARP as a client can result in more throughput and more sessions per second then during normal usage in a real life scenario. Pushing the amount of sessions up could stress the intermediate hardware to their limits.

Chapter 6

Results

Real world tests proposed in chapter 5 are executed from within the network of Nikhef to the network of company x during a maintenance window. For the methodology containing the use cases DPDK was used as a framework to find the limitations of the hardware in the path. Pktgen was used for T1 and T2. WARP was used to get the results for T3. A combination of WARP and NGINX was used to get the results for T4.

6.1 Infrastructure

The network at company x as it is shown in figure 5.1 is a simplified representation. The detailed infrastructure used during the real world test is displayed in figure 6.1. Since all the network hardware is redundant and a single device failure cannot result in total downtime the network and therefore the monitoring gets more complicated. The server is connected to a data center layer that is spread over two physical locations, one serving as the active and the other as the passive environment. To minimize broadcast traffic between the data centers an overlay technique is used. This overlay blocks broadcast storms at one location spreading to the other location. During the tests we did not reach hardware limits based on capacity. Therefore, the overhead from the overlay technique is not a problem during these tests. The network is not in production and the server used for the tests is the only server connected to the data center switches. When traffic does not arrive at the destination, detailed measurements are needed at every device for every link to determine where the traffic gets dropped.

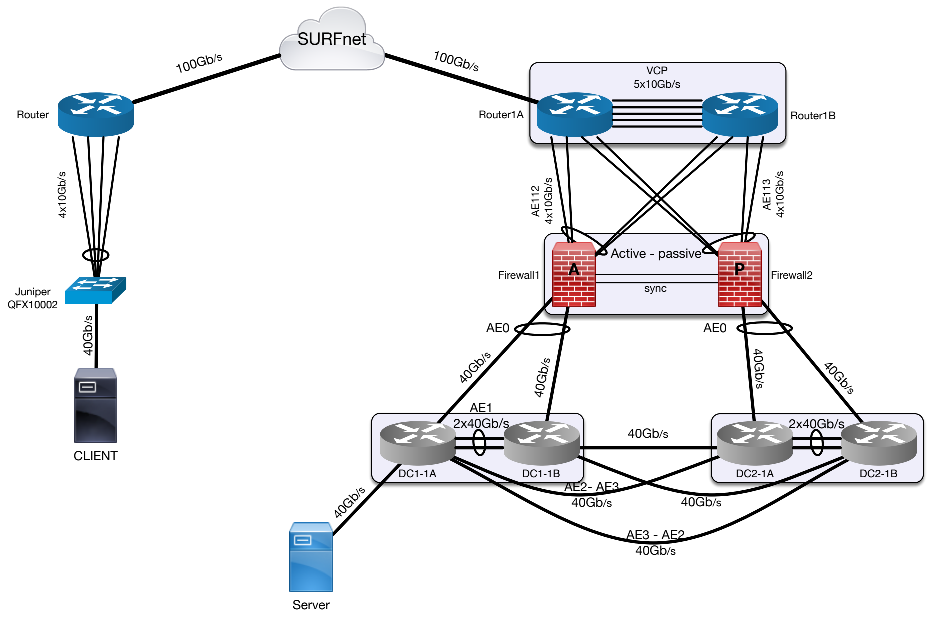


Figure 6.1: Detailed visualization of real world scenario.

6.2 Monitoring

During the test cases, we executed the tests while monitoring all of the interfaces used for the path from client to destination. The experimental environment was monitored using the SNMP collector. SURFstat monitors the link utilization between Nikhef and company x. Company x used an SNMP collector to monitor all the links from the Internet Service Provider to the destination of the test. Interfaces from possible backup paths were monitored as well. Due to the complexity of the network, graphs displaying multiple links are created to present a total picture of the used bandwidth during the maintenance window. Figures shown in this chapter display the graphs of the throughput that was sent out of the aggregated interfaces connecting router1 to the firewall cluster and the links connecting the firewalls to the DCI switches inside the data center. SURFstat provided figure 6.2 displaying the generated throughput between source and destination networks. This figure is used to see if the measured incoming throughput was the same as the measured outgoing bandwidth from Nikhef's network.

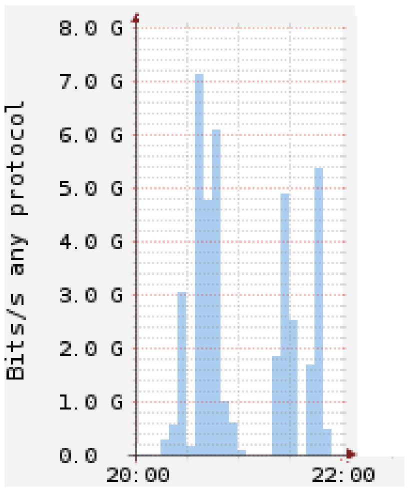


Figure 6.2: Bandwidth utilization of real world tests from Nikhef to company X during the time window the use cases were executed (figure provided by SURFstat).

6.3 Data Plane Development Kit based tools

The results of the DPDK based applications pktgen and WARP are explained during the performance tests in the real world scenario.

6.3.1 pktgen

Due to hashing algorithms used inside the core network of Nikhef, the real world test using pktgen could never reach more than 10Gb/s. The production network of Nikhef could not be altered during the tests.

The hashing algorithm used by the switch should be configured to divide the load coming in from the internal network. When most of the traffic is generated by one source to one destination using a single session, the throughput for a single flow will never reach above 10Gb/s in the network setup used during the tests. Knowing the traffic patterns during the design phase of a network can overcome these issues. Using multiple streams between client and server can improve the performance but this cannot be achieved using pktgen. Multiple clients generating traffic towards one destination can fill the capacity of multiple links to reach 40Gb/s.

6.3.2 WARP

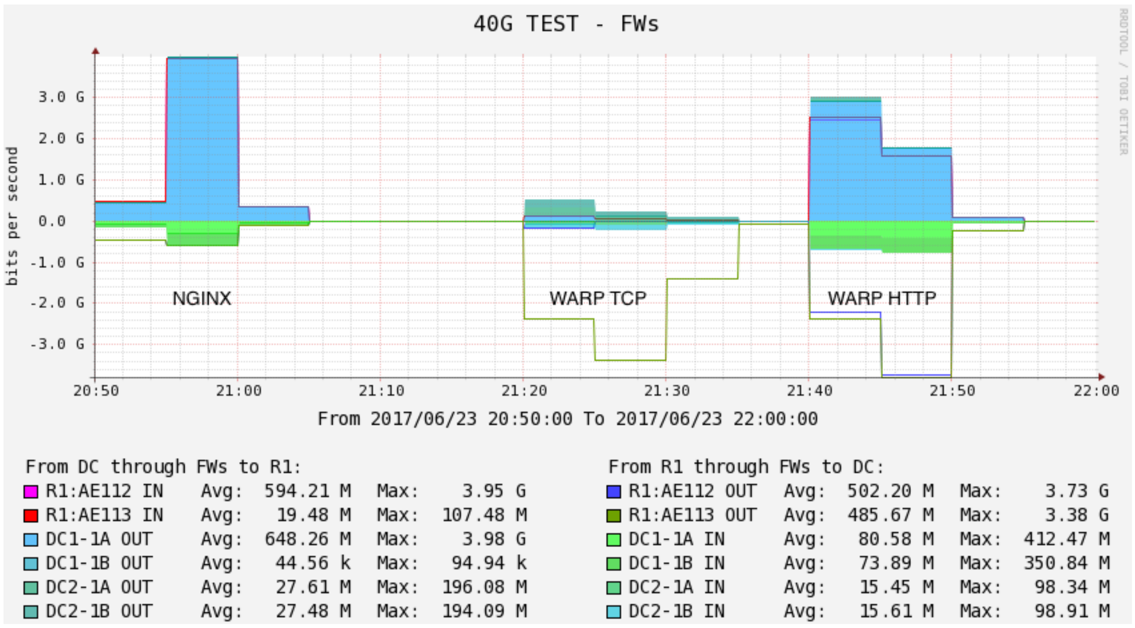
From the benchmark results performed on WARP in chapter 4 it is known WARP is capable of generating a million session per second for raw TCP and HTTP requests.

WARP was used to run three tests. WARP needs to be commanded using its own syntax. The command file that was used for the NGINX server test and the WARP HTTP server test can be found in appendix B.3. As WARP runs on top of DPDK, the DPDK commands point to resources that will be claimed by DPDK. As an example:

```
./build/warp17 -c ff -n 4 -m 32768 -- --qmap-default max-c \
--tcb-pool-sz 32768 --cmd-file test-client-nginx-http.txt
```

DPDK is started using 8 cores and 4 memory channels loading 32GB of memory, a default optimal mapping for CPU to NIC binding is chosen and the 32MB of memory is used per TCP control block size and a command file is used to start WARP.

Figure 6.3 shows the results of the three experiments executed with the use of WARP, to get the results for use cases three and four. Measurements were taken at the interfaces used to handle the data from the client going into the network towards the destination during the three upcoming tests. A first test where WARP was used to retrieve a 500 Kbyte file from an NGINX web server running on server A was executed. This NGINX server was tuned for performance[20]. The file was placed on a RAM disk to make sure disk IO would not be the bottleneck during the performance tests. The first test is executed starting at 20:50 and it ran until 21:00. The request size is increased every 90 seconds with an interval of 30 seconds between the tests. During this test the following request sizes where used: 64, 256, 512, 1024 and 2048 bytes. Figure 6.4 shows that the amount of traffic leaving the server goes above 4Gb/s while only 0.6 Gb/s of requests are coming in on the receiving end. This matches the values shown in figure 6.3.
A choice to rate limit at the NGINX server at the receiving end (allowing the server to accept a maximum of 50.000 concurrent sessions) was made to restrict the service from being overloaded by the client. Although the server was limited to 50.000 concurrent sessions, all the available sessions were used by WARP which made the service unavailable for other users. This proves the infrastructure and the application are capable of handling 50.000 sessions.



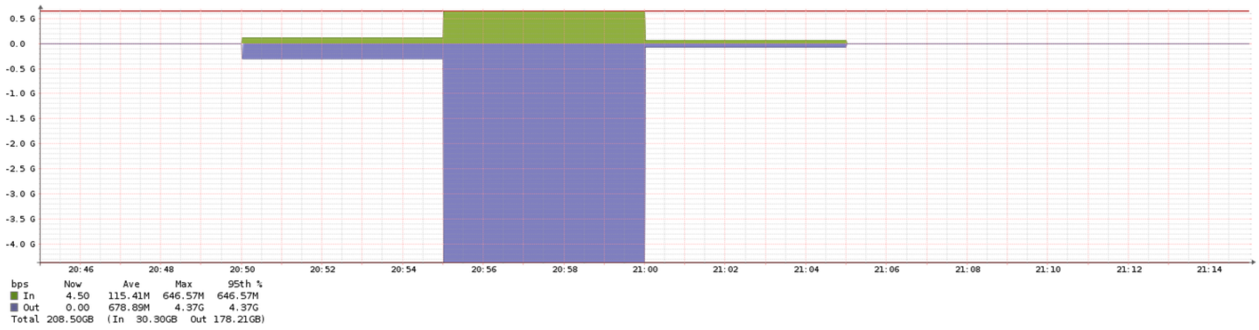
From DC through FWs to R1:

🟪	R1:AE112 IN	Avg: 594.21 M	Max:	3.95 G
🟥	R1:AE113 IN	Avg: 19.48 M	Max:	107.48 M
🟦	DC1-1A OUT	Avg: 648.26 M	Max:	3.98 G
🟦	DC1-1B OUT	Avg: 44.56 k	Max:	94.94 k
🟩	DC2-1A OUT	Avg: 27.61 M	Max:	196.08 M
🟩	DC2-1B OUT	Avg: 27.48 M	Max:	194.09 M

From R1 through FWs to DC:

🟦	R1:AE112 OUT	Avg: 502.20 M	Max:	3.73 G
🟩	R1:AE113 OUT	Avg: 485.67 M	Max:	3.38 G
🟩	DC1-1A IN	Avg: 80.58 M	Max:	412.47 M
🟩	DC1-1B IN	Avg: 73.89 M	Max:	350.84 M
🟩	DC2-1A IN	Avg: 15.45 M	Max:	98.34 M
🟦	DC2-1B IN	Avg: 15.61 M	Max:	98.91 M

Figure 6.3: Time line of the performed tests using WARP displaying the bandwidth usage over time



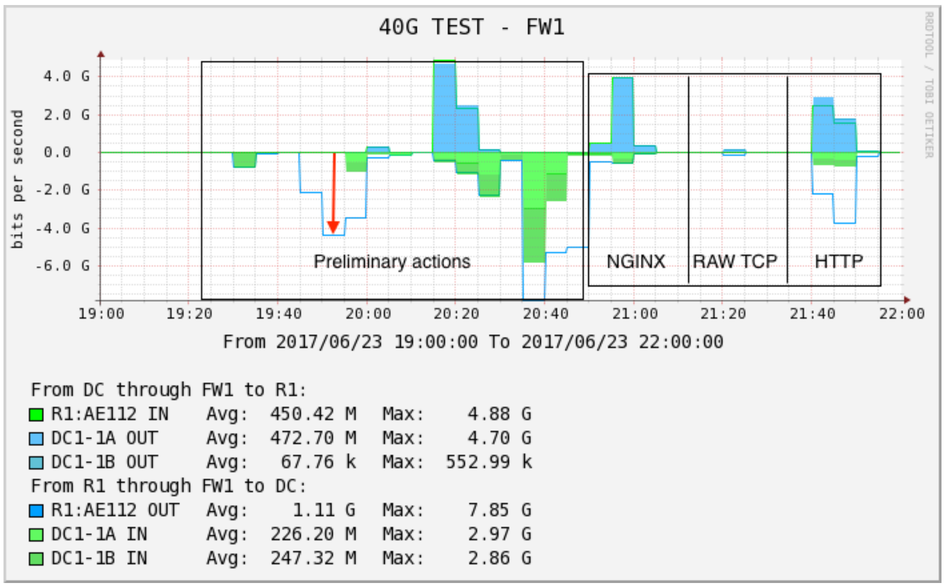
bps	Now	Ave	Max	95th %
🟩 In	4.50	115.41M	646.57M	646.57M
🟦 Out	0.00	678.89M	4.37G	4.37G

Total 208.50GB (In 30.30GB Out 178.21GB)

Figure 6.4: Bandwidth utilization server A during NGINX test, measurements from servers perspective

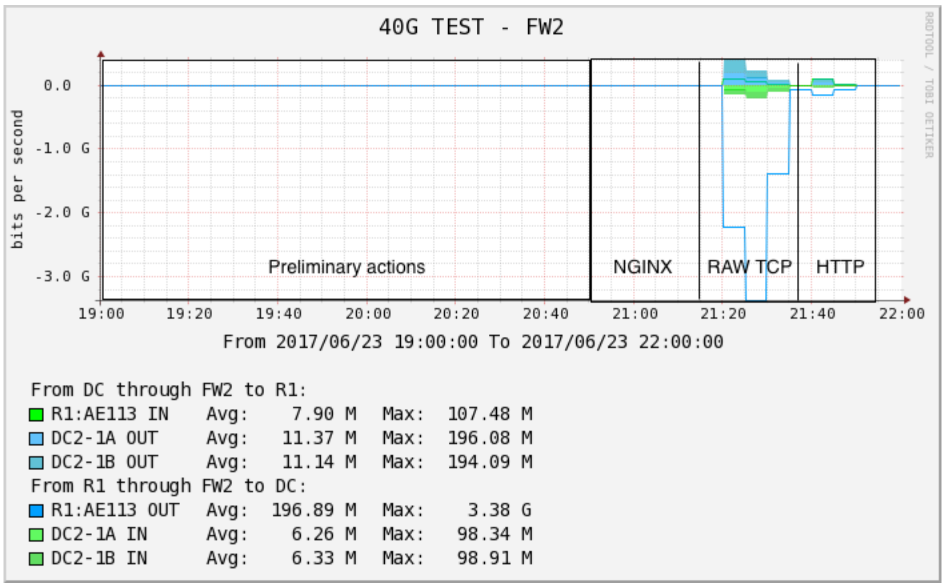
At 21:20 test 2 was started, to determine the limitation of handling layer 4 data as detailed in paragraph 5.2.4. Generating a maximum amount of raw TCP session from client to server both running WARP and using 32GB of memory and all cores to generate traffic. Request and response sizes chosen for this tests are: 64, 256, 512, 1024 and 2048 bytes. The bandwidth utilization measurements are from the uplink and downlink interfaces of the devices in the path as shown in figure 6.1 represented by "AE112" and "AE113". Two separate graphs are displayed for these links, figure 6.5 shows all the interfaces connected to firewall1 and figure 6.6 shows all the interfaces connected to firewall2. The graphs display traffic that went out of one interface and is expected to come into the interface at the other side when the firewall forwards the traffic, when this does not happen white gaps can seen as marked with the red arrow in figure 6.5. The blue line represents data that left the router interface and the green bar represent the data that was received by the data center switches. These white gaps in the graph represent the data that got lost due to failing hardware during the execution of the tests. Data points above zero represent data that went from server to client and data points below zero represent data from client to server.

Firewall sessions statistics were not registered due to logging problems in the firewall environment that we were not able to repair before the end of the maintenance window. Chapter 4 shows that the benchmark value of 1 million sessions per second can be generated, where only small differences are observed at different packets sizes. The graph in figure 6.5 doesn't display any traffic at the start of this second test at 21:20. This is the moment the raw TCP test was started while figure 6.2 displays traffic towards the tested network.



```
From DC through FW1 to R1:
🟩 R1:AE112 IN     Avg:  450.42 M   Max:    4.88 G
🟦 DC1-1A OUT      Avg:  472.70 M   Max:    4.70 G
🟦 DC1-1B OUT      Avg:   67.76 k   Max:  552.99 k
From R1 through FW1 to DC:
🟦 R1:AE112 OUT    Avg:    1.11 G   Max:    7.85 G
🟩 DC1-1A IN       Avg:  226.20 M   Max:    2.97 G
🟩 DC1-1B IN       Avg:  247.32 M   Max:    2.86 G
```

Figure 6.5: Bandwidth utilization of links between router1, firewall1 and DC1 during the time window of the tests, spikes represent an executed test.



```
From DC through FW2 to R1:
🟩 R1:AE113 IN    Avg:    7.90 M   Max:  107.48 M
🟦 DC2-1A OUT     Avg:   11.37 M   Max:  196.08 M
🟦 DC2-1B OUT     Avg:   11.14 M   Max:  194.09 M
From R1 through FW2 to DC:
🟦 R1:AE113 OUT   Avg:  196.89 M   Max:    3.38 G
🟩 DC2-1A IN      Avg:    6.26 M   Max:   98.34 M
🟩 DC2-1B IN      Avg:    6.33 M   Max:   98.91 M
```

Figure 6.6: Bandwidth utilization of links between router1, firewall2 and DC2 during the time window of the test, the spike represents the moment of a fail over to the secondary machine.

The graph in figure 6.6 displays the generated traffic. When the test started, the active firewall crashed and a failover to the passive machine is the result. Log messages retrieved from connected routers also display BGP session failures from the formerly-active firewall. This graph also displays the difference between traffic sent from the router to the firewall and the traffic received by the downstream switch. Input is 3Gb/s and output is in the range of 200 - 300 Mb/s. The firewall was configured to accept all the traffic from the source range of the test server inside the network.

The firewalls are not capable of handling this amount of sessions per second. Due to time restrictions this test could not be performed again to pinpoint the maximum number of sessions. When the test was finished, the firewalls recovered.

At 21:40 the third and last test is started between server and client, again both running WARP. Generating the maximum amount of HTTP sessions using 32GB of memory and all available cores. Generating a GET request and responding with a 200-OK message from the server running WARP instead of NGINX. From the experimental phase of this research it is known that WARP is capable of handling more requests per second than NGINX can. This last test is executed to find the limits of intermediate hardware when HTTP sessions are opened up in a fast rate. Message sizes for this tests are : 64, 256, 512, 1024 and 2048 bytes. Request and response size are equal. With the information from the benchmark in chapter 4 the limitations of client and server are known. From test two it is known the firewalls cannot handle the load when to many sessions per second come in. The same behavior is expected during this last test. Figure 6.5 shows 4Gb/s going into the firewall and only 500Mb/s of HTTP traffic arriving at the other side. This means that the firewall was not able to handle the amount of new sessions per second which caused it to stop functioning[13]. Management sessions broke down and traffic got lost as the figures display.

In a real world scenario like the one we used in these test, limitations can be revealed by executing the test described in chapter 5. The unexpected fail over of the firewall when one million session were sent to the destination server is a perfect example of finding the weakest link in the chain. It was not the bandwidth nor the amount of packets per second that made the firewall crash, but the amount of sessions are the limitation for tests two and three.

This proves that DPDK and WARP can be utilized to perform end-to-end application layer testing. WARP can generate the load to a service in order to see if the service can handle the expected load. WARP can also be used to find the limitation of the infrastructure towards an application.

Chapter 7

Conclusion

From the tool assessment performed in chapter 4 and the tests described in chapter 5, results are gathered and described in chapter 6. The results, executed according to current standards and best practices, allow to draw conclusions on tool suitability and the necessary characteristics of high-bandwidth session based throughput tests in a real-world environment.
When it comes to generating session based high bandwidth throughput testing, DPDK should be used in combination with pktgen in order to reach hardware limitations. Using DPDK in combination with WARP creates the possibility to generate traffic at the application layer. The kernel based tools could not provide the results this research was looking for.

During this research, the limitations of the hardware used in the experimental setup were found by executing the tests. The tests can be used as a guideline to find the hardware limits in the path from client to server. T1 revealed a limitation for the amount of packets per second in the PCI Express bus. T2 was used to get the maximum possible throughput from client to server, the hashing settings were shown to be a limitation in the setup as expected. T3 revealed the hardware limits for client and server with regards to the amount of sessions and bandwidth usage. Next to the hardware limits T3 revealed a limitation of a stateful firewall in the path towards the destination, the overload behavior of the firewall is also known by executing this test.
T4 stressed an application to get the performance limits for this specific application.

These tests should be used to get a better insight in the limitations of an infrastructure that is used to provide services. Combining DPDK with pktgen and WARP can reveal the limits of an infrastructure. To perform high bandwidth session based throughput tests up to layer 3, pktgen on top of DPDK is capable of reaching hardware limits. For application layer link testing, WARP is the framework to use. Support for other applications must be added to WARP in order to improve the employability, but the start looks promising. The use of different kernels did not show any major differences in the results of executed kernel based application tests during this research.

The exact limit of the firewall was not found, it is only known that one server using WARP can generate the amount of sessions per second to make the system fail. By increasing the amount of sessions step by step we could have pinpointed the amount of sessions where the firewall started failing.

7.1 Suitability of the Data Plane Development Kit

The Data Plane Development Kit is still a work in progress, and today we see only the beginning of its potential being harnessed for network load testing. New tools that use the power of DPDK are introduced every year: Pktgen (2013), MoonGen(2014), T-rex (2015), WARP(2016). The possibilities to test up to layer 7 in the OSI model are now becoming available to system and network administrators. Current tooling is capable of generating a million sessions per second using simple server hardware. DPDK applications supporting IPv6 and multiple application layer protocols are needed to improve infrastructures in order to offer services.

7.2 Future Work

The hashing algorithm used at Nikhef's core network limited the performance tests to 10Gb/s. Using 4 clients or changing the hashing settings should result in more bandwidth utilization. By doing this, the other limitation this research was looking for such as the amount of packets per second being a bottleneck can be reached. Further analysis on the Data Center Infrastructure layer at company x can be performed.

During this project an attempt was made to use an IBM Power8 machine (server E) to generate traffic at 100Gb/s. Because of problems during compilation and memory allocation this attempt had to be abandoned due to time constraints.

This project used HTTP version 1.1 for application testing. Support for more protocols need to be added to WARP to make it more powerful. Currently WARP supports IPv4 only. When IPv6 is supported, the performance should be tested using IPv6. Monitoring in WARP should be improved, currently the API provides the only way of getting detailed results. NGINX was made available for DPDK recently. Running WARP towards a DPDK NGINX server should provide the capabilities of NGINX when it does not rely on kernel interrupts.

DPDK supports multiple NICs. During the project an effort was made to start generating traffic over 100Gb/s Mellanox cards. This was successful up to 60Gb/s TCP traffic, until the system crashed for reasons that could not be determined within the scope of this project. The proposed tests in this research paper need to be run using the Mellanox cards. Support and limitations for different 100Gb/s cards need to be researched.

The Generation 3, 8 lane PCI express cards are a limiting factor as shown in this paper. Further investigations could look into the limitations of 16-lane PCIe and its associated scaling behavior.

Intel offers a guide to improve the throughput for the XL710 40Gb/s card for the Linux kernel[16]. This guideline provides kernel settings that might improve the results for the kernel based tools. During this research the guideline was not used to improve the kernel settings, the reason for this is that the settings are dependent on the application that is ran on top of the kernel. This research, due to time constraints focused on the

DPDK tooling. Tweaking the kernel for all the tools in table 3.1 is out of scope for this research while it would be very interesting to know if the proposed settings from the guide will impact the performance of the kernel based tools.

Bibliography

[1] iperf - the tcp, udp and sctp network bandwidth measurement tool. URL https://iperf.fr.

[2] Welcome to nginx wiki! — nginx. URL https://www.nginx.com/resources/wiki/.

[3] Hping - active network security tool, 2006. URL http://www.hping.org.

[4] Bonesi - the ddos botnet simulator, 2008. URL https://github.com/Markus-Go/bonesi.

[5] Dpdk (data plane development kit), 2013. URL http://dpdk.org.

[6] pktgen-dpdk - traffic generator powered by dpdk, 2013. URL http://dpdk.org/browse/apps/pktgen-dpdk/.

[7] libmoon: libmoon is a library for fast and flexible packet processing with dpdk and luajit, 2014. URL https://github.com/libmoon/libmoon.

[8] emmericp/moongen: Moongen is a fully scriptable high-speed packet generator built on dpdk and luajit, 2014. URL https://github.com/emmericp/MoonGen.

[9] Juniper/warp17: The stateful traffic generator for layer 1 to layer 7, 2016. URL https://github.com/Juniper/warp17.

[10] Computing - cern, 2017. URL https://home.cern/about/computing.

[11] Dpdk supported network interface cards, 2017. URL http://dpdk.org/doc/nics.

[12] Ethernet frame explanation, 2017. URL https://en.wikipedia.org/wiki/Ethernet_frame.

[13] Fortigate 3200d data sheet, 2017. URL https://www.fortinet.com/content/dam/fortinet/assets/data-sheets/FortiGate_3200D.pdf.

[14] Hashing algorithms, 2017. URL https://www.juniper.net/documentation/en_US/junos/topics/concept/load-balance-technique-overview.html.

[15] Dpdk performance guide for xl710, 2017. URL http://dpdk.org/doc/guides/linux_gsg/nic_perf_intel_platform.html?highlight=xl710.

[16] Linux guide for intel xl710 nic, 2017. URL https://www.intel.com/content/dam/www/public/us/en/documents/reference-guides/xl710-x710-performance-tuning-linux-guide.pdf.

[17] Joint engineering team (jet) - nitrdgroups, 2017. URL https://www.nitrd.gov/nitrdgroups/index.php?title=Joint_Engineering_Team_(JET).

[18] Warp benchmark script, 2017. URL https://github.com/Juniper/warp17/blob/15d0f5a51619259bd75dca42a9ccbccedb94880b/examples/python/test_2_perf_benchmark.py.

[19] Moongen benchmark script, 2017. URL https://github.com/emmericp/MoonGen/blob/b5c6c5f7c82b153deecbefcf694c4121c81f3ca5/examples/l3-tcp-syn-flood.lua.

[20] Nginx tuning, 2017. URL https://www.nginx.com/blog/tuning-nginx/.

[21] *Sandvine report on protocol usage.* 2017. URL https://www.sandvine.com/downloads/general/global-internet-phenomena/2016/global-internet-phenomena-report-latin-america-and-north-america.pdf.

[22] Juniper qfx10000 pic port speed, 2017. URL https://www.juniper.net/documentation/en_US/release-independent/junos/topics/reference/specifications/port-panel-qfx10002-36Q.html.

[23] Esnet - science dmz, 2017. URL https://fasterdata.es.net/science-dmz/.

[24] Product specification for chelsio t580, 2017. URL https://www.chelsio.com/nic/unified-wire-adapters/t580-cr/.

[25] Product specifications for intel xl710, 2017. URL http://ark.intel.com/nl/products/83967/Intel-Ethernet-Converged-Network-Adapter-XL710-QDA2.

[26] Ethernet Alliance. Ethernet jumbo frames, 2017. URL http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf.

[27] R. Asati, C. Pignataro, F. Calabria, and C. Olvera. Device reset characterization. RFC 6201, March 2011.

[28] S. Bradner. Benchmarking Terminology for Network Interconnection Devices. RFC 1242 (Informational), July 1991. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc1242.txt. Updated by RFC 6201.

[29] S. Bradner and J. McQuaid. Benchmarking Methodology for Network Interconnect Devices. RFC 2544 (Informational), March 1999. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc2544.txt. Updated by RFCs 6201, 6815.

[30] S. Bradner, K. Dubray, J. McQuaid, and A. Morton. Applicability statement for rfc 2544: Use on production networks considered harmful. RFC 6815, RFC Editor, November 2012. URL http://www.rfc-editor.org/rfc/rfc6815.txt. http://www.rfc-editor.org/rfc/rfc6815.txt.

[31] B. Constantine, G. Forget, R. Geib, and R. Schrage. Framework for TCP Throughput Testing. RFC 6349 (Informational), August 2011. ISSN 2070-1721. URL `https://www.rfc-editor.org/rfc/rfc6349.txt`.

[32] Paul Emmerich, Sebastian Gallenmller, Daniel Raumer, Florian Wohlfart, and Georg Carle. *MoonGen: A Scriptable High-Speed Packet Generator*. 1 edition, 2015. URL `https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/MoonGen_IMC2015.pdf`.

[33] Jelte Fennema. Modifying existing applications for 100 gigabit ethernet. URL `www.delaat.net/rp/2015-2016/p94/report.pdf`.

[34] Chelsio Communications Inc. *FreeBSD 40GbE Netmap Performance*. Chelsio, 2015. URL `https://www.chelsio.com/wp-content/uploads/resources/T5-40Gb-FreeBSD-Netmap.pdf`.

[35] Soumya Mahalakshmi, BS Amulya, and Minal Moharir. A study of tools to develop a traffic generator for l4–l7 layers. In *Wireless Communications, Signal Processing and Networking (WiSPNET), International Conference on*, pages 114–118. IEEE, 2016.

[36] David Murray and Terry Koziniec. The state of enterprise network traffic in 2012. In *Communications (APCC), 2012 18th Asia-Pacific Conference on*, pages 179–184. IEEE, 2012.

[37] Thomas Stockhammer. Dynamic adaptive streaming over http–: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 133–144. ACM, 2011.

[38] Daniel Turull, Peter Sjödin, and Robert Olsson. Pktgen: Measuring performance on high speed networks. *Computer communications*, 82:39–48, 2016.

Appendix A

Acronym	**Definition**
DTN	Data Transfer Node
CLI	Command Line Interface
PPS	packets per second
Gb/s	Gigabit per second
DPDK	Data Plane Development Kit
ISP	Internet service provider
QSFP	Quad Small Form-factor Pluggable
DOS	Denial Of Service
DDOS	Distributed Denial Of Service
FCS	Frame check sequence
MTU	Maximum Transmission Unit
OSI	Open System Interconnection
NIC	Network Interface Card
TCP	Transport Control Protocol
UDP	User Datagram Protocol
ICMP	Internet Control Message Protocol
OS	Operating System
LAN	Local Area Network
VLAN	Virtual LAN
IP	Internet Protocol
HTTP	Hyper Text Transport Protocol
MTU	Maximum Transmission Unit
SNMP	Simple Network management Protocol
RFC	Request For Comment
DUT	Device Under Tests
CPU	Central Processing Unit
UC	Use Case
API	Application Programmable Interface
EVPN	Ethernet Virtual Private Network
DWDM	Dense Wave Division Multiplexing

Table A.1: Used Acronyms

Appendix B

B.1 Software

Software used during research:

- Ubuntu 16.04 LTS
- FreeBSD 11.0
- DPDK 16.11
- pktgen(dpdk) 3.3.4
- WARP 1.4
- Moongen
- iPerf 3.1.3
- hping 3
- Bonesi V0.3

B.2 benchmark test

Benchmark script for WARP:
This script is provided with the application source[18].

```
import sys
import time

from functools import partial

sys.path.append('../../python')
sys.path.append('../../api/generated/py')
sys.path.append('../../ut/lib')

from warp17_api import *

from b2b_setup import *

from warp17_common_pb2    import *
from warp17_l3_pb2        import *
from warp17_server_pb2    import *
from warp17_client_pb2    import *
from warp17_app_raw_pb2   import *
from warp17_app_http_pb2  import *
from warp17_test_case_pb2 import *
from warp17_service_pb2   import *

# 4M sessions
sip_cnt = 1
dip_cnt = 1
sport_cnt = 40000
```

```
dport_cnt = 100

sess_cnt = sip_cnt * dip_cnt * sport_cnt * dport_cnt
serv_cnt = dip_cnt * dport_cnt

expected_rate = 1000000
run_cnt = 3

def die(msg):
    sys.stderr.write(msg + ' Should cleanup but we just exit..\n')
    sys.exit(1)

def get_server_test_case(protocol, app, req_size, resp_size):
    l4_scfg = L4Server(l4s_proto=protocol,
                       l4s_tcp_udp=TcpUdpServer(tus_ports=b2b_ports(dport_cnt)))
    app_scfg = {
        RAW: AppServer(as_app_proto=RAW,
                       as_raw=RawServer(rs_req_plen=req_size,
                                        rs_resp_plen=resp_size)),
        HTTP: AppServer(as_app_proto=HTTP,
                        as_http=HttpServer(hs_resp_code=OK_200,
                                           hs_resp_size=resp_size))
    }.get(app)

    return TestCase(tc_type=SERVER, tc_eth_port=1, tc_id=0,
                    tc_server=Server(srv_ips=b2b_sips(1, sip_cnt),
                                     srv_l4=l4_scfg,
                                     srv_app=app_scfg),
                    tc_criteria=TestCriteria(tc_crit_type=SRV_UP,
                                             tc_srv_up=serv_cnt),
                    tc_async=False)

def get_client_test_case(protocol, app, req_size, resp_size):
    l4_ccfg = L4Client(l4c_proto=protocol,
                       l4c_tcp_udp=TcpUdpClient(tuc_sports=b2b_ports(sport_cnt),
                                                tuc_dports=b2b_ports(dport_cnt)))
    rate_ccfg = RateClient(rc_open_rate=Rate(),
                           rc_close_rate=Rate(),
                           rc_send_rate=Rate())

    delay_ccfg = DelayClient(dc_init_delay=Delay(d_value=0),
                             dc_uptime=Delay(),
                             dc_downtime=Delay())

    app_ccfg = {
        RAW: AppClient(ac_app_proto=RAW,
                       ac_raw=RawClient(rc_req_plen=req_size,
                                        rc_resp_plen=resp_size)),
        HTTP: AppClient(ac_app_proto=HTTP,
                        ac_http=HttpClient(hc_req_method=GET,
                                           hc_req_object_name='/index.html',
                                           hc_req_host_name='www.foobar.net',
                                           hc_req_size=req_size))
    }.get(app)

    return TestCase(tc_type=CLIENT, tc_eth_port=0,
                    tc_id=0,
                    tc_client=Client(cl_src_ips=b2b_sips(0, sip_cnt),
                                     cl_dst_ips=b2b_dips(0, dip_cnt),
                                     cl_l4=l4_ccfg,
                                     cl_rates=rate_ccfg,
                                     cl_delays=delay_ccfg,
                                     cl_app=app_ccfg),
                    tc_criteria=TestCriteria(tc_crit_type=CL_ESTAB,
                                             tc_cl_estab=sess_cnt),
                    tc_async=False)

def configure_server_port(warp17_call, protocol, app, req_size, resp_size):
    pcfg = b2b_port_add(eth_port=1, def_gw=Ip(ip_version=IPV4, ip_v4=0))
    b2b_port_add_intfs(pcfg,
                       [
                        (Ip(ip_version=IPV4, ip_v4=b2b_ipv4(eth_port=1, intf_idx=i)),
                         Ip(ip_version=IPV4, ip_v4=b2b_mask(eth_port=1, intf_idx=i)),
                         b2b_count(eth_port=1, intf_idx=i)) for i in range(0, dip_cnt)
                       ])

    if warp17_call('ConfigurePort', pcfg).e_code != 0:
        die('Error configuring port 1!')

    scfg = get_server_test_case(protocol, app, req_size, resp_size)
    if warp17_call('ConfigureTestCase', scfg).e_code != 0:
        die('Error configuring server test case')

def configure_client_port(warp17_call, protocol, app, req_size, resp_size):
    pcfg = b2b_port_add(eth_port=0, def_gw=Ip(ip_version=IPV4, ip_v4=0))
    b2b_port_add_intfs(pcfg,
                       [
                        (Ip(ip_version=IPV4, ip_v4=b2b_ipv4(eth_port=0, intf_idx=i)),
                         Ip(ip_version=IPV4, ip_v4=b2b_mask(eth_port=0, intf_idx=i)),
                         b2b_count(eth_port=0, intf_idx=i)) for i in range(0, sip_cnt)
                       ])

    if warp17_call('ConfigurePort', pcfg).e_code != 0:
        die('Error configuring port 0!')

    ccfg = get_client_test_case(protocol, app, req_size, resp_size)
    if warp17_call('ConfigureTestCase', ccfg).e_code != 0:
        die('Error configuring client test case')


def run_test(protocol, app, req_size, resp_size):
    env = Warp17Env(path='./test_2_perf_benchmark.ini')
    warp17_pid = warp17_start(env=env, exec_file='../../build/warp17',
                              output_args=Warp17OutputArgs(out_file='/tmp/test_2_perf.out'))
    warp17_wait(env=env, logger=LogHelper(name='benchmark',
                                          filename='/tmp/test_2_perf.log'))

    warp17_call = partial(warp17_method_call, env.get_host_name(),
                          env.get_rpc_port(), Warp17_Stub)

    configure_server_port(warp17_call, protocol, app, req_size, resp_size)
    configure_client_port(warp17_call, protocol, app, req_size, resp_size)
    timeout_s = int(sess_cnt / float(expected_rate)) + 2

    if warp17_call('PortStart', PortArg(pa_eth_port=1)).e_code != 0:
        die('Error starting server test cases!')

    if warp17_call('PortStart', PortArg(pa_eth_port=0)).e_code != 0:
        die('Error starting client test cases!')
    time.sleep(timeout_s)

    result = warp17_call('GetTestStatus', TestCaseArg(tca_eth_port=0,
                                                       tca_test_case_id=0))
    if result.tsr_state != PASSED:
        die('Test case didn\'t pass: ' + str(result))

    start_time = result.tsr_stats.tcs_start_time
    end_time = result.tsr_stats.tcs_end_time

    # start and stop ts are in usecs
    duration = (end_time - start_time) / float(1000000)
    rate = sess_cnt / duration
    txr = result.tsr_link_stats.ls_tx_pkts / duration
    rxr = result.tsr_link_stats.ls_rx_pkts / duration
    link_speed_bytes = float(result.tsr_link_stats.ls_link_speed) * 1024 * 1024 / 8
    tx_usage = min(float(result.tsr_link_stats.ls_tx_bytes) * 100 / duration / link_speed_bytes, 100.0)
    rx_usage = min(float(result.tsr_link_stats.ls_rx_bytes) * 100 / duration / link_speed_bytes, 100.0)

    warp17_stop(env, warp17_pid, force=True)
    return (rate, txr, rxr, tx_usage, rx_usage)

def run_test_averaged(descr, protocol, app, req_size, resp_size, run_cnt):
    results = [run_test(protocol, app, req_size, resp_size)
               for i in range(0, run_cnt)]
    avgs = [sum(result, 0.0) / run_cnt for result in zip(*results)]

    # Print as csv
    print '%(descr)s,%(req_size)u,%(resp_size)u,%(rate).0f,%(txr).0f,%(rxr).0f,%(txu).2f,%(rxu).2f' % \
           {
            'descr': descr, 'req_size': req_size, 'resp_size': resp_size,
            'rate': avgs[0], 'txr': avgs[1], 'rxr': avgs[2],
            'txu': avgs[3], 'rxu': avgs[4]
           }

def run():

    # Print csv header
    print 'Description, req_size, resp_size, rate, tx pps, rx pps, tx usage, rx usage'

    # TCP RAW
    tcp_raw_cfg = [(8, 8), (16, 16), (32, 32), (64, 64), (128, 128),
                   (256, 256), (256, 512), (256, 1024), (256, 2048), (256, 4096),
                   (256, 8192), (512, 8192), (1024, 8192), (2048, 8192)]

    for (req_size, resp_size) in tcp_raw_cfg:
        run_test_averaged('TCP request={req!s}b response={resp!s}b'.format(req=req_size,
                                                                        resp=resp_size),
                          TCP, RAW, req_size, resp_size, run_cnt)

    # HTTP
    http_cfg = [(8, 8), (16, 16), (32, 32), (64, 64), (128, 128),
                (256, 256), (256, 512), (256, 1024), (256, 2048), (256, 4096),
                (256, 8192), (512, 8192), (1024, 8192), (2048, 8192)]

    for (req_size, resp_size) in http_cfg:
        run_test_averaged('HTTP request={req!s}b response={resp!s}b'.format(req=req_size,
                                                                         resp=resp_size),
                          TCP, HTTP, req_size, resp_size, run_cnt)
```

```
if __name__ == '__main__':
    run()
```

Benchmark script for Moongen:
This script is provided with the application source[19].

```
local mg                = require "moongen"
local memory    = require "memory"
local device    = require "device"
local stats             = require "stats"
local log               = require "log"

function configure(parser)
        parser:description("Generates TCP SYN flood from varying source IPs, supports both IPv4 and IPv6")
        parser:argument("dev", "Devices to transmit from."):args("*"):convert(tonumber)
        parser:option("-r --rate", "Transmit rate in Mbit/s."):default(10000):convert(tonumber)
        parser:option("-i --ip", "Source IP (IPv4 or IPv6)."):default("10.0.0.1")
        parser:option("-d --destination", "Destination IP (IPv4 or IPv6).")
        parser:option("-f --flows", "Number of different IPs to use."):default(100):convert(tonumber)
end

function master(args)
        for i, dev in ipairs(args.dev) do
                local dev = device.config{port = dev}
                dev:wait()
                dev:getTxQueue(0):setRate(args.rate)
                mg.startTask("loadSlave", dev:getTxQueue(0), args.ip, args.flows, args.destination)
        end
        mg.waitForTasks()
end

function loadSlave(queue, minA, numIPs, dest)
```

```
--- parse and check ip addresses
local minIP, ipv4 = parseIPAddress(minA)
if minIP then
        log:info("Detected an %s address.", minIP and "IPv4" or "IPv6")
else
        log:fatal("Invalid minIP: %s", minA)
end

-- min TCP packet size for IPv6 is 74 bytes (+ CRC)
local packetLen = ipv4 and 60 or 74

-- continue normally
local mem = memory.createMemPool(function(buf)
        buf:getTcpPacket(ipv4):fill{
                ethSrc = queue,
                ethDst = "12:34:56:78:90",
                ip4Dst = dest,
                ip6Dst = dest,
                tcpSyn = 1,
                tcpSeqNumber = 1,
                tcpWindow = 10,
                pktLength = packetLen
        }
end)

local bufs = mem:bufArray(128)
local counter = 0
local c = 0

local txStats = stats:newDevTxCounter(queue, "plain")
```

```
        while mg.running() do
                -- fill packets and set their size
                bufs:alloc(packetLen)
                for i, buf in ipairs(bufs) do
                        local pkt = buf:getTcpPacket(ipv4)

                        --increment IP
                        if ipv4 then
                                pkt.ip4.src:set(minIP)
                                pkt.ip4.src:add(counter)
                        else
                                pkt.ip6.src:set(minIP)
                                pkt.ip6.src:add(counter)
                        end
                        counter = incAndWrap(counter, numIPs)

                        -- dump first 3 packets
                        if c < 3 then
                                buf:dump()
                                c = c + 1
                        end
                end
                --offload checksums to NIC
                bufs:offloadTcpChecksums(ipv4)

                queue:send(bufs)
                txStats:update()
        end
        txStats:finalize()
end
```

B.3 scripts used during the tests

Script used during the real world pktgen test:
It starts of sending 64 byte frames, where after 400 byte frames are send to the destination.

```
set ip src 0 1.2.3.4/24
set ip dst 0 4.3.2.1
set mac 0 ff:ff:ff:ff:ff:ff
proto tcp 0
set 0 size 64
sleep 30
start 0
sleep 90
stop 0
set 0 size 400
sleep 30
start 0
sleep 90
stop 0
```

Scripts to perform WARP tests during real world test:
Server responding to requests from client using WARP:

```
#Set client IP on interface
add tests l3_intf port 0 ip 1.2.3.4 mask 255.255.255.0
add tests l3_gw port 0 gw 1.2.3.1

add tests server tcp port 0 test-case-id 0 src 1.2.3.4 1.2.3.4 sport 80 180
set tests server http port 0 test-case-id 0 200-OK resp-size 64
```

```
add tests server tcp port 0 test-case-id 1 src 1.2.3.4 1.2.3.4 sport 80 180
set tests server http port 0 test-case-id 1 200-OK resp-size 256

add tests server tcp port 0 test-case-id 2 src 1.2.3.4 1.2.3.4 sport 80 180
set tests server http port 0 test-case-id 2 200-OK resp-size 512

add tests server tcp port 0 test-case-id 3 src 1.2.3.4 1.2.3.4 sport 80 180
set tests server http port 0 test-case-id 3 200-OK resp-size 1024

add tests server tcp port 0 test-case-id 4 src 1.2.3.4 1.2.3.4 sport 80 180
set tests server http port 0 test-case-id 4 200-OK resp-size 2048

start tests port 0
show tests ui
```

Client requesting a file from a server (can be run to an NGINX server or to a WARP server):

```
add tests l3_intf port 0 ip 4.3.2.1 mask 255.255.255.0
add tests l3_gw port 0 gw 4.3.2.254

add tests client tcp port 0 test-case-id 0 src 4.3.2.1 4.3.2.1 sport \
10000 50000 dest 1.2.3.4 1.2.3.4 dport 80 180
set tests client http port 0 test-case-id 0 GET "1.2.3.4" \
/files/500K.img req-size 64

set tests timeouts port 0 test-case-id 0 init 30
set tests timeouts port 0 test-case-id 0 uptime 1
set tests timeouts port 0 test-case-id 0 downtime 0
set tests criteria port 0 test-case-id 0 run-time 90
```

```
add tests client tcp port 0 test-case-id 0 src 4.3.2.1 4.3.2.1 sport \
10000 50000 dest 1.2.3.4 1.2.3.4 dport 80 180
set tests client http port 0 test-case-id 0 GET "1.2.3.4" \
/files/500K.img req-size 256

set tests timeouts port 0 test-case-id 1 init 30
set tests timeouts port 0 test-case-id 1 uptime 1
set tests timeouts port 0 test-case-id 1 downtime 0
set tests criteria port 0 test-case-id 1 run-time 90

add tests client tcp port 0 test-case-id 0 src 4.3.2.1 4.3.2.1 sport \
10000 50000 dest 1.2.3.4 1.2.3.4 dport 80 180
set tests client http port 0 test-case-id 0 GET "1.2.3.4" \
/files/500K.img req-size 512

set tests timeouts port 0 test-case-id 2 init 30
set tests timeouts port 0 test-case-id 2 uptime 1
set tests timeouts port 0 test-case-id 2 downtime 0
set tests criteria port 0 test-case-id 2 run-time 90

add tests client tcp port 0 test-case-id 0 src 4.3.2.1 4.3.2.1 sport \
10000 50000 dest 1.2.3.4 1.2.3.4 dport 80 180
set tests client http port 0 test-case-id 0 GET "1.2.3.4" \
/files/500K.img req-size 1024

set tests timeouts port 0 test-case-id 3 init 30
set tests timeouts port 0 test-case-id 3 uptime 1
set tests timeouts port 0 test-case-id 3 downtime 0
set tests criteria port 0 test-case-id 3 run-time 90
```

```
add tests client tcp port 0 test-case-id 0 src 4.3.2.1 4.3.2.1 sport \
10000 50000 dest 1.2.3.4 1.2.3.4 dport 80 180
set tests client http port 0 test-case-id 0 GET "1.2.3.4" \
/files/500K.img req-size 2048

set tests timeouts port 0 test-case-id 4 init 30
set tests timeouts port 0 test-case-id 4 uptime 1
set tests timeouts port 0 test-case-id 4 downtime 0
set tests criteria port 0 test-case-id 4 run-time 90

start tests port 0
show tests ui
```