



---

TECHNISCHE UNIVERSITÄT MÜNCHEN  
DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

**Performance Analysis of Middlebox  
Functionality**

Simon Sternsdorf







---

TECHNISCHE UNIVERSITÄT MÜNCHEN  
DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Performance Analysis of Middlebox Functionality  
Leistungsanalyse der Funktionen von Middleboxes

*Author*      Simon Sternsdorf  
*Supervisor*   Prof. Dr.-Ing. Georg Carle  
*Advisor*      Florian Wohlfart  
*Date*          September 12, 2017





---

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, September 12, 2017

---

Signature



## **Abstract**

Abstract eng





## **Zusammenfassung**

Zusammenfassung de



# Contents

1	Introduction	1
1.1	Motivation . . . . .	1
1.2	Goal of the thesis . . . . .	1
1.3	Outline . . . . .	2
2	Background	3
2.1	NAT . . . . .	3
2.2	NAT model . . . . .	5
2.3	Performance testing . . . . .	6
2.4	Data Plane Development Kit . . . . .	7
3	Methodology	9
3.1	General Idea . . . . .	9
3.1.1	Blackbox testing . . . . .	10
3.1.2	Software . . . . .	11
3.2	Test Methodology . . . . .	11
3.2.1	Experimental Setup . . . . .	12
3.2.2	MoonGen Traffic Generator . . . . .	12
3.2.3	mOS . . . . .	12
3.2.4	Open VSwitch . . . . .	12
4	Evaluation and Analysis of results	15
4.1	Firewall tests . . . . .	15
4.2	NAT tests . . . . .	15
5	Conclusion	17
5.1	Future Works . . . . .	17
	Bibliography	19



## List of Figures

2.1	A simple NAT with one public IPv4 address . . . . .	4
2.2	Intel® Data Plane Development Kit . . . . .	7
3.1	Our testsetup . . . . .	10
3.2	Distributed OpenVSwitch . . . . .	12
3.3	OpenVSwitch with and without DPDK . . . . .	14



## List of Tables





# Chapter 1

## Introduction

### 1.1 Motivation

Middleboxes are mediating devices used by both End-user Internet Service Providers and normal home users. The requirements ISPs have for Middleboxes are of course vastly different from the requirements of private users. Thus the implementations differs greatly as well. Middleboxes for home users do not have high performance requirements. They conduct mostly very simple tasks for a low amount of devices. This is changing of course, as more and more web-enabled devices are used in modern households. Still the required performance is low in contrast to at an ISP for example. Especially carrier grade network address translation is used to provide ipv4 connectivity for mobile phones, since IPv4 addresses are getting rare [1]. The Middleboxes used are mostly implemented in hardware, which has assets and drawbacks. Those drawbacks are significant. Middleboxes specifically produced for ISPs are expensive both in acquisition and maintainance, also they usually have to be replaced to introduce new features [2]. Also they are difficult to scale with higher or lower demand. All these problems are avoidable through network function virtualization. And the long-term plan is indeed to replace these hardware middleboxes with all-purpose hardware that is cheap and easily replaceable [3]. The networking functions would be implemented in software. 7 of the worlds largest telecoms network operators are in an standards group for virtualization of network functions. So the topic is already being discussed in ISPs [4]

### 1.2 Goal of the thesis

The goal of this thesis is to test different software Middlebox implementations. We will install different middlebox implementations in our testbed. Then we will test the packet processing capability, try to find bottlenecks for the performance when

processing packets. We will evaluate our results. Additionally we want to evaluate if software Middleboxes are competitive with hardware implemented Middleboxes and could replace them in the foreseeable future.

### 1.3 Outline

The thesis reads as follows. The second chapter introduces the theoretical concept of NAT and a NAT model which we used in our tests. Also it defines performance testing. Additionally the Data Plane Development Kit is introduced, DPDK. The third chapter informs the reader about the general idea behind our tests. Further it presents the software used for the tests. This includes the software running on the device under test, as well as the software used to run the tests. It explains the methodological approach used in this thesis. Here it explains the setup for the experiment. In chapter 4 are the collected results of the Firewall and NAT tests with a brief analysis of the result. Finally chapter 5 summarizes the outcome and gives possible future works of this thesis.

## Chapter 2

### Background

This chapter gives a overview over network address translation and the NAT model we will assume in this thesis. Also it will explain our approach to performance testing. Finally the chapter outlines the Data Plane Development Kit, developed by Intel [5].

#### 2.1 NAT

Network address translation NAT was first described 1993 and written into RFC 3022 [6] in 2001. It was proposed as an temporary solution for the shortage of IPv4 addresses. It should slow down the need for IPv4 addresses of private customers and businesses [7]. It does this by working as a connector between two different networks with different IPv4 address spaces. Mostly it translates between the address space of the Internet and a private network. Since NAT is used so broadly it is one of the most common middleboxes.

NAT in private households is in many instances implemented directly in the router. The home ususally only gets one IPv4 address from its ISP. The router then interconnects the home network to the Internet via an ISP. It translates the private IP addresses of the home network to enable them to share the single IPv4 address [7][Page 168]. In corporate networks it basically fulfills the same purpose. The main difference is that the border router manages multiple public IPv4 addresses and manages the correct translation between them and the private IPv4 addresses in the private network. Here we see the simple version with only one public IPv4 address.

A NAT middlebox manages the translation between the different IPv4 address spaces. To achieve this the middlebox has to save a mapping of the private IP addresses to the public ones. In the simplest imaginable case we have as many public IPv4 addresses as we have private ones. In that case the mapping is simply a bijection. When the NAT middlebox receives a packet from a new private IP  $S$  in the internal network it maps it



Figure 2.1: A simple NAT with one public IPv4 address [7][Page 168]

to a not used public IP from its address pool. This mapping is saved. To translate the packet the NAT middlebox has to :

1. Replace the original source IP from the packet with the mapped public IP
2. Completely recompute the IP header checksum, as not only the Time To Live header field changes, but also the source IP header field [8][Page 435]
3. Recompute the checksum in the TCP or UDP header if existent. The checksum of these protocols computes the checksum over the whole packet

When an answer from the Internet arrives the NAT middlebox has to do the same process only with replacing the destination IP address with the source address of the private host. This is done with the same mapping. Afterwards the packet is forwarded to the host in the private network.

In the realistic case where we have less public IP addresses than private ones the translation occurs over the IP and the port number. An NAT middlebox that uses this translation method maps the internal IP and the internally used port number of the TCP or UDP packet to a public IP from the IP pool available to the NAT middlebox and the first available port number [7][Page 169]. The entries in the mapping table are removed by the system after either the TCP connection is closed or the connection is idle for a longer time. Here the NAT middlebox functions similarly to a stateful firewall, which will become important later. When the NAT middlebox has to handle packets from the Internet, it looks up a mapping from its state table for the destination IP address and the destination port. If a matching mapping exists the packet is translated accordingly and forwarded to the matching internal host. If no such mapping exists the packet gets discarded, as there is no way to determine the correct internal host.

NAT has two main disadvantages: Opening TCP connections from the Internet to an internal network is very difficult. This means that for example FTP users behind NAT have problems. In active mode, the FTP client first establishes a control connection to the server. After that the client listens on a random port for the incoming data connection from the server. If the client sits behind a NAT this connection will not work [9]. The other disadvantage is that NAT breaks the end-to-end transparency of the IP layer

and the application layer. This problem occurs when the IP address is used in the application layer, as the NAT only replaces the IP in the IP header. This can be avoided with an Application Level Gateway installed at the NAT. However it is not feasible to install an an ALG for every application that relies on the IP in layer 7 [7][Page 169].

IPv6 would make NAT middleboxes obsolete, although some people argue NAT still has some use cases when only IPv6 is used. This is discussed in RFC 5902 [10]. With IPv6 enough addresses are available to give each device a unique global IP.

## 2.2 NAT model

This section is about the NAT model we will use in this thesis and why it is important for the performance tests we did. This model is based on the basic concept of how a NAT middlebox works. Roughly speaking we will split up NAT in different components. This way we can get an better understanding of which part of the NAT is actually responsible for how much of the time we need to forward a packet.

In the model the NAT middlebox performs 3 tasks excluding the sheer forwarding of the packet:

- Parse the packet. This means it has to check the IP header for the source and destination IP address and also the port numbers. Here the NAT compares to a stateless firewall, which has to do the same things.
- Holding a state. The NAT middlebox has to remember the mapping it created for a connection. This is similar to the state-holding of a stateful firewall. A stateful firewall tracks the state of every single TCP connection that goes through it and keeps a 5-tuple for every open connection. This 5-tuple persists of the source and destination IP, the source and destination port and the protocol used for the connection. This makes 5-tuple unique to its connection [7][Page 167].
- Rewriting either the source or destination IP address, and rewriting the source or destination port respectively. When the NAT middlebox has to translate the packet to forward it either to the Internet or the private network it has to replace the right IP address and the right port.

This simple model for a NAT middlebox will us help hopefully to determinate the different factors that influence the performance of the NAT middlebox.

## 2.3 Performance testing

Performance testing is the term that designates a testing technique to determine how a system reacts in term of steadiness and amenability under different amounts of load. With this testing we can learn about the quality of our system and its attributes [11]. There are different forms of performance testing.

- Load testing. Load testing is a very simple form of performance testing where the system is tested under specific work loads. All the parts of the system get monitored. With this one can predict how the system will react in the future under concrete work loads.
- Soak testing. Here the system is monitored under an continuous load. The load is set to the excepted average work load the system has to handle. With this testing the endurance of the system is determined. Critical parts like the memory of the system are monitored to find possible performance issues or errors under uninterrupted use.
- Spike testing. As the name suggests this testing revolves around a sudden spike of new users in the system. This is simulated by suddenly increasing the number of virtual users and monitoring the system. This test shows if a system can manage such a strong spike of new users and also if the system can handle the continuous workload that comes with a bigger user base.
- Stress testing. Stress testing is used to determine the maximum workload a system can handle. Also it shows how a system performs with a workload well above its limits. This is done by monitoring the system and increasing the workload until a part of the system fails or cannot complete the work correctly.

We will take a closer look at stress testing since it is the main form of performance testing we used here. In a stress test the condition of our system is progressively worsened until the performance is too low as that we could use the system or the system fails entirely. To exactly determine which element of the system failed we have to have indivisual stressors that are varied, and see which effect they have. Stress testing can often take a long time, but is really important to determine the degree of graceful degradation of a system. Graceful degradation describes the capability of a system to still be somewhat operational even under extreme circumstances. This is important for critical systems which always have to be operational [12].

## 2.4 Data Plane Development Kit

The Intel Data Plane Development Kit, Intel DPDK, is a group of programming libraries available in source code. These libraries enable faster basic data plane functions on Intel processors. These libraries are especially designed to optimize the packet processing capabilities on IA processors. These are processors from the Xeon series. These libraries allow for fast implementation of packet forwarding functions [13]. The most important DPDK elements can be seen in 2.2

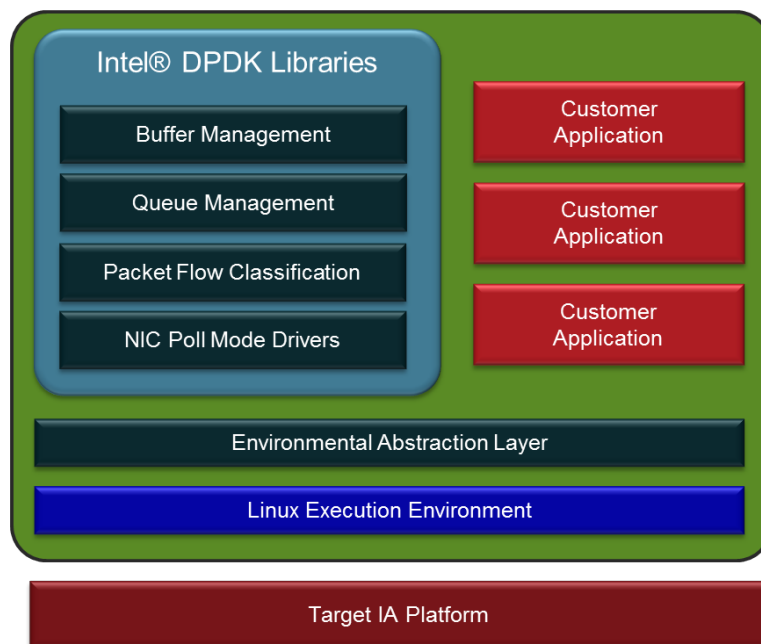


Figure 2.2: Intel® Data Plane Development Kit [13]

This shows the different DPDK libraries for buffer management, queue management, packet flow classification and poll mode drivers for different network interface cards. DPDK provides a model with a low overhead to enable a optimal data plane performance. It includes an Environment Abstraction Layer, which includes platform specific code. This simplifies application porting [13].

The buffer management includes the DPDK Memory Manager, which allocates pools of objects in the memory. This is realized through a ring to store free objects, which accelerates the whole process. Additionally the DPDK Buffer Manager reserves buffers with a fixed size. This additionally accelerates the system, as we no longer have to constantly allocate new buffers or free buffers which are no longer in use.

The packet flow classification includes the Intel DPDK Flow Classifier, which uses Streaming SIMD Extensions to speed up the process of placing packets in the right flow to get them processed. This additionally improves the throughput of the system [13].

The poll drivers included in DPDK work for Gigabit Ethernet and 10 Gigabit Ethernet and work with asynchronous signal mechanisms. This improves the performance for pipe-lining packets.

Since 2013 exists [dpdk.org](http://dpdk.org) [5], a Open Source Project founded by 6WIND [13]. Through this project source code, a documentation and example applications are available. This code is pretty much up-to date with the Intel DPDK releases and provides high processing-performance. The open-source project sees a lot of interest and is used already in several big projects and of course many smaller programs. One of them is for example MoonGen, which we will discuss in the Methodology chapter.

As a short conclusion in DPDK, we can see that DPDK could be vital to accelerate packet processing and handling in software middleboxes. It has great capabilities and sees a lot of ongoing development from Intel [13].



## Chapter 3

# Methodology

Before we can start testing the performance of a software middlebox of our choice, we have to decide on test parameters. This involves how we should conduct the measurements, what we want to measure, how the measurements should look like etc. For example we have to decide on a load generator, what kind of test environment, how we want to collect and visualize the data that we derive, and most importantly the middlebox software we want to survey.

In this chapter we talk about the research methodology and how to build it [14][Page 23].

### 3.1 General Idea

To test a software middlebox we defined a specific test setup which will help us derive the best results. The exact setup will be described in the following section.

The short abstract of it is: We will conduct blackbox tests of a middlebox software we installed on the DUT, the device under test. This DUT is connected to our testserver. This testserver runs a load generator to stress test our middlebox software.

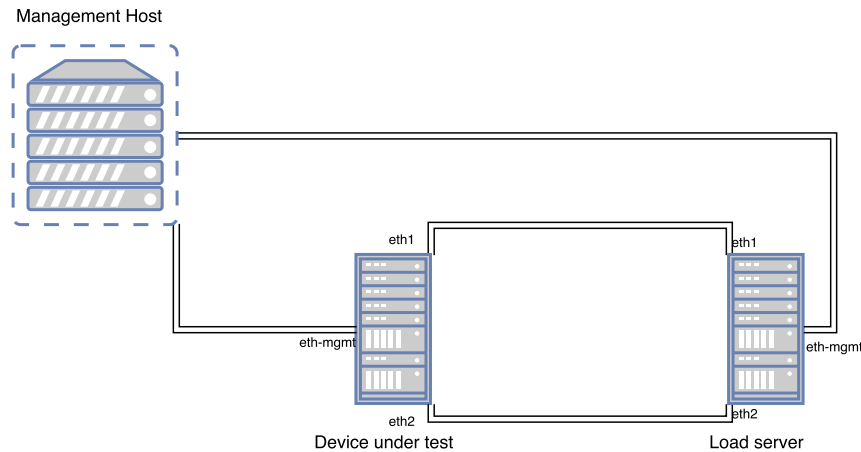


Figure 3.1: Testsetup

### 3.1.1 Blackbox testing

Blackbox testing is a method to test software in which the internal setup of the DUT is not known to the tester. This testing method is named blackbox testing as the interior of the DUT is not visible to the tester as the interior of a blackbox is not visible [15]. This testing method is mostly used to find:

- Performance errors
- Initialization and termination errors
- Database errors (External and internal)
- Interface errors

[15]

This method has mostly advantages for the tester:

- The tester not need to know how the software that they test is implemented. This means the tester must not be a software developer.
- The tester can be someone independent from the developer, removing possible bias from the tests and how the are conducted
- The tests are automatically from the viewpoint of a user

Some disadvantages of blackbox testing include:

- Only a limited amount of inputs can be tested, a full test of the software is not possible. This means that many possible cases for the software will not be tested.
- Test cases are difficult to design without the exact specifications of the software.
- Possibility to have redundant tests that the developer already performed.

[16]

### 3.1.2 Software

For the tests we need different software. For one we need the middlebox software. We want to test the different 'states' of NAT, so our software middlebox needs to be very flexible. As our main features we need a stateless firewall and the NAT functionality. The software needs to run on our hardware under Linux. Preferably the software should use DPDK or similar software to increase the performance of the packet forwarding. Otherwise if the software uses the Linux Kernel it should at least have the option to use DPDK in the future. To test the performance of the DUT we need a load generator. The load generator needs to generate enough load to saturate a 10 gigabit Ethernet connection, as our two servers are connected with 10 gigabit Ethernet. Also our load generator must be able to accurately measure the latency of our packets. Important for the tests are the different metrics we can define with the load generator. To fully test the middlebox we need to be able to change the metrics of our packet stream very freely. This includes the packet sizes, different ports, different IP addresses and different transmit rates.

## 3.2 Test Methodology

This section introduces the specific software we chose for the testsetup. We will discuss the advantages and disadvantages of the choices we made. Also we will explain the experimental setup we arranged to test the middlebox software. A great deal of the setup was already established by the testbed.

### 3.2.1 Experimental Setup

### 3.2.2 MoonGen Traffic Generator

### 3.2.3 mOS

#### 3.2.3.1 General Information

#### 3.2.3.2 Problems

### 3.2.4 Open VSwitch

Open VSwitch is the middlebox software we settled on using. In its core, OVS is a very useful virtual switch. It is licensed under the Apache open source license. Open VSwitch supports most of the standard management interfaces and protocols. It works in the most commonly used virtualization environments like Xen, KVM and Virtualbox as well in virtual management systems. It is mostly used to virtualize multiple servers at once [17].

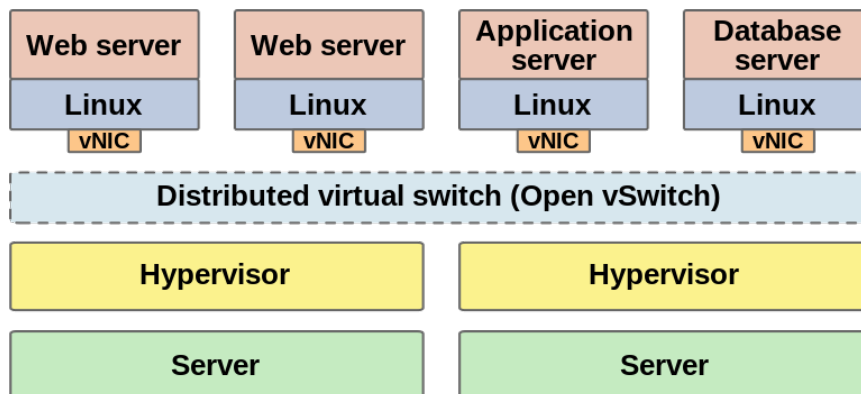


Figure 3.2: Distributed OpenVSwitch [18]

OpenVSwitch enables logical abstraction between different virtual machines and handles the offloading to the actual switching hardware. To control this OVS lies on top of the virtualization layer of the hypervisor and provides interfaces to the virtual machines. The virtual machines contain virtual interfaces. These enable communication between the VMs running on the same hypervisor or connected in the same virtual network and forward the traffic to the actual network interface of the machine. Here OVS acts as a bridge between the VMs and the physical interfaces. In short it interconnects the virtual interfaces with the physical interfaces [17].

### 3.2.4.1 Advantages of Open VSwitch

Open VSwitch is used as an alternative to the basic Linux bridge for modern hypervisors. OVS has some clear cut advantages over the classic Linux bridge mostly in terms of performance.

- Fast self-regulation when the network environment changes, realized through sFlow and Netflow.
- Very easy configuration and automation of network systems. Slow and fast network states can be migrated between instances.
- OVS can manipulate packets to enable a better administration of different flows. This means especially changing tags in packets or reordering them. This can speed up the network processes.

[17]

### 3.2.4.2 OVS with DPDK support

Open VSwitch forwards packets over the kernel data path, which means it uses the standard Linux kernel. This method divides the packet processing in a slow path and a fast path. The fast path matches incoming packets on a flowtable to determine the forwarding rule for the specific packet. The first packet from a new flow does not match anything in the flow table and is sent via the slow path in the user space daemon. After the user space has processed the packet, the user space daemon gets the information for the flow table from him and includes them in the current flow table. All subsequent packets of the flow can be handled via the fast path again.

By using this approach native OVS is a lot faster than normal user space packet forwarding. Still, the context switching required for new flows limits the realistically achievable throughput. This means native OVS is not feasible for use cases where a high throughput is mandatory [19].

With the use of DPDK in OVS, the slow path can make use of the Poll Mode Drivers. These enable the fast forwarding of packets from the user space to the interfaces, without the need for extra interrupt handling of the kernel. This means we essentially bypassing the whole kernel network stack. By incorporating DPDK with the Poll Mode Driver in Open VSwitch the slow path essentially becomes the fast path, as in both cases the path of the packet is the same.

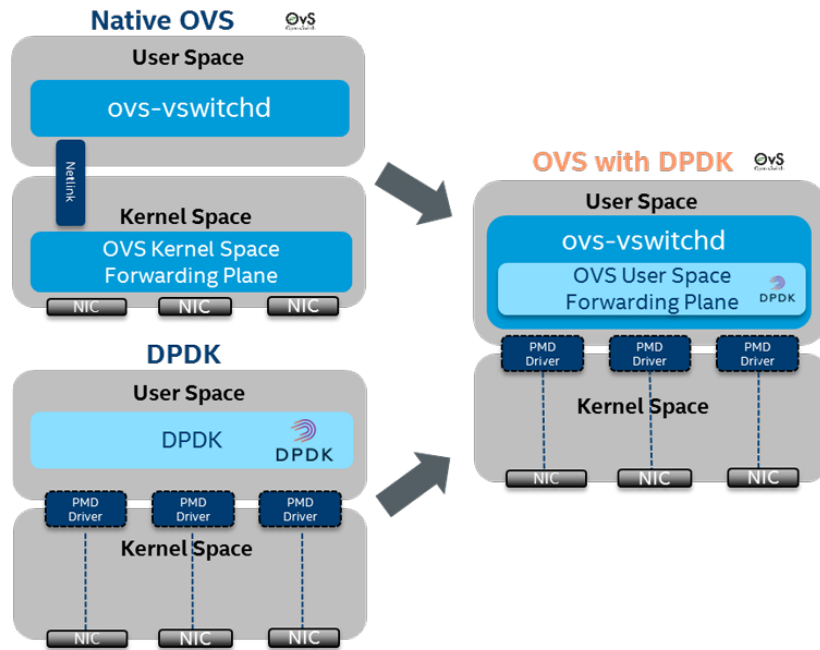


Figure 3.3: OpenVSwitch with and without DPDK [19]

This can be seen in 3.3. Sadly the NAT extension of OVS does not support DPDK just yet. The support is announced for version 2.7 which will probably be released in 2018 [20].

## Chapter 4

### Evaluation and Analysis of results

#### 4.1 Firewall tests

#### 4.2 NAT tests





## Chapter 5

## Conclusion

### 5.1 Future Works



## Bibliography

- [1] “Carrier grade network address translation,” <https://www.a10networks.com/resources/glossary/carrier-grade-network-address-translation>, visited: 20.08.2017 13:50.
- [2] “Network functions virtualisation,” [https://portal.etsi.org/nfv/nfv\\_white\\_paper.pdf](https://portal.etsi.org/nfv/nfv_white_paper.pdf), visited: 20.08.2017 13:40.
- [3] M. A. e. a. Joao Martins, “Clickos and the art of network function virtualization,” 2014.
- [4] “Leading operators create etsi standards group for network functions virtualization,” <http://www.etsi.org/news-events/news/644-2013-01-isg-nfv-created>, visited: 20.08.2017 15:50.
- [5] “Dpdk website,” <http://dpdk.org/>, visited: 21.08.2017 15:50.
- [6] “Rfc 3022,” <https://tools.ietf.org/html/rfc3022.html>, visited: 23.08.2017 15:50.
- [7] O. Bonaventure, *Computer Networking: Principles, Protocols, and Practice*. The Saylor Foundation, 2011.
- [8] A. S. Tanenbaum, *Computer networks*, 1996, no. 04; TK5105. 5, T35 1996.
- [9] “Ftp connection modes (active vs. passive),” [https://winscp.net/eng/docs/ftp\\_modes](https://winscp.net/eng/docs/ftp_modes), visited: 22.08.2017 15:50.
- [10] “Rfc5902,” <https://tools.ietf.org/html/rfc5902>, visited: 23.08.2017 15:50.
- [11] “Performance testing,” [https://www.tutorialspoint.com/software\\_testing\\_dictionary/performance\\_testing.htm](https://www.tutorialspoint.com/software_testing_dictionary/performance_testing.htm), visited: 24.08.2017 17:30.
- [12] M. Rouse, “Definition: Stress testing,” <http://searchsoftwarequality.techtarget.com/definition/stress-testing>, visited: 24.08.2017 15:50.
- [13] “Roving reporter: Dpdk goes open-source,” <https://embedded.communities.intel.com/community/en/software/blog/2013/05/16/roving-reporter-dpdk-goes-open-source>, visited: 26.08.2017 13:30.

- [14] O. Schmidt, “A programming language for protocol-independent packet processing: Analyzing and evaluating p4,” 2017.
- [15] “Black box testing fundamentals,” <http://softwaretestingfundamentals.com/black-box-testing/>, visited: 29.08.2017 14:30.
- [16] “Black box testing, its advantages and disadvantages,” <https://www.codeproject.com/Articles/5579/Black-Box-Testing-Its-Advantages-and-Disadvantages>, visited: 29.08.2017 13:30.
- [17] R. Pothuraju, “Measuring and modeling of open vswitch performance: Implementation in kvm environment,” 2016.
- [18] J. M., “Virtual networking in linux,” 2010.
- [19] R. G., “Open vswitch\* with dpdk overview,” <https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview>, 2016.
- [20] “Roadmap for dpdk usespace nat,” <https://mail.openvswitch.org/pipermail/ovs-discuss/2016-September/042581.html>.