BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

AUTOMATED TESTING OF OS LINUX PROTOCOL STACK BEHAVIOUR DURING CONGESTION

AUTOMATIZOVANÉ TESTOVÁNÍ CHOVÁNÍ PROTOKOLOVÉHO ZÁSOBNÍKU

OS LINUX VE STAVU ZAHLCENÍ SÍTĚ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

ADRIÁN TOMAŠOV

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. ONDREJ LICHTNER

BRNO 2018

Bachelor's Thesis Specification/21223/2017/xtomas32

Brno University of Technology - Faculty of Information Technology

Department of Information Systems Academic year 2017/2018

Bachelor's Thesis Specification

For:	**Tomašov Adrián**
Branch of study:	Information Technology
Title:	**Automated Testing of OS Linux Protocol Stack Behaviour During Congestion**
Category:	Networking

Instructions for project work:

1. Study and describe network congestion and it's effects on the functionality of the TCP/IP stack.
2. Design and create a network suitable for testing and monitoring network overload.
3. Observe the behavior of the created network during different levels of congestion.
4. Design and implement a testsuite for automated testing of the network stack supporting different network congestion scenarios.
5. Evaluate the data from automated testing.

Basic references:

- Beaker Project documentation [online]. Available on: https://beaker-project.org/
- Bradner, S. and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices", RFC 2544, March 1999.
- Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

Requirements for the first semester:
Items 1 and 2.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Lichtner Ondrej, Ing.**, DIFS FIT BUT
Beginning of work:	November 1, 2017
Date of delivery:	May 16, 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

[illegible]

Dušan Kolář
Associate Professor and Head of Department

Abstract

This thesis is focused on observing and simulating network congestion in laboratory conditions, and on automated testing of the protocol stack in the Linux operating system during network congestion. We perform a set of experiments to find the impact of network congestion on the protocol stack. The simulation and emulation method of this network using physical device will be described. The outcome of this thesis are various configurations of devices and emulators for network congestion together with measurements and evaluation of results. These configurations will be used for automated testing of the kernel of the Linux operating system to catch development errors, network protocol stack errors and card driver error earlier.

Abstrakt

Táto práca sa zaoberá štúdiom a simuláciou zahltenej siete v laboratorných podmienkach a následne automatizovaným testovaním protokolového zásobníka v operačnom systéme Linux na tejto sieti. Na základe sady experimentov zistíme, aký dopad na správanie protokolového zasobníku má zahltenie siete. Následne bude popísaný spôsob simulácie a emulácie takejto siete fyzickým zariadením. Výstupom tejto práce budú rôzne konfigurácie strojov a emulátorov pre zahltenie siete, a k ním priložená sada meraní s vyhodnotením výsledkov. Tieto konfigurácie budú použité v automatizovanom testovaní jadra operačného systému Linux, aby sa chyby vo vývoji a pri implementácii sieťových protokolov a ovládačov pre sieťové karty našli rýchlejšie.

Keywords

Network congestion, CI testing, continous integration testing, OSI model, TCP, UDP, iPerf3, Beaker, Linuxové jadro, protokolový zásobník.

Kľúčové slová

Zahltenie siete, CI testovanie, testovanie priebežným integrovaním, OSI model, TCP, UDP, iPerf3, Beaker, Linux kernel, protocol stack.

Reference

TOMAŠOV, Adrián. *Automated Testing of OS Linux Protocol Stack Behaviour During Congestion.* Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondrej Lichtner

Rozšírený abstrakt

V tejto práci sa zaoberáme dopadom zahltenia siete na výkonnosť protokolového zásobníka v jadre Linuxu. Zahltenie siete nastáva v momente, keď sieťové zariadenia alebo služby vyžadujú väčšiu šírku pásma ako poskytnutá sieť dokáže preniesť. Tento jav je očakávaný hlavne z dôvodu, že poskytovatelia sieťových služieb poskytujú väčšiu šírku pásma ako ich sieť naozaj dokáže preniesť. Poskytovatelia predpokladajú, že koncový užívatelia nebudú používať celú poskytnutú šírku pásma po celý čas, ale len krátkych opakovaných intervaloch.

Podobnú situáciu môžeme vidieť aj v dátových centrách, kde niekoľko serverov je zapojených do jedného prístupového prepínača. Tento prepínač má väčšinou len niekoľko spojení s distribučným prepínačom, cez ktorý zariadenia komunikujú s dátovými servermi alebo inými zariadeniami na internete. Keď pripojené servery požadujú väčšiu priepustnosť ako je dostupná nastane zahltenie.

Zákazníci spoločnosti Red Hat používajú operačný systém Red Hat Enterprise Linux väčšinou v dátových centrách a cloudoch, kde zahltenie nastáva často. Chcú si byť istí, že výkonnosť tohto produktu je dostačujúca aj v týchto podmienkach a hlavne, že sa výkon v rámci vývoja nebude zhoršovať.

V tejto práci sme rozdelili naše pozorovania na dve časti. V prvej časti sme pozorovali dopad zahltenia na fyzickú sieť (sieťové karty, prepínače). Chyby v sieti, ktoré sme hľadali sú: strata dát, vysoké oneskorenie, zmena poradia odoslaných dát a poškodenie dát počas prenosu. Pre účely pozorovania sme vytvorili experimentálnu sieť pozostávajúcu z dvoch serverov a jedného prepínača. Prvý server bol pripojený pomocou dvoch liniek v LACP skupine a druhý pomocou šiestich liniek, aby bolo možné vytvoriť zahltenie siete. Naše experimenty sme vykonávali pomocou niekoľkých programov, ktoré generovali sieťovú prevádzku a potom sme skontrolovali záznamy v sieťových prvkoch. Zistili sme, že v zahltenej sieti sa objavuje strata dát a oneskorenie. Na tieto chyby sa zameriame v ďalších pokusoch. V druhej časti pozorovaní sme sa zamerali na správanie protokolového zásobníka v jadre Linuxu počas zahltenia siete. V experimentoch sme použili TCP a UDP transportný protokol, pretože sa najčastejšie používajú pri komunikácií. Všetky pokusy sme robili dva krát, pre každý protokol zvlášť. Pomocou programu iPerf3 sme spúšťali niekoľko dátových tokov, ktoré posielali dáta rôznou rýchlosťou, aby sme vyskúšali rôzne úrovne zahltenia. Z týchto experimentov sme zaznamenávali rýchlosť a počet odoslaných a prijatých dát. Pomocou programu ping sme merali oneskorenie v sieti.

Z výsledkov meraní UDP protokolu je jasne vidieť, že s rastúcim pomerom vyžadovanej a dostupnej šírky pásma rastie aj počet stratených dát a oneskorenie v sieti. Percento stratených dát sa asymptoticky blíži k 100%. Oneskorenie rastie až do hodnoty, ktorú môžeme vypočítať ako veľkosť fronty portu / šírka pásme.

Vo výsledkoch meraní TCP protokolu je oveľa menej stratených dát, pretože TCP má funkcie, ktoré kontrolujú prípadnú stratu v toku dát. Pri strate znížia rýchlosť prenášania dát, aby predišli ďalšej strate. Priemerné oneskorenie počas testov má nižšiu hodnotu ako pri UDP testoch. Je to spôsobené práve znižovaním rýchlosti prenosu dát. Detailná analýza oneskorenia jedného TCP toku ukázala, že oneskorenie sa počas testu mení.

Na základe nameraných výsledkov sme vytvorili testovací scenár, ktorý simuluje zahltenie siete. Pre simuláciu takejto siete sme použili Attero-X, ktoré vie obmedziť dátovú priepustnosť a pridať oneskorenie do siete. Tento scenár pozostáva z viacerých skúmaných oneskorení a priepustností. Scenár v prvom kroku nastaví simulátor siete a potom spustí testovacie prípady pre jednotlivé transportné protokoly. Tento scenár budeme spúšťať na páre strojov s rovnakou konfiguráciou komponentov. Sieťové karty používané pre tento

scenár sú od spoločnosti Intel s ovládačom ixgbe, pretože ma dlhodobo stabilné výsledky výkonosti.

Pre vyhodnotenie užitočnosti a dôveryhodnosti tohto testovacieho scenára sme spustili testy nad štyrmi distribúciami operačného systému RHEL. Tieto distribúcie sa líšili vo verziách nainštalovaných balíkov, takže aj verziou jadra. Výsledky testov sme porovnali našim tímovým nástrojom, ktorý vygeneroval web stránku s výsledkami v grafickej podobe. Výsledky TCP testu vyzerajú stabilne a očakávane. V jednom teste sme dokonca odhalili aj možnú regresiu. Tento scenár testuje protokolový zásobník počas zahltenia dokonca aj v sieťach s veľmi vysokým oneskorením s vysokou priepustnosťou.

Výsledky UDP ukazujú zlý návrh tohto testovacieho prípadu. Pri návrhu sme použili rovnaký konfiguračný objekt ako pri TCP, čo sa ale neosvedčilo. Priepustnosť s nízkymi veľkosťami správ je malá pretože procesor nestíha spracovávať toľko dát. Veľké správy sa musia rozdeliť na menšie časti a ak sa aspoň jedna časť stratí, celá správa je nepoužiteľná, a to ma za následok malú priepustnosť.

Výstupom tejto práce je rozšírenie testovacieho programu o nový testovací scenár StaticCongestion a testovací prípad iPerf3TCPTest. Tento scenár bude zaradený do testovania postupným začleňovaním a bude spúšťaný niekoľko krát denne, aby odhalil chyby vo vývoji v čo najmenšom čase.

Automated Testing of OS Linux Protocol Stack Behaviour During Congestion

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing.Ondrej Lichtner. The supplementary information was provided by Ing.Adam Okuliar All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Adrián Tomašov
May 15, 2018

Acknowledgements

I would like to thank to my supervisor Ing.Ondrej Lichtner for advising me during creation of this work. I also would like to thank to Ing.Adam Okuliar, whose gives me necessary information important to this thesis. I would like to include special notes of thanks to Martina and Jozef, who keep me motivated and optimistic during most parts of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis we focus on network performance during network congestion. In Red Hat Kernel Performance Quality Qngineering team there is currently no testcase designed for automated testing of kernel protocol stack during network congestion. The goal of this work is to create test scenarios and use them to increase the coverage of tested scenarios and by that to increase the quality of the Linux kernel network stack.

In chapter 2, we discuss network basics which are important to understand properly this thesis (e.g. OSI model, TCP). We also introduce the most important problematic which is network congestion, described in local and remote networks with practical examples when such situations can occur. Finally we explain the most common source of network congestion.

Chapter 3 brings important information about the testing methods related to our research. Although we can identify several possible errors during network congestion, this thesis focuses on the four most common errors. In every section we include the theory behind explaining the possibility of occurrence of specific error in first four layers of OSI model. Higher layers are not a part of the kernel, but the user space, and are beyond the focus of this thesis. The result of this chapter is data used for simulating of network congestion. The various methods of network congestion simulation will be discussed further in chapter 5.

In chapter 4, we observe behavior of the Linux protocol stack during congestion. The experiments focus on two communication protocols: user datagram protocol, transmission control protocol. The observed errors during congestion are data loss and network delay. In TCP protocol behavior test we are interested in congestion window variable in network with changing over-subscription ratio.

In chapter 5, we discuss test scenario requirements based on results of behavior tests from chapter 4. We discuss several implementation possibilities of these requirements in new test scenario. The most important part is simulation method of network during congestion.

Chapter 6 brings an insight to automated testing using continuous integration development method. We discuss implementation details of our testing Beaker task, which prepares persistent configuration for servers and run performance test of those servers after reboot.

In chapter 7, we discuss results of new test scenarios and compare several kernel versions. Finally, we evaluate the asset of these scenarios and consider their inclusion into CI testing process.

Chapter 2

Network model

This chapter provides basic knowledge important for understanding the main concepts of this thesis. In section 2.1, there is a quick overview of the OSI model application onto network focused on Ethernet networks. This description concentrates mostly on application of this model, not the ISO/OSI standard itself.

The next topics explained in this chapter are network congestion and congestion control algorithms, which create core of this thesis.

2.1 OSI model application in network

Open systems Interconnection model is a conceptual model that divides the network communication into seven specific layers for better understanding of interactions that happen within the network. Abstracting and standardizing different responsibilities to separate independent layers is the biggest asset of this model. This level of abstraction is highly used for different media types in communication (e.g. Smartphone that communicates with web server uses several different media types: radio, cooper cable, and various encapsulations: 802.11 WiFi[1], PPP[2], 802.3 Ethernet). Each layer serves a different purpose, which will be explained later in this section. We will focus only on Ethernet networks, which are important for this work. Layer names and protocol examples are shown in table 2.1. OSI model was developed by International Standards Organization (ISO) [9].

No.	Layer	Protocol example	Devices	Data unit name
7	Application	DNS, SMTP, POP3, HTTP	-	Data
6	Presentation	SSL, TLS	-	Data
5	Session	NetBIOS, PPTP	-	Data
4	Transport	TCP, UDP, RTP	Statefull firewall	Segment
3	Network	IPv4, IPV6, IPX, ICMP	L3 switch, router	Packet
2	Data link	ARP, ATM, PPP,	L2 switch, NIC	Frame
1	Physical	Bluetooth, Ethernet, ISDN	Hub, repeater	Bits

Table 2.1: Visual representation of layers in OSI model with protocol examples.

[1]802.11 is Standard for local wireless communication.
[2]PPP - point to point protocol

Data units have specific name on different layers. Various name of data is used to clearly specify data structure according to protocols in higher layers. Layers are able to add headers to store important information for current layer. The example of data structure name is shown in figure 2.1.

2.1.1 Physical layer

The first layer is responsible for converting data from upper layers into a physical signal which can be carried by transfer media (ethernet cable, optical cable, wireless signal). The signal represents the data encoded into bits. Speed and reliability of transmission is determined by physical properties of the transmission media.

2.1.2 Data link layer

The second layer is responsible for communication within local networks and physical addressing. It is responsible for detection of errors which might happen in communication in Layer 1. CRC algorithm is most commonly used to check errors in data transfer. In Ethernet networks an address called `MAC address` is used. This address has to be unique in local network and it is usually a part of NICs[3] firmware. If two stations want to communicate in local network [4] they need to know the `MAC address` of the opposite station. If two stations want to communicate over IP in local network and they do not know `MAC address` of opposite site, they need to discover it. For that reason `ARP` protocol was created. (There is ICMPv6 protocol which is alternative to IPv4 in IPv6 network address space.) Its purpose is to translate Network layer address (IPv4) to `MAC address`. From the perspective of end-stations this layer of OSI model is the last layer implemented in the hardware (NIC). Higher layers are implemented in software usually in kernel protocol stack.

This layer adds Ethernet header where the destination and source MAC addresses are specified together with others specific flags. It includes FCS[5] field, which is important for data integrity check. Data together with upper layer headers are encapsulated into payload/data field.

2.1.3 Network layer

The third layer brings logical addressing into OSI model. The biggest responsibility of this layer is end to end addressing in local and remote networks. Each end station needs to have IPv4 or IPv6 address for communication with other end-stations. Routers, which interconnect networks, operate on this layer. While the packet is traveling through the network, this layer together with the upper ones stay almost unchanged. This behavior is necessary in communication between remote networks. Routing protocols operate in this layer too. Their purpose is to find the best path for data, which travels between remote networks. Routers direct traffic according to destination IP address.

The network layer adds the internet protocol header with following important fields : source IP address, destination IP address, header checksum and data. The header checksum checks an integrity of header itself to avoid further sending of invalid data. The header of upper layer together with payload are encapsulated into the data field.

[3]Network interface card

[4]Local network is meant to be a single broadcast domain.

[5]FCS - file checksum

2.1.4 Transport layer

The fourth layer identifies the service or process, to which data is being be delivered. This is done by using port numbers, which specify the unique process or service on a host. Data lost during transmission can be sent again to guarantee data delivery. Communication reliability is provided by several protocols. The most common used protocol for this type of communication is TCP, which will be described further in section 2.3.1. If reliability of data delivery is not required, UDP protocol, which does not resend the lost data, can be used. This is the last layer, which is usually implemented in the kernel of the operating system.

This is the first layer that adds header over the transferred data. The headers of both protocols contains fields with source and destination port number and several specific flags.

2.1.5 Upper layers in OSI model

The fifth layer together with sixth and seventh is usually implemented in the application. Compare to the OSI model, the `TCP/IP model` compress these layers into one `Application layer`. The fifth layer (Session layer) is responsible for creating, maintaining and closing connections between applications.

The sixth layer (Presentation layer) is responsible for syntax processing of message data [22]. The original data can be converted into a different format, which is not in conflict with other layers or network format. Data can be compressed or encrypted in this layer as well.

The seventh layer (Application layer) creates the interface for processes and services running in the system. Usually there is a standardized protocol, which can transfer data between end-processes. This layer also checks the syntax of communication's protocol. The application layer is responsible for identifying participants in the communication.

2.2 Network congestion

Routers and switches use `Best-effort delivery` method to transfer the data. This method does not prioritize packets. Intermediary devices do not guarantee packet delivery, but try to deliver packets as quickly as possible [20]. Routers direct packets according to entries in the routing table. In case of large networks (e.g. ISPs[6]) routing tables can be very large. Most of the time they do not load-balance[7] the traffic between more routes, so data traveling from user to server uses the same route every time.

Congestion problems occur when too many users want to use the same path in the network. Users have enormous requirements on network throughput while accessing internet services (e.g. web, instant messaging, video streaming). When the requirement throughput is higher than available throughput of some device, congestion occur on that device. The incoming traffic can not be sent further, because the outgoing interface queue is already full. That means some part of incoming traffic is dropped by the router. As a consequence the user's application or the server has to resend the data, which further complicates the situation.

[6]Internet service providers

[7]Distribute the traffic between more routes

2.2.1 Oversubscription

Congestion usually happens in internet networks, because ISPs provide to customers more bandwidth than they are able to guarantee in their infrastructure [27]. This situation applies to networks in data centers as well. The speed of inter switch link is usually much slower than the bandwidth required by devices connected to switch.

Oversubscription exists due to costs savings [12]. Network and service architects assume the service or the network are not fully utilized all the time. This is based on measurements of the average utilization of a specific service. By using the oversubsription method they are able to provide services for more customers with the same infrastructure, but with a higher average utilization. Big companies (e.g. Cisco, IBM) publish white papers[8] which describe „The best Oversubscription practices“ for different type of networks. These describe the design patterns of infrastructures for oversubscription with recommended oversubscription ratios [12].

Let's imagine a network with twenty five customers, each of them has provided speed of internet connection 10Gbps. That is 250Gbps of required bandwidth, but the backbone of ISP network is only 160Gbps. ISPs presume that all customers do not use the full bandwidth capacity at once. However, sometimes the customer requirements on bandwidth are higher than the network provider can provide. This usually happens for a short period of time. These relatively small data burst are buffered and resent with delay or completely dropped. The arithmetic mean of traffic speed passing through the ISP network should be lower than the theoretical capacity of the network. The figure 2.1 shows how the transmission speed is changed while crossing the device that buffers the over limit data.

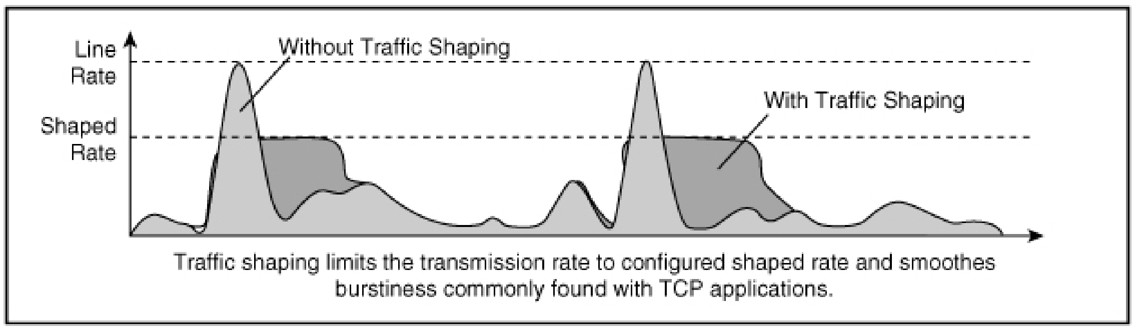

Figure 2.1: The graph shows ingoing and outgoing throughput of intermediary device that use data buffering method [26]

.

Network congestion can be present in data center network as well. Queuing principles described above are now applied to network switches. Switches receive the frame in the same way as the router, which means into incoming interface queue. In contrast to router, switch decides where to forward the frame based on information from the second layer, instead of the third layer. After processing, the frame is queued into an outgoing interface queue, where network congestion can occur. This mostly happens on an interface or interface group, which points to the neighbor switch (interface group connection example is show in figure 2.2). Let's imagine that the switch has 48 ports of speed 1Gbps. Four ports are grouped together to provide connection to the other switch. Bandwidth of this interface

[8]White paper is technical documentation.

bundle is 4 Gbps (8 Gbps in both directions). There are 44 devices connected to those remaining ports in the switch. This topology is shown in figure 2.2. These devices want to communicate with internal servers, placed behind the local access switch. Devices can not use their full speed of connection at once, because the connection to other switch creates a bottleneck for the data traffic.

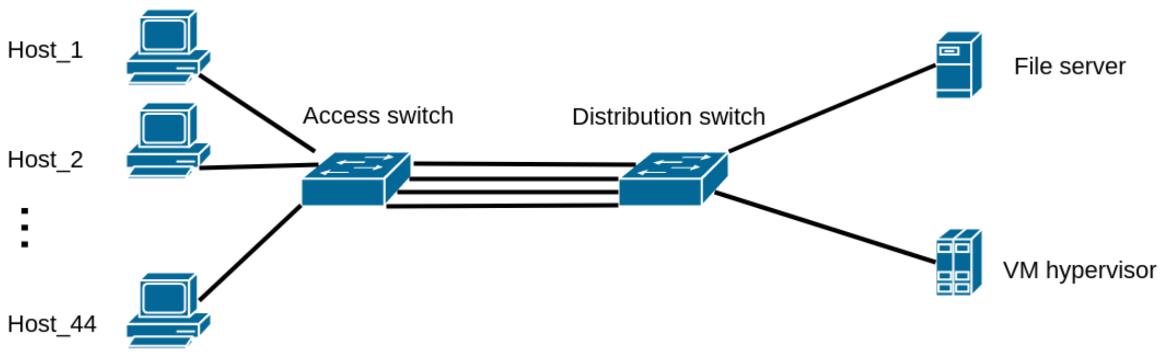

Figure 2.2: Forty four hosts are connected to access switch and only four switch ports are designated to be uplink ports.

2.3 Congestion control algorithms

Several methods were developed to avoid the network congestion. The most useful method is `TCP` protocol. It is designed for reliable data delivery even in case the network nodes are communicating during network congestion. TCP implements congestion control mechanisms and is able to maintain throughput of the traffic between participants. This protocol allows to send as much data as the network is able to transmit, which is equal to the bottleneck bandwidth [9]. More detailed description is in the section 2.3.1.

Compared to TCP, UDP does not maintain the sending speed and sends as much application data as the NIC of the device allows. If all hosts in the exemplary network from figure 2.2 start sending files via for example TFTP[10] protocol, the access switch will drop many UDP datagrams, because of an exhausted capacity of the switch connection. Only small percentage of UDP datagrams will successfully travel through the connection between switches. From this reason the network needs efficient methods or protocols, which are able to handle the transmission speed.

2.3.1 TCP - Transmission control protocol

Transmission control protocol can be found on the fourth layer of OSI model. It creates an interface between the userspace application and the internet protocol. TCP is a connection-oriented protocol, which means both participants have to create and accept the

[9]Bottleneck bandwidth between points A and B is the slowest bandwidth of single link in path between A and B.

[10]TFTP protocol is trivial file transfer protocol. It use UDP for transferring files.

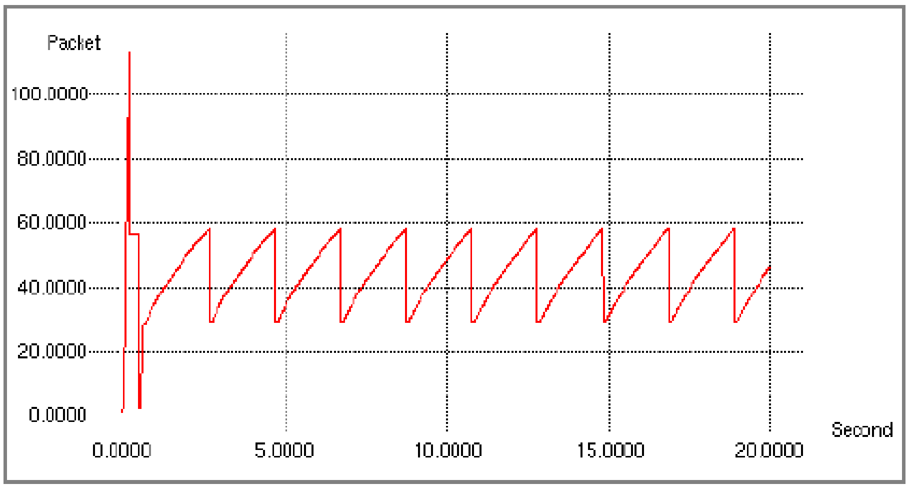

Figure 2.3: The example of sliding window changes by congestion control algorithm [6]
.

new connection. Both participants also have to agree on closing the connection. If only one of the hosts requests connection termination, the connection becomes half closed until the second host stop sending data and request connection termination as well. The reason for this, is to avoid situation, where one host is sending data and the second one is discarding it without informing the sender.

Data travels between processes in the same or different host in multinetwork enviroment [23].

Within the scope of this thesis the main characteristics are reliable data transfer and flow control[11] over non-reliable connectionless internet protocol. TCP implements reliable data transmission using special `ACK` flag and `ACK number`. These attributes check which data arrives to the receiver or which data is lost in the network. TCP will keep trying to send the lost data, until it arrives to the receiving process or the connections time out. TCP also has a feature responsible for black box analysis, verifying the data integrity by calculating and comparing a checksum.

The communication stream of this protocol has an attribute called `window size`. It represents the amount of data possible to send, while TCP is waiting for the acknowledgement. If an acknowledgement arrives in time, TCP sends more data of size equal to window size and waits for their acknowledgement. If an acknowledgement does not arrive, the lost data is resent. Window size value can be changed while processes are communicating through TCP connection. The throughput of a TCP connection is proportional to the windows size value.This feature is called sliding window, used for flow control. The sender and receiver are able to change the transmission speed by changing the window size. Communication speed increases with increasing window size and vice versa. Thanks to the sliding window, receiver is allowed to lower the transmission speed, if data comes too quickly for processing.

The TCP protocol uses several congestion control algorithms [24]. These algorithms are responsible for avoiding network congestion by changing the window size value. They are also able to raise the transmission speed as high as the network is able to transmit, if needed. The algorithms increase the window size, while the data arrives to the receiver without a

[11]Flow control is process, which allow to control transmission speed.

problem. When data is lost the congestion control algorithms assume that the network is overloaded and lower the window size. When this happens, the transmission speed decreased. Network development community designed and implemented many congestion control algorithms. We describe three most commonly used algorithm in sections below. The example of `New Reno` congestion control algorithm in graph is visible in figure 2.3.

Tahoe

This congestion avoidance algorithm consists of three phases: slow-start phase, congestion avoidance phase and fast retransmit. As mentioned above, this algorithm maintains sending speed by changing *congestion window (cwnd)* value. It represents amount of data, which can be sent and then sender waits for acknowledgment.

The first phase is slow-start, which is used directly after connection establishment or when packet loss occurs. In this phase *cwnd* value rises exponentially. In the beginning of this phase *cwnd* value is set to *Maximum segment size (MSS)* and initial value of ssthresh can be arbitrarily high (e.q. advertised window) [24].

$$cwnd_0 = MSS \tag{2.1}$$

Then with every received acknowledgment of sent packet, the *cwnd* is increased by *MSS*.

$$cwnd_{n+1} = cwnd_n + MSS \tag{2.2}$$

This phase stops when *cwnd* is larger than *Slow-start threshold (ssthresh)* and congestion avoidance takes place.

The congestion avoidance phase uses *cwnd* value from previous phase. In this phase transmission speed gets closer to speed, which may cause the congestion. Therefore, *cwnd* value rises lineally and is incremented by $MSS*MSS/cwnd$ every time the acknowledgment is received [24].

$$cwnd_{n+1} = cwnd_n + \frac{MSS * MSS}{cwnd_n} \tag{2.3}$$

Tahoe detects congestion by receiving three duplicate acknowledgments of lost packet or expiring timer, which waits for acknowledgment. This situation triggers fast retransmit phase, which immediately sends all data in sliding window, sets *sstresh* to value *cwnd/2* and *cwnd* to value *MSS*.

$$sstresh = \frac{cwnd}{2}, \quad cwnd = MSS \tag{2.4}$$

After fast retransmit Tahoe algorithm continues with slow-start phase again.

Reno

This algorithm extends behavior of Tahoe algorithm. It adds feature called *fast recovery*. When three duplicate acknowledgments are received, the Reno shifts from congestion avoidance phase to fast recovery. This phase sets *sstresh* to *cwnd/2* and *cwnd* sets to *sstresh* + three times *MSS* [24]. Then Reno continues in congestion avoidance phase. This feature rises overall throughput of TCP connection, because it has higher congestion window after packet loss, therefore Reno sends more data after congestion.

$$sstresh = \frac{cwnd}{2}, \quad cwnd = sstresh + 3 * MSS \tag{2.5}$$

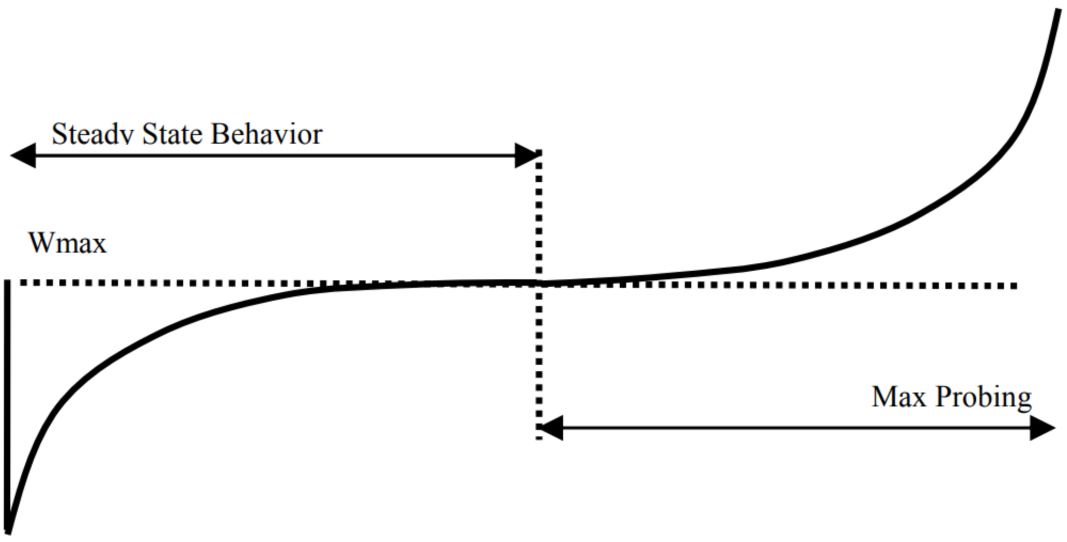

Figure 2.4: Graph of CUBIC function from equation 2.6, which calculates W_{cubic} used for cwnd increment.

CUBIC

CUBIC algorithm is optimally designed for *long fat networks*, which have usually high link speeds and high round trip times [21]. CUBIC inherits behavior from BIC algorithm [21], but it uses simplified function, that calculates congestion window. The design process keeps these principles [25] :

1. Function with a concave and a convex part is used to achieve stability and better network utilization.
2. In the networks with short round trip times, CUBIC emulates behavior of standard TCP protocols.
3. Share bandwidth linearly among TCP connections with different round trip time.
4. To keep balance between the scalability and speed of convergence, CUBIC sets the multiplicative decrease factor appropriately.

In the beginning of TCP communication using CUBIC congestion avoidance algorithm, slow start algorithm is used. When slow start reaches congestion, *cwnd* values is stored into W_{max}, which indicates the last congestion *cwnd*. TCP CUBIC uses following cubic function for calculating of increment to *cwnd*,

$$W_{cubic}(t) = C * (t - K)^3 + W_{max} \tag{2.6}$$

where C is a scaling factor, t is elapsed time from the last congestion event, W_{max} is cwnd value right before the last congestion and K is time period that $W_{cubic}(t)$ function takes to achieve the last *cwnd* (stored in W_{max}) [25]. This makes CUBIC independent on the actual RTT of network. This function is shown in figure 2.4. The K value is evaluated with this equation:

$$K = \sqrt[3]{W_{max} * \frac{1 - \beta}{C}} \tag{2.7}$$

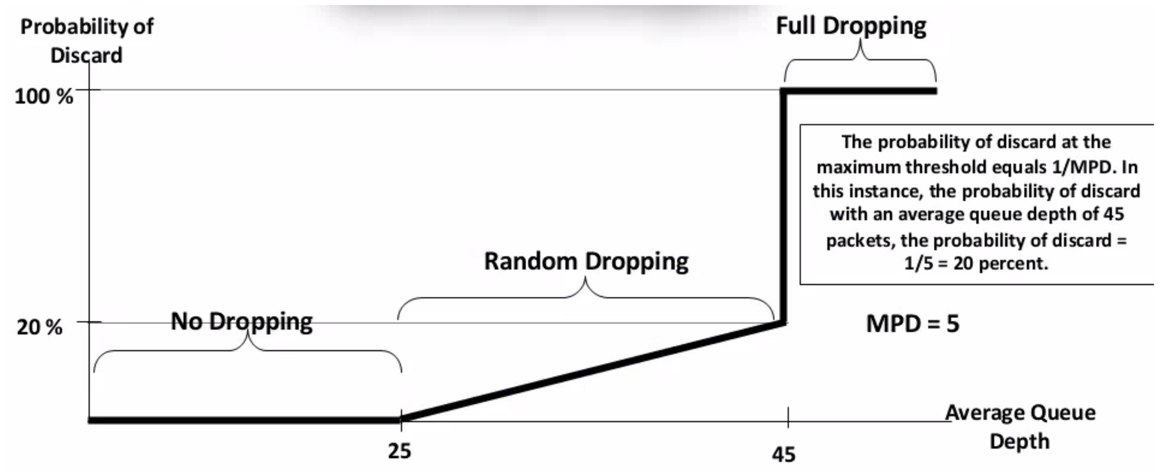

Figure 2.5: Graph of probability of packet drop according to queue length [7]

.

where β is the CUBIC multiplicative decrease factor, used to set new *cwnd*, when congestion occurs.

$$cwnd = W_{max} * \beta \tag{2.8}$$

The function shown in equation 2.6 is divided by W_{max} into two parts: concave and convex part. In the concave part, cubic tries to slowly reach the point of last congestion. If congestion occurs, W_{max} is updated by current *cwnd* and *cwnd* is updated by equation 2.8. If congestion does not occur, function comes to convex part and tries to achieve maximal *cwnd*. The convex part is used mostly if some TCP stream finishes communication on the shared connection and releases his allocated bandwidth.

Further explanation of CUBIC algorithm is beyond the scope of this thesis and can be found in **RFC 8312** [25].

2.3.2 TCP global synchronization

TCP is very efficient in maintaining congestion in the network. Let's imagine a scenario of Internet network from section 2.2. If there is a huge amount of TCP connections communicating through the shared link, the phenomenon called `TCP global synchronization` occurs. The value of windows size attribute becomes equal across all TCP connections, which lowers average throughput of TCP streams.

Random early detection is a mechanism, which tries to avoid the TCP global synchronization. It is implemented by dropping randomly chosen data when the outgoing interface queue of congested device is at least partially filled (queue length must exceed the minimum threshold) [11]. The probability of dropping data increases with increasing queue length until the maximum threshold is exceeded. This is the moment when the intermediary devices (e.g. routers) start to drop every packet. The graph example of probability function is shown in figure 2.5. Random drop of packets in different TCP streams tells the TCP protocol that congestion occurs. Congestion control algorithms start avoiding congestion, but at different time, which means the value of window size attribute is different across the TCP streams and TCP global synchronization does not occur. This helps to keep higher average throughput of TCP streams.

Chapter 3

Network congestion impact on physical network

In this chapter we will observe a real life network and impact of congestion on it. Network congestion can affect different layers of network. We describe various problems, which may occur in the network. Our research is focused on four main types of errors, described in table 3.1.

The congestioned network is observed to find any occurrence of these errors. Other errors (e.g. jitter, etc.) are not part of this work. The experiments will use our testing switched network without routers, therefore it crosses the first two layers of OSI model[1], because these layers belong to kernel space and device hardware. Higher layers are investigated in following chapter.

This chapter is divided into sections by layers of OSI model. Each section includes theoretical analysis of each listed error and results of measurements of our tests on the experimental network.

A quick overview of measured results with description is visible in the last section of this chapter in table 3.2.

Error type	Description
Data loss	Amount of data that are lost during congestion.
Delay	Duration of delay caused by network congestion.
Data corruption	Data is corrupted during transmission (some data bits are changed)
Data reorder	Data of one flow is divided into several smaller parts and sent, but received in a different order.

Table 3.1: Error types that are observed in the network during congestion

3.1 The network used for experiments

For our experiments we used network with real servers described further in this section. The experimental network tries to simulate the utilization of the network described in section 2.2. The network diagram shown in figure 3.1 is an abstract diagram to a real scenario shown

[1]This model is described in section 2.1

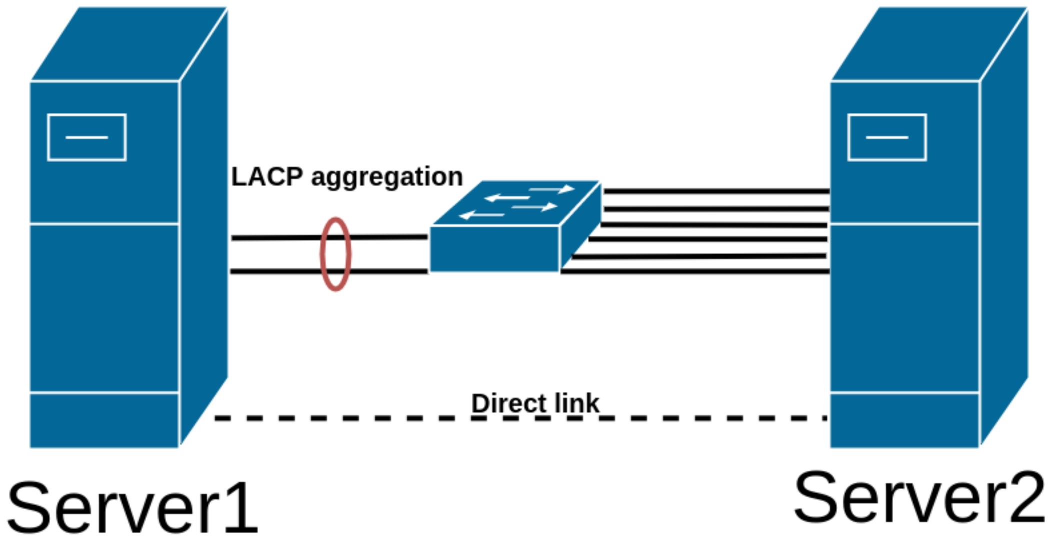

Figure 3.1: The diagram of testing network.

in figure 2.2. In this network there are two servers connected through a switch. The switch has 48 ports with 1Gb link speed. The server marked as *Server1* is connected to the switch with two physical links. Those links are aggregated together with `LACP` protocol described in section 3.1.1. A hash function distributes data according to `source IP address` between the two links. The second server called *Server2* is connected to a switch using six 1Gb connections. To reach simulated host's requirements the connection to the first server is slowed down to 100Mb per physical interface, which creates an aggregated interface of 200Mb link speed in each direction (400Mb in both directions). The ratio of available to required bandwidth, which emulates an over-subscription ratio from section 2.2.1, is 1:11 (4:44) in the real scenario shown in figure 2.2. The ratio in our testing environment can be variably modified from 1:1 to 1:30 (200:6000), so it is possible to simulate exactly the same ratio as is shown in the real scenario.

3.1.1 LACP - link aggregation control protocol

Good practice in local area networks is creating backup connections between intermediary devices, which eliminate a single point of failure. Adding backup connections into the network creates bridging loops, which might cause broadcast storms in the network. There is a solution of this problem. It is spanning tree (STP) protocol[2], which blocks the additional connections between switches to eliminate the bridging loop [19].

To utilize links blocked by STP, link aggregation takes place, which creates a virtual link bundled from physical interfaces, where traffic load can be shared. The STP protocol recognizes the virtual link as a single connection, which means both physical connections are functional and used. An example of an aggregated link in network is shown in figure 3.1.

An example of link aggregation is the use of the LACP[3] protocol, that maintains membership of physical connections in a virtual link. It is responsible for advertising the LACP

[2]Spanning tree protocol is defined by `IEEE 802.1D` standard [2].
[3]The LACP protocol is defined in `IEEE 802.3ad` standard [19].

virtual connection to the neighbor device. It sends LACP packets between those devices, which exchange necessary information for successful connection [19]. For successful creation of an LACP virtual link both sides of the connection have to be configured to use this protocol (e.g. „Server1“ and the switch have to use LACP protocol in aggregated connection).

3.1.2 Software

Servers used in this setup use RHEL-7.4 operating system. Public alternative of this system is CentOS, which is derived from RHEL sources[4]. Default congestion control algorithm in this operating system is CUBIC, briefly described in section 2.3.1.

3.2 Physical layer

As we mentioned already at the beginning of this chapter, this work is focused on four types of errors that can happen during network congestion. In this chapter we examine the possible errors in testing environment. This section will focus firstly on the theoretical part and afterwards we will share the results of the experiments which had been run.

Loss

In point-to-point Ethernet networks with full-duplex transmission mode the probability of data loss is very low, but it depends on quality of cable, cable length and environment, where cable is installed.

Data loss can be caused only by different transmission speed of NICs. However, nowadays it is impossible to create such situation, because NICs negotiate the transmission speed before the link becomes operational. If there is a speed mismatch, NICs can not communicate through this link. Simply said the NICs are built to receive as much data as they are able to send. Data loss might happen on WiFi[5] networks or other physical medium and it is not part of this work.

Delay

Delay can occur due to the characteristic of propagation over media. We call this propagation delay. It can be expressed by formula, shown in equation 3.1 [10].

$$\begin{aligned} delay &= \frac{distance}{speed} \\ delay_{3m} &= \frac{3m}{177000\frac{km}{s}} \doteq 16.94ns \\ delay_{500m} &= \frac{500m}{177000\frac{km}{s}} \doteq 2824.86ns \end{aligned} \tag{3.1}$$

The calculated propagation delay is too small in Ethernet networks, even if very long cable connections with repeaters are used, so it can be ignored in this work.

[4]https://www.centos.org/about/

[5]Wireless local area network.

Corruption

Data corruption used to be a big problem of second-class and low cost un-shielded cables. Data can be corrupted by electromagnetic interference from external sources or by the cable itself. UTP cables consist of four pairs of wires, which can also influence themselves. These days pairs of wires are twisted together and each pair has a different number of twists, which eliminates the electromagnetic influence. This problem is only related to copper cables.

Probability of data corruption in optical cables by electromagnetic or radio-frequency interference is very low. We do not investigate transmission behavior of optical cables in this thesis.

Reorder

Due to the nature of the cable medium in point-to-point Ethernet network, the reorder of data in this layer of OSI model can not happen. The reason is simple - each cable has just two ends, which means if you send data in specific order from one device to another, data is received exactly in the same order.

3.3 Measured results on physical layer

We did few experiments with the network during the congestion. According to limited options for testing in physical layer, we tried to find only corruption error. For these tests we use network topology shown in figure 3.1. The tests were performed exclusively using the *direct link* connection. To perform this analysis it was necessary to keep sending a large amount of data[6] to get eligible results. According to results there were not any errors during the test. The counters from interface are shown in listing 3.1

```
[root@uklizec1 ~]# ifconfig enp4s0f1
<output omitted>
    RX packets 1663126831 bytes 1979120929996 (1.8 TiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1590262300 bytes 1892412137783 (1.7 TiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Listing 3.1: „The interface coutners from server1“

3.4 Data link layer

To test this layer of OSI model we connected the servers through a switch. The direct point-to-point connections are ignored. To simulate the network congestion the server called *Server2* sends several TCP streams to server called *Server1* using `netperf` [7] benchmark [18]. Then *Server2* tries to simulate network congestion using `trafgen`, which produces a huge amount of UDP traffic. The measurements which are important to this layer are gathered and analyzed in both environments.

[6]The test transferred 1.8TB of data in 1 day.

[7]Netperf is a benchmark used to measure the network performance and will be discussed later.

Loss

Data loss might occur in network with a shared path, which can create a bottleneck, visible in figure 2.2. This error type occurs when excessive amount of data income. This data is not possible to distribute further because the outgoing interface is overloaded. For more detailed description see section 2.2.

Delay

In any network delays are caused by incoming and outgoing interface buffers. Incoming data is buffered before being processed. After processing data is queued into an outgoing buffer of outgoing interface. The time data spends in those buffers is the cause of these delays. During network congestion the delay is much longer - buffers are more utilized and fuller, because they need to process large amount of data. The calculation of nodal delay is shown in equation 3.2 [5].

$$d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop} \tag{3.2}$$

Corruption

Data corruption does not happen in this layer of OSI model. There is only a function for verifying the data integrity by checking checksum. The intermediary devices calculate its own checksum and if these two match, the data is proceeded to further processing. If these two numbers do not match, corruption error occurs in physical layer and the frame is dropped.

Reorder

We limit our speculations about reorder error to network shown in figure 3.1, where is only one possible link for communication. Reorder error is searched only in one data flow. The most used queuing mechanism is FIFO[8], which keeps the data order. This mechanism does not allow to create data reorder error in the network. Some of the NICs and switches use more FIFO queues within the one interface and data from one flow is distributed between the queues, which could create reorder error. Another possibility where reordering could happen are aggregated links. These links create a virtual interface, recognized as only one link by the intermediary device. For sending data to this virtual interface there is a hash function that distributes data to its physical interfaces. The hash function distributes data according to their addresses(MAC, IP, port number) contained in protocol headers. After distributing by hash function, the basic FIFO queuing mechanism is used again.

3.5 Measured results on data link layer

For searching data loss errors, the logs from NICs of the servers and the switch interfaces were checked to detect if any error occurred during network congestion.

[8]First in first out.

Data loss

In logs there are frame drops in outgoing queue of the switch interface, which points towards the first server. This confirms our expectations based on theory explained in section 2.2. The show command applied in the switch is shown in listings 3.2.

```
duchodce(config-if)#do show inter port-chann 1
<output omitted>
Input queue: 0/75/0/0 (size/max/drops/flushes); Total output drops:
    ↪ 35933202
```

Listing 3.2: Output drops captured in the switch.

Data corruption error

We did not find any data corruption errors in the logs from NICs and switch. Output of show command applied in the switch is pasted below.

```
[root@uklizec1 ~]# ifconfig bond0
<output omitted>
RX packets 7300325393 bytes 8693376275152 (7.9 TiB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 8597112660 bytes 8676956810520 (7.8 TiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Listing 3.3: The log from interface in uklizec1 LACP interface

Data reorder error

To test data reorder we developed a simple special utility which uses `UDP socket`. This utility consists of two programs (`sender, receiver`). Within the payload of our UDP datagram we transmit increasing sequence number with an unimportant payload. The receiver program receives the datagram and checks the sequence number by comparing it to the previously received sequence number. If the number from the current frame is different than last number +1, we have detected a data reorder error. We ran our experiments for one hour and did not detect a single reorder error.

Source codes of these two programs can be found in appendixes on CD.

Delay in the congested network

The delay in the network was measured using the `ICMP`[9] protocol, which is able to determine the nodal delays. Based on the test that we ran for about one hour, the average delay in non-fully utilized network was $0.169ms$. Compare to congested network this measured delay is significantly lower. In fully utilized network with TCP streams the average delay is $27.364ms$ (this value is the result of the test executed for about an hour). The average delay in network congested by UDP stream is $37.085ms$. The biggest part of the measured delay is caused by overloaded buffer. The delay rises with length of outgoing interface buffer, until the buffer gets full. Once the buffer is full, the delay remains at its maximum value.

Ping test output of non-congested network is pasted below. There is a very small delay in the network, because buffers in the switch are almost empty.

[9]Internet control message protocol is an error detection protocol defined in RFC 792.

```
--- 192.168.10.11 ping statistics ---
36000 packets transmitted, 36000 received, 0% packet loss, time 3599899ms
rtt min/avg/max/mdev = 0.098/0.169/0.638/0.021 ms
```

Listing 3.4: The round trip time of the non-congested network.

Ping test output of TCP congested network is pasted below. Average round trip time is enormously bigger compare to round trip time in non-congested network.

```
--- 192.168.10.11 ping statistics ---
36000 packets transmitted, 35779 received, 0% packet loss, time 3621928ms
rtt min/avg/max/mdev = 9.366/27.364/62.892/7.744 ms
```

Listing 3.5: The round trip time of the network congested by TCP protocol.

Ping test output of UDP congested network is pasted below. Average round trip time is even bigger than in TCP congested network.

```
--- 192.168.12.1 ping statistics ---
128310 packets transmitted, 67541 received, 47% packet loss, time 1199992ms
rtt min/avg/max/mdev = 0.575/37.085/40.816/8.157 ms, pipe 8
```

Listing 3.6: The round trip time of network congested by UDP protocol.

3.6 Evaluation of the results

In the first layer of OSI model there were no serious errors that could be simulated by the network emulator. We were not able to simulate the packet reorder error in network with LACP aggregated interface. Possibility of packet reorder error occurs in WAN[10] network, where MPLS[11] technology is used or in routed(L3) networks where routing protocols use loadbalance feature to utilize the links more efficiently.

We were not able to simulate packet corruption error caused by hardware in the first or second layer of OSI model either. The quality of today's copper cables is much higher. For longer distances are usually used optical cables, which are capable to transfer data for greater distances with higher speed. The optical cables do not suffer from crosstalk or EMI[12]/RFI[13].

From the results measured in section 3.4 describing errors in the data layer is clearly visible that the error with the biggest impact to the network was data loss. The reason why this happened was a larger amount of data coming to the switch, which had no resources to distribute this data further. Some data was lucky enough to be queued to outgoing buffer. However, this almost full buffer caused significantly higher delay. These two errors are worth to be investigated further under various circumstances (over-subscription ratio, transport protocol, etc.). These errors are great candidate for network congestion simulation in created testing network, which we will discuss and describe in chapter 5.

The overview of measured results is visible in table 3.2.

[10] World area network
[11] Multi protocol layer switching
[12] Electro magnetic interference
[13] Radio frequency interference

	Network hardware	
OSI layer / Error	Physical	Data link
Loss	✘	✔ - TX queue <0% - 100%)
Delay	✔ - small, propagation (17ns)	✔ - TX queue OS buffers
Corruption	✘	✘
Reorder	✘	✘

✘ - the error was not found

✓ - the error occurs

Table 3.2: Errors that were investigated in experiments with the network during congestion.

Chapter 4

Linux protocol stack behavior during network congestion

In this chapter we describe several experimental test scenarios, which try to analyze behavior of protocol stack in network during congestion. Our interest is focused on two categories of protocols: connection-less (UDP) and connection-oriented (TCP) protocols. These protocols are observed in testing network with different values of oversubscription ratio, where every abstract customer tries to achieve full provided bandwidth.

The results of measurements from this chapter will be used in the next chapter for designing test scenario for CI testing of kernel project.

4.1 Network used for experiments

The network used for testing is easily changed compared to testing network[1] used for measurements of impact to physical network. According to experiments from chapter 3, we decided to focus only on two errors (data loss, transmission delay), because only these two really happen in Ethernet network during congestion. It also appeared that the data reorder error does not occur, therefore we replaced aggregated link by LACP with connection that consists of single wire. Link speed of all interfaces in the switch is set to 100 Mbps. On all interfaces of both servers we turn off *tcp-segmentation-offload* to be able to capture each TCP segment and evaluate network delay by Wireshark. It evaluates the delay by subtracting time stamp of sent segment from time stamp of received acknowledgment. If *tcp-segmentation-offload* is turned on, NIC takes care of segmentation and in traffic capture we see only single large segment. In this situation Wireshark is not able to exactly evaluate round trip time in the network.

4.2 iPerf3 - testing tool

iPerf3 is network testing tool, which measures various characteristics of network, but is mostly used for measure achievable network bandwidth [3]. It can use several protocols (TCP, UDP, SCTP) for testing and reports the amount of transferred data, the amount of lost data, average bandwidth, and many others. It is also able to set options for specific testing protocol (e.g. maximum segment size, sizes of buffers, etc.). iPerf3 can store output log in *JSON* format, which is easy to process in evaluation in CI testing.

[1]Network diagram of previous network is shown in figure 3.1

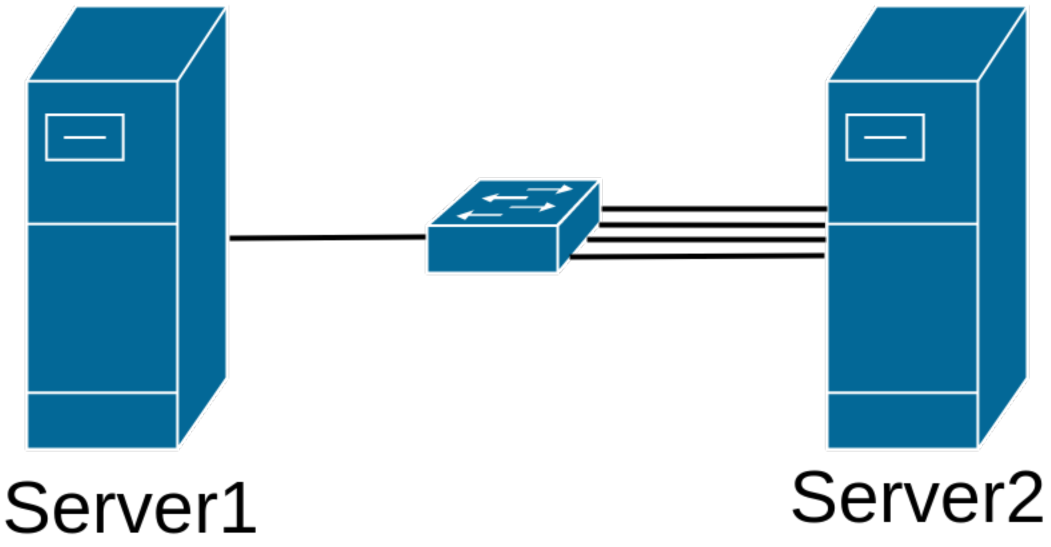

Figure 4.1: The diagram of testing network after changes.

4.3 Behavior of connection-less and connection-oriented protocols

For the network communication are used two different groups of protocols. The main differences between these groups are reliable data delivery and establishment of connection for data transmission. These protocols exist in the fourth layer of OSI model. In this section we are going to observe behavior of User Datagram Protocol (UDP) and Transmission Control Protocol (TCP).

UDP does not guarantee data delivery and does not use any control flags to inform sender that receiver lost the data. UDP just sends datagrams to specific destination port and does not care about anything else. UDP finds its justification in several services and solutions, where data loss is accepted or required.

The first highly used scenario of UDP usage is voice over IP, which is very sensitive to network delay. If small amount of data is lost during voice call over IP, the users would not even recognize the data is lost. It is useless to resend lost data because it is too late for decoding voice signal to receiving user.

The second very often used example of UDP communication are video streams, which are similar to voice communication. Streamed video data is divided into smaller segments and sent to network. Video receiver is still able to reconstruct the original video even if small percentage of data is lost.

The last example of usage of UDP protocol is VXLAN tunnel. This type of tunnel varies from others. Each endpoint may have more tunnel destinations hidden behind one IPv4/IPv6 address. VXLAN tunnel uses UDP protocol to separate traffic from different hosts especially in virtual environment.

As we can see UDP protocol is highly used for different services, therefore UDP protocol is the subject of our experiments to discover its behavior during network congestion.

The **connection oriented protocols** are often used for reliable network communication. They are used by various applications, which are not sensitive to delay and also needs reliable data delivery. There are various protocols which meet requirements of reliable and connection-oriented protocol (e.g. TCP, SCTP, RDP). The most used connection-oriented protocol is Transmission Control Protocol (TCP). This protocol is used for web browsing, file transfer, mail communication, telnet and many others. All of these services need reliable data delivery to keep an integrity of transmitted information.

Reliability and congestion control

Before sending data, TCP establishes connection, which is used for tracking sent and lost data during communication. The reliability is guaranteed by retransmission of lost data, which may cause additional delay in communication. TCP is described in section 2.3.1. TCP tries to avoid congestion in the network using special algorithms, which limit the amount of transmitted data.

4.3.1 Testing methodology

For the observations of UDP and TCP behavior we simulate two potential customers by UDP/TCP streams of various bandwidth. Experiments are focused mostly on two errors : data loss and network delay. Programs used for testing and measurements are `ping` and `iperf3` [2] and `netperf`. Two UDP/TCP streams run in parallel from *Server2*, where several `Ipsef3` services run in background to *Server1*, the initiator of stream. This test simulates two virtual machines in cloud environment, that try to send stream of data in constant speed to external recipients. They both have the same shared connection to the outside network, which creates a bottleneck for network connectivity of virtual machines.

Throughput

We run several tests with different over-subscription ratio, accomplished by various UDP streams bandwidth. Bash script used for starting the UDP streams is visible in Listings 4.1. The script has one command line argument - the bandwidth. This script runs two parallel `iperf3` UDP streams with the same bandwidth. The logs from tests inform about how much data is sent and how much data is lost with sending and receiving bandwidth.

```bash
#!/bin/bash
iperf3 --client 192.168.12.11 --bind 192.168.12.1 --udp --reverse --time 20
    ↪  --bandwidth $1 --port 10001 | tee -a udp1_$1.log &
iperf3 --client 192.168.13.11 --bind 192.168.13.1 --udp --reverse --time 20
    ↪  --bandwidth $1 --port 10002 | tee -a udp2_$1.log &
```

Listing 4.1: Script used for starting two parallel UDP stream.

Latency

While UDP/TCP test is running, the ping test, which measures round trip time in the network, is running as well. The ping command is ran in *Server2* and travels in the same wire as the first UDP stream. The round trip time measures the transmission delay in both ways of transfer (*Server2* -> *Server1* -> *Server2*). This delay is very small in non congested network, therefore it is neglected. The ping test returns statistic about round trip time (minimal, maximal, average, standard deviation), which shows fullness of switch

[2]This program is described in section 4.2.

buffers during the test. This test runs 15 seconds and sends ping packet of size 1450 bytes every 1ms. Output of every test is saved for further investigation.

```
ping 192.168.12.1 -s 1450 -i 0.001 -w 15
```

Listing 4.2: Ping command used in UDP behavior testing.

In the TCP test we additionally run `netperf` test called *TCP_CRR* (connect/request/response) to be sure the results from ping are correct. This test creates new TCP connection for each request to server and measures time, which server takes to respond to the request. This test is usually used to simulate *HTTP* requests [18].

```
netperf -t TCP_RR -H 192.168.12.11 -- -r 1400,1400
```

Listing 4.3: Ping command used in UDP behavior testing.

The result of `netperf` test is in *transactions per second* unit, which is converted to delay by equation 4.1.

$$dealy_{CRR} = \frac{1}{transaction \quad per \quad second} \tag{4.1}$$

Window scaling

The TCP has congestion control algorithms[3], which can modify transmitting throughput by changing congestion window to effectively utilize network. The experiment, which observes behavior of these algorithms, consists of two TCP streams, sharing same connection. The over-subscription ratio in this set is 2. The first test runs for 30 seconds and the second stream runs for only 15 seconds, but it is ran 5 seconds later than the first one. The example of testing script is visible in listings 4.4.

```
#!/bin/bash
iperf3 --client 192.168.12.11 --bind 192.168.12.1 --reverse --time 30 --
    ↪ bandwidth $1 --port 10001 &
sleep 5
iperf3 --client 192.168.13.11 --bind 192.168.13.1 --reverse --time 15 --
    ↪ bandwidth $1 --port 10002 &
sleep 25
```

Listing 4.4: This script runs two TCP streams to create network congestion.

Whole test is captured by *tcpdump* for further analysis by *Wireshark*[4], which is able to evaluate round trip time of TCP communication by calculating difference of timestamp of sent data and timestamp of received data acknowledgment.

Congestion window variable is not included inside the TCP communication, but it is a part of socket attributes in the linux kernel. For tracking this variable we use specialized kernel module called *tcpprobe*, which appends information about specific socket to a file [4].

Workflow

The testing workflow is described step by step in algorithm 1. By using this algorithm we observe behavior of UDP protocol in congested and non-congested network. In the first

[3]The congestion control algorithms are described in section 2.3.1.

[4]Wireshark is network analyzing and troubleshooting tool [15].

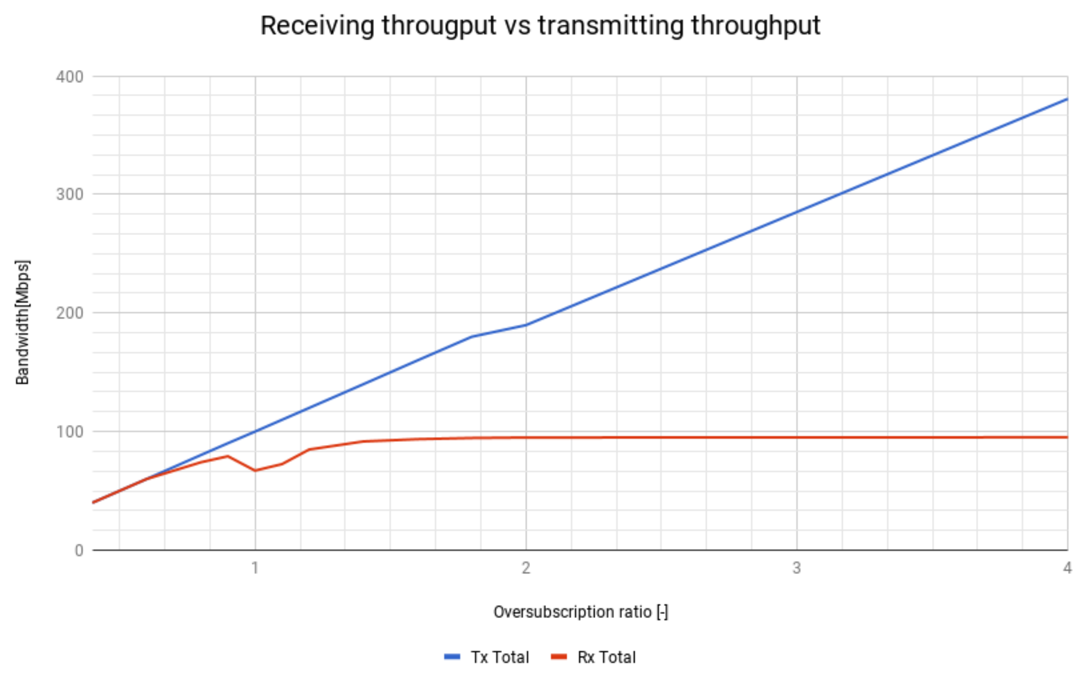

Figure 4.2: Sending and receiving throughput of UDP test in different over-subscription ratio.

place we need to create a list of over-subscription ratios, which will be tested. Then follows the algorithm 1 to get results of UDP behavior tests.

For TCP behavior test we use the same list of over-subscription ratios by algorithm 2. iPerf3 *–udp* parameter is removed to run TCP stream test and *netperf TCP_CRR* test runs together with ping.

4.4 Results evaluation of connectionless protocols

Selected over-subscription ratio values are : [0.4, 0.6, 0.8, 0.9, 1.0, 1.1, 1.2, 1.4, 1.6, 1.8, 2.0, 3.0, 4.0]. These values capture network before and during congestion. For measurement is used algorithm 1, with defined values. Detailed description of measured results is in the following sections focused on throughput, data loss and round trip time during different over-subscription ratios.

Throughput and loss

From measurement is visible that sending and receiving throughput in the network, where the over-subscription ratio is small (0.4, 0.6), has at the same value with no data loss. The sending and receiving throughput from observations are in table 4.1 and chart from this value is visible in figure 4.2. Table 4.2 together with table 4.3 contain number of transmitted and received data. There is real and theoretical data loss as well. The real data loss for current over-subscription ratio is calculated by equation 4.2.

$$Loss_{real} = \frac{data_{sent} - data_{received}}{data_{sent}} \tag{4.2}$$

Algorithm 1: UDP behavior testing algorithm.

```
UDPTest (ratios)
    input  : list of over-subscription ratios, which are going to be measured
    output: records of send/receive bandwidth and amount of data, statistics of
            round trip time
    start iperf3 services in Server1 on ports 10001, 10002, 10003, 10004;
    foreach ratio ∈ ratios do
        if ratio <= 2 then
            $B \leftarrow ratio * 100Mbps/2$;
            // run 2 instances of iperf3 UDP stream test with B bandwidth
            iperf3 –client 192.168.12.11 –bind 192.168.12.1 –udp –reverse –time 20
             –bandwidth B –port 10001 & ;
            iperf3 –client 192.168.13.11 –bind 192.168.13.1 –udp –reverse –time 20
             –bandwidth B –port 10002 & ;
        else
            if ratio <= 3 then
                $B \leftarrow ratio * 100Mbps/3$;
                // run 3 instances of iperf3 UDP stream test with B
                   bandwidth
                iperf3 –client 192.168.12.11 –bind 192.168.12.1 –udp –reverse –time 20
                 –bandwidth B –port 10001 & ;
                iperf3 –client 192.168.13.11 –bind 192.168.13.1 –udp –reverse –time 20
                 –bandwidth B –port 10002 & ;
                iperf3 –client 192.168.14.11 –bind 192.168.14.1 –udp –reverse –time 20
                 –bandwidth B –port 10003 & ;
            else
                $B \leftarrow ratio * 100Mbps/4$;
                // run 4 instances of iperf3 UDP stream test with B
                   bandwidth
                iperf3 –client 192.168.12.11 –bind 192.168.12.1 –udp –reverse –time 20
                 –bandwidth B –port 10001 & ;
                iperf3 –client 192.168.13.11 –bind 192.168.13.1 –udp –reverse –time 20
                 –bandwidth B –port 10002 & ;
                iperf3 –client 192.168.14.11 –bind 192.168.14.1 –udp –reverse –time 20
                 –bandwidth B –port 10003 & ;
                iperf3 –client 192.168.15.11 –bind 192.168.15.1 –udp –reverse –time 20
                 –bandwidth B –port 10004 & ;
        wait 2 seconds; // wait till UDP streams start
        /* start ping test for 15 seconds with message size 1450 and
           interval 1ms                                              */
        ping 192.168.12.1 -s 1450 -i 0.001 -w 15 ;
        record ping RTT output (min, avg, max, stdev);
        foreach instance ∈ iPerf3RunningInstances do
            record results of the test ()send/receive bandwidth, sent/received data) ;
```

Algorithm 2: TCP behavior testing algorithm.

```
TCPTest (ratios)
    input  : list of over-subscription ratios, which are going to be measured
    output: records of send/receive bandwidth and statistics of round trip time
    start iperf3 services in Server1 on ports 10001, 10002, 10003, 10004;
    foreach $ratio \in ratios$ do
        if $ratio <= 2$ then
            $B \leftarrow ratio * 100Mbps/2$;
            // run 2 instances of iperf3 TCP stream test with $B$ bandwidth
            iperf3 –client 192.168.12.11 –bind 192.168.12.1 –reverse –time 20
             –bandwidth $B$ –port 10001 & ;
            iperf3 –client 192.168.13.11 –bind 192.168.13.1 –reverse –time 20
             –bandwidth $B$ –port 10002 & ;
        else
            if $ratio <= 3$ then
                $B \leftarrow ratio * 100Mbps/3$;
                // run 3 instances of iperf3 TCP stream test with $B$
                   bandwidth
                iperf3 –client 192.168.12.11 –bind 192.168.12.1 –reverse –time 20
                 –bandwidth $B$ –port 10001 & ;
                iperf3 –client 192.168.13.11 –bind 192.168.13.1 –reverse –time 20
                 –bandwidth $B$ –port 10002 & ;
                iperf3 –client 192.168.14.11 –bind 192.168.14.1 –reverse –time 20
                 –bandwidth $B$ –port 10003 & ;
            else
                $B \leftarrow ratio * 100Mbps/4$;
                // run 4 instances of iperf3 TCP stream test with $B$
                   bandwidth
                iperf3 –client 192.168.12.11 –bind 192.168.12.1 –reverse –time 20
                 –bandwidth $B$ –port 10001 & ;
                iperf3 –client 192.168.13.11 –bind 192.168.13.1 –reverse –time 20
                 –bandwidth $B$ –port 10002 & ;
                iperf3 –client 192.168.14.11 –bind 192.168.14.1 –reverse –time 20
                 –bandwidth $B$ –port 10003 & ;
                iperf3 –client 192.168.15.11 –bind 192.168.15.1 –reverse –time 20
                 –bandwidth $B$ –port 10004 & ;
        wait 2 seconds; // wait till TCP‘ streams start
        /* start ping for 15 sec with msg size 1450 and interval 1ms   */
        ping 192.168.12.1 -s 1450 -i 0.001 -w 15 &;
        /* start netper TCP_CRR test to measure delay in the network
           differently                                                  */
        netperf -H 192.168.12.1 -l 15 -t TCP_CRR ;
        record ping RTT output (min, avg, max, stdev);
        record netperf TCP_CRR output ;
        foreach $instance \in iPerf3RunningInstances$ do
            record results of the test (send/receive bandwidth) ;
```

	TX throughput					RX throughput				
ratio	A	B	C	D	Total	A	B	C	D	Total
0.4	20	20	0	0	40	20	20	0	0	40
0.6	30	30	0	0	60	30	30	0	0	60
0.8	40.1	40.1	0	0	80.2	35.7	38.4	0	0	74.1
0.9	45.1	45.1	0	0	90.2	42.3	36.9	0	0	79.2
1	50	50	0	0	100	35.8	31.3	0	0	67.1
1.1	55	55	0	0	110	33.9	38.7	0	0	72.6
1.2	60	60	0	0	120	40.1	44.8	0	0	84.9
1.4	69.9	70.1	0	0	140	45.5	46.2	0	0	91.7
1.6	80	80.1	0	0	160.1	48.5	45.1	0	0	93.6
1.8	89.9	90.1	0	0	180	41.8	52.8	0	0	94.6
2	95.4	94.4	0	0	189.8	39.1	55.9	0	0	95
3	94.5	95.4	95.4	0	285.3	24.7	46	24.5	0	95.2
4	95.4	95.4	94.5	95.4	380.7	20.3	10.4	33.4	31.1	95.2

Table 4.1: The sending and receiving bandwidth of UDP test.

Theoretical data loss, used for comparing the real loss, is calculated by function, described in equation 4.3, where n is current ratio.

$$f_{TheoreticalLoss}(n) = \begin{cases} 0 & \text{if } n < 1 \\ 1 - \frac{1}{n} & else \end{cases} \tag{4.3}$$

The comparison of real and theoretical data loss is visible in chart in figure 4.3. It is clearly visible that the data loss approximates to value 1, which means 100% data loss.

With rising over-subscription ratio data loss error occurs. However, it appears with ratios [0.8, 0.9, 1.0], which should not cause data loss. This is due to implementation of sending algorithm in iPerf3, which sends burst of data in the first time period and in the second iPerf3 waits and sends nothing, because it sends more data in the first time period than it should. This behavior is similar to pulse wave modulation known in embedded systems. Captured example is visible in figure A.1. If iPerf3 wants to send data stream of 50Mbps it will send UDP stream of 50+Mbps in the first time period and then stop sending till overall average throughput will become 50Mbps again. According to testing algorithm, there are two UDP streams in one testrun, so if they both try to send 50Mbps (over-subscritpion ratio 1.0) and send time slots overlap each other, the data loss appears in the outgoing buffer of switch interface.

When over-subscription ratio crosses value 1.0, the network would get to congested state. The results from the test proves it. In figure 4.2 is evident that the receiving bandwidth (over-subscription ratio : 1.1, 1.2, 1.4) is lower than transmitting bandwidth. But receiving bandwidth is even lower than link speed of interface. This behavior is due to iPerf3 sent policing algorithm again.

Receiving throughput converges to link speed with higher over-subscription ratio (1.6, 1.8, 2, 3, 4). In the testruns, where the iPerf3 sent algorithm was not used, the real and theoretical loss is almost the same value. The UDP streams were sent at full bandwidth and decision about data drop remained to the switch, which drop data equally.

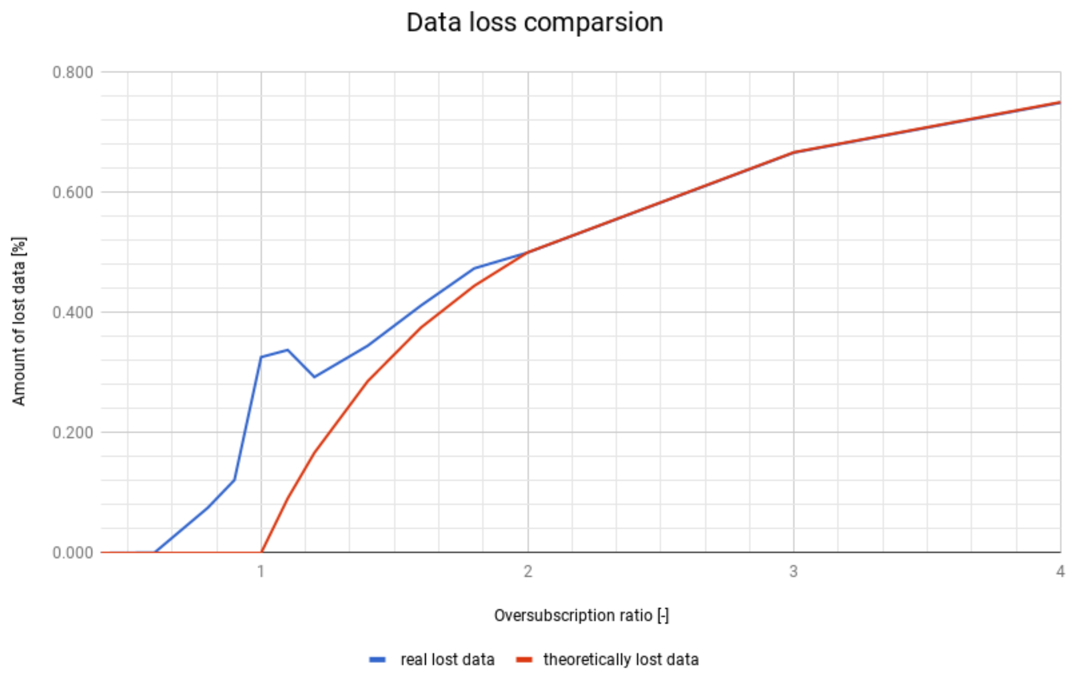

Figure 4.3: Amount of lost data in different over-subscription ratio during UDP behavior test.

	TX messages				
ratio	A	B	C	D	Total
0.4	34555	34556	0	0	69111
0.6	51860	51861	0	0	103721
0.8	69184	69185	0	0	138369
0.9	77837	77837	0	0	155674
1	85759	86328	0	0	172087
1.1	94962	94398	0	0	189360
1.2	103634	103665	0	0	207299
1.4	120757	120846	0	0	241603
1.6	136659	138065	0	0	274724
1.8	154743	155515	0	0	310258
2	164658	163031	0	0	327689
3	163086	164659	164674	0	492419
4	164660	162906	163087	164671	655324

Table 4.2: Number of sent messages during UDP behavior test.

	RX messages					Loss	
ratio	A	B	C	D	Total	Real	Theoretical
0.4	34555	34556	0	0	69111	0.000	0.000
0.6	51860	51802	0	0	103662	0.001	0.000
0.8	61633	66325	0	0	127958	0.075	0.000
0.9	73008	63768	0	0	136776	0.121	0.000
1	61892	54126	0	0	116018	0.326	0.000
1.1	58613	66801	0	0	125414	0.338	0.091
1.2	69242	77383	0	0	146625	0.293	0.167
1.4	78581	79727	0	0	158308	0.345	0.286
1.6	83698	77931	0	0	161629	0.412	0.375
1.8	72133	91226	0	0	163359	0.473	0.444
2	67435	96497	0	0	163932	0.500	0.500
3	42588	79493	42339	0	164420	0.666	0.667
4	35090	17896	57722	53720	164428	0.749	0.750

Table 4.3: Number of received messages and data loss in UDP behavior test in different over-subscription ratio.

Round trip time

The average round trip time in the network with different over-subscription ratio is visible in figure 4.4. The data used to create the chart is in table 4.4. With rising over-subscription ratio the average round trip time converges to value $42.9 \pm 0.2ms$. This delay is caused by full-filled buffer of outgoing switch interface. The test-runs with over-subscription ratio in the interval $< 0.4, 1.8 >$ have big standard deviation in results of ping command. As discussed above in the throughput section, the iPerf3 use special algorithm to achieve given bandwidth. When data burst of both UDP streams overlap, buffer is filled fully and delay of value $42.9 \pm 0.2ms$ appears in the network. When data streams gets idle, buffers are practically empty and there is no delay caused by interface buffer. The average round trip time refers to mean of repeated values $[0.2ms, 42.9ms]$ by Monte Carlo method [14].

4.5 Results evaluation of connection oriented protocol

For observing connection-oriented protocol behavior we use the same over-subscription ratio list as in connection-less protocols test. The list can be found in section 4.4. For measurement is used algorithm 2, with defined values. Detailed description of measured results is in the following sections focused on throughput, amount of retransmitted data and round trip time during different over-subscription ratios.

Throughput

In the results from TCP test is visible the sending throughput is the same as receiving throughput. With over-subscription ratios, which should not create congestion[5], sending and receiving throughput are equal to requested value. Results from throughput tests can be visible in table 4.5 and chart from this table is visible in figure 4.5.

[5]The ratio is lower than 1.

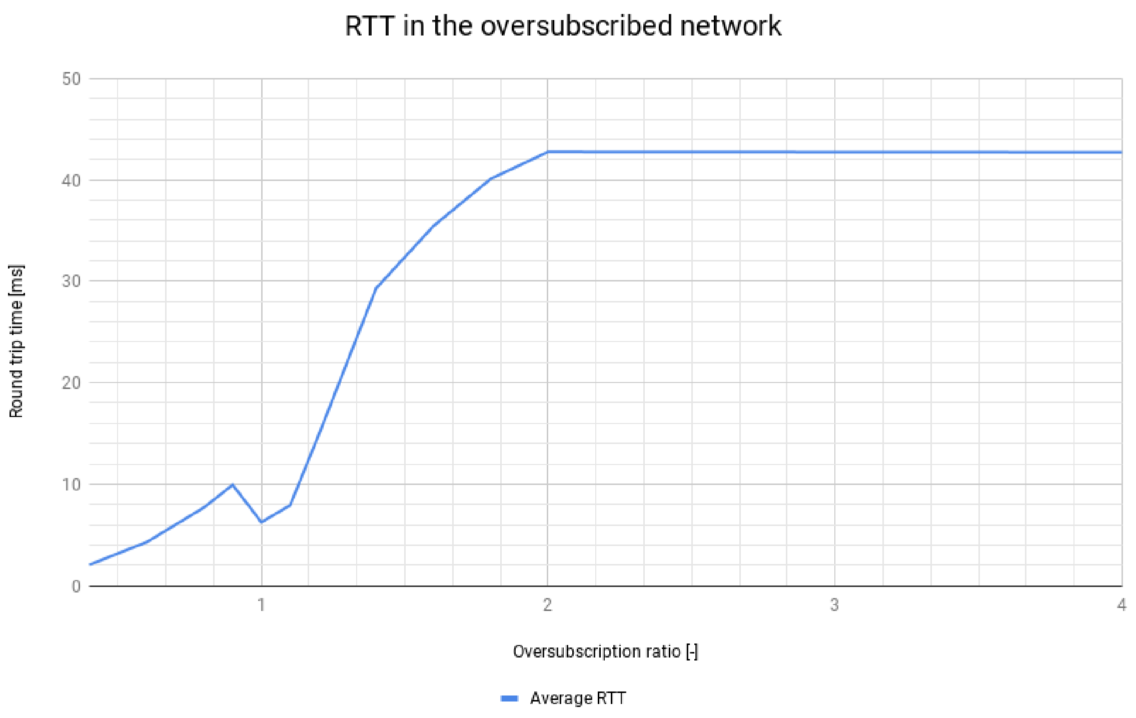

Figure 4.4: Round trip time in different over-subscription ratio during UDP behavior test.

	Latency			
ratio	min	avg	max	stdev
0.4	0.639	2.051	28.063	4.966
0.6	0.64	4.282	38.749	8.885
0.8	0.624	7.701	42.942	12.732
0.9	0.627	9.934	43.041	14.136
1	0.636	6.25	42.948	12.849
1.1	0.623	7.908	43.018	14.367
1.2	0.629	14.859	43.042	17.649
1.4	0.647	29.312	43.034	17.411
1.6	0.676	35.455	43.189	14.195
1.8	0.682	40.113	42.998	8.835
2	42.582	42.777	43.024	0.263
3	42.589	42.772	42.983	0.284
4	42.492	42.73	42.934	0.197

Table 4.4: The network delay measured in the network during UDP behavior test.

When network congestion occurs[6], congestion control algorithms start to lower transmission speed to avoid unnecessary data loss. They try to fully utilize network, therefore total sending/receiving throughput is near to bottleneck bandwidth.

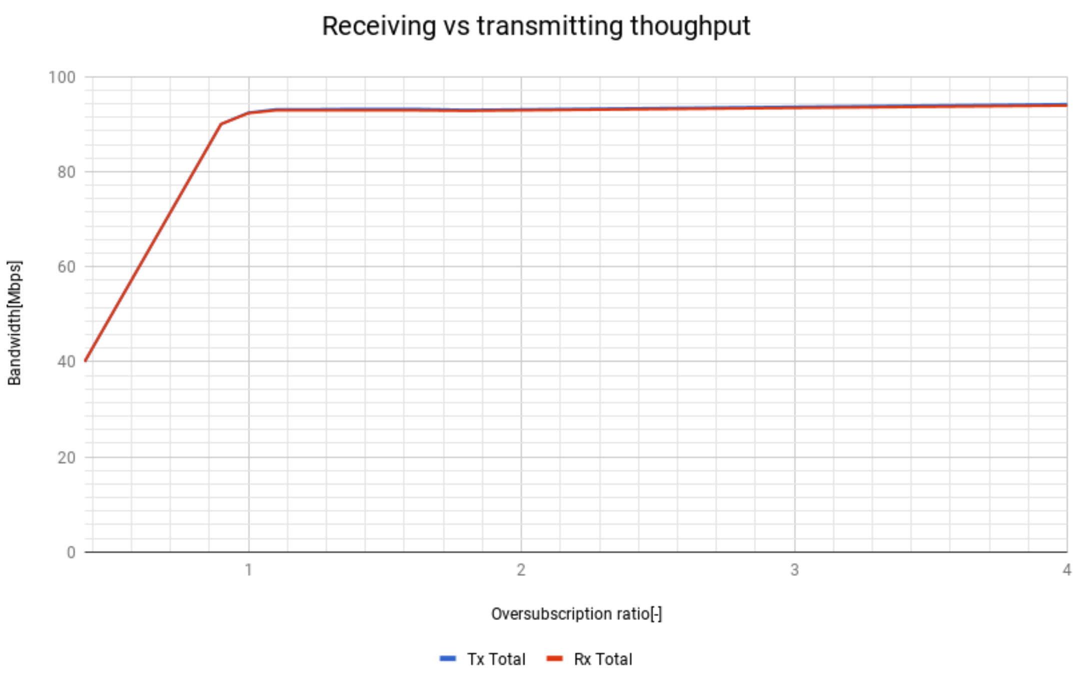

Figure 4.5: Sending and receiving throughput during connection oriented protocol test.

	TX throughput					RX throughput				
pf	A	B	C	D	Tx Total	A	B	C	D	Rx Total
0.4	20	20	0	0	40	20	20	0	0	40
0.6	30	30	0	0	60	30	30	0	0	60
0.8	40	40	0	0	80	40	40	0	0	80
0.9	45	45	0	0	90	45	45	0	0	90
1	50	42.4	0	0	92.4	49.9	42.4	0	0	92.3
1.1	46.5	46.6	0	0	93.1	46.4	46.5	0	0	92.9
1.2	40.9	52.2	0	0	93.1	40.8	52.1	0	0	92.9
1.4	51.7	41.5	0	0	93.2	51.6	41.3	0	0	92.9
1.6	57	36.2	0	0	93.2	56.9	36	0	0	92.9
1.8	37.8	55.2	0	0	93	37.7	55.1	0	0	92.8
2	48.2	44.9	0	0	93.1	48.1	44.8	0	0	92.9
3	30.2	36.2	27.9	0	94.3	30	35.9	27.7	0	93.6
4	21.5	23.5	26.5	22.7	94.2	21.4	23.4	26.4	22.7	93.9

Table 4.5: Sending and receiving throughput in different over subscription ratio during connection oriented protocol test.

[6]The ratio is higher or equal to 1.

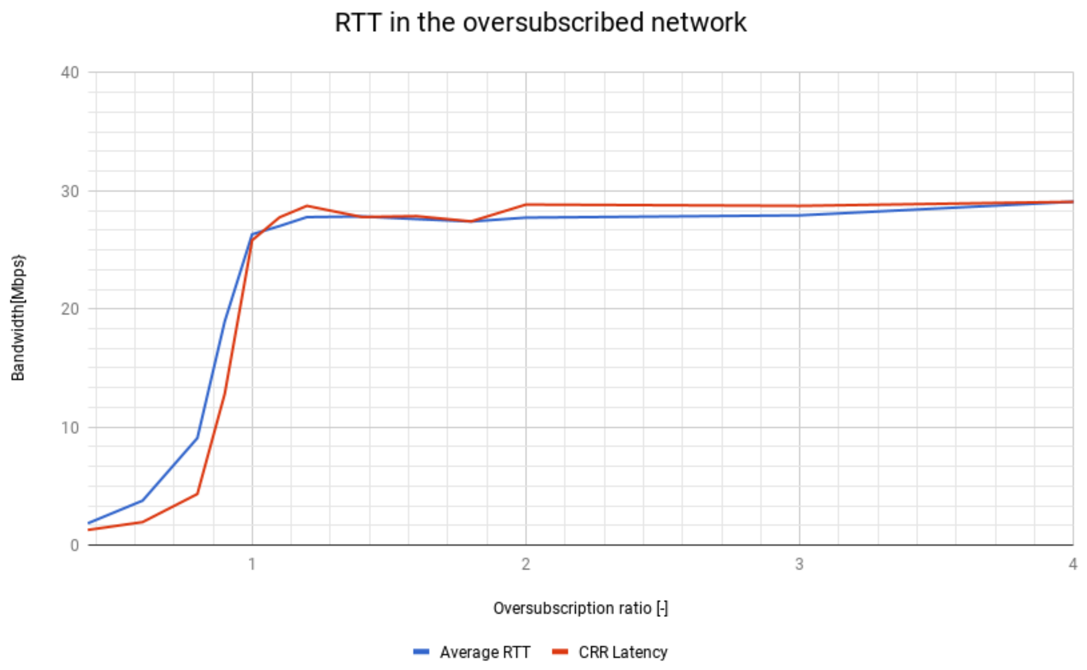

Figure 4.6: Round trip time in different over-subscription ratio during connection oriented protocol test.

Round trip time

The network device cannot send data at specific speed. It can only send at link speed (or it can not send data at all). If network hardware wants to slow down transmission speed to specific rate, it regularly switches between sending and silent period to reach the ratio between sending and silent period, which corresponds to transmission speed at specific rate. When several streams share the same connection with the same or lower link speed and want to use this method to lower transmission rate, congestion may occur. Results from tests with oversubscription ratio [0.4, 0.6, 0.8, 0.9] prove it. The results of tests are visible in figure 4.6. The round trip time in the network rises with higher oversubscription ratio. There is high standard deviation of results caused by the above described behavior. When only one of two streams sends data, the round trip time is low. When two streams send data at the same time, congestion occurs and buffers get full, which correspond to higher round trip time.

The results with ratios, which cause congestion, show the average delay is $27.5 \pm 2ms$. This value corresponds to fullness of outgoing switch interface buffer, where data is temporarily stored until interface can send it. TCP tries to fully utilize network, but tries to avoid congestion as well, therefore throughput of streams still get bigger or lower. This competition of bandwidth makes the streams send data at bottleneck bandwidth speed. Interface buffer does not get empty, but creates delay.

	Latency				CRR		TX Retransmits			
pf	min	avg	max	stdev	tpm	delay	A	B	C	D
0.4	0.623	1.856	25.458	4.07	772.15	1.295	0	0		
0.6	0.626	3.777	33.614	7.456	510	1.961	5	6		
0.8	0.624	9.076	32.468	11.06	230.48	4.339	8	8		
0.9	0.619	18.956	32.285	10.589	78	12.821	13	29		
1	9.195	26.325	32.756	4.372	38.73	25.820	13	30		
1.1	19.082	27.023	31.988	3.14	36.02	27.762	17	23		
1.2	15.765	27.783	31.928	2.712	34.8	28.736	17	21		
1.4	18.688	27.837	31.892	2.921	35.98	27.793	16	31		
1.6	19.293	27.615	31.975	2.974	35.89	27.863	13	28		
1.8	19.809	27.412	32.136	2.901	36.47	27.420	23	18		
2	22.398	27.739	31.921	2.571	34.67	28.843	21	17		
3	21.409	27.931	31.72	2.433	34.8	28.736	22	15	18	
4	19.332	29.084	31.968	1.779	34.39	29.078	27	26	25	38

Table 4.6: Latency in the network during connection oriented protocol test.

Window scaling

In figure 4.8 is visible rapid growth of round trip time at fifth second. At this time the second iPerf3 stream starts to send data, creates congestion and fills interface buffer, which creates high delay. Due to congestion some data is lost, therefore congestion avoidance algorithm starts to control transmission speed. It rapidly increases congestion window to achieve similar throughput in delayed network. The congestion window during test is visible in figure 4.7. The maximal theoretical throughput of TCP with current congestion window and round trip time can be calculated by equation 4.4.

$$throughput <= \frac{cwnd}{RTT} \tag{4.4}$$

During congestion TCP algorithm maintains transmission speed and round trip time is about $27 \pm 2ms$ as we expected. Created over-subscription ratio is 2.

When second TCP stream ends and over-subscription ratio gets back to 1, which means there should not be network congestion, some data is still in interface outgoing buffer and causes delay in the network. The first TCP stream sends data at link speed without creating congestion, whereupon the switch does not have the opportunity to empty the buffer and delay in the network still exists.

4.6 Conclusion

From the results of connection-less oriented protocols is clearly visible, that the delay caused by outgoing switch buffer exists in the network, where over-subscription ratio is equal or higher to 1. Based on low standard deviation we expected the delay during network congestion would be constant. Part of lost data increases with over-subscription ratio as well, because UDP protocol does not implement any congestion control mechanism.

From the results of connection oriented protocols is visible that the available bandwidth is shared between all TCP connections. Amount of lost data is much lower compared to connection-less protocols, because TCP implements congestion control algorithms, which

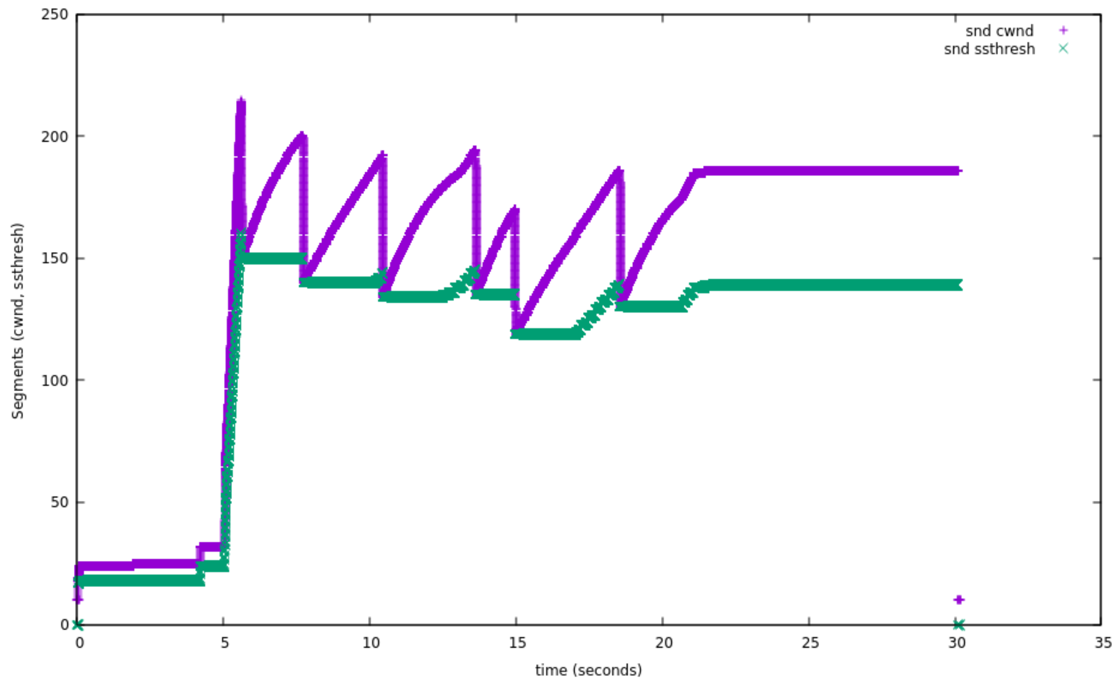

Figure 4.7: TCP congestion window during window scaling test.

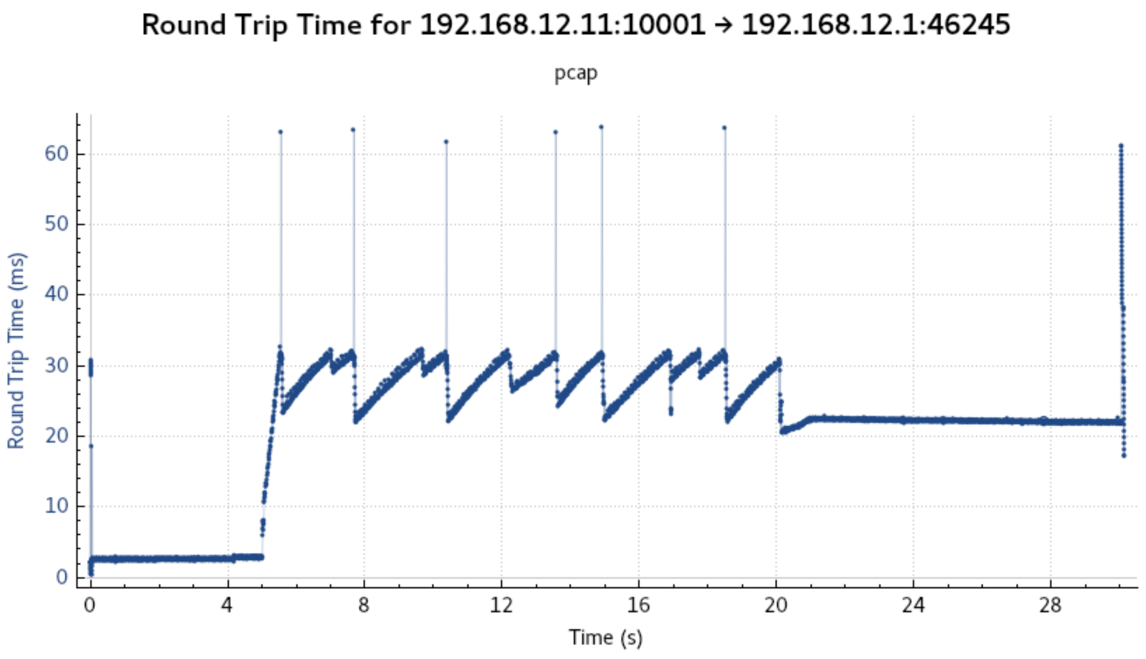

Figure 4.8: TCP round trip time during window scaling test.

try to avoid unnecessary data loss. Due to these algorithms, delay during network congestion caused by connection-oriented protocols is developed not statically but dynamically.

The window scaling test shows us the most interesting and unexpected behavior. The network delay exists even network congestion is finished. When the second TCP stream ends communication, the over-subscription ratio is decreased to 1 and the first TCP stream raises its transmission speed to bottleneck bandwidth. Network should not be congested, but the delay caused by outgoing interface buffer is still high.

These results give us description of protocol stack behavior during congestion. We are able to create suitable test case/test cases, which test protocol stack performance during network congestion.

Chapter 5

Test case for protocol stack in congested network

In this chapter we define test requirements for test scenarios, which will be created for automated testing of the linux protocol stack. The test requirements are based on our network setup during experiments from chapter 4 and we consider the usability of this test scenario in CI testing.

In section 5.2, we discuss several possibilities of implementation of created test scenarios. We based our decision mostly on scalability option of current solution.

5.1 Test case requirements

Test case suitable for CI testing, which tests part of kernel protocol stack we are interested in - especially congestion control algorithms, has to meet these requirements :

- `The test case must simulate network congestion by increasing delay and decreasing available throughput.`
- `The network impairments must not change during the test case.`
- `The test case must use IPv4 and IPv6 protocol.`
- `The test case must test TCP and UDP protocol.`

The test case has to meet these requirements to fit our testing process :

- `The test case must bind testing tool to specific CPU for stability of results.`
- `The test case must use pair of two servers with exactly the same hardware configuration (CPU, MB, NICs).`

5.2 Test case design

In this section we consider possibilities of creating the test case. We focus mostly on implementation of network simulation, network performance measuring tools and network setup used for testing.

5.2.1 Congestion simulation

The first problem is how to create network congestion without using too many servers/devices, necessary for testing congestion control algorithms. Quality of this simulator has the biggest impact to the validity of network congestion model.

This simulator causes delay, which is constant during test case run. This rule applies to throughput limit as well.

Netem

In the Linux operating system community develops a special enhanced traffic queue called `netem` [17]. It can limit transmission speed, add delay to communication and much more. This makes it a great candidate for simulating network congestion using only software.

Attero

The second option for simulating network congestion is to use hardware solution, independent on current version of operating system, which may have some bugs even in netem simulator. The SPIRENT company created a hardware solution called `Attero` for simulating network errors and special conditions. This device is able to create delay and limit transmission speed as well, which satisfies the first requirement. It gives us a scalability option for the next development of more complex test cases in the future.

5.2.2 Testing tool

There are several benchmarks for network performance testing (e.g. netperf, iperf3, ttcp) [1] and many of them meet requirements for testing tool. We decided to use iPerf3 (described in section 4.2) to keep consistency of measurements from chapter 4 and newly created measurements. This tool is well supported and new features are still being developed. The biggest benefit is that the output log can be encoded to JSON format, which is very easy to store and also to load result to be processed later on. It also gives us scalability option for the next test cases created in the future (e.g. iPerf3 can omit the first n seconds of test, use n parallel streams, and other features).

5.2.3 Network setup

For this test case we create special hardware configuration shown in figure 5.1. This setup consists of two exactly the same servers and Attero-X emulator. Whole network environment used for testing is sealed off from other traffic, which may influence the results of tests. This isolation protects the whole lab network from congestion state.

A network interface card should have stable driver in kernel to avoid interference of measurements by defective kernel module. We use 10Gb NIC with ixgbe driver from Intel, which has stable and good performance results[2] in non congested network.

[1]https://wiki.linuxfoundation.org/networking/performance_testing

[2]This argument is based on our performance test of current NIC, but the results are not public.

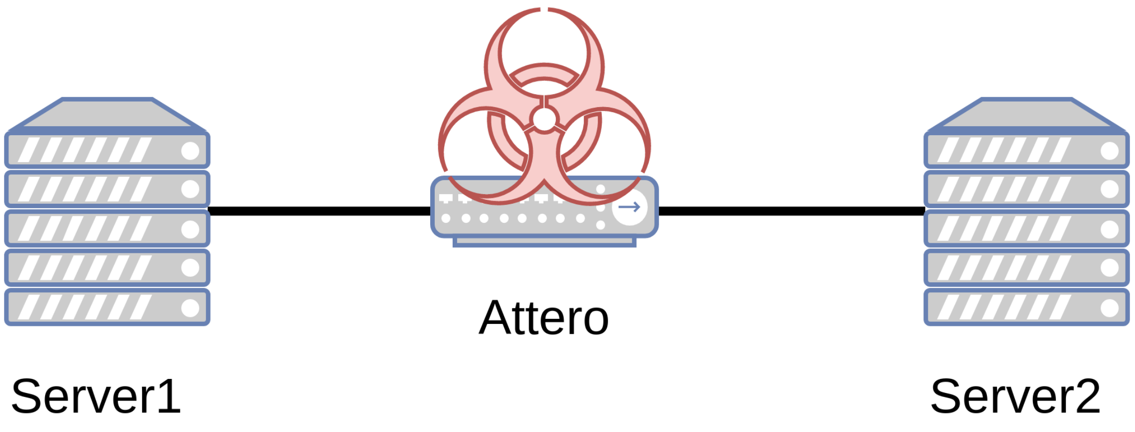

Figure 5.1: Network diagram show connection setup of servers and network simulator.

5.3 Test scenario

The test cases will simulate two different network impairments: delay and bottleneck bandwidth. These errors are defined by tester and can be variably changed. In our newly created test scenario we simulate various over-subscription ratios: 2, 4, 8, which give us bandwidth on 10 Gb link: 5 Gbps, 2.5 Gbps, 1.25 Gbps respectively.

In this scenario we use several delays: 0.3ms, 3ms, 30ms. These delays are based on measurements from chapter 4, where 30ms delay is rounded delay measured in 100 Mbps link. Other delays simulate latency of links with higher speed and the same buffer length ($1Gbps \rightarrow 3ms$, $10Gbps \rightarrow 0.3ms$).

Our idea of testing process is shown in algorithm 3. The designed algorithm iterates through every simulated delay and over-subscription ratio and calls iPerf3 TCP and UDP test case. The test scenario settings are stored in configuration object, where can be found the list of simulated delays, list of simulated over-subscription ratios, list of tested message sizes, number of iperf3 test for one message size and many others.

Algorithm 3: StaticCongestion

```
StaticCongestion (cfg)
    input  : Configuration object of test case
    output: Measured throughput of each iPerf3 stream.
    foreach $delay \in cfg.delays$ do
        foreach $ratio \in cfg.ratios$ do
            SetupAttero(delay, ratio);
            iPerfTest(cfg, tcp);
            iPerfTest(cfg, udp);
        end
    end
```

Algorithm 4 explains how iPerf3 measures the network. The network is already set, when iPerfTest is run, therefore this algorithm do not setup any network impairments. This test iterates through several message sizes and for each message size runs five iperf3 tests to verify the stability of results. Results of each iperf3 test are saved to be processed later.

```
Algorithm 4: iPerfTest
iPerfTest (cfg, stream)
    input  : Configuration object of test case
    output: Measured throughput of each iPerf3 stream.
    res = ResultContainer();
    foreach size ∈ cfg.messsage_sizes do
        foreach run ∈ cfg.runs do
            if stream == tcp then
                iperf3 –client cfg.dst.ip –bind cfg.srcip –time cfg.testduration –len size;
            else
                iperf3 –client cfg.dst.ip –bind cfg.srcip –time cfg.testduration –len size
                  –udp;
            end
            store result of iPerf3 test to res;
        end
    end
    return res;
```

Chapter 6

Implementation

In this chapter we describe CI process implemented for kernel testing used by our team. We use several modules/programs in CI process with key testing project called Beaker. These programs are quickly described in section 6.1.

We implement new test scenario and test case into this CI process based on test scenario design from chapter 5. Implementation details of this test case are in section 6.2 together with description of project responsible for preparing server for test and executing network performance tools.

6.1 Automated testing

Software needs to be tested to maintain the quality and reliability as high as possible. In old software development models (e.g. Waterfall model) the testing phase takes place after development of project requirements. If the testers find a wrong behavior (bug), the software development process steps back into the development phase, fixes the bug and comes back into testing phase. This process is quite slow and extends the period between software releases. The next step in evolution of software development management are Agile methodologies. These iterative methodologies (e.g. Scrum) make the time period between releases shorter and they easily adapt to evolving customer's needs through project development. In agile, the testing phase does not take place after development as in Waterfall model, but testing and development phases exist in parallel.

6.1.1 Continuous integration

The developers merge their work into one repository several times a day to detect integration errors and also to provide the newest progress in their work to testers. This process is called **Continuous integration**. The greatest benefit of this process is earlier detection of possible bugs [8], if tests are run with the new version of project. Continuous integration does not necessarily add more quality assurance to project, but with suitable automated tests can save the time wasted by the integration of code and testing the project manually.

6.1.2 CI testing of kernel performance

Continuous integration process is used in kernel development, which helps to find bad commit/version. A pipeline, that triggers and evaluates the test results, used by our team, is shown in figure 6.1. New kernel version is build by Brew, which informs Jenkins (described

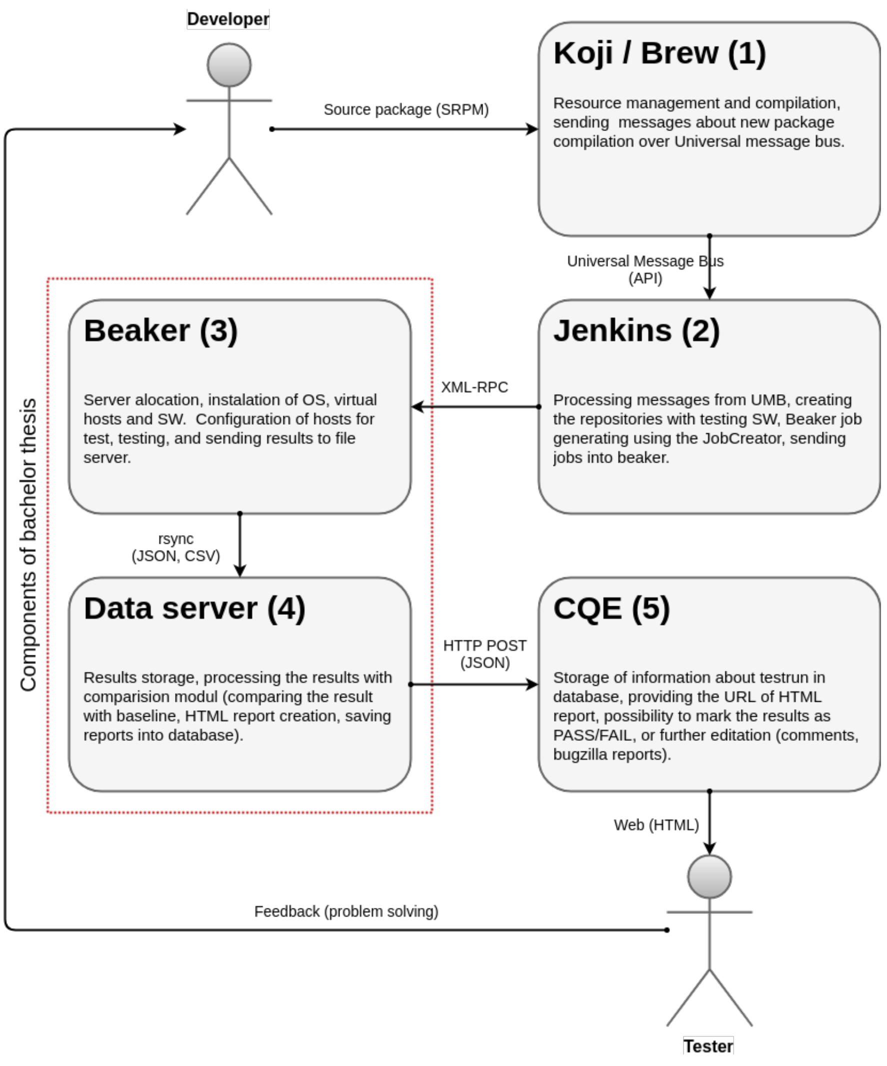
Developer
Source package (SRPM)
Koji / Brew (1)
Resource management and compilation, sending messages about new package compilation over Universal message bus.
Universal Message Bus (API)
Jenkins (2)
Processing messages from UMB, creating the repositories with testing SW, Beaker job generating using the JobCreator, sending jobs into beaker.
XML-RPC
Beaker (3)
Server allocation, instalation of OS, virtual hosts and SW. Configuration of hosts for test, testing, and sending results to file server.
rsync (JSON, CSV)
Data server (4)
Results storage, processing the results with comparision modul (comparing the result with baseline, HTML report creation, saving reports into database).
HTTP POST (JSON)
CQE (5)
Storage of information about testrun in database, providing the URL of HTML report, possibility to mark the results as PASS/FAIL, or further editation (comments, bugzilla reports).
Web (HTML)
Components of bachelor thesis
Feedback (problem solving)
Tester

Figure 6.1: The process of continuous integration testing used in our team [28].

in section 6.1.4) about new version of kernel. Jenkins runs Jobcreator (described in section 6.1.6), which generates test job for each test case. These jobs are submitted to beaker (described in 6.1.5), which performs test case execution. Output data from test cases is stored into data server, where program named reporting (described in section 6.1.7) compares test case result to baseline and creates web report. This report is stored into database (described in section 6.1.8), where it is evaluated whether it passed or failed performance criteria.

6.1.3 Koji/Brew

Koji is system for building packages [13]. It was created by Fedora project. It focuses only on RPM packages. Koji uses Mock[1] to build packages for different architectures and distributions. The build process is independent of the host operating system, which runs Koji.

[1]https://github.com/rpm-software-management/mock/wiki

6.1.4 Jenkins

Jenkins is an open-source project, which helps with automation of CI/CD process. The major component of this project is automation server, which can be used to automate all tasks needed in CI/CD process (e.g. building, testing, delivering or deployment tasks) [1]. This automation server can be easily extended by plugins, which adjust the server to project requirements.

The biggest benefit of this project is that testers can spend more time with test development and results analysis and they can forget about the test orchestration or running the tests manually.

6.1.5 Beaker

Beaker is a tool for automated managing of servers and computers in lab environment. It is mostly used for automated installation of system and task execution. Beaker provides web user interface for maintaining hardware inventory, system provisioning, task scheduling and viewing task results [16].

Beaker has its own TFTP server with available image of operating systems for automated installation. This is provided through PXE[2] during automated installation process. User of Beaker system has to ensure that installed system has correctly set boot order (system should try to boot from network at the first place), otherwise automated installation will not work. Beaker must have access to the management of installed device to be able to power on/off or reboot the current system automatically. When Beaker starts installation process of some system, it powers on this system and creates special configuration file for this system in the TFTP server. Server downloads this file via PXE and starts the installation process according to this file.

Beaker Job

Beaker Job contains recipe set and description system or several systems, where recipe set is being run. System can be specified by several parameters, for example hostname, if recipe should run on specific system, or hardware restrictions, or random system, which satisfies these restrictions.

The recipe set contains one or more recipes, which are run in parallel [16]. Each recipe contains test system specifications (e.g. hostname, number of CPU, NICs, amount of RAM, etc.), specification of distribution and ordered sequence of tasks. Example of a Beaker job for one specific server is shown in listings B.1.

Beaker Task

Beaker recipe task is the smallest unit of work and status (PASS/FAIL) is reported to Beaker server [16]. In tasks, users are able to implement their own executable programs or tests, which will run inside jobs. Beaker provides unified calling methods using `Makefile`. In the beginning of task execution it calls `make build` to install or prepare application for execution. In the next step it calls `make run`, which should execute task. User or tester is responsible for implementing of `build` and `run` dependencies.

[2]Preboot execution environment.

Beaker scheduler

Beaker is responsible for matching the correct system and executes job on that systems. For this purpose Beaker scheduler was created. This module creates simple FIFO queue, which cannot be prioritized (only if jobs match the same system) [16].

Beaker keeps hardware specifications of each system in a database. Based on these specifications scheduler choose proper system for current job.

When scheduler executes job on target system, as the first step it installs test distribution. Then tasks of current job are being run and evaluated sequentially. Each task is saved in Beaker database server and target system installs current task from this database. Beaker project was developed especially for software testing and developers supposed the task was test scenario or test case. Therefore Beaker stores result (PASS/FAIL) of each executed task to database.

6.1.6 JobCreator

JobCreator is our internal tool generating jobs for Beaker system. The job describes the whole automated process, which contains installed distribution, installed and executed tasks and others. This job is stored in XML format. JobCreator is able to submit job directly using remote procedural call, which transfers job encoded in XML via HTTP protocol.

6.1.7 Reporting

The main role of reporting module is to compare several results and create web report, which shows the results graphically. Most of the time it compares just two results : new results from CI test of kernel and baseline. These comparisons are stored in data server and links to these reports are stored in **CQE database** . This module supports generating of results manually for special purposes (e.g. report of several kernels of specific test case, report on demand).

6.1.8 CQE database

CQE is database for links to reports and information about testrun (e.g. rpms, distribution, tags, hostnames of devices). On the top of this database web interface is implemented. It can be used to filter results according to users needs. This database serves to testers and to managers as well. We can filter results, that need to be reviewed, or we can create CQE link with only specific kernel/distribution as a status of our testing.

6.2 Test case implementation

In this section we quickly describe implementation of our testing task to Beaker called `network_perftest`. It is used during the whole testing phase. It setups testing servers and runs sequentially thousands of performance tests with various options. Output of these tests are processed during testing phase and the most important parts are stored for comparison to baseline result.

6.2.1 Testing

For implementation of various network setups our team created python project, responsible for setup servers with variable operating system and many others. For each scenario

Name	Description
SSH keys	We add the same ssh key during setup to enable the access to server without password.
Kdump	Kdump is set to send vmcore of crashed kernel to data server for later analyzes.
Sysctl variables	These variables are set during setup phase.
Packages	List of packages, which will be installed during setup by yum.
Services	Testing program starts/stops various services running on the system (e.g. lldpad, netperf, irqbalance).
Repositories	These repositories contains required packages, therefore are added to system.
IP addresses	Each interface has assigned unique IPv4 and IPv6 address, therefore testing program is able to exactly find correct interface for data stream of performance test.
Paths	Paths is list of pairs (source, destination) of IPv4/IPv6 addresses, which describe route of testing stream for performance measurement tool.
Test cases	List of test cases executed during this scenario.
Logging	Several show commands are executed in the end of testing phase. They are stored together with test results for later analyzes if problem occurs.

Table 6.1: The most important system settings, which are adjusted during setup.

and each server used by this scenario we create special configuration object, which stores information needed for system setup. Quick example of the most important adjustments are shown in table 6.1. All the set adjustments are persistent and after setup phase the system is rebooted and no additional adjustments are set.

The test case has configuration object as well. It specifies: paths[3] for performance tests, duration of single performance test, tested message sizes and number of executed performance test for each message size. This checks deviation of measured results.

The testing phase takes place after reboot. The server pair is prepared according to test scenario needs, therefore the testing phase just runs performance tests. It reads test cases from configuration and starts to perform test cases sequentially. Each test case iterates through: paths, message sizes and number of test run and run performance test for each combination of values. Test case object reads these values from configuration and it may be different across all test cases.

Performance test is executed by using special object. It abstracts creation of shell command and execution of this command in new process running in the system. The process is created by using `subprocess` python library, which allows spawning new process, catch the return code and connect the input/output/error pipes[4]. All necessary variables are set in the main test case cycle and test object is executed. When performance test finishes the execution, output is parsed by the performance test object itself and returns data in dictionary.

[3]Path is pair of source and destination address.
[4]https://docs.python.org/2/library/subprocess.html

The test case creates tree structure of results in python dictionaries, which are stored together with the next test case results in XML format in the end of testing. The testing program adds meta information about testing environment and runs all logging commands, which will be packed to results as well. The final result directory contains following files:

- **store.xml** - results of performance tests
- **meta.xml** - information about system under test
- **attachments.tgz** - logs from various services and programs

6.2.2 Synchronization

The installation process usually takes different time. If one server of testing pair is installed and set earlier, testing process starts earlier as well. This causes several faulty measurements in the beginning of testing. To avoid this situation our team developed a synchronization tool, which ensures the both systems are ready before the testing phase starts. This tool creates synchronization barrier, which means both testing servers ask the synchronization server if the second server is ready to test.

6.2.3 Spirent Attero-X

Attero-X is a key component of `StaticCongestion` test scenario. It simulates bottleneck bandwidth and delay in the network. This test case is added into automated CI testing, therefore configuration of Atterro-X has to be automated as well. Calnex and Spirent companies developed attero python library, which makes API for communication with Attero-X. This library was modified by our team to fit into CI process. We created several abstract functions, which wrap the creation of Attero-X commands and network communication with program that controls Attero-X. In `StaticCongestion` scenario, Attero-X is configured before running test case.

6.2.4 Features implemented for this thesis

Creation of `StaticCongestion` test scenario requires implementing several python classes and objects. In the first step, we had to create class, which wraps calling of iPerf3 performance tool. Usage of this class instance is described in section 6.2.1. This object has the same attributes as iPerf3 command line arguments. During command line creation, responsible method just takes object's attributes and adds these attributes to command in required format.

In the second step, we needed to create configuration objects for designated server pair with configuration for `StaticCongestion` test scenario. This configuration object copies almost all settings from `All NIC's drivers` test scenario. Only differences are path list, which contains only single route traveling through Attero-X, and test scenario list, which contains iPerf3 TCP and UDP StaticCgonestion test scenarios.

In the last step, we had to implement test scenario classes. For better maintainability we created an abstract class called StaticCongestion, that just prepares Attero-X according to scenario configuration object and calls test case method, which is not implemented. To test TCP and UDP protocol we created two classes (one for each protocol), which inherit from StaticCongestion class and implement test case method. When test driver calls run method of TCP or UDP StaticCongestion, it prepares Attero-X and executes test case measurements.

Chapter 7

Evaluation of testing scenario results

In this chapter we discuss reports created from test scenario called StaticCongestion. We had run this scenario four times with different distributions and kernels to check if performance of protocol stack changed during the development. Kernel versions and distributions used for testing are shown in table 7.1. We used these distributions to check performance difference between many patches. In CI testing the amount of tested patches is much lower.

Generated web reports are very large, therefore are not part of this printed document and can be found in included CD.

Distribution	**kernel version**
RHEL-6.7	kernel-2.6.32-573.el6.x86_64
RHEL-6.9	kernel-2.6.32-696.el6.x86_64
RHEL-7.3	kernel-3.10.0-514.el7.x86_64
RHEL-7.5	kernel-3.10.0-862.el7.x86_64

Table 7.1: Kernel versions in various RHEL GA distributions.

7.1 RHEL-6.7 vs RHEL-6.9

These two distributions are different only in version of installed packages, which contain kernel as well. To check kernel versions of these distributions see table 7.1. These two kernels differs in 123 versions and too many patches.

7.1.1 TCP

In the TCP test cases, performance of these distributions is very similar. In test cases with $30ms$ delay, results do not achieve bottleneck bandwidth. The Bandwidth-delay product is probably too large and decreasing TCP throughput. The test result of this measurement is visible in figure 7.1. Acceptable regression is no more than 5% and this report shows that the newer RHEL-6 kernel passes the test.

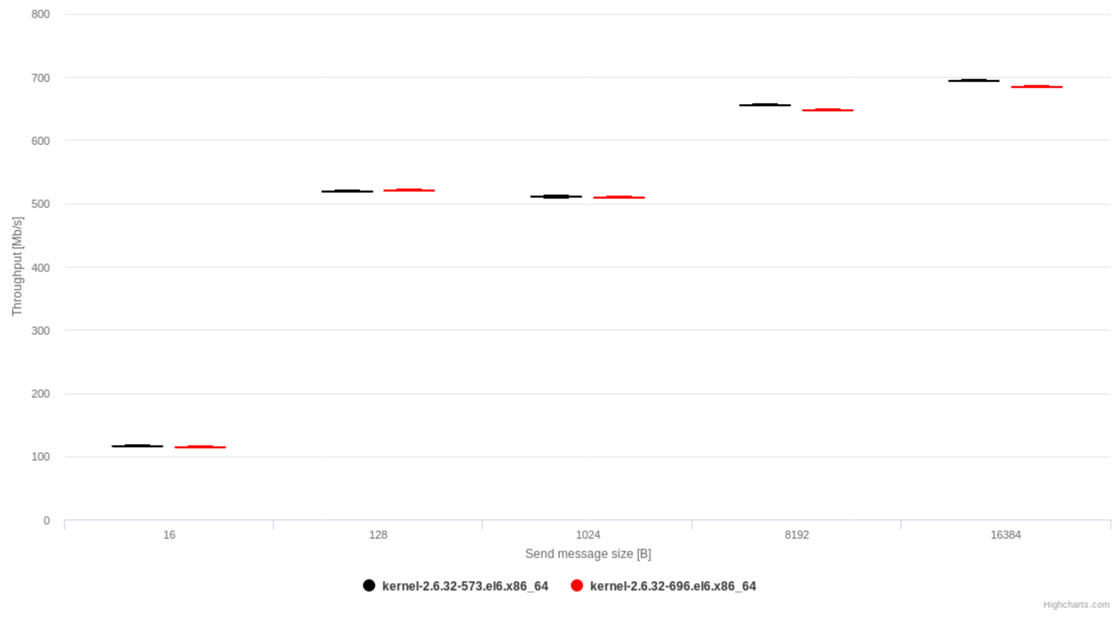

Figure 7.1: Low TCP throughput due to high bandwidth-delay product in test case: Network impairment emulation above IPv4 over Intel Corporation 82599ES[delay: 30 ms, bottleneck: 2500000 kbps]

7.1.2 UDP

Performance results of UDP protocol show very low throughput in test cases various oversubscription ratios. It looks like the delay has no performance effect on throughput. In non-processed results we can see high percentage of lost data (50% in small message size without, 95±2in higher message size). UDP is not able to reconstruct fragmented datagram if one part is missing, therefore amount of lost data is enormously high. It is clearly visible that UDP protocol suffers during network congestion, because it has no congestion control mechanism.

7.2 RHEL-7.3 vs RHEL-7.5

These distributions differ only in version of packages. Kernel used in these distributions is based on fork of 3.10 upstream kernel. Full kernel versions of these distributions are shown in table 7.1.

7.2.1 TCP

In TCP stream results with $0.3ms$ delay, we can see similar performance between these distributions on higher message sizes. In lower message sizes, performance is slower, which might be caused by NIC driver.

In TCP stream results with $3ms$ delay and $5Gbps$ bottleneck bandwidth, there is regression at all message sizes. Test results in chart are shown in figure 7.2. Performance with lower bottleneck bandwidth looks similar and passes testing criteria.

RHEL7 has similar performance as RHEL6 with 30ms delay. Long fat networks have enormous bandwidth-delay products, therefore congestion window variable is probably set to the highest value. This stops raising congestion window, which is proportional to throughput (equation 4.4 shows evaluation of throughput). Therefore TCP stream does not achieve even bottleneck bandwidth transmission rate.

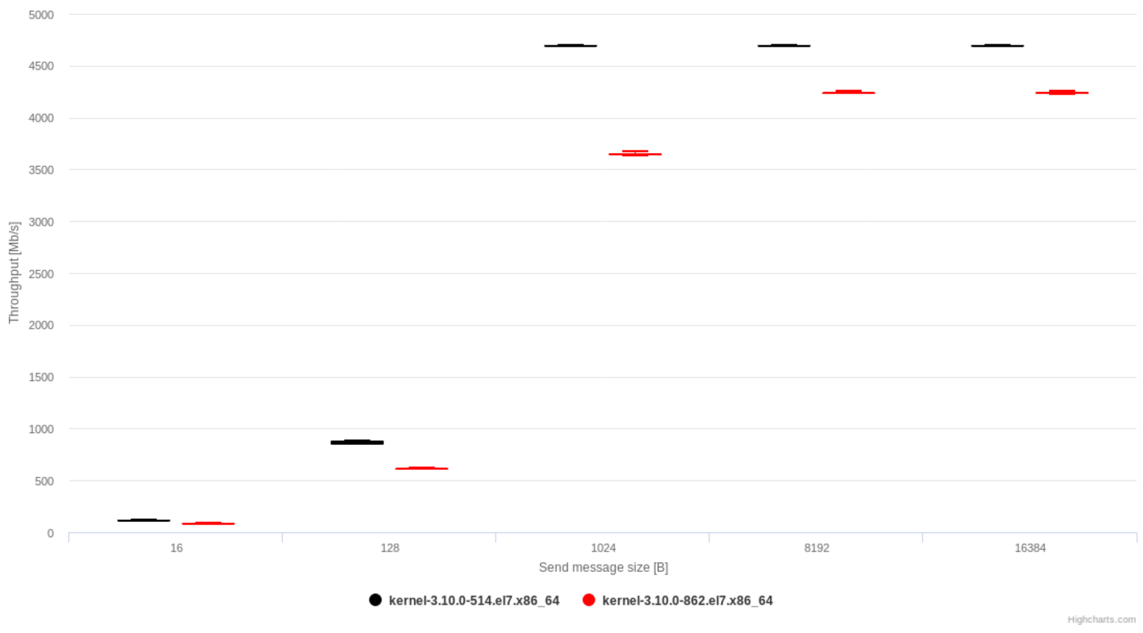

Figure 7.2: Regression in TCP test: Network impairment emulation above IPv4 over Intel Corporation 82599ES[delay: 3 ms, bottleneck: 5000000 kbps]

7.2.2 UDP

UDP stream test shows similar results as RHEL6. Network throughput is also very low in simulated networks with various delays and over-subscription ratios. Our thoughts about this behavior are in section 7.1.2.

7.3 Test scenario conclusion

Designed test cases are successfully implemented in development branch of our testing project. Web reports of experiments results bring more detailed information about protocol stack behavior during network congestion.

Results from TCP test case are stable (low standard deviation) and expected. They show us behavior of congestion control algorithms in network during congestion and in long fat networks.

From results of UDP test case is clearly visible the bad design of test case. Methods applied on TCP test case can not be shared with UDP test case. iPerf3 UDP test usually uses message size equal to MTU or 1460B. Very low message size consumes lot of CPU time and higher message sizes has to be fragmented (it is offloaded to NIC). If a fragment is lost the whole UDP datagram is lost.

Chapter 8

Conclusion

The goal of this thesis was to study and understand networks during congestion and to create suitable test scenario for testing Linux kernel protocol stack under these circumstances. Red Hat customers using Red Hat Enterprise Linux operating system want to be sure about network performance of this product. They create hundreds of VMs or containers inside dozens of servers, all connected into one switch. This switch creates bottleneck, where congestion may occur.

For understanding the behavior of each network device, we created lab environment simulating over-subscribed network in datacenter with several file servers. We simulated scenario with many users downloading files from file servers at one time, which caused network congestion. In our lab we had two servers simulating users and file servers. Total throughput of file server could be 6 Gbps, but user server was connected to switch using 100Mbps connection. It is obvious that congestion occurred in outgoing interface buffer in the switch. In this network we observed statistics of the congested interface buffer and throughput of UDP and TCP protocol. On servers we ran iPerf3 performance tool and ping to observe throughput and latency. Congestion window was observed by special kernel module reading this variable from socket descriptor when data arrived to server. In TCP protocol we were interested especially in congestion control algorithms, how efficiently they were able to utilize congested network. These observations and measurements done in network during congestion are described in chapters 3 and 4.

The most important outcome of this thesis are test scenario designs, which test Linux kernel protocol stack during congestion with various delays and over-subscription ratios. We considered two possibilities of congestion simulation: software and hardware method. We chose hardware method, because we already had network emulation device (Attero-X).

We implemented several python modules to enhance existing testing project with new test scenarios. These test scenarios firstly set up network, servers and network emulator. After this phase they ran iPerf3 performance tests using TCP and UDP protocol. Results from iPerf3 tests are compared to baseline to detect regression.

To see the asset of new test scenarios we ran several OS distributions with various kernel versions and generated web reports with comparisons. Results of TCP test case are as we expected and stable. This test scenario will be added into CI testing of protocol stack and will be run several times a day. Results of UDP test case show very low performance caused by bad design of UDP test case. It is not possible to use the same testing configuration as in TCP test case.

8.1 Possible enhancement to this thesis

This topic offers another possibilities for further research, which we had to leave aside, because it would exceed the scope of this thesis.

- **UDP test scenario fix** - To find proper configuration (message sizes, over-subscription ratios, delays) with low standard deviation of test results. Test scenario results should show adequate throughput of UDP protocol.
- **Observe network with routers configured with RED** - Random early detection may significantly decrease delay in the intermediary device. Probability function decides packet loss, when queue is partially filled. This informs congestion control algorithm about congestion, which starts to lower transmission speed. Earlier packet drop may cause lower overall throughput. This type of network should be observed to understand protocol stack behavior in this network and to check the performance impact.
- **Observe network with enabled load-balancing** - Routing protocols may find more than one equal route to the same destination. Routers are able to use more routes to the same destination by using load-balancing feature, which can use round robin algorithm to choose the path. This algorithm does not guarantee the usage of the same path for the same data flow, therefore out of order delivery may occur.
- **Create Dynamic Congestion test scenario** - Network congestion is not constant event. Due to the reaction of congestion control algorithms it is a recurring event. Therefore over-subscription ratio and delay change in time. New Dynamic Congestion test scenario should change delay and bottleneck bandwidth during the test according to configuration pattern. This should create the same network behavior for each test.

Bibliography

[1] *Jenkins User Documentation.* [Online; visited Mar 9, 2018].
Retrieved from: https://jenkins.io/doc/

[2] *A802.1D-2004 - IEEE Standard for Local and metropolitan area networks: Media Access Control (MAC) Bridges.* IEEE. 2004. ISBN 9780738139821.

[3] Jon Dugan and Seth Elliott and Bruce A. Mah and Jeff Poskanzer and Kaustubh Prabhu: iPerf3.
Retrieved from: https://iperf.fr/

[4] Timo Dorr: tcp_probe_fixed.
Retrieved from: https://github.com/Dynalon/tcp_probe_fixed

[5] Abdou, A.; Matrawy, A.; van Oorschot, P. C.: *Accurate One-Way Delay Estimation With Reduced Client Trustworthiness.* IEEE Communications Letters*. vol. 19. 2015: pp. 735–738.*

[6] Abed, D.; Ismail, M.; Jumari, K.: A Survey on Performance of Congestion Control Mechanisms for Standard TCP Versions. vol. 5. 12 2011: pp. 1345–1352.

*[7] Abík, A.: Random Early Detection (RED). 2015. [Online; visited Jan 13, 2018].
Retrieved from: http://network.jecool.net/random-early-detection-red/*

*[8] Azeri, I.: What is CI/CD? [Online; visited Mar 11, 2018].
Retrieved from: https://www.mabl.com/blog/what-is-cicd*

*[9] Beal, V.: The 7 Layers of the OSI Model. September 1999. [Online; visited Jan 13, 2018].
Retrieved from: https://www.webopedia.com/quick_ref/OSI_Layers.asp*

*[10] Bekman, S.: What is propagation delay? (Ethernet Physical Layer). [Online; visited Jan 13, 2018].
Retrieved from: http://stason.org/TULARC/networking/lans-ethernet/3-11-What-is-propagation-delay-Ethernet-Physical-Layer.html*

*[11] Chen, J.; Hu, C.; ; et al.: Self-Tuning Random Early Detection Algorithm to Improve Performance of Network Transmission. 2010. [Online; visited Jan 13, 2018].
Retrieved from:
http://downloads.hindawi.com/journals/mpe/2011/872347.pdf*

[12] Cisco Systems, I.: Oversubscription and Density Best Practices. 2015. [Online; visited Jan 17, 2018].

Retrieved from: https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/storage-networking-solution/net_implementation_white_paper0900aecd800f592f.html

[13] Fedora community: Koji. 2014. [Online; visited Apr 24, 2018].
Retrieved from: https://fedoraproject.org/wiki/Koji

[14] Frenkel, D.: Introduction to Monte Carlo methods. vol. 23. 01 2004.

[15] Gerald Combs: Wireshark.
Retrieved from: https://www.wireshark.org/

[16] Hat, R.: Beaker. 2011. [Online; visited Apr 23, 2018].
Retrieved from: https://beaker-project.org/

[17] Hemminger, S.; Ludovici, F.; Pfeifer, H. P.: tc-netem (8) - Linux Man Pages. 2011. [Online; visited Apr 16, 2018].
Retrieved from:
https://www.systutorials.com/docs/linux/man/8-tc-netem/#lbAU

[18] Hewlett-Packard Company: Netperf.
Retrieved from: https://hewlettpackard.github.io/netperf/

[19] Hucaby, D.: CCNP Routing & Switching SWITCH 300-115. Cisco Press. 2015. ISBN 1587205602, 9781587205602.

[20] Huston, G.: Best Efforts Networking. July 2001. [Online; visited Jan 13, 2018].
Retrieved from: https://www.potaroo.net/ispcol/2001-09/2001-09-best.pdf

[21] Injong Rhee, Lisong Xu: CUBIC: A New TCP-Friendly High-Speed TCP Variant . [Online; visited Apr 3, 2018].
Retrieved from: http://www4.ncsu.edu/~rhee/export/bitcp/cubic-paper.pdf

[22] Mitchell, B.: The Layers of the OSI Model Illustrated. [Online; visited Jan 13, 2018].
Retrieved from:
https://www.lifewire.com/layers-of-the-osi-model-illustrated-818017

[23] Postel, J.: TRANSMISSION CONTROL PROTOCOL. RFC 793. RFC Editor. September 1981.
Retrieved from: http://www.rfc-editor.org/rfc/rfc793.txt

[24] Postel, J.: TCP congestion control. RFC 5681. RFC Editor. September 2009.
Retrieved from: http://www.rfc-editor.org/rfc/rfc5681.txt

[25] Rhee, I.; Xu, L.; Ha, S.; et al.: CUBIC for Fast Long-Distance Networks. RFC 8312. IETF. February 2018.
Retrieved from: https://tools.ietf.org/html/rfc8312

[26] Triplett, J.: Is Your ISP Limiting or Shaping Your Traffic? 2015. [Online; visited Jan 13, 2018].
Retrieved from: http://www.ninjasysadmin.com/is-your-isp-limiting-or-shaping-your-traffic/

[27] Welzl, M.: Network Congestion Control: Managing Internet Traffic. John Wiley & Sons. 2005. ISBN 0470025298, 9780470025291.

[28] Šabart, O.: TESTING OPEN VSWITCH AND DPDK. Master's Thesis. Brno University of Technology. Brno. 2017.

Appendix A

Wireshark charts

In this chapter there is shown UDP send algorithm during transmission captured by wireshark.

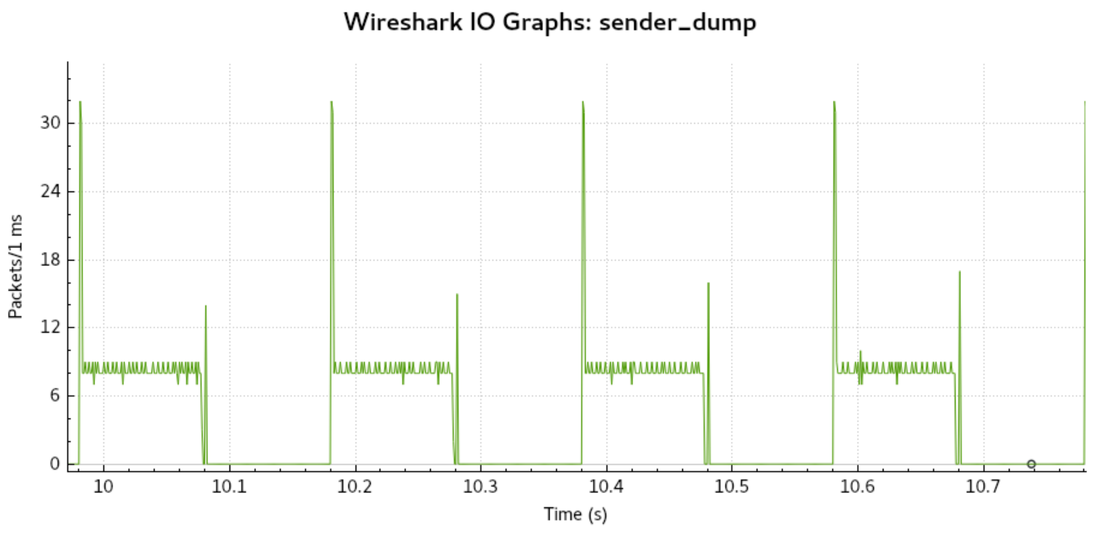

Figure A.1: Sending UDP stream by iPerf3.

Appendix B

Beaker examples

Listings B.1 shows beaker job just for one server, otherwise the example will be very long.

```
<?xml version='1.0' encoding='ASCII'?>
<job user="atomasov" group="perf-test" retention_tag="60days">
<whiteboard>**NET** - *RHEL-7.5 (CI Kernel: ‘None‘, Family: ‘RHEL7‘)* - **
    ↪ StaticCongestion** - hosts=[[brusic1.slevarna.tpb.lab.eng.brq.redhat
    ↪ .com](https://beaker.engineering.redhat.com/view/brusic1.slevarna.
    ↪ tpb.lab.eng.brq.redhat.com), [brusic2.slevarna.tpb.lab.eng.brq.
    ↪ redhat.com](https://beaker.engineering.redhat.com/view/brusic2.
    ↪ slevarna.tpb.lab.eng.brq.redhat.com)], tag=‘None‘, devel=‘False‘
---
</whiteboard>
<recipeSet priority="Normal">
<recipe kernel_options="" kernel_options_post="" ks_meta="beah_no_ipv6"
    ↪ role="" whiteboard="[brusic1.slevarna.tpb.lab.eng.brq.redhat.com](
    ↪ https://beaker.engineering.redhat.com/view/brusic1.slevarna.tpb.lab.
    ↪ eng.brq.redhat.com)">
<watchdog panic="None"/>
<packages>
<package name="pciutils"/><package name="screen"/><package name="tcpdump"/>
    ↪ <package name="iptraf-ng"/><package name="tmux"/><package name="mc"/
    ↪ ><package name="psmisc"/><package name="vim"/><package name="wget"/>
    ↪ <package name="lldpad"/><package name="tuned"/><package name="rsync"
    ↪ /><package name="xterm"/>
</packages>
<ks_appends/>
<repos/>
<distroRequires>
<and>
<distro_name op="=" value="RHEL-7.5"/>
<distro_variant op="=" value="Server"/>
<distro_arch op="=" value="x86_64"/>
</and>
<distro_virt op="=" value=""/>
</distroRequires>
```

```
<hostRequires force="brusic1.slevarna.tpb.lab.eng.brq.redhat.com"/>
<partitions><partition fs="xfs" name="/boot" size="4" type="part"/></
    ↪ partitions>
<task name="/distribution/install" role="STANDALONE"><params/></task>
<task name="/distribution/crashes/enable-abrt" role="STANDALONE"><params/><
    ↪ /task>
<task name="/performance/perf_synchronization" role="STANDALONE"><params><
    ↪ param name="SYNC_ARGS" value="--server netperf-services.slevarna.tpb
    ↪ .lab.eng.brq.redhat.com --port 8000"/></params></task>
<task name="/performance/libres" role="STANDALONE"><params/></task>
<task name="/performance/configure_attero" role="STANDALONE"><params><param
    ↪  name="ATTERO_ACTION" value="stop"/></params></task>
<task name="/performance/network_perftest" role="STANDALONE"><params><param
    ↪  name="NETWORK_PERFTEST_ARGS" value="--meta Family RHEL7 --meta
    ↪ Workflow compose setup StaticCongestion"/></params></task>
<task name="/performance/network_perftest" role="STANDALONE"><params><param
    ↪  name="NETWORK_PERFTEST_ARGS" value="--meta Family RHEL7 --meta
    ↪ Workflow compose run --sync --scenario NetperfTCPSanity
    ↪ StaticCongestion"/></params></task>
<task name="/distribution/utils/reboot" role="STANDALONE"><params/></task>
<task name="/performance/network_perftest" role="STANDALONE"><params><param
    ↪  name="NETWORK_PERFTEST_ARGS" value="--meta Family RHEL7 --meta
    ↪ Workflow compose run --sync --submit StaticCongestion"/></params></
    ↪ task>
</recipe>
</recipeSet>
</job>
```

Listing B.1: Example of Beaker Job in XML for one server only.

Appendix C

CD Contents

Included CD corresponds to this tree structure:

```
/
├── BeakerJob/ ........................ Beaker job example
├── Reports/ .......................... Generated web reports
│   ├── RHEL-6.7-vs-RHEL-6.9
│   └── RHEL-7.3-vs-RHEL-7.5
├── Results/ .......................... XML data from test
├── README.txt ........................ Instructions for running test scenarios
├── codes/ ............................ source codes
│   ├── configure_attero/ ............. Python library for setting Attero-X
│   ├── libres/ ....................... Library for storing test results.
│   ├── reporting/ .................... Tool for generating web reports
│   ├── sequence-checker/ ............. Program for checking data loss and reorder error
│   └── testing/ ...................... Beaker task for network testing
└── latex/ ............................ Source code of this thesis
```