

DEPARTMENT OF INFORMATICS

Master Thesis

MultiPath TCP-communication (in NorNet Core)

By: Kristian B. Ingebretsen and Daniel Selvik

Supervisor: Øyvind Ytrehus

June 1, 2016

Preface

This thesis has been written in co-operation between *Kristian B. Ingebretsen* and *Daniel Selvik*, and is a result of our work and research in the period of time between August 2015 and June 2016. The work carried out is the final step for obtaining the *Master of Science* (MSc) degree in Informatics from The University of Bergen. All textual content is co-produced using *ShareLaTeX*, an online L^AT_EX editor, which have allowed for real-time collaboration. This has made it easier for both to contribute, make suggestions and discuss matters during the writing process.

The research question in this thesis was formulated together with our supervisor, *prof. Øyvind Ytrehus*, to whom we would like to express our deepest gratitude. He introduced us to the topic and gave us the opportunity to conduct our master thesis at the Department of Informatics. We are also grateful for his invaluable guidance and engagement throughout the year.

Furthermore, we would like to thank *Simula Research Laboratory* for providing us with access to the NORNET testbed. A special thanks to *Ahmed Elmokashfi* and *Thomas Dreibholz*, for answering our inquiries and for dedicating their time to help us.

Bergen, 1st of June 2016

Kristian B. Ingebretsen

Daniel Selvik

Abstract

Technology is constantly evolving, and we are currently witnessing a digital revolution with a tremendous growth of interconnected devices. The scale of the Internet and the amount of transported data is constantly increasing. The need for reliability is also increasing, as users are constantly on the move, and are more dependent on cloud services. Utilizing the connectivity redundancy of devices is an interesting approach to deal with this increasing resource demand. In addition to ISPs and data centers implementing redundancy, lately we have also seen that end-user devices also do the same. The term *multihoming* describes the practice of being connected to multiple networks simultaneously. However, the most used transport protocols today, *TCP* and *UDP*, are not able to effectively take advantage of multihoming, as they are single-path transport protocols.

In this thesis we will explore the newly proposed *MultiPath TCP* protocol, an extension of *TCP*, which enables the concurrent use of multiple network interfaces. This makes it possible for one data stream to be transferred over multiple paths on the Internet. In order to evaluate the performance of this protocol, we will test it according to its design goals set by the *Internet Engineering Task Force*, in the NORNET CORE testbed.

Contents

Preface	i
Abstract	i
List of Figures	viii
List of Tables	x
Listings	xii
Acronyms	xiv
1 Introduction	1
1.1 Initial Motivation	2
1.2 Objective	3
1.3 Thesis Outline	4
2 Background	5
2.1 The TCP/IP Protocol Suite	5
2.1.1 Architecture	6
2.1.1.1 Application Layer	8
2.1.1.2 Transport Layer	9
2.1.1.3 Network Layer	10
2.1.2 Encapsulation	11
2.2 Transmission Control Protocol (TCP)	13
2.2.1 Important Concepts	14
2.2.2 The TCP Window Principle	15
2.2.3 TCP Segment Format	17
2.2.4 TCP Congestion Control Algorithms	18
2.2.4.1 Slow Start	19
2.2.4.2 Congestion Avoidance	20
2.2.4.3 Fast Retransmit	21

2.2.4.4	Fast Recovery	22
3	Multihoming: State of the Art	24
3.1	Resource Pooling Principle	25
3.2	Application Layer	26
3.3	Link Layer	27
3.4	Network Layer	28
3.4.1	Mobile IP (MIP)	28
3.4.2	Site Multihoming by IPv6 Intervention (Shim6)	30
3.4.3	Host Identity Protocol (HIP)	31
3.5	Transport Layer	33
3.5.1	Stream Control Transmission Protocol (SCTP)	34
3.5.1.1	Basic SCTP Features	34
3.5.1.2	SCTP Multistreaming Feature	35
3.5.1.3	SCTP Multihoming Feature	36
4	MultiPath TCP	37
4.1	Design Goals	38
4.1.1	Functional Goals	38
4.1.2	Compatibility Goals	39
4.1.2.1	Application Compatibility	39
4.1.2.2	Network Compatibility	40
4.1.2.3	Compatibility With Other Network Users	41
4.1.2.4	Security Goals	42
4.1.2.5	Congestion Control Algorithm Goals	42
4.2	Terminology	43
4.3	Protocol Operation	44
4.3.1	MPTCP Options	44
4.3.2	Connection Establishment	45
4.3.3	Starting a New Subflow	47
4.3.4	Exchange of Data	48
4.3.5	Prioritizing of Subflows	49
4.3.6	Closing a Connection	51
4.3.7	Coupled Congestion Control Algorithm	52
4.4	Failure Handling	55

4.4.1	Middleboxes	55
5	NORNET CORE: A Multihomed Research Testbed	57
5.1	The Design	57
5.2	The Implementation	61
5.2.1	Testbed Management	61
5.2.2	The Sites	61
5.2.3	Tunneling Setup	62
5.2.4	Addresses in NORNET CORE	63
5.2.5	Accessing the Testbed Slivers	63
5.2.6	Virtualization	66
5.3	The NORNET MPTCP Implementation	66
5.3.1	Installation	67
5.3.2	Configuration	67
5.3.2.1	Congestion Control	68
5.3.2.2	Path Manager	69
5.3.2.3	Scheduler	70
6	Methodology and Results	72
6.1	Expectations	72
6.2	Evaluation Tools	73
6.2.1	Gathering Experimental Data	75
6.3	Analysis of Packet Capture	79
6.4	Experiments and Results	84
6.4.1	MPTCP Subflow Analysis	85
6.4.1.1	Subflow Routing	87
6.4.2	Throughput	88
6.4.2.1	MPTCP vs. TCP Throughput, Independent Flows	88
6.4.3	Fairness	93
6.4.3.1	MPTCP vs. TCP, Two Concurrent Flows	93
6.4.3.2	MPTCP vs. TCP, Four Concurrent Flows	96
6.4.4	Latency	99
6.4.5	Connection Handover	107
6.4.6	Congestion Control Algorithms	108
6.4.7	Schedulers	110

6.5	Summary and Evaluation	111
7	Conclusion and Future Work	114
7.1	Conclusion	114
7.2	Future Work	115
	Bibliography	117
	Appendix A NetPerfMeter Vector Parser	123

List of Figures

2.1	TCP/IP Architecture and Encapsulation	11
2.2	Overview of a TCP Connection	14
2.3	The TCP Window Principle	16
2.4	The TCP Segment Format	17
2.5	The TCP Options Format	18
2.6	TCP Fast Retransmit	22
3.1	Resource Pooling Principle	25
3.2	Mobile Host Sending and Receiving Data	29
3.3	Architectural Comparison of IP and HIP	32
4.1	Comparison of TCP and MPTCP Protocol Stack	38
4.2	The Traditional Internet Architecture	41
4.3	The Real Internet Architecture	41
4.4	MPTCP Connection and Subflow Establishment	46
4.5	MPTCP Subflow Priority System	50
4.6	MPTCP Shared Bottleneck Problem	52
5.1	NORNET CORE Architecture	59
5.2	NORNET Tunneling	60
5.3	NORNET CORE Site Schematic View	62
5.4	NORNET Gatekeeper	64
5.5	NORNET Sliver	65
6.1	NetPerfMeter Passive Side	75
6.2	NetPerfMeter Active Side	77
6.3	Tcptrack During an MPTCP Transfer	79

6.4	Wireshark screenshot - MP_CAPABLE signal	80
6.5	Wireshark screenshot - ADD_ADDR signal	81
6.6	Wireshark screenshot - MP_JOIN signal	82
6.7	Wireshark screenshot - DSS signal	83
6.8	Wireshark screenshot - The Closure of a Connection	84
6.9	Subflow throughput analysis: floeibanen to bymarka	86
6.10	Independent throughput: lungegaardsvannet to nordlys	89
6.11	Independent throughput: rennesoey to floeibanen	90
6.12	Independent throughput: nordlys to kettwig	92
6.13	Fairness, two concurrent flows: rennesoey to lungegaardsvannet	94
6.14	Fairness, two concurrent flows: nordberg to kettwig	95
6.15	Fairness, four concurrent flows: rennesoey to lungegaardsvannet	97
6.16	Fairness, four concurrent flows: lungegaardsvannet to kettwig	98
6.17	Latency: ekeberg to bymarka (TCP: Uninett→Uninett)	100
6.18	Latency: ekeberg to bymarka (MPTCP: Uninett→Uninett subflow)	101
6.19	Latency: ekeberg to bymarka (MPTCP: All subflows)	102
6.20	Latency: nordlys to lungegaardsvannet (TCP: Uninett→Uninett)	103
6.21	Latency: nordlys to lungegaardsvannet (MPTCP: All subflows)	104
6.22	Latency: rennesoey to bymarka	105
6.23	Latency: rennesoey to lungegaardsvannet (MPTCP: All subflows)	106
6.24	Connection Handover: One Backup Link	108
6.25	MPTCP Congestion Controls Performance	109
6.26	Scheduler: LowRTT vs. roundrobin	111

List of Tables

4.1	MPTCP Option Subtypes	45
5.1	NORNET CORE Deployment Status, April 2016	58
6.1	Subflow throughput analysis: floeibanen to bymarka	86
6.2	Independent throughput: lungegaardsvannet to nordlys	89
6.3	Independent throughput: rennesoey to floeibanen	91
6.4	Independent throughput: nordlys to kettwig	92
6.5	Fairness, two concurrent flows: rennesoey to lungegaardsvannet	94
6.6	Fairness, three concurrent flows: rennesoey to lungegaardsvannet	94

Listings

6.1	Example of <i>traceroute</i> from <i>rennesoey</i> to <i>nordberg</i>	88
-----	---	----

List of Acronyms

3G Third Generation

4G Fourth Generation

ACK Acknowledgement

ADSL Asymmetric Digital Subscriber Line

API Application Programming Interface

ARP Address Resolution Protocol

Balia Balanced Linked Adaption Congestion Control Algorithm

CMT Concurrent Multipath Transfer

CRC Cyclic Redundancy Check

DCCP Datagram Congestion Control Protocol

DNS Domain Name System

DSN Data Sequence Number

DSS Data Sequence Signal

FTP File Transfer Protocol

GRE Generic Routing Encapsulation

HIP Host Identity Protocol

HMAC Hash Message Authentication Code

HTTP Hypertext Transfer Protocol

IANA Internet Assigned Numbers Authority

ICMP Internet Control Message Protocol

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IMS IP Multimedia Subsystem

IP Internet Protocol

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

IRTF Internet Research Task Force

ISP Internet Service Provider

LACP Link Aggregation Control Protocol

LAN Local Area Network

LIA Linked Increase Algorithm

LTE Long-Term Evolution

LXC Linux Containers

MIP Mobile IP

MIPv6 Mobile IP version 6

MPRTP MultiPath Real-time Transmission Protocol

MPTCP MultiPath Transmission Control Protocol

MSS Maximum Segment Size

MTU Maximum Transmission Unit

NAT Network Address Translation

NIC Network Interface Controller

OLIA Opportunistic Linked Increase Algorithm

OSI Open Systems Interconnection

P2P Peer To Peer

PLC PlanetLab Central

REAP Reachability Protocol

RFC Request For Comments

RTA Real-Time Application

RTP Real-time Transmission Protocol

RTT Round-Trip-Time

SCP Secure Copy

SCTP Stream Control Transmission Protocol

SDP Session Description Protocol

SHA Secure Hash Algorithm

Shim6 Site Multihoming by IPv6 Intermediation

SIP Session Initiation Protocol

SMTP Simple Mail Transfer Protocol

SSH Secure Shell

TCP Transmission Control Protocol

TLD Top-level Domain

UCL Université catholique de Louvain

UDP User Datagram Protocol

ULID Upper Layer Identifier

VoIP Voice over IP

VPN Virtual Private Network

WAN Wide Area Network

Wi-Fi Equivalent to WLAN (Wireless Local Area Network)

wVegas Weighted Vegas (Delay-based Congestion Control for MPTCP)

WWW World Wide Web

Chapter 1

Introduction

During the last decade we have witnessed a digital revolution, with an enormous increase of interconnecting devices that communicate over the Internet. We rely on our devices every day - our smartphones, tablets and personal computers. Our devices again rely on server parks and huge content delivery networks, which are satisfying our need for information and entertainment. Our desire to stay connected and reachable the whole time, demands that our devices have reliable connections. Recently, end-user devices have started to employ some kind of connection redundancy, in the form of at least two different built-in (network) interfaces. The set of interfaces that a device is equipped with varies from device to device. Smartphones and tablets typically have 3G/4G and a Wi-Fi interface, and laptops can additionally have an Ethernet interface. However, the currently used transport protocols doesn't allow us to benefit from multiple interfaces. Today's web technologies have simply not adapted to the evolution of devices.

For over 30 years, the *Transmission Control Protocol* (TCP) has been the standard reliable transport protocol. Major Internet applications like the *World Wide Web* (WWW), e-mail and file transfers relies upon the services of TCP. TCP provides services like reliable transmission, flow- and congestion control, but a TCP connection is bound to the client and server IP addresses at the time of connection establishment. This implies that it only allows a single path between a source and a destination. With the rapid growth of devices and traffic experienced today, especially coming from multimedia streaming services, the

scalability of the Internet is put to the test. Although TCP provides mechanisms that assure flow and congestion control, the protocol itself cannot guarantee real-time delivery in situations where links become unavailable or critically congested.

With the objective to work around these TCP limitations, various researchers have proposed solutions at different network layers. However, in order for the development of a multipath protocol to be realistic, it needs to be easily deployable, without having to perform modifications on existing infrastructure. With this understanding, the *Internet Engineering Task Force* (IETF) has established a workgroup to develop a standardization for a new multipath protocol. This ongoing work has resulted in *MultiPath TCP* (MPTCP), an extension of the TCP protocol. Given that it is an extension makes it easier to deploy, since it provides backward compatibility with existing network structure.

Theoretically, MPTCP can possibly increase throughput, reliability and robustness in end-to-end connections. However, these benefits may be at the expense of increased latency. In this thesis, we seek an answer to whether these statements are accurate under realistic network conditions.

1.1 Initial Motivation

Cloud computing is rapidly gaining more importance for consumers. The ease of managing huge amounts of data and the reliability factor associated with cloud computing, are parts of the reason why cloud computing and similar services are of growing importance. However, a stable Internet connection is essential for applications of cloud computing. Knowing that critical applications have to be stable, initially motivated us to look further into multihoming.

The same factors also apply to portable devices widely deployed all over the world. Because some of these devices are constantly on the move, robustness and reliability are possibly more important than the theoretic improvement in throughput. For these devices, seamless application connectivity is of great importance.

After learning about all these gains related to multihoming, it became clearer to

us that the development of a transport protocol that is capable of utilizing all available links is definitely an important area of commitment. Optimizing the multihoming area can prepare the Internet for the future ahead.

The Department of Informatics' association with *Simula Research Laboratory* made it possible for us to get access to a *Future Internet* research testbed consisting of multihomed sites, namely the NORNET CORE. NORNET was built to allow researchers to perform experiments with multihomed systems. An experimentation platform like NORNET is able to tell us whether MPTCP truly has benefits like better throughput and seamless connection handover. Given this unique opportunity, together with our growing interest in the multihoming area, our curiosity is looking forward to combine MPTCP with an up-to-date testbed.

1.2 Objective

The purpose of this thesis is to study, analyze and evaluate the new emerging MPTCP protocol, an extension of the standard TCP protocol. Our main goal of this work is to evaluate the performance of a functional MPTCP implementation in the NORNET testbed, by performing a selection of experiments. These experiments will hopefully enlighten both positive and negative aspects of the MPTCP protocol.

To achieve the best possible understanding of MPTCP before conducting our experiments, we will at first focus our attention on background information and the underlying technologies on which MPTCP depends. We will then present state-of-the-art research on different solutions to multihoming from various research communities.

In the second phase of this thesis, the design and implementation of NORNET will be described. Finally, an MPTCP implementation will be evaluated in the NORNET environment.

1.3 Thesis Outline

The thesis is structured as follows:

Chapter 2: We go into detail on the background material to get a better understanding of the technologies which MPTCP is dependent on. This is where the basics will be discussed and established.

Chapter 3: An overview over the most recognized related work and different state-of-the-art multihoming implementations is presented.

Chapter 4: We will in detail present and describe different aspects of *MultiPath TCP*, as it is currently proposed by the *IETF*.

Chapter 5: We present the testbed setup and analyze the NORNET CORE framework implemented with MPTCP in more detail.

Chapter 6: We will explain the methodology during our experiments. Our methods for data collection from the experiments will also be given. We then present and analyze our results.

Chapter 7: Finally, we conclude the thesis and our findings, with a summary of the work developed, in addition to some thoughts on future work.

Chapter 2

Background

In this chapter, we will present some theory on underlying network technologies. In the first section, we explain the key principles of the *TCP/IP protocol suite*, with an extra emphasis on the *transport layer* and standard *TCP*, the most prominent transport protocol today. With an understanding of the protocol suite, we dig deeper into standard *TCP*, on which *MPTCP* is highly dependent. *TCP* will be thoroughly investigated, covering the most important aspects, especially those which affect *MPTCP* a great deal. In order to understand the capability of *TCP/IP* applications, an understanding of the core protocol functions must be established.

2.1 The TCP/IP Protocol Suite

A network is a group of connected, communicating devices such as computers and phones. An internet (lowercase i) is two or more networks that can communicate with each other. The most notable is called the *Internet* (uppercase I), composed of hundreds of thousands of interconnected networks and billions of devices worldwide. The Internet uses the *TCP/IP* protocol suite (*TCP/IP*) [49] to make communication possible.

TCP/IP is the set of communication protocols that implements the protocol stack

on which the whole Internet and most commercial networks run. The suite is named after its core protocols, TCP and *Internet Protocol* (IP). A protocol is required when two entities need to communicate - a set of rules that governs communication. It defines several aspects of the communication, like what, how and when. For example, in a face-to-face communication between two persons, there are rules that define how two persons should start communicating. This reasoning also holds for computer networks, as two entities in a computer network cannot simply send arbitrary data to each other and expect to be understood. For the communication to conform, both entities must agree on a protocol.

When communication is intricate or complex, there are some advantages by dividing the complex task of communicating into several layers of abstraction. Several protocols are therefore needed, typically one or more per layer. For example, the transport layer consist of more than one transport protocol, specifically TCP and *User Datagram Protocol* (UDP) [38]. By dividing the communication software into layers, the protocol stack allows for division of labor, ease of implementation and code testing. Each of the layers defines a part of the process of moving information across a network.

Most networking software follow this philosophy. The original TCP/IP was defined as a four-layer suite, built on top of the hardware. Today, it is thought of as having five layers, as the hardware that the original suite was built upon, is now seen as a layer itself.

Especially two models are mentioned in the literature, and both has been constructed to abstractly describe the communication in a computer network. The two models are the TCP/IP and the *Open Systems Interconnection* (OSI) model. The shared properties of these models are their layered structure, having the layers arranged based on its specific functionality. The model that dominated data communication and networking before 1990 was the OSI model, but as the TCP/IP was used and extensively tested in the Internet, TCP/IP became the dominant commercial architecture. For these reasons, TCP/IP is the primary focus of this section.

2.1.1 Architecture

The TCP/IP model is composed of five ordered layers [32]:

- Application Layer
- Transport Layer
- Network Layer
- Data Link Layer
- Physical Layer

While the lower layer protocols are responsible for physical transmission of data, the layers near the top are logically closer to the user application.

The layers only pass data and network information with those directly above or below, and it is made possible by concise interfaces between each pair of adjacent layers. The structure of TCP/IP is said to be hierarchical, which means that each upper level protocol is supported by one or more lower level protocols. Within each computer or any device, each layer calls upon the services of the layer just below it, e.g. the *application layer* uses the services provided by the *transport layer*, which in turn uses the services of the *network layer*. As long as a layer provides the expected services to the layer above it, the specific implementation of its functions can be modified or replaced without requiring changes to the surrounding layers. This is an important feature that is essential to the prevalence of MPTCP as a TCP extension.

It is possible to structure the layers of TCP/IP even more, as thinking of the three bottom (data link, physical- and network) layers as the network support layer. These layers deal with the electrical specifications, transport timing, transport reliability, physical connections and physical addressing. The application layer and the network support layer are linked together by the transport layer, which ensures that what the lower layers have transmitted is in a form that the application can use. The different layers are implemented differently, as the upper layers are almost always software, lower layers are a combination of software and hardware. The physical layer is essentially hardware.

Further we will describe the three top layers a little closer.

2.1.1.1 Application Layer

The application layer is the highest level layer of TCP/IP. This layer allows a user to access the different services on the global Internet. It receives data from user applications and issues requests to the transport layer. An application is a user process cooperating with another process, usually on a different host. Examples of application protocols include the *Hypertext Transfer Protocol* (HTTP) [8], *Simple Mail Transfer Protocol* (SMTP) [30] and *File Transfer Protocol* (FTP) [39]. These application protocols are defined at this layer to provide access to the *World Wide Web*, electronic mail and file transfer. The unit of communication at the application layer is called a message.

The interface between the application- and transport layer is defined by port numbers and sockets. The concept of ports and sockets is needed to determine which local process at a given host actually communicates with which process, at which remote host, using which protocol.

Port: Each process that wants to communicate with another process identifies itself to the TCP/IP stack by one or more ports. A port is a 16-bit number used by the host-to-host protocol to identify which higher-level protocol or application program it must deliver incoming messages to. There are two types of ports:

Well-known: Well-known ports belong to standard servers, for example SMTP uses port 25 and HTTP uses port 80. The well-known port numbers range between 1 and 1023 and are assigned by the *Internet Assigned Number Authority* (IANA). The reason for using well-known ports is allowing clients to find servers without configuration information.

Ephemeral: Some clients do not need well-known port numbers because they are the ones initiating communication with servers, and the port numbers are allocated automatically by the client itself. When the client is automatically allocated a port, it is also done by random, so an ephemeral port is essentially a random port used to communicate with a known server port. The ephemeral ports have values greater than 1023, normally in the range of 1024 to 65535. The ports in this range can be used by ordinary user-developed programs on most systems.

Example: If a client wants to open a connection to an FTP server, the connection would look like:

192.168.1.100:34234 → 192.168.1.200:21

Here, 21 is the standard FTP port the client is connecting to; 34234 is the ephemeral port used on the client machine.

Socket: A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data are destined to be sent to. Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. A client knows the host name of the machine on which a server is running and on the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also binds to a local ephemeral port number that is assigned by the system. Upon acceptance of the connection, the server gets a new socket bound to that port and also has its remote endpoint set to the address and port of the client. On the client side, a socket is successfully created, so that the client and server can now communicate by writing to or reading from their sockets.

2.1.1.2 Transport Layer

The transport layer provides end-to-end data transfer, by delivering data from the application layer to the correct remote host and application. The transport layer implementation is usually found in the end-user computers, and not in the routers, as these depend on lower layer protocols.

As the layer beneath, the network layer, is responsible for sending individual datagrams between A and B, the transport layer is responsible for delivering the whole message, which is called a *segment* (for TCP) or *datagram* (for UDP). A segment may consist of a few or tens of datagrams. For the transport layer to send data to a remote host, the TCP segments need to be broken into datagrams. After dividing the segments into datagrams, each datagram has to be delivered to the network layer below for transmission.

The transport layer protocols offer an abundance of services to the application layer. The services include connection-oriented communication, reliability, end-to-end integrity, flow control, congestion avoidance and so forth. Because of the important services that the transport layer offer and the location of TCP, this is the most important and relevant layer for this thesis. This is because MPTCP also operates precisely here, even though MPTCP aims at being transparent to both higher and lower layers. A detailed description of TCP and MPTCP will be provided in section 2.2 and chapter 4, respectively.

Another transport layer protocol is UDP, which provides connectionless, unreliable best-effort service. As a result, applications using UDP as their transport protocol have to provide their own end-to-end integrity, flow control etc, if this is desired. Due to the lack of services UDP provide, it is suitable for applications that need a fast transport mechanism and can tolerate possible loss of some data.

2.1.1.3 Network Layer

At the network layer (also called the internetwork layer), the virtual network image of an internet is provided. The *Internet Protocol* (IP) is the most important protocol in this layer, and is the transmission mechanism used by TCP/IP. Unlike TCP, IP does not provide reliability, flow control or error correction.

IP provides a routing function that attempts to deliver transmitted messages to their destination IP address. As described in the previous section, a TCP segment is broken into multiple IP datagrams, each of which is transported separately by IP. The content of datagrams that start at the sender and reach the receiver are not changed by intermediate routers, though they inspect the source and destination addresses of packets to find the best possible path.

In order for IP to uniquely identify the target hosts, the protocol is in need of some kind of forwarding address, specifically the IP address. This address is a numerical label assigned to each device participating in an *Internet Protocol* network. Specifically, an IP address identifies an interface that is capable of sending and receiving IP datagrams. The first designers of the IP defined an IP address as a 32-bit number [26], known today as *IPv4*. Because of the growth of the Internet and the future insufficiency of addresses, a

new version of the IP protocol (*IPv6*) was developed [17]. IPv6 uses a 128-bit number.

In addition to the core protocols IPv4 and IPv6 in the network layer, other important protocols exist, like *Internet Control Message Protocol (ICMP)*, which is primarily used for error and diagnostic functions, and *Address Resolution Protocol (ARP)*, which is providing resolution of network layer addresses into link layer addresses.

2.1.2 Encapsulation

Along with the descending, sequential order of a data unit, a header is also added at each layer, the so called encapsulation. The original data packet expands in terms of added headers as it travels down the stack. The term encapsulation comes from the fact that each layer can not differentiate the data part and header part. The formatted data is ultimately changed into an electromagnetic signal and transported onward a physical link.

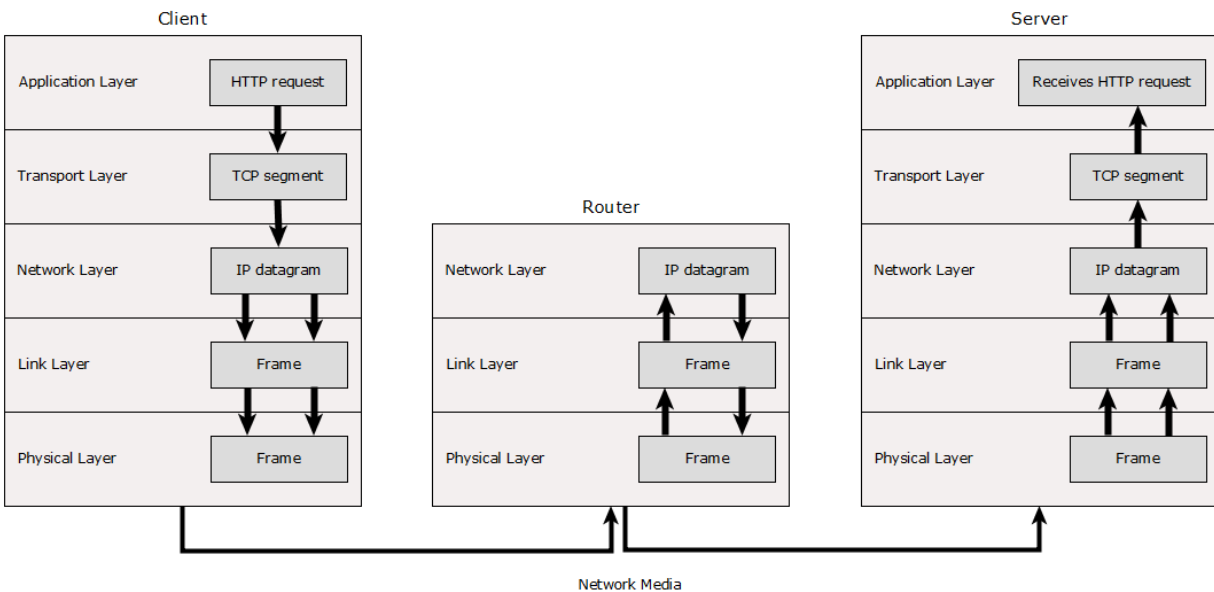


Figure 2.1: TCP/IP Architecture and Encapsulation

1 - Application Layer: The journey of the packet begins when a user sends a message or issues a command that must access a remote host. The application layer formats the

packet so that it can be handled by the appropriate transport layer protocol, either TCP or UDP.

- 2 - Transport Layer:** When the data has arrived at the transport layer, TCP or UDP starts the process of encapsulation. TCP is a connection-oriented protocol because it ensures the successful delivery of data at the receiving host. Figure 2.1 shows how TCP receives the HTTP request. TCP then divides the data received from the application layer into segments and attaches a header to each segment. These headers contain sender and recipient ports, ordering information and a checksum. The checksum is used for detecting errors that might have occurred. Since TCP is connection-oriented, the TCP connection must be established before sending. All TCP connections need to be initiated by a three-way handshake.
- 3 - Network Layer:** When TCP has established the connection to the remote host, it passes their segments and packets down to the *internet layer*, where they are handled by the IP protocol. Each segment is prepared for delivery by formatting it into multiple IP datagrams and determining the IP addresses for the datagrams. The IP then attaches an IP header containing the sending and receiving IP address, datagram length and sequence order. This information is added in case the datagrams must be fragmented due to exceeding the *maximum transmission unit* (MTU) for network packets.
- 4 - Data Link Layer:** The IP datagrams are passed down to the link layer, where Data-link layer protocols, such as Ethernet, encapsulates the IP datagrams into a frame and then attaches a third header and a footer. Each frame contains source and destination address, and error-checking data (CRC) so that damaged frames can be detected and discarded. Then the data link layer passes the frames to the physical layer.
- 5 - Physical Layer:** After receiving the frames, the physical layer converts the IP addresses into the hardware addresses appropriate to the network media. The frames are then sent out over the network media.
- 6 - Intermediate routers:** As the packet moves across the Internet towards the destination computer, the packet has to pass through routers connecting the networks the packet is traversing. Unlike the sending and receiving computer, a router only possess some of the layers of the TCP/IP stack, specifically the network-, data link-

and physical layer. A router contains only these layers, as the network layer is responsible for choosing the path through the Internet that the packet will follow. As the packet moves through the protocol stack in reverse order to the network layer, features specific to the network the packet is leaving are removed. At the network layer, IP will choose the path or route the packet should follow next. After this, the packet then moves down through the protocol layers on that router to add the features specific to the network the packet is entering.

7 - Receiver: When the packet arrives at the receiving host, the packet again moves through the TCP/IP stack in reverse order. Each protocol at the receiving host strips off header information attached to the frame. At the physical layer the *Cyclic Redundancy Check (CRC)* is computed and sent together with the frame to the Data link layer. The CRC for the frame is verified for its correctness, and the frame header is stripped off. The Data link layer then sends the frame to the network layer which reads information in the header to identify the transmission and determine if it is a fragment. If it is a fragment, then it is reassembled to the original datagram, if it is not, then the header is stripped off and is passed to the transport layer. The TCP in the transport layer reads the header to determine which application layer protocol should receive the data. Then TCP strips off its related header and sends the message or stream up to the appropriate application.

2.2 Transmission Control Protocol (TCP)

TCP was firstly described in 1981, by RFC 793 - *Transmission Control Protocol* [27]. TCP is by far the most widely used transport protocol today. Unlike UDP, TCP provides considerably more services for the applications. Specifically, this includes flow and congestion control, and reliability. As TCP is a connected-oriented protocol, its primary purpose is to provide a reliable connection between two services, and assure in order byte delivery of data over a possible unreliable underlying network, such as an IP network.

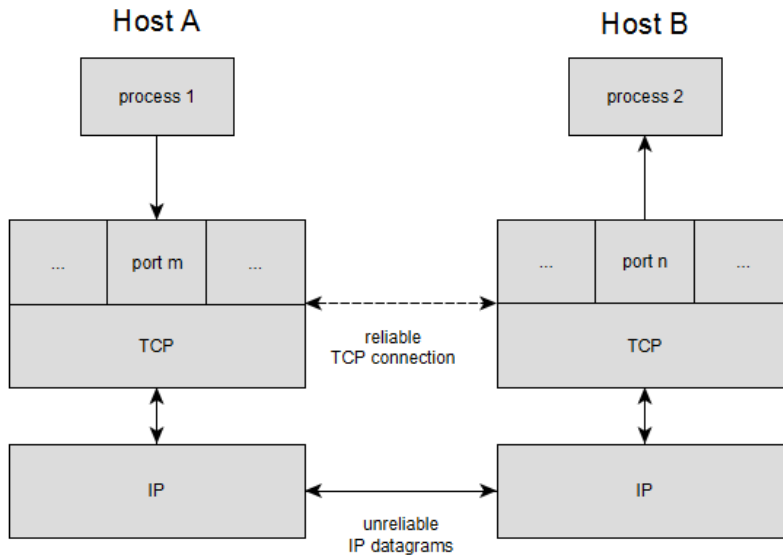


Figure 2.2: Overview of a TCP Connection

2.2.1 Important Concepts

TCP provides the following services to the applications which rely on it:

Stream data transfer: Applications using TCP is not responsible for dividing the bytes into blocks or datagrams. TCP does this by grouping the data into TCP segments, which are passed on to the IP layer for transmission. In other words, TCP assures the application that a contiguous stream of data is transmitted through the network. How the data is segmented, is up to itself.

Reliability: A sequence number is assigned to each byte transmitted, and a positive acknowledgment (ACK) is expected from the transport layer at the receiving part. If the ACK is not received within a timeout interval, the data will be retransmitted. Due to the fact that data is transmitted in blocks as TCP segments, only the sequence number of the first data in the segment is sent to the destination host. TCP at the receiving part use the sequence numbers to rearrange the segments if they arrive out of order, as well as eliminating duplicate segments.

Flow control: When TCP at the receiving end is sending an ACK back to the sender, it also

indicates to the sender the number of bytes it can receive, beyond the last received TCP segment, without causing overrun and overflow in its internal buffers. This information lies in the ACK in the form of the highest sequence number it can receive without complications. This mechanism is also called the window-mechanism, and will be discussed in detail later.

Multiplexing: Multiplexing, which can be described as the process of combining two or more data streams into a single physical layer connection, is achieved through the use of source and destination ports. These port numbers allow TCP to establish a number of virtual connections over a physical connection, and multiplex the data stream.

Logical connections: The reliability and flow control mechanisms described above require that TCP initializes and maintains certain status information for each of the data streams. The combination of this status information, including sequence numbers, sockets, and window sizes, is called a logical connection. Each of the connections are uniquely identified by the pair of sockets used by the sending and receiving process.

Full duplex: TCP offers full-duplex service, which means that data can flow in each direction concurrently. Each TCP endpoint has its own sending and receiving buffer, thus segments move in both directions.

Error free data transfer: Error-free data transfer is guaranteed by TCP. It does this by calculating a 16-bit checksum over the TCP packet (header and data). At the receiving end, if the checksum does not match the contents of the packet, it is discarded. Because the sending side does not receive an acknowledgement of the discarded packet, it is retransmitted.

2.2.2 The TCP Window Principle

In order to utilize the available network bandwidth, TCP groups packets that are being transmitted in a window of a predefined size. The sender can then fill up the window with packets, but a timeout timer must be started for each of them. The receiver must acknowledge each packet received, with the sequence number of the last received packet

indicated. When the sender receives the ACK, the window is slid, and more packets can be transmitted. Because TCP provides a byte-stream connection, sequence numbers are assigned to each byte in the stream. The contiguous byte stream is then divided into TCP segments when transmitted. The window principle is used at the byte level, which means that the segments sent and ACKs received will carry byte-sequence numbers and the window size is therefore expressed as a number of bytes, rather than a number of packets.

The size of the window is determined by the receiver at the time of connection establishment, but can vary during the data transfer. Every ACK message will include the window size that the receiver is able to process at that point in time - an important factor in the flow control.

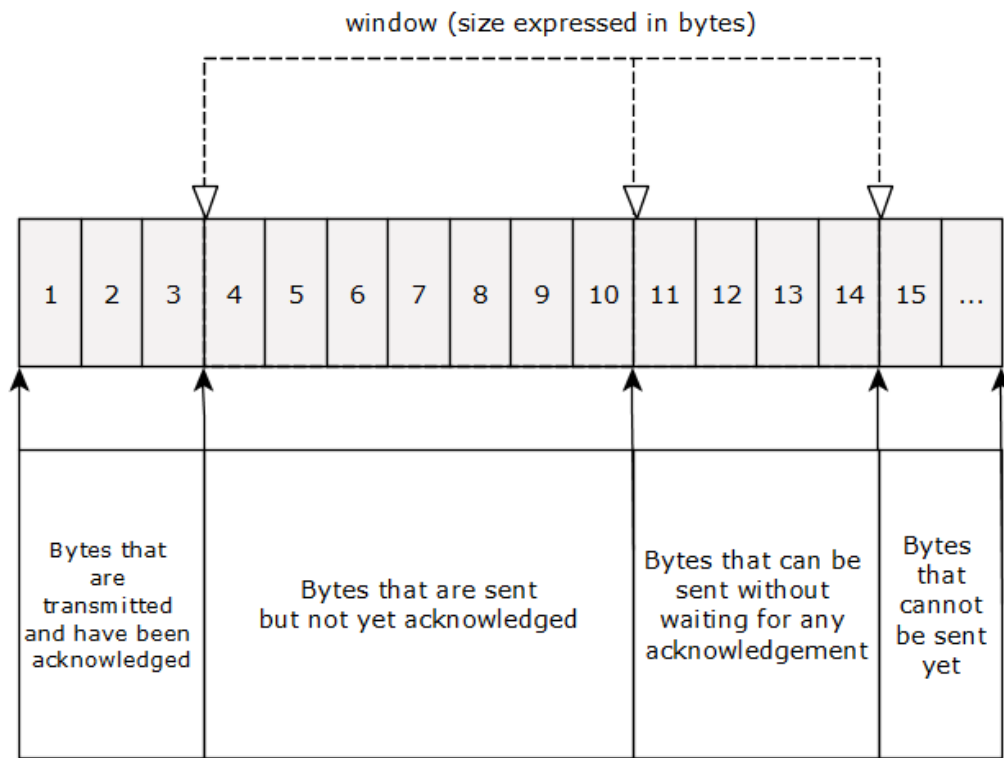


Figure 2.3: The TCP Window Principle

2.2.3 TCP Segment Format

0	1	2	3
0123456789	0123456789	0123456789	0123456789
Source Port		Destination Port	
Sequence Number			
Acknowledgement Number			
Data Offset	Reserved	U R G	A C K
		P S H	R S T
		S Y N	F I N
		Window	
Checksum			Urgent Pointer
Options		Padding
Data Bytes			

Figure 2.4: The TCP Segment Format

Source port: The 16-bit source port number, used by the receiver to reply.

Destination port: The 16-bit destination port number.

Sequence number: The sequence number of the first data byte in this segment. If the SYN control bit is set, the sequence number is the initial sequence number (n) and the first data byte is $n + 1$.

Acknowledgment number: If the ACK control bit is set, this field contains the value of the next sequence number that the receiver is expecting to receive.

Data offset: The number of 32-bit words in the TCP header. It indicates where the data begins.

Reserved: Six bits reserved for future use, must be zero.

URG: Indicates that the urgent pointer field is significant in this segment.

ACK: Indicates that the acknowledgment field is significant in this segment.

PSH: Push function.

RST: Resets the connection.

SYN: Synchronizes the sequence numbers.

FIN: No more data from sender.

Window: Used in ACK segments. It specifies the number of data bytes, beginning with the one indicated in the acknowledgment number field that the receiver (the sender of this segment) is willing to accept.

Checksum: The 16-bit one's complement of the one's complement sum of all 16-bit words in a pseudo-header, the TCP header, and the TCP data. While computing the checksum, the checksum field itself is considered zero. The pseudo-header is the same as that used by UDP for calculating the checksum. It is a pseudo-IP-header, only used for the checksum calculation.

Urgent pointer: Points to the first data octet following the urgent data. Only significant when the URG control bit is set.

Options: Just as in the case of IP datagram options, options can be either:

- A single byte containing the option number.
- A variable length option in the format as shown in figure 2.5 beneath.

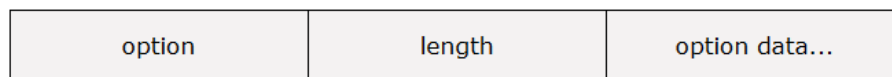


Figure 2.5: The TCP Options Format

2.2.4 TCP Congestion Control Algorithms

The responsibility of the TCP congestion control algorithm [4] is to prevent a sender from overrunning the capacity of a link. TCP has the ability to adjust the sender's rate to better

fit the network capacity and thus avoiding potential congestion situations. A number of congestion control enhancements have been added and suggested to TCP over the years, and this is still an important matter of research and discussion. Nevertheless, modern implementations of TCP contain the following four important algorithms as basic Internet standards:

- Slow start
- Congestion avoidance
- Fast retransmit
- Fast recovery

2.2.4.1 Slow Start

In the first implementations of TCP, the connection is started by the sender injecting multiple segments into the network, up to the window size announced by the receiving part. It is possible that this works OK when the two hosts are on the same LAN, but what happens if there are a number of routers and slow links between the sender and the receiver? Problems can arise, and performance is undoubtedly degraded when packets get dropped due to intermediate routers that can't handle the ongoing traffic.

To avoid this, TCP uses an algorithm called *slow start*. It operates by making sure that the rate at which new packets are sent, doesn't exceed the rate at which the acknowledgments are returned by the other end. The slow start algorithm adds another window to the sender's TCP: the congestion window, often called *cwnd*. When a new connection is established with another host, the congestion window is initialized to one segment, typically 536 or 512.

Each time an ACK is received at the sending part, the congestion window is increased by one segment. The sender cannot transmit segments exceeding the size of the congestion window. The sender starts the transmission by sending one segment, and when the ACK is received, the congestion window is incremented from one to two, and then two segments can be sent. When the ACK's for those two segments are received, the congestion window is incremented to four, and so on. This results in an exponential growth of the window. However, at some point the capacity of the network will be

reached, and packets will eventually be discarded or dropped. This can as an example happen due to slow *Wide Area Network* (WAN) links, or because of overloaded routers somewhere on the path. When packets are getting lost, the sender knows that the congestion window has increased too much.

2.2.4.2 Congestion Avoidance

Most often, the loss of a packet signifies congestion in the network somewhere between the source and the destination. There are two indications of packet loss:

- A timeout occurs
- Duplicate ACKs are received

As mentioned, congestion avoidance and slow start are independent algorithms with different objectives, but in practice, they are implemented together. Congestion avoidance and slow start's responsibility is to monitor and adjust two variables for each TCP connection:

- The congestion window, called *cwnd*
- The slow start threshold size, called *ssthresh*

The combination of these two algorithms works as follows:

1. Initialization for a given connection sets *cwnd* to one segment and *ssthresh* to 65535 bytes.
2. The TCP output routine never sends more than the lower value of *cwnd* or the receiver's advertised window.
3. When a congestion occurs (timeout or duplicate ACK), one-half of the current window size is saved in *ssthresh*. Additionally, if the congestion is indicated by a timeout, *cwnd* is set to one segment.

4. When new data is acknowledged by the other end, increase *cwnd*, but the way it increases depends on whether TCP is performing slow start or congestion avoidance. If *cwnd* is less than or equal to *ssthresh*, TCP is in slow start; otherwise, TCP is performing congestion avoidance.

The slow start procedure continues until TCP is halfway to the point where it was when the congestion occurred (half the window size was recorded in *ssthresh* in *step 2*, and then the congestion avoidance takes over. Instead of the exponential incrementation in slow start, the congestion avoidance algorithm now increments the *cwnd* by $\frac{segsz \times segsz}{cwnd}$ each time an ACK is received, where *segsz* is the segment size and *cwnd* is maintained in bytes. This results in a linear growth of *cwnd*, compared to the exponential growth of the slow start algorithm. The increase in *cwnd* should now be at most one segment each *round-trip time* (RTT), regardless of how many ACKs received in that RTT.

2.2.4.3 Fast Retransmit

The fast retransmit algorithm makes sure that TCP does not have to wait for a timeout to resend lost segments. When an out-of-order segment is received, TCP generates and sends a duplicate ACK of the last ACK sent. The duplicate ACK should not be delayed. The purpose of this duplicate ACK is to tell the other part that a segment was received out of order and to let it know what sequence number is expected next.

Due to the fact that TCP does not know whether a duplicate ACK is caused by a lost segment or just a reordering of segments, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost. TCP then performs a retransmission of what appears to be the missing segment, without waiting for a transmission timer to expire. The fast retransmit principle is illustrated in figure 2.6.

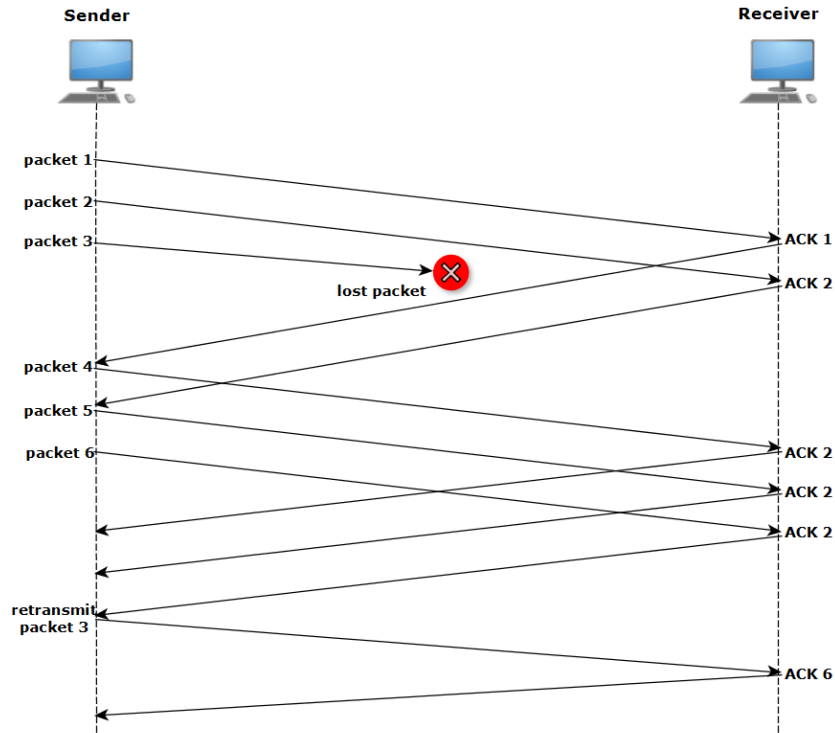


Figure 2.6: TCP Fast Retransmit

2.2.4.4 Fast Recovery

When fast retransmit has sent what appears to be the missing segment, congestion avoidance is initiated, not slow start. This is basically the fast recovery algorithm. Fast recovery is an improvement that makes it possible for TCP to maintain high throughput under moderate congestion, especially for large windows.

The main reason for not initiating slow start in this case can be explained by the fact that the receipt of the duplicate ACKs tells TCP more than just a packet has been lost. Because the receiver can only send the duplicate ACK when an out-of-order segment is received, that received segment has left the network and is now in the receiver's buffer. This means that it's still being transmitted data between the source and the destination, and TCP does not want to reduce the flow of data by going into slow start. The fast retransmit and fast recovery are usually implemented together and operates in the following way:

1. When the third duplicate ACK in a row is received, set *ssthresh* to one-half the current congestion window, *cwnd*, but no less than two segments. Retransmit the missing segment. Set *cwnd* to *ssthresh* plus three times the segment size. This inflates the congestion window by the number of segments that have left the network and the other end has cached.
2. Each time another duplicate ACK arrives, increment *cwnd* by the segment size. This inflates the congestion window for the additional segment that has left the network. Transmit a packet, if allowed by the new value of *cwnd*.
3. When the next ACK arrives that acknowledges new data, set *cwnd* to *ssthresh* (the value set in *step 1*). This ACK is the acknowledgment of the retransmission from *step 1*, one RTT after the retransmission. Additionally, this ACK acknowledges all the intermediate segments sent between the lost packet and the receipt of the first duplicate ACK. This step is congestion avoidance, because TCP is down to one-half the rate it was at when the packet was lost.

Chapter 3

Multihoming: State of the Art

The Internet has become a redundant, multipathed network over the years, far from what the designers imagined. The protocols allowing communication over the Internet, have all been designed with single-path communication in mind. There exists a gap between the single-path transport and the multipathed network that has emerged. Filling this gap would allow to pool the resources of the different available paths, which is known as the *resource pooling principle* [57].

The multihoming problem has received a lot of attention in the research community over the last years. Researchers argue different implementation solutions to this problem. As described in chapter 2, the different layers and protocols of the TCP/IP stack have different functionality and characteristics. Some of the researchers have seen possibilities in the network layer to provide the benefits of multipath transport. Some have even implemented their solution in the application layer. Currently, we find the most promising implementations in the transport layer, such as the *Stream Control Transmission Protocol* (SCTP) and *MultiPath TCP* (MPTCP).

The following sections will describe the *resource pooling principle* and state-of-the-art approaches for achieving its benefits.

3.1 Resource Pooling Principle

Being a network of networks, the Internet consists of multiple paths between communicating hosts. These paths have different characteristics, such as bandwidth and delay. *Resource pooling* means making a collection of these networked resources and make them behave as though they make up a single pooled resource, and aims to increase reliability, flexibility and efficiency. In case one of the paths experience an issue, the traffic can be moved to another path, thus achieving resilience to network failures. In a conventional single-homed session, the failure of a single path can isolate an end system, while failures in single-homed machines within the network core could cause temporary unavailability of transport. Resource pooling also enables moving traffic from congested paths to less congested paths, thus balancing the load more efficiently.

Resource pooling benefits can be summarized as follows:

- Robustness against component failures
- Better ability to handle localized surges in traffic
- Higher utilization of available infrastructure
- Balancing load between various parts of the network

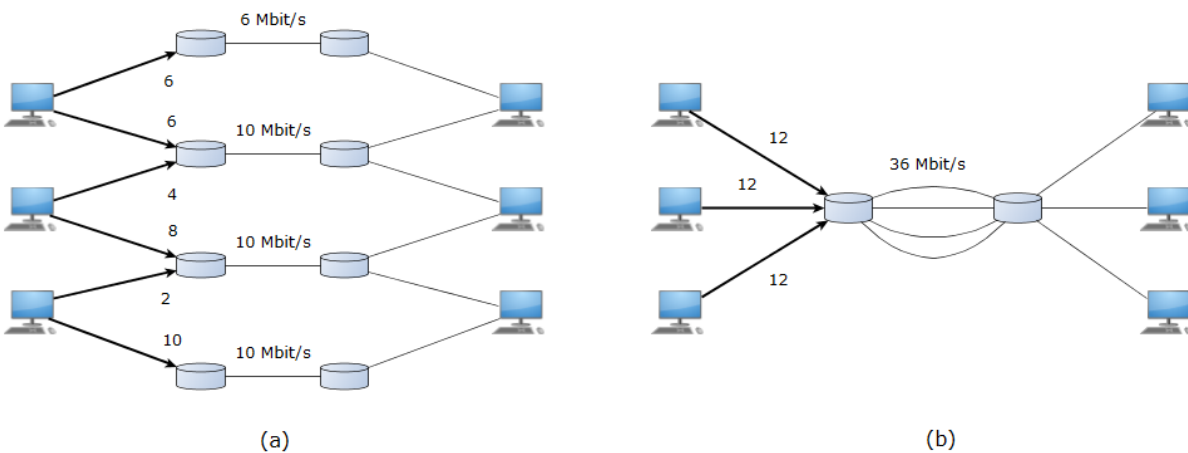


Figure 3.1: Resource Pooling Principle

In figure 3.1, the entire 36 Mbit/s bandwidth in (a) can be shared fairly and each flow can achieve a higher throughput than it could over a single path. This is as they were load balancing over the four links in (b). Since each node is connected to multiple links, it is also more robust to any link failure, but not as robust as in (b).

The implementation solutions to the multihoming problem are all examples of some type of resource pooling. There are clear differences between the implementations, but the *resource pooling principle* is what binds them all together. The following subsections will describe the various approaches in deeper detail, categorized by the layer of implementation.

3.2 Application Layer

There exist a few approaches to resource pooling in the application layer, even though implementing multi-access protocols at this layer is not hard. One can naturally bind sockets to different interfaces with the socket API and then send data to different sockets, which will, with appropriate routing configuration, follow different paths. This approach has some inconvenience associated with it, as the developer must implement services like congestion control and scheduling by himself.

Session Initiation Protocol (SIP) [42] and *IP Multimedia Subsystem* (IMS) are signaling protocols to negotiate media sessions, that typically include phone calls and media streaming sessions. These protocols can interestingly enough support the use of multiple contact IP addresses for the registration of one or more IMS user agents on terminals. This enables the creation of multiple IMS signal paths. To negotiate media streams, *Session Description Protocol* (SDP) [24] messages are carried on the paths. The negotiations of the media streams enable multipath transport at the application layer. However, to negotiate multimedia streams, *Real-time Transmission Protocol* (RTP) [45] is used, but only single-path communication is supported. On the other hand, a protocol that enables multipath RTP is under discussion within the IETF [47] [48]. In [46], Varun Singh et al. says that MP RTP allows utilization of multiple paths without performance degradation compared to suitable single-path cases. It also enables load distribution between available interfaces, as well as resource pooling in diverse scenarios. MP RTP may also allow mobile hosts to

access the Internet on the move, providing seamless handovers between networks.

Peer-to-Peer (P2P) protocols, like *BitTorrent*, are also an approach to benefit from multihoming, though it is a bit different from the other approaches. The *BitTorrent* approach involves downloading separate chunks of a file from different peers located anywhere. It is different from node-to-node multipath approaches as it achieves resource pooling from opening a single connection per peer, while other approaches opens several connections to one single peer. When peers are downloading chunks from different people, they are also uploading chunks to other peers, making it possible for files to be downloaded and shared simultaneously by a very large number of peers [16].

3.3 Link Layer

At the link layer, link aggregation techniques are used to aggregate the capacities of different interfaces to the same switch. The goal is to provide redundancy in case of a link failure, as well as to increase throughput. The link aggregation feature has been defined by the *Institute of Electrical and Electronics Engineers* (IEEE), in the *IEEE 802.3ad* [33] [13]. Only the link capacity of a single hop is able to be pooled at this level. Also, it requires specific configuration on both the host and the connected switch. Though, there exists an automatic configuration protocol for this configuration, known as the *Link Aggregation Control Protocol* (LACP) [14]. LACP offers a mechanism to control the aggregation of several physical ports to form a single logical channel. Resource pooling through link aggregation can be achieved in multiple modes. Distribution of frames by round-robin over the links is one option. However, this approach increases the probability of frame reordering, which can result in performance problems for TCP's fast retransmission algorithm [9], as it uses arrival of duplicate ACKs to detect packet loss, which can be caused by reordering.

Today, link layer aggregation is widely used, despite that one of its downsides is that it only allows to aggregate the capacity of the next hop. Some *Internet Service Providers* (ISPs) for instance, uses link aggregation to increase cumulative bandwidth between two switches, which can help improve cost effectiveness. It is also used to provide the servers with higher network-access. Some approaches, targeting the home and business market,

manages the incoming ISP connections, by sending and receiving traffic on the best link possible [19]. This can potentially improve application performance by prioritizing and optimizing traffic. However, this approach does not fully comply with the *resource pooling principle*. Rather than using multiple links at the same time, it tries to constantly use the best one. Since link aggregation only affects the next hop, it doesn't bring any benefits if the bottleneck of the communication is not in the next hop. Link aggregation is mostly restricted to *local area networks* (LANs), and doesn't support the multipath diversity as well as the layers above.

3.4 Network Layer

It seems more natural to benefit from multiple paths by implementing the solution in the network layer. Intuitively, this seems to be the most clear-cut. A single TCP connection would be sufficient, and the packets would be distributed across different flows, with the help of the network layer. There are especially three recognized solutions that try to achieve all, or some, of the benefits of multihoming at the network layer, such as *Mobile IP* (MIP) [11], *Site Multihoming by IPv6 Intervention* (Shim6) and *Host Identity Protocol* (HIP). These solutions will be described in the following subsections.

3.4.1 Mobile IP (MIP)

Mobile IP (MIP) is an IETF standard protocol that allows mobile device users to relocate from one subnet to another while maintaining a permanent IP address. The result is continuous Internet connectivity. There also exists an implementation of the next generation of IP, specifically the *Mobile IPv6* [12].

Mobile IP introduces the following entities; the mobile node, a home agent, and a foreign agent. Regardless of its current position, the mobile entities are identified by its home address, belonging to the home agent. The home agent is a router on the mobile node's home network. When the node moves away from the home network, it is identified by a care-of-address. This address can identify the current network location, which is on the so-called foreign agent. The foreign agent provides routing services to the mobile

node. Mobile IP specifies how the home agent routes datagrams to the mobile node, as the foreign agent detunnels and delivers them.

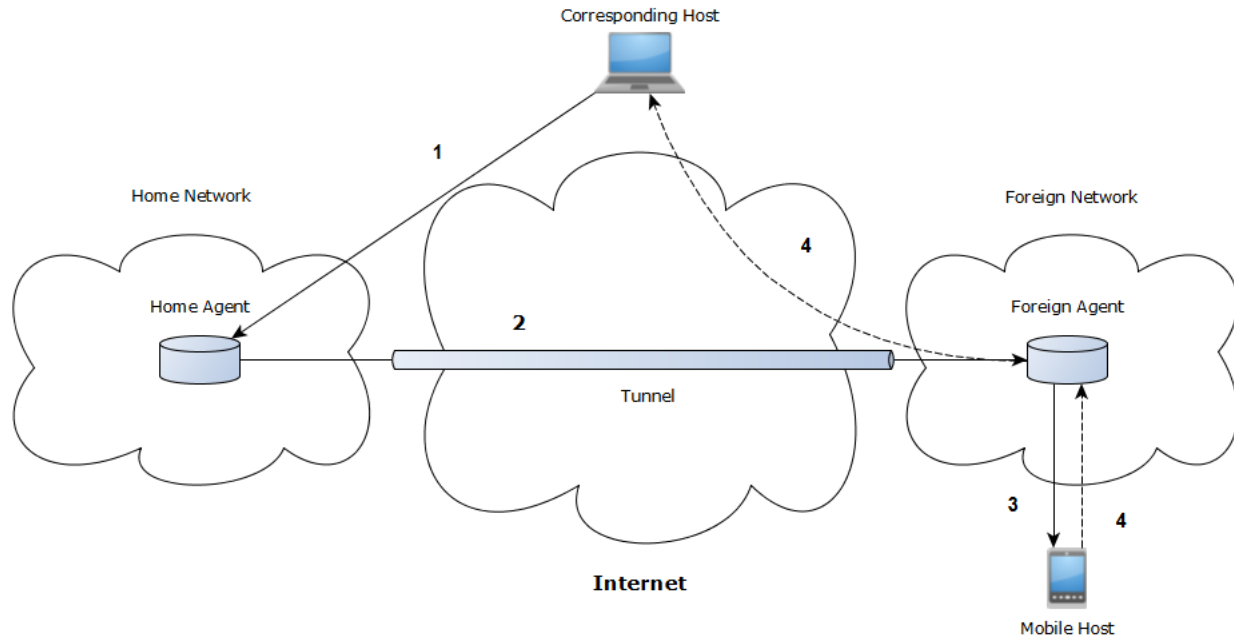


Figure 3.2: Mobile Host Sending and Receiving Data

Figure 3.2 shows how Mobile IP operates:

1. The home agent receives datagrams going to the mobile node (home address), via standard IP routing
2. The home agent tunnels the datagrams to the care-of-address (foreign agent)
3. Datagrams are detunneled by the foreign agent and delivered to the mobile node
4. If the mobile are the one sending datagrams, standard IP will forward them to their destinations. The foreign agent then function as the mobile node's default router

Mobile IP allows for undisrupted TCP connections. However, it does not comply with the *resource pooling principle*. Since it does not allow resource pooling, and thus not increase

the bandwidth, the Mobile IP we know today is only capable of giving us some of the benefits of multipath.

However, in [59] it is described how a Mobile IP extension is able to manage multiple simultaneous connections with foreign agents. For now, it is only compatible with Mobile IPv4. In the extension, multiple paths can be used for packets to and from the mobile host, therefore it also complies with the *resource pooling principle*. The mobile host is able to register multiple care-of-addresses at the home agent by listing all the reachable networks. It only chooses the networks supporting the best connectivity. The connectivity is evaluated by monitoring the deviation in arrival times between advertisements. With a list of care-of-addresses, the home agent can now reach the mobile host on each of these. The MIP extension will manage the network mobility and multihoming, so the IP routing protocols are unaware. Its compliance to the *resource pooling principle* is able to support an increase in throughput and provide a more reliable connection than the original MIP.

3.4.2 Site Multihoming by IPv6 Intervention (Shim6)

Mobile IP, despite its lacking support of multipath communication, is a very successful solution in IP-based networks. Unfortunately, partly because of the need of an intermediary router, performance can be poor in terms of handover delay [2].

With this problem in mind, there is specified a purely end-to-end based solution, namely *Site Multihoming by IPv6 Intervention (Shim6)* [36], an extension to IPv6. The Shim6 protocol is a network layer protocol for providing locator agility below the transport layer protocols. This provides multihoming for IPv6 with fail-over and load-sharing properties, in the sense that it allows existing communication to continue when a host experiences link failure.

A part of the Shim6 approach involves detecting when pair of interfaces (or addresses) between two nodes experience a failure. If there is such a detection, Shim6 will allow to hand over traffic from one IPv6 address to another. This signaling is done by bringing a clear split between the locator and identifier part of an IP address. In mobile environments, this is especially helpful for hosts that have to manage many IP addresses. If a host has several IPv6 addresses, the connecting application will use one of these as

the *upper layer identifier* (ULID). For the upper layers, the ULID of a Shim6 host stays unchanged even if the active IP addresses (locators) are changed, intentionally or not. At the host, mapping between the identifier and locators is performed at the Shim6 sublayer. The Shim6 sublayer is responsible for changing locators, while keeping the identifiers constant.

The *REACHability Protocol* (REAP) has been designed to be used with Shim6, and implements failure detection and recovery capabilities through locator pair exploration functions [5]. Upon failure detection, REAP is able to find a new working path by probing the available locator pairs. The Shim6 layer is then told to change the current locators, after in which communication can continue without change in the application.

Studies have shown that Shim6 is indeed realizable and also deployable in a multihoming context [7]. There are currently some public Linux implementations of Shim6, e.g. the *LinShim6* [6] and the one described in [1].

3.4.3 Host Identity Protocol (HIP)

During IETF meetings in 1998-1999, the *Host Identity Protocol* (HIP) was formed by the HIP working group. The HIP architecture is described in [34].

The primary concept of HIP, is to enhance the original TCP/IP architecture, by decoupling the network layer and the transport protocols. In this new architecture, *Host Identifiers* (HI) are introduced, by implementing the same identifier/locator split as in Shim6. A *host identifier* is the public cryptographic key component of a public/private key pair. As in Shim6, the new host identifiers take over the prior identification role of IP addresses, which is now used for determining topological locations in the network. Identification of TCP connections, applications and sockets is now achieved with HIs. Two hosts cannot have the same host identifier.

A four-way handshake between two HIP hosts is run when initializing a connection, also called the *base exchange*. During the exchange, the hosts identify each other using public key cryptography and exchange *Diffie-Hellman* public values. The host initiating a connection is labeled as the *Initiator*, the peer is the *Responder*. The Initiator first

sends a trigger packet to the Responder (I1), containing a *Host Identity Tag* (HIT) of the Responder. The HIT is a 128-bit hashed encoding value of the HI [31]. A hashed encoding allows for some advantages; a fixed length makes for easier protocol encoding and better packet size cost. The Responder starts the actual exchange by sending a packet (R1) containing a cryptographic challenge (puzzle) that the Initiator must figure out before continuing the exchange. Based on the trust level of the Initiator, the puzzle difficulty is adjusted accordingly. R1 also contains the initial Diffie-Hellman parameters. Now, the Initiator have to solve the puzzle, which it displays to the Responder in the I2 packet. I2 also contains a Diffie-Hellman parameter. The Responder discards I2 if it does not contain the correct solution to the puzzle. If it's correct, the R2 packet concludes the exchange.

The result of adding a new name space to the architecture is that applications that open connections and send packets, now bind to the HIs instead of IP addresses. The HIP sublayer links IP addresses together, so that multiple IP addresses correspond to one Host Identity. The locators do not need to be known to the upper layers, only the Host Identity, as the applications are unaware of the architectural change, which leads to practical division of labor and backwards compatibility. Additionally, HIP is constructed so that it is entirely backward compatible with the already deployed IP infrastructure. Consequently, when an existing e-mail client connects to an e-mail server, a reference to the server's public key is handed over to its operating system. This means that the client wants to open a (secure) connection to the server holding the corresponding private key. Even though both the client and server are mobile and changing their whereabouts, the connection can be kept open. If a subset of the addresses become unavailable or a more favored address becomes available, existing connections can easily be moved to another address. Address changes are therefore straightforward when a node relocates while a connection is already active. HIP also allows for load balancing over available links.

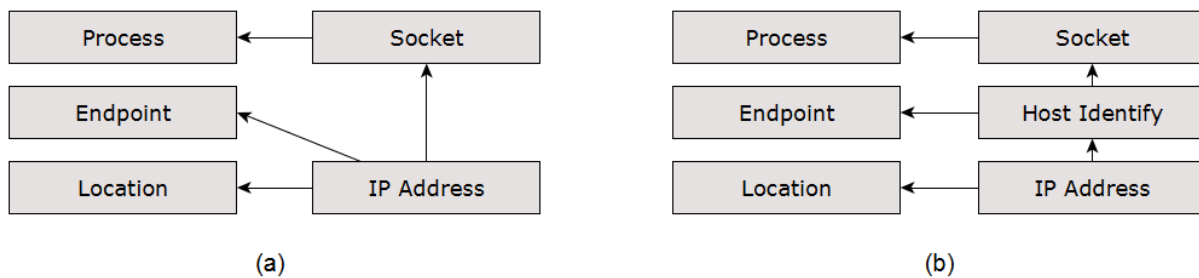


Figure 3.3: Architectural Comparison of IP and HIP

As seen in figure 3.3, in the conventional TCP/IP architecture (a), IP addresses are used as both locators and identifiers. However, in the HIP architecture (b), the endpoint names and locators are separated, but IP addresses still act as locators. The HIs take the role of endpoint identifiers. Note that an HI is reachable through different interfaces simultaneously.

Another consequence of the decoupling of the network- and transport layer, is that new possibilities for network layer mobility and host multihoming rise at a low infrastructure cost. With HIP, a number of previously hard networking problems suddenly become much easier. Mobility, multihoming and security integrate neatly into the new architecture. Since multiple interfaces can be associated dynamically to the same HI, multihoming implementation becomes trivial. The usage of cryptographic identifiers provides a basis for strengthening the trust between hosts.

The mobility and multihoming extensions to HIP [35] defines the `LOCATOR` parameter, which allows a HIP host to notify peers about alternative addresses at which it may be reached. A host achieves mobility by sending a `HIP UPDATE` packet containing a `LOCATOR` parameter to notify the peers of the new address. When a host is multihomed, it has multiple locators simultaneously rather than sequentially, as in the case of mobility. By using the `LOCATOR` parameter, a host can inform its peers of multiple locators at which it can be reached simultaneously. Today, the *Internet Research Task Force* (IRTF) is looking further into new possibilities and impacts of HIP.

3.5 Transport Layer

A multipath solution sitting at the transport layer is aware of path characteristics, which is an advantage over other solutions. When scheduling traffic over different paths, it allows for the protocols to take this information into account.

3.5.1 Stream Control Transmission Protocol (SCTP)

The *Stream Control Transmission Protocol* (SCTP) [50], is a reliable transport protocol operating on top of a connectionless packet network such as IP, and is a proposed standard by the IETF [51]. It exists at an equivalent level with TCP and UDP, which provide transport layer functions to many Internet applications. Equivalent to the TCP, SCTP offers a reliable transport service to applications, ensuring no errors and in-order delivery. SCTP is also a session-oriented protocol, implying that a relation is established between the hosts prior to data transmission, and this relationship is preserved until all data transmission has been successfully completed. SCTP was originally designed for transporting telephony over IP, hence it provides a number of functions that are critical for telephony signaling transport. These functions can at the same time benefit other applications needing transport with additional performance and reliability.

3.5.1.1 Basic SCTP Features

Unicast: SCTP is a unicast protocol, therefore it only supports data exchange between exactly two endpoints, even though these can be represented by multiple IP addresses.

Reliable: SCTP provides reliable transmission to the application layer, as it detects when data are discarded, duplicated or corrupted, and it retransmits damaged data as necessary. SCTP transmission is also full duplex, meaning both parties can communicate with each other simultaneously.

Message-oriented: SCTP is message-oriented and supports framing of individual message boundaries. In comparison, the stream-oriented nature of TCP is often an inconvenience, as applications must add their own record-marking to delineate their messages to ensure that a complete message is transferred in a reasonable time.

Rate adaptive: SCTP implements congestion control, and will scale back data transfer based on the load conditions in the network. It is also fair to other traffic.

3.5.1.2 SCTP Multistreaming Feature

It is the multistreaming function provided by SCTP that gives it its name, *Stream Control Transmission Protocol*. The multistreaming feature allows to subdivide the data into multiple streams. Each stream has the property of independently sequenced delivery, meaning any loss will only affect delivery with that stream. In contrast, TCP utilizes a single data stream. As described in section 2.2.1, TCP ensures byte sequence preservation. This can be beneficial for delivery of e.g. a file, but it can cause additional delay when the network experiences loss. When this happens, TCP will delay the delivery until correct sequencing is restored. A strict sequence preservation is not vital for all applications. In e.g. telephony, only messages affecting the same resource need to maintain sequence order. Other messages are loosely correlated and can be delivered without maintaining the same order. Another example of multistreaming is the transmission of multimedia documents, such as a web page. Since web pages consist of objects of different sizes and data types, multistreaming allows these objects to be partially ordered at the receiver. This may reduce delay and result in an improved user experience.

SCTP achieves the multistreaming feature by creating an independence between data transmission and data delivery. More specifically, unlike TCP, each payload in SCTP uses two sets of sequence numbers, a *Transmission Sequence Number* that oversees the message transmission and the detection of packet loss, and the *Stream Sequence Number/Stream ID* pair, which is used to determine the sequence of received data. The independence between transmission and delivery allows the receiver to determine immediately when a gap in the transmission sequence occurs, and also if the following messages are within the same stream. If the following messages are within the same stream, there is going to be a corresponding gap in the *Stream Sequence Number*. Other messages from other streams will not show a gap. The receiver can then continue to deliver messages from the unaffected streams, while buffering messages in the affected stream until retransmission occur.

3.5.1.3 SCTP Multihoming Feature

Another core feature of SCTP is multihoming, i.e. the ability for a host to support multiple IP addresses. The benefits of multihoming are discussed in section 3.1. In traditional single-homed sessions, the failure of the single path can completely isolate a host. All addresses a host is available on is exchanged during the connection establishment in order for SCTP to support multihoming.

The earlier versions of SCTP only supported multiple IP addresses in fail-over situations. However, it is the recent extensions that have enabled SCTP to support the use of multiple paths concurrently [28]. Regardless of this, SCTP has not been universally deployed. One reason is that a majority of firewalls and NAT (*Network Address Translation*) boxes are inadequate of processing the SCTP packets, and therefore discard them. Another reason is that each application must code to a different socket than of TCP. Consequently, SCTP is not backward compatible to current applications. These problems combined lead to the classic chicken-and-egg problem; the network middleboxes do not support SCTP in their firewalls because no applications are using this protocols, and no application is using SCTP because the firewalls discard SCTP packets.

As we will describe in section 4.1.2.2, the Internet today is infested with middleboxes that interfere with our data transport. TCP and UDP are the only protocols we can expect gaining adequate acceptance by the middleboxes. Therefore, a promising solution to the multipath problem runs on top of TCP, namely *MultiPath TCP* (MPTCP). MPTCP has gained a lot of attention from the research community in recent years. MPTCP is also a transport layer solution, and will be thoroughly described in the next chapter.

Chapter 4

MultiPath TCP

This chapter will present and describe the most important concepts of the current MPTCP proposal by IETF, as defined in *RFC 6824* [21]. We will look at advantages, potential drawbacks, and give an overview of its general operation. We will also take a look at the proposed *Coupled Congestion Control Algorithm* for MPTCP, described in *RFC 6356* [40]. Finally, we will take a look at how MPTCP handles failures, specifically concerning middleboxes.

MultiPath TCP is an extension of the existing TCP which achieves *resource pooling* and resilience to failures by utilizing diverse paths on the Internet. Unlike ordinary TCP, MPTCP allows sending and receiving data using different interfaces, with different IP addresses, simultaneously. In addition, it also provides compatibility with the application layer and network layer - every modification is done at the transport layer. As a result of this, MPTCP will appear to applications just as standard TCP, and it also maintains backward compatibility with existing network designs and segments can traverse the Internet as it is currently structured. Beneath, figure 4.1 give a comparison of the protocol stacks of respectively standard TCP and MPTCP.

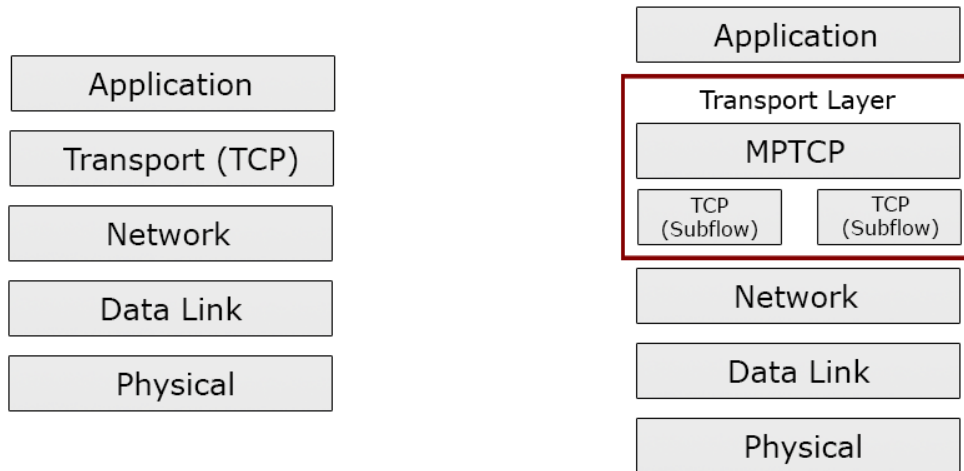


Figure 4.1: Comparison of the TCP (left) and MPTCP (right) Protocol Stack

4.1 Design Goals

As noted in *RFC 6182*, the development of MPTCP aims to meet both functional and compatibility goals. The functional goals consist of features that MPTCP must provide, and the compatibility goals determine how MPTCP should appear to entities that interact with it. This section is based on the goals as they are stated in [20], and will outline the main principles behind them.

4.1.1 Functional Goals

When taking advantage of transmission over multiple paths, MPTCP has the following two functional goals:

Improve Throughput: MPTCP must support the concurrent use of multiple paths. To meet the minimum performance incentives for deployment, a MultiPath TCP connection over multiple paths SHOULD achieve no worse throughput than a single TCP connection over the best constituent path.

Improve Resilience: MPTCP must support the use of multiple paths interchangeably for resilience purposes, by permitting segments to be sent and re-sent on any available path. It follows that, in the worst case, the protocol **MUST** be no less resilient than regular single-path TCP.

If the distribution of traffic among available paths are done in accordance with the *resource pooling principle*, the use of MPTCP should eventually improve overall network performance over the Internet, as traffic will be more balanced by reducing load on congested links, and taking advantage of spare capacity on other links.

4.1.2 Compatibility Goals

In order to be deployable without complications in today's network infrastructure, MPTCP is required to meet several compatibility goals. The following sections will divide the compatibility goals into categories.

4.1.2.1 Application Compatibility

This section refers to the appearance of MPTCP to the applications - both regarding the API that can be used and the expected services that are provided.

It is essential that MPTCP follows the exact same service model as TCP, and provide in-order, reliable, and byte-oriented delivery. It is also required that an MPTCP connection provides the application with no worse throughput and resilience compared to the best available single-path TCP connection. However, MPTCP may not be able to provide the same level of consistency of throughput and latency as a single TCP connection [43]. Extensive backward compatibility to existing TCP APIs must be retained, so that existing applications can take advantage of multipath transport just by upgrading transport layer-components in the operating systems of the end hosts. Although, this backward compatibility should not exclude the possibility of using an advanced API to allow multipath-aware applications to have specific preferences, or for users to do configurations to their systems that differs from the default settings.

MPTCP should desirably implement some sort of similar session continuity as TCP. The architecture of TCP allows active sessions to handle minor connectivity breaks by keeping the state of the host static if a possible timeout occurs. However, with MPTCP a new problem needs to be addressed; if a connectivity break occurs in a multipath capable connection, which of the interfaces appear afterwards? With regular single-path TCP, you will only have one interface, and thus the address of this interface will remain constant. In the case of MPTCP, every interface will have its own address. Therefore, it's desirable (but not required) to implement some kind of support for the mentioned session continuity, which is based on the *break-before-make*-principle. The *break-before-make* term means that the failed connection is completely killed before a new one is established.

4.1.2.2 Network Compatibility

The network compatibility goals refer to how MPTCP should behave in traditional network architecture. The main goal that MPTCP aims to meet is to retain compatibility with the Internet infrastructure as it exists today, without needing to do modifications to network devices which operate at the network layer and the layers below that. To meet this goal, MPTCP is constrained to appear exactly as TCP on the media layers, and using established TCP extensions where necessary. This requirement makes MPTCP able to traverse predominant middleboxes, such as NATs, firewalls and proxy servers. If insurmountable incompatibilities arise for the multipath extension on a path, MPTCP must fall back to regular TCP to retain network compatibility. As mentioned, the only modifications required to support MPTCP remain at the transport layer. However, some information about the underlying network layer is required, for MPTCP to satisfy the goal of seamlessly work with interchangeably IPv4 and IPv6 networks.

Despite the almost exact appearance as TCP, problems may arise regarding prevailing middleboxes. The architecture shown in figure 4.2 was for a long time the default Internet structure, but this no longer reflects the truth about network operation. Occasionally middleboxes interfere with the transport layer, as illustrated in figure 4.3. The fact that this possibly can complicate MPTCP deployment and impact the general use of MPTCP, is a challenge. The main problem we are facing is that middleboxes can strip or make changes to the TCP header (section 2.2.3). All classes of middleboxes can deliberately drop packets carrying unknown TCP options. As we will see in section 4.3.1,

MPTCP uses a new *TCP Option Kind*, which carries information that is essential for successful MPTCP operation. Middleboxes should preferably just be concerned about the delivery of datagrams in the network layer, and thus be forwarding TCP segments without modifying or removing unknown TCP options. Unfortunately, this is not the reality.

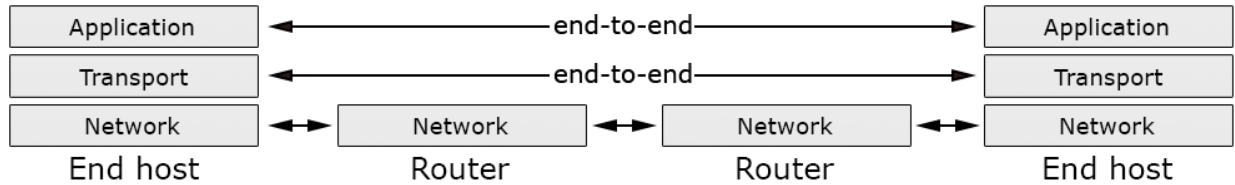


Figure 4.2: The Traditional Internet Architecture

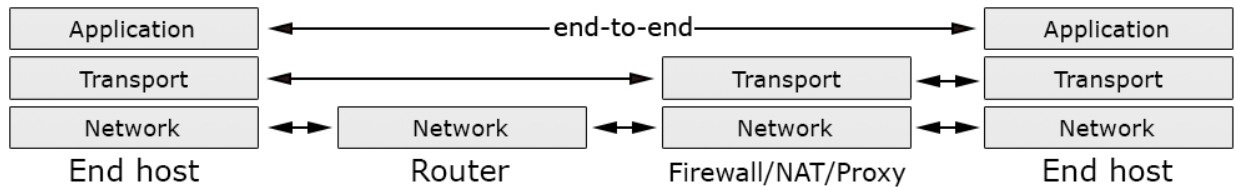


Figure 4.3: The Real Internet Architecture

The purpose of the behavior of middleboxes is mainly related to the attempt of optimizing performance or enhancing security, but naturally this behavior needs to be considered and accounted for in order to minimize the chances of failure during MPTCP deployment. In section 4.4.1, we will take a closer look at error handling related to middleboxes.

4.1.2.3 Compatibility With Other Network Users

The main goal of this section is to ensure that the architecture enables new MPTCP sub-flows to coexist with existing single-path TCP-flows, without competing for bandwidth in an aggressive way. In the cases where an overloaded link is shared, MPTCP flows are required not to deliberately harm other users transmitting using a single-path TCP-connection, beyond the impact of a competing single-path TCP-flow. As a corollary, multiple MPTCP flows over a shared bottleneck must divide the bandwidth equally with

other single-path TCP flows, the same way as single-path TCP-flows would impact the bandwidth.

4.1.2.4 Security Goals

From a security perspective, MPTCP is required to provide a reliable service with the same level of security as regular, single-path TCP. However, the extension of a single-path TCP service to a service with multipath capabilities, will introduce a number of new threats [41]. Still, the security goals will be achieved by taking advantage of existing TCP security mechanisms, and combining them with new security measures of protection against the new multipath threats identified.

4.1.2.5 Congestion Control Algorithm Goals

This section presents the primary goals for the congestion control algorithm selected for MPTCP. As mentioned, fairness when transmitting through a shared bottleneck is just one of the goals required for MPTCP. To obtain these desirable qualities, there exist three goals for the multipath congestion control algorithm:

Improve Throughput: A multipath flow should perform at least as well as a single path flow would on the best of the paths available to it.

Do no harm: A multipath flow should not take up more capacity from any of the resources shared by its different paths than if it were a single flow using only one of these paths. This guarantees it will not unduly harm other flows.

Balance congestion: A multipath flow should move as much traffic as possible off its most congested paths, subject to meeting the first two goals.

These three goals for the congestion control algorithm support and correspond to many of the goals stated in the previous sections. The combination of the first and second goal ensure fairness at a potential shared bottleneck, as stated as a goal for MPTCP in section 4.1.2.3. The first functional goal defined for MPTCP, is also in accordance with the

first goal for the congestion control algorithm - to assure that a service with multiple flows should never perform more poorly than a regular single-path flow. The third goal emphasizes the concept of *resource pooling* - if a service transmitting over multiple paths sends more data through the least congested path, it will clear away traffic from the most congested paths. Among other things, this will eventually improve robustness and overall throughput. The best way to achieve *resource pooling* is to effectively manage to *couple* the congestion control loops for the different subflows.

4.2 Terminology

Before going into detail on how MPTCP operates, we need to define and explain a number of terms that are essential to this protocol.

Path: A path is defined as a sequence of links between a sender and a receiver. When a packet is sent, it traverses through many links between different routers before it arrives at the destination. The total route this packet has traveled, is defined as the path. In order to achieve *resource pooling*, an important concept of MPTCP is that the different subflows will benefit from transmitting over different paths. This concept is of course dependent on the infrastructure of the network and how the routing is configured. It is obvious that two different paths can share one or more links, but as long as they have at least one link which differs, there can be benefits, at least for reliability and robustness.

Subflow: A subflow is defined as a flow of TCP segments operating over an individual path, which again forms part of a larger MPTCP connection, naturally consisting of several subflows. A subflow of MPTCP is initiated and terminated similar to a regular single-path TCP-connection. As a consequence, data belonging to the same MPTCP connection can consist of packets with different source and destination IP addresses and ports.

MPTCP Connection: The MPTCP connection is a set of one or more subflows, and forms the basis for communication between two end-hosts. The connection and the application socket is one-to-one mapped.

Token: The token is defined as a locally unique identifier given to a multipath connection by a host. The token can also be referred to as the *Connection ID*.

Host: The host is referred to as the end host operating an MPTCP implementation, and is either initiating or accepting an MPTCP connection.

Path management: The path managers are responsible for how the available paths between hosts should be utilized.

Packet scheduling: The packet schedulers are responsible for how packets are scheduled over the available paths. The packet scheduler divides the byte-stream from the application into segments and then schedules the segments in accordance with the principles behind the currently used scheduler.

4.3 Protocol Operation

This section will present and give a description of the different mechanisms which are essential for MPTCP operation, as specified in *RFC 6824*. These key parts of the protocol operation allow MPTCP to create, maintain and close connections that can consist of several subflows.

4.3.1 MPTCP Options

For signaling between end-hosts, MPTCP uses the *TCP Options* field. The TCP Option Kind reserved for MPTCP is 30, and is assigned by the *Internet Assigned Numbers Authority* (IANA) [25]. The use of these option subtypes is essential for the support of multipath-specific functionality and they consist of a single numerical type for MPTCP, with 4-bit “sub-types” for each MPTCP message.

Value	Symbol	Description
0x0	MP_CAPABLE	Multipath Capable
0x1	MP_JOIN	Join Connection
0x2	DATA_SEQUENCE_SIGNAL	Data ACK and Data Sequence Mapping
0x3	ADD_ADDR	Add Address
0x4	REMOVE_ADDR	Remove Address
0x5	MP_PRIO	Change Subflow Priority
0x6	MP_FAIL	Fallback
0x7	MP_FASTCLOSE	Fast Close
0x8-0xe	Unassigned	
0xf	Reserved for private use	

Table 4.1: MPTCP Option Subtypes

The subtypes specified in table 4.1 will be essential in the following description of the different MPTCP mechanisms. When a reference to an MPTCP option is made in this thesis, the symbolic name is used, such as `MP_CAPABLE`. This does, however, refer to the TCP Option with the subtype value of the symbolic name.

4.3.2 Connection Establishment

This subsection will describe how an MPTCP connection is initiated, and we will also see how new subflows are established and added to an existing connection.

To initialize the MPTCP connection, an MPTCP-enabled host opens a regular TCP-connection by sending an `ACK` packet with the `MP_CAPABLE` option. In the cases where the other host also supports MPTCP, the `SYN/ACK` response will contain the `MP_CAPABLE` option as well. The three-way-handshake is then completed by the final `ACK` packet which also contains the `MP_CAPABLE` option. Now, both hosts know that the other host is capable of performing multipath transmission, and corollary intends

to do so in this connection. In cases where the SYN/ACK response doesn't contain the MP_CAPABLE option, it is assumed that the passive opener does not support MPTCP, and the connection continues as a regular single-path TCP connection.

During the three-way-handshake, the two hosts will also exchange keys and agree on a cryptographic algorithm to be used for the connection. For now, the only cryptographic algorithm specified by RFC 6824 is the HMAC-SHA1 algorithm. The initiating host uses the MP_CAPABLE option in the SYN packet to send a 64-bit key that is generated specifically for this MPTCP connection. The generation method for this key is implementation specific, but the key must be hard to guess, and it must be unique for the sending host at any one time. This unique and random key is then hashed with a one-way hash function. The resulting HMAC-value (*Hash-based Message Authentication Code*) is the 32-bit token for this connection, and all future subflows will use this token to identify the connection. As we will see, this subflow handshake mechanism ensures that every subflow is added to the right connection - this is essential in cases where a host handles thousands of connections.

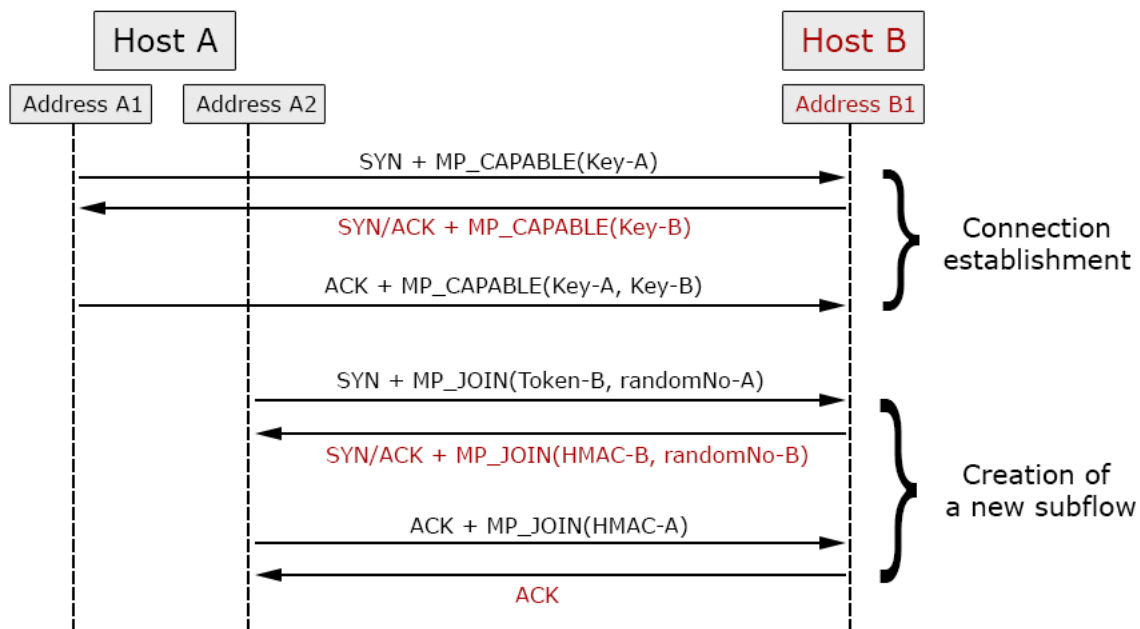


Figure 4.4: MPTCP Connection and Subflow Establishment

After the MPTCP connection has been established between two hosts, new subflows can be added to the connection. The `MP_CAPABLE` option is only used in the initialization of the MPTCP connection, and each host is given a token in order for other subflows to be able to identify the connection. Both two hosts can at any time during the connection initiate the establishment of new subflows, and they can be created using the `MP_JOIN` option.

4.3.3 Starting a New Subflow

We can of course assume that a multihomed host has knowledge of its own address(es). While both hosts are allowed to initialize the process of a starting a new subflow, the most logical and normal will be that the the host that originally initiated the connection take this responsibility.

The creation of a new subflow is started as a TCP three-way-handshake, with a normal TCP `SYN/ACK` exchange. The first `SYN`-packet is naturally sent from the additional interface with the new address. Unlike in the initiation of the connection, now the `MP_JOIN` TCP option is used, and the token for the connection is sent with it to identify which connection to be joined by this new subflow. The first `MP_JOIN SYN`-packet sends not only the token, which we know is derived from the other host's key and is static for each connection, but also a random number and an *Address ID*. The random number, called a *nonce*, is a security precaution, which prevents replay attacks on the authentication method. As mentioned, a cryptographic algorithm was exchanged during the initial connection establishment - the *HMAC-SHA1* being the only one currently available. When using this, the hosts will exchange the *nonces*, combine them with the keys from the connection establishment, generate a *SHA1* hash from the result, and finally exchange the hash values. This method will prevent an attacker from creating a new subflow to a multipath capable host, unless the attacker managed to sniff the original key from the handshake during the connection establishment.

The *Address ID* can be described as an identifier of the source address from where a subflow is initiated. As we know, middleboxes can change or remove the source address in the IP header of a packet, but using the *Address ID* in the `MP_JOIN` option assures that the receiver knows the address of the interface where the subflow originated. When the

final ACK is sent, the subflow is created and ready for data transfer. The whole exchange process is shown in figure 4.4.

The term *path management* refers to the information exchange regarding additional paths between hosts. This is basically done by exchanging information about your additional address(es). One host can notify the other host about an additional address using the `ADD_ADDR` option on an existing subflow, containing the new address. The other host may then decide to initiate the creation of a new subflow to the address received. This can be useful if a multihomed host is behind a NAT, which may prevent the host from creating new subflows with others.

If one or more of the interfaces of a host should start to encounter problems, there is also the `REMOVE_ADDR` option, which can be used to notify the other host that the address is no longer reachable and should be removed.

4.3.4 Exchange of Data

In this section the general MPTCP operation for data transfer will be explained. As we know, an MPTCP implementation takes an input data stream from an application and splits the data across one or more subflows. To ensure reliable and in-order delivery, data transmitted over multiple subflows need sufficient control information to be successfully reassembled at the recipient. Thus, MPTCP uses a 64-bit *data sequence number* (DSN) to number all the data being sent over the different active subflows, and each subflow has its own 32-bit sequence number space, just as regular TCP.

All data packets transmitted contain the `DATA_SEQUENCE_SIGNAL` (DSS) TCP option, which carries the *Data Sequence Mapping*. The data sequence mapping consists of the *data sequence number* and the *subflow sequence number*, in addition to a length for which this mapping is valid. This mapping maps the subflow sequence space to the data sequence space, and ensures that data from different subflows are reassembled correctly and delivered in-order. In the case of failure, the mapping also ensures that data can be retransmitted on another subflow (mapped to the same DSN). However, *RFC 6824* does not specify how the data should be scheduled between the different subflows at the sender, this aspect is implementation-specific.

The `DATA_SEQUENCE_SIGNAL` option also carry the *Data ACK* field, a connection-level acknowledgment for the received data sequence number. This is needed in order to assure that MPTCP provide full end-to-end resilience, and acts as a cumulative `ACK` for the connection as a whole. TCP ensures that the data segments sent over subflows are acknowledged as usual, but it's also desired that MPTCP acknowledges the reassembled data stream. Different subflows may have different RTTs, and this could lead to holes in the data-level sequence numbers, and would eventually result in the *head-of-line blocking problem* [44]. This acknowledgment is necessary for the sender to know at what point it can free data from the buffer. The *Data ACK* field specifies the next data sequence number it expects to receive on either active subflow.

Head-of-line blocking problem: The subflows of MultiPath TCP will naturally go through paths with different network characteristics. As packets are multiplexed across the available paths, the paths delay differences can potentially cause out-of-order delivery at the receiver. As MPTCP ensures in-order delivery, the packets that went across the low RTT path will have to "wait" for the packets coming from paths with high RTT. This phenomenon is known as the *head-of-line blocking problem*.

4.3.5 Prioritizing of Subflows

How an MPTCP connection choose to schedule traffic over its subflows is implementation specific, and cannot be dictated by the applications. Local policies decide how the traffic is divided over the available paths, but in most cases the goal is to maximize throughput and balance congestion.

Regardless of the scheduling mechanisms, it could be useful for applications or end users to have some kind of control of which subflow to utilize. A possible scenario could be a situation where the main subflow is the only one used for regular traffic, but a secondary subflow is ready for use in the event of failure of the main subflow. This solution can for example be favorable in cases where the secondary subflow has a high monetary cost, or any other negative path properties such as variable delay or unstable throughput.

Because of this, MPTCP has implemented the `MP_PRIO` option. This option supports two levels of subflow priority: *normal priority* and *backup priority*. If an `MP_PRIO` option is sent with the *B-flag* (backup flag) set to 1, it indicates that the path should not be used for data traffic unless there are no other subflows in the connection where $B = 0$. In other words, the subflow *can* be used in cases where every other subflow with *normal priority* fails. Even though a subflow is in *backup priority*-mode, it is still possible to *receive* data from the subflow. The principle is illustrated in figure 4.5. The *B-flag* with the priority setting can also be notified during the creation of a new subflow, in the `MP_JOIN`-option.

Note that the `MP_PRIO` option with a priority setting is a request from a data receiver to a data sender, but the sender should comply with the request. Although, it is possible that a data sender has local policies overriding the fact that a subflow is configured to *backup priority*. In situations where you want to change the priority of a subflow that is temporary unavailable due to unexpected link problems, it is possible to send the `MP_PRIO`-option on another subflow and specify the address of the desired interface by setting the *Address ID* field.

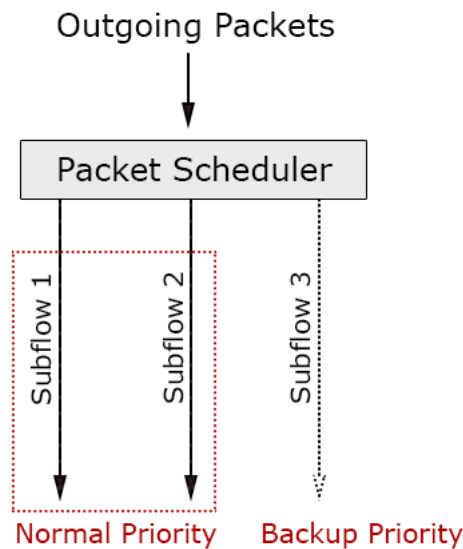


Figure 4.5: MPTCP Subflow Priority System

4.3.6 Closing a Connection

This section will describe the process related to the closing of an MPTCP connection. In a regular TCP connection, a `FIN` announces to the receiver that the sender has no more data to transmit. However, as MPTCP subflows operate independently, a `FIN` would only affect the one specific subflow on which it was sent. The semantics of a `FIN` on a subflow remains as with ordinary TCP - before both hosts have acknowledged each other's `FINs` the subflow remains open.

For MPTCP, an equivalent mechanism for closing the whole connection is needed, and this is referred to as the `DATA_FIN`. The `DATA_FIN` operates the same way as the regular TCP `FIN` - it is an indication that the sender has no more data to send. In other words could it also be a verification that all data has been successfully received.

The `DATA_FIN` is sent to the other host by setting the *F-flag* in the `DATA_SEQUENCE_SIGNAL` option to 1. When the first `DATA_FIN` is sent from a host, the return of both `DATA_ACK` and `DATA_FIN` from the other host is triggered. It is only needed for the first `DATA_FIN` to be sent on one subflow, and once it has been acknowledged, all remaining subflows must be closed according to standard TCP `FIN`-procedure. This is done as a courtesy to allow middleboxes to clean up state even if an individual subflow has failed. The MPTCP connection is considered closed once both hosts `DATA_FINs` have been acknowledged by `DATA_ACKs`.

The `MP_FASTCLOSE` option is only used by specific implementations, but acts as a *reset* to allow the abrupt closure of the whole MPTCP connection. The option indicates to the host that the connection will be abruptly closed and no data will be accepted anymore. `MP_FASTCLOSE` can be compared to the regular TCP reset (`RST`) signal, however, this would only close the concerned subflow and not affect the remaining subflows in the connection.

4.3.7 Coupled Congestion Control Algorithm

In order for MPTCP to comply with the design goals for congestion control, especially regarding bottleneck fairness, regular TCP congestion control algorithms cannot be used individually on each subflow [53]. As defined by *RFC 5681* [3], regular TCP congestion control guarantees fairness between all the data flows. Implementing this solution in each MPTCP subflow, would give the multipath flow an unfair share when paths taken by its different subflows share a common bottleneck. Say that you have a number of subflows that share the same bottleneck link or middlebox as several other single-path TCP data flows - then your number of MPTCP subflows should theoretically achieve the same throughput as every other single-path flow, and that matter of fact does not meet the MPTCP congestion control design goals. Taking this circumstance into consideration, MPTCP congestion control on one subflow can not be independent of the other subflows. As we can see in figure 4.6, the host benefiting from multipath transport with per-flow congestion control is unfair to the other single-path flow.

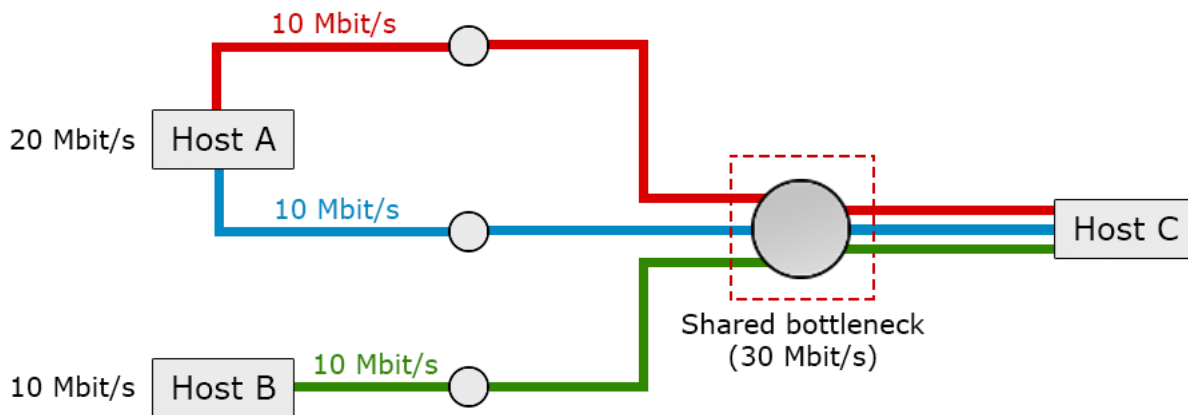


Figure 4.6: MPTCP Shared Bottleneck Problem

This subsection describes the proposed solution to this bottleneck fairness-problem, by introducing the *Coupled Congestion Control Algorithm*, defined by *RFC 6356*. This window-based congestion control takes the congestion level of the other subflows into account, and has been designed to comply with the design goals presented in section 4.1.2.5.

This algorithm operates by coupling the additive increase function of the con-

gestion windows of the individual subflows, but unmodified TCP behaviour is adopted in case of packet loss. The algorithm relies on the traditional TCP mechanisms (presented in section 2.2.4) to detect loss, to retransmit data, etc. The goal of the congestion control is to make sure that the aggregate throughput of the MPTCP subflows is equal to what a single-path TCP flow would achieve on the best path available. In order to estimate the throughput of a regular TCP flow, it is required to compute the target rate by estimating loss rates and RTTs. Then, the overall aggressiveness (the α parameter) is adjusted adequately, to achieve the desired rate.

The effect of the mechanism mentioned above is dependent on whether MPTCP subflows influence the link loss rates or not. In cases where MPTCP does not influence the link loss rate, the throughput will be equal to TCP on the best available path. In the opposite scenario, where the MPTCP subflow indeed influences the loss rates on the path, the throughput of the MPTCP subflow will probably be slightly higher than the single-path TCP would achieve on any of the paths in use.

Compared to regular TCP mechanisms, the only modification done to the coupled algorithm is actually in the increase phase of the congestion avoidance state. As mentioned in section 2.2.4, the congestion avoidance state specifies how the congestion window should increase when an ACK is received.

Beneath is the proposed *cwnd* increase formula from RFC 6356. Let $cwnd_i$ denote the congestion window of the subflow i . The sum of every congestion window of the subflows that belong to the connection is denoted by $cwnd_{total}$. The loss rate, RTT, and maximum segment size on subflow i are denoted by respectively P_i , RTT_i , and MSS_i . In this algorithm it is assumed that the congestion window is maintained in bytes. The acknowledged data, in bytes, is denoted by B .

- For each ACK received on subflow i , increase $cwnd_i$ by the following formula:

$$cwnd_i = cwnd_i + \min\left(\frac{\alpha \times B \times MSS_i}{cwnd_{total}}, \frac{B \times MSS_i}{cwnd_i}\right) \quad (1)$$

By the second term in the formula (1) stated above we observe that the *cwnd* will never

increase faster than the $cwnd$ of a regular TCP flow. By taking the minimum of the two terms, we can guarantee that the multipath flow won't be more aggressive than an ordinary TCP flow under the same conditions, hence fulfilling goal 2 (*do no harm*).

The term of importance in the formula (1) is the α -parameter, which describe the aggressiveness of the multipath flow. In order for goal 1 (*improve throughput*) to be met, the value of α has to be calculated so that the total throughput of the multipath flow is equal to the throughput of a TCP flow running on the best available path.

The calculation of the α -parameter is stated below in (2), and the value is calculated based on observed properties of the subflows of the MPTCP connection, where RTT_i is the observed round-trip-time of the subflow i .

$$\alpha = cwnd_{total} \cdot \frac{\max_i \left(\frac{cwnd_i}{RTT_i^2} \right)}{\left(\sum_i \frac{cwnd_i}{RTT_i} \right)^2} \quad (2)$$

Note:

\max_i denotes the maximum value of all possible values of i , which represents the subflows.

\sum_i denotes the summation of all possible values of i , which represents the subflows.

From this algorithm we observe that the window increase is lower than one *maximum segment size* (MSS), which means that the subflows of an MPTCP connection will reach maximum throughput slower than regular single-path TCP flows - $cwnd$ grows slower. However, this is an intended behavior. The algorithm also makes sure that the $cwnd$ of subflows that are heavily congested increase at a slower rate compared to a less congested subflow. This is in accordance with the *resource pooling principle* - traffic is shifted from congested links to less congested links, hence balancing congestion more effectively in a network.

When the total throughput of a multipath flow approaches the throughput of what a regular single-path TCP flow would get on the best flow, the *coupled congestion control*-algorithm gradually makes the α -parameter less aggressive, and by doing so, the growth rate of the *cwnd* of each subflow is decreased. If congestion arises on one or more of the subflows, both MPTCP and TCP's competing flows will halve their *cwnd*, but given the fact that TCP's *cwnd* increase faster than the *cwnd* of MPTCP subflows, the single-path flow will to some degree gain capacity previously possessed by the multipath flow. This behavior is eventually resulting in fairness between the different flows - links are shared evenly.

4.4 Failure Handling

4.4.1 Middleboxes

As mentioned in section 4.1.2.2, MPTCP can have some difficulties regarding middleboxes that strip or modify information in the TCP header, such as the TCP options. These challenges are of course accounted for in the development of MPTCP. As stated in the *design goals*, MPTCP should always fall back to ordinary TCP if some unexpected behavior is experienced.

When an MPTCP connection is initiated, the first SYN packet contains the MP_CAPABLE TCP option. If this TCP option is removed by a middlebox, the returning SYN/ACK packets will not contain MP_CAPABLE either, and MPTCP should fall back and continue the session using regular TCP.

The same goes for the establishment of new subflows - if the MP_JOIN option is removed from the outgoing SYN packet, the reply will be either a SYN/ACK or a RST (connection failure). In the case of a SYN/ACK response, the initiator will send a RST, simply because the reply did not contain the MP_JOIN option and the connection token, and thus the subflow is not multipath capable and the establishment fails. This failed subflow establishment does not affect other aspects of an ongoing MPTCP connection. If an MP_JOIN-request fails, the initiator should not attempt the same MP_JOIN-request

again to the same address, unless the other host sends an `ADD_ADDR` containing updated address information.

Situations can occur where NATs might change the source address and source port of packets, which means that it exists a possibility that a host does not know its public-facing address used for signaling MPTCP options. In cases like this it can be problematic for other hosts to establish subflows. Therefore, the *Address ID*-field has to be used during the `MP_JOIN`-request, signaling the hosts source address and port.

Another obstacle that can be met, is firewalls that possibly perform sequence number randomization on the TCP header for security reasons [22]. As MPTCP uses relative sequence numbers in the data sequence mapping, this should not be a problem. Firewalls can also place a limitation on the amount of incoming connections permitted. In these cases, it is necessary for the "protected" host to signal its address(es) using the `ADD_ADDR`-option, so other hosts can connect to additional interfaces behind the firewall. If unforeseen occurrences arise, that may be caused by middleboxes or not, the standard MPTCP procedure is, regardless of cause, to fall back to regular TCP.

Chapter 5

NORNET CORE: A Multihomed Research Testbed

Today, our society heavily relies on the diverse services the Internet provides. However, failures are unavoidable, and can have some serious consequences. When our networks experience failures, it is important to mitigate the consequences as much as possible. We also need to ensure the availability of especially critical services.

The Internet is big and complex, and carries a magnitude of critical services. This acts as an obstacle in terms of doing changes to the current infrastructure, as new protocols, algorithms and software have to be comprehensively tested and proved before being deployed. Not to mention, these tests must be done in realistic environments for the results to be legitimate and credible. To satisfy this obligation in the context of multihoming, the NORNET [23] testbed has been realized by *Simula Research Laboratory*.

5.1 The Design

The NORNET testbed is made up of two parts, NORNET CORE and NORNET EDGE. NORNET EDGE is the wireless part of the testbed, and consists of several hundred nodes. It

is a wireless infrastructure for conducting measurements and experiments with cellular networks, such as *Telenor*, *Telia* and *OneCall*.

NORNET CORE is the wired part of NORNET, where the sites are connected by wired Internet connections. Initially, NORNET CORE consisted of 10 programmable sites that was geographically spread across large parts of Norway, mainly at universities and research institutions. There were additionally two sites in Germany and China, but today, the NORNET CORE network consist of 18 sites in several different countries. Most of the sites are connected to at least two wired *Internet Service Providers* (ISPs). Table 5.1 shows the deployment status of NORNET CORE in April 2016. The ten initial sites from September 2013 have expanded to 11 domestic sites (*no. 1 to no. 11*) and nine international sites (*no. 30 to no. 200*).

No.	Site	ISP1	ISP2	ISP3	ISP4
1	Simula Research Laboratory	Uninett	Kvantel	Telenor	PowerTech
2	Universitetet i Oslo	Uninett	Broadnet	PowerTech	
3	Høgskolen i Gjøvik	Uninett	PowerTech		
4	Universitetet i Tromsø	Uninett	Telenor	PowerTech	
5	Universitetet i Stavanger	Uninett	Altibox	PowerTech	
6	Universitetet i Bergen	Uninett	BKK		
7	Universitetet i Agder	Uninett	PowerTech		
8	Universitetet på Svalbard	Uninett	Telenor		
9	Universitetet i Trondheim	Uninett	PowerTech		
10	Høgskolen i Narvik	Uninett	Broadnet	PowerTech	
11	Uni. i Oslo og Akershus	Uninett			
30	Karlstads Universitet	SUNET			
40	Universität Kaiserslautern	DFN			
41	Hochschule Hamburg	DFN			
42	Universität Duisburg-Essen	DFN	Versatel		
43	Universität Darmstadt	DFN			
88	Hainan University	CERNET	China Unicom		
100	University of Kansas	KanREN			
160	Korea University	KREONET			
200	National ICT Australia	AARNet			

Table 5.1: NORNET CORE Deployment Status, April 2016

Each of the sites consists of a set (of any size) of servers and a switch, where one of the

servers acts as a *tunnelbox*. The tunnelbox is responsible for connecting its own site to other sites in NORNET CORE, by using the available set of ISPs. The site's tunnelbox creates connections, or more precisely *tunnels* by using all possible combinations of local and remote ISPs at the other site's tunnelboxes. The topology of the NORNET CORE is therefore a fully connected mesh, which means there are direct links between any two nodes in the network. The other servers at a given site that are not the tunnelbox are called *nodes*, and are connected by the switch. These additional nodes host the virtual resources that are made available for researchers to perform experiments. Most often, a site consists of at least three servers, but this number can however vary from site to site. The NORNET CORE architecture is shown in figure 5.1.

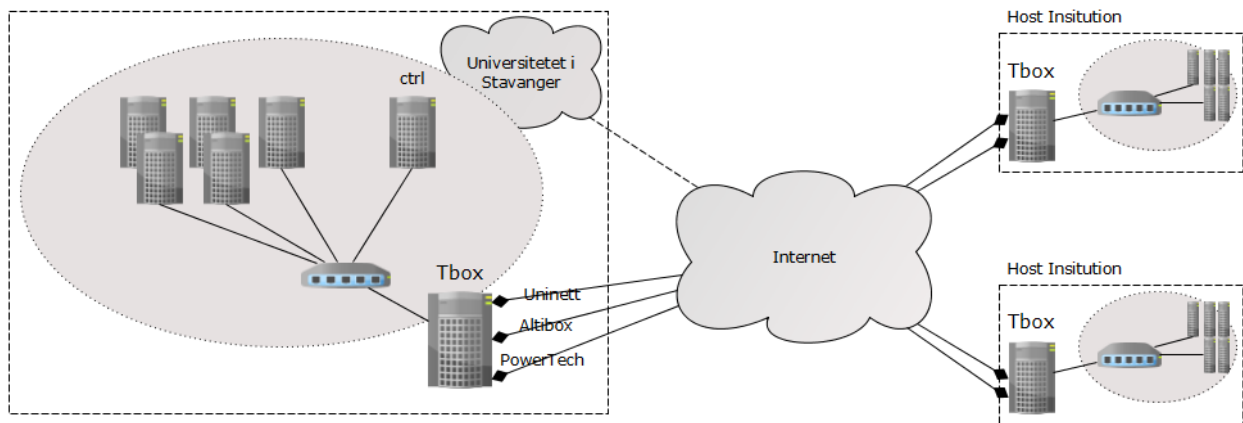


Figure 5.1: NORNET CORE Architecture

The administration of resources and users is realized by the use of a *PlanetLab/OneLab* installation. Nodes that are based on the PlanetLab/OneLab software are Linux-based physical machines that run virtual machines. Initially, the NORNET CORE was based on the original PlanetLab software, but today it is based on a PlanetLab code branch, namely OneLab. OneLab is a European testbed initiative that is built on the PlanetLab core, but has several advantages over it. Originally, the PlanetLab software applied the Linux-VServer, which is no longer a part of the mainline kernel, and is therefore troublesome to use. The mainline Linux kernel today prefers the approach of *Linux Containers (LXC)*, providing relatively similar functionality. LXC is an operating-system-level virtualization method for running multiple Linux systems on a control host using a single Linux

kernel. As the development of PlanetLab/OneLab also went in the LXC direction, an LXC-based OneLab build is the research testbed platform used by NORNET CORE. In addition to the LXC providing a much easier possibility to user state-of-the-art Linux kernels and software, it also provides a significantly improved network handling in comparison to the original PlanetLab.

As mentioned earlier, most sites are connected to at least two ISPs. Each ISP i is given a NORNET *Provider Index*, denoted $P_i \in [1, 255] \subset \mathbb{N}$. A site a is identified by a unique identification number, called the NORNET *Site Index*, denoted $S_a \in [1, 255] \subset \mathbb{N}$, and are connected to its ISPs, denoted $\hat{P}_a = \{P_{a1}, P_{a2}\}$. If S_a , with its two ISPs, is connected to S_b with $\hat{P}_b = \{P_{b1}, P_{b2}, P_{b3}\}$, there would be $|\hat{P}_a| \times |\hat{P}_b| = 2 \times 3 = 6$ possible paths from S_a to S_b . Figure 5.2 illustrates this.

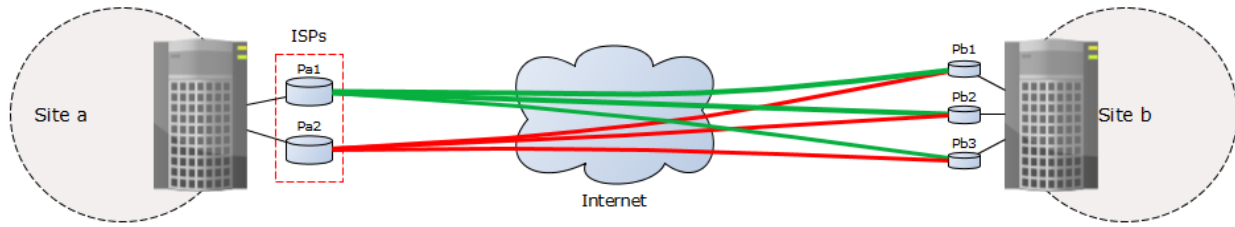


Figure 5.2: NORNET Tunneling

This means, traffic from S_a can use the two outgoing providers \hat{P}_a , and traffic received at S_b can come in from its three incoming providers. All six possible paths from S_a to S_b are represented by static tunnels among the corresponding sites' provider endpoints. Of course, these tunnels will differ in both throughput and delay, as the NORNET testbed prefers to have a mix of different connection types. That means, while it will still be connected using high-speed connections, it is also desired to add the type of connectivity that is provided to "regular" users, such as consumer-grade broadband Internet access. This selection of different Internet access types, will allow for representative network evaluation experiments. For consumers, a typical scenario would be to compare a high-speed fiber-link with a more limited ADSL-link.

5.2 The Implementation

5.2.1 Testbed Management

The nodes that are based on the PlanetLab/OneLab platform are centrally administrated by a *PlanetLab Central* (PLC). The PLC takes care of the managing of user accounts, sites, nodes and the so-called *slices*. It also provides a web-based administration interface, managing the *PLCAPI*. A slice is a reserved set of resources in the testbed. A node's resources can be split into multiple slices. For each slice, the physical node will instantiate its own virtual machine, denoted as a *sliver*. A node is therefore shared among all researchers having slivers on it. The PLC also include a *tag* feature, which allows custom extensions by adding tags to the database. Tags are additional information to be managed, e.g. testbed-specific configuration parameters. Custom, NORNET-specific configurations are then realized by appropriate tags, e.g. tunnelbox configurations (such as ISPs, addresses etc.). A tunnelbox can also use the *PLCAPI* to obtain configuration data for setting up interfaces, routing policies, as well as for dynamically providing information about any changes to the list of connected ISPs.

5.2.2 The Sites

Each NORNET CORE site needs appropriate hardware to provide resources for researchers. Therefore it was decided to deploy the setup shown in figure 5.3. At the time the *NorNet Core Handbook* [18] was written, all NORNET CORE sites physically consisted of four *HP DL320* servers, all with the same hardware specifications (e.g. two Ethernet interfaces). However, since server 1 is intended to work as a tunnelbox for the site, it is also equipped with an additional *Network Interface Controller/Card* (NIC), with four Ethernet ports. These ports are used to connect to the different ISPs, such as *Uninett*, *Kvantel*, *Telenor* and *PowerTech*. The switch connects all of the servers to the tunnelbox, which acts as a router. The fourth server is equipped with a global Uninett IP-address to provide access to a site, in case the tunnelbox is not working properly.

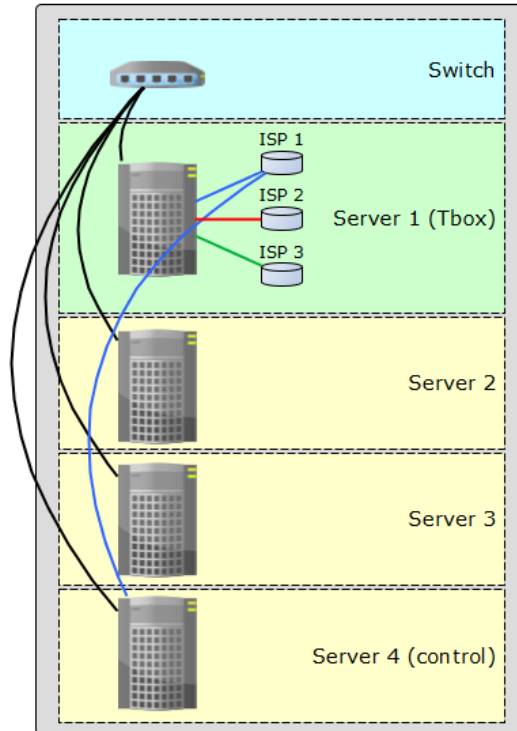


Figure 5.3: NORNET CORE Site Schematic View

The servers are installed with *Ubuntu Linux 12.04 LTS*, a server installation that is described as "minimal". The purpose of this is to offer the basic configuration tasks and *Secure Shell* (SSH) for remote login, as well as running *VirtualBox* to host virtual machines. This means that the tunnelbox and all the research systems are realized as virtual machines, which allows for good hardware resource utilization. It also allows researchers to easily backup and restore virtual systems, without requiring physical access to the hardware itself.

5.2.3 Tunneling Setup

To set up IPv4 tunnels between two hosts, *Generic Route Encapsulation* (GRE) is used. GRE is a tunneling protocol that can encapsulate a wide variety of network layer protocols inside virtual point-to-point links over an IP network. IPv6 uses direct tunneling over IPv6, but if there no IPv6 interconnection between two hosts, GRE is used.

To be able to use the tunnels appropriately, the routing has to be configured. For multihomed hosts, the default routing (longest prefix matching) is not sufficient. All packets for a destination address would then take the same route via the same ISP by default. When a host is multihomed, the source address must also be taken into consideration. Packets with source address *Uninett*, should go over the *Uninett* interface, as should packets with source address *Altibox* go over the *Altibox* interface. By applying IP-rules, i.e. own routing tables, for each ISP, this functionality is implemented.

5.2.4 Addresses in NORNET CORE

The tunnelbox also functions as a DNS server at the sites - providing the internal, private NORNET CORE network addresses. To make it easier for researchers to work with the nodes at the different sites, each site has its own second-level domain below the *top-level-domain* (TLD) `.nornet` - as an example `uib.nornet` for nodes located at the University of Bergen. Every node has its own name, for example the node *lungegaardsvannet*, located at the University of Bergen, can be reached at the address `lungegaardsvannet.uib.nornet`. In order to use these internal addresses, we must of course go through the *Gatekeeper* server, which acts as a tunnel from the outside Internet to the internal network of NORNET CORE.

5.2.5 Accessing the Testbed Slivers

In order to perform experiments in the NORNET CORE testbed, a user account for the PLC server is necessary. The new user then has to store a *Secure Shell* (SSH) key on the PLC server, as it will be used to authenticate users when accessing slivers. A user is then affiliated with one or more new or existing slices, where experiments can be done. A slice will most often represent one project. A new slice, *uib-mptcp*, was created in the context of this thesis.

To access the NORNET CORE network, the user can simply connect a computer to the local NORNET network at one of the sites. For now, external users has to connect via the Gatekeeper server (*gatekeeper.nntb.no*), but eventually it is intended to make the

network available via a *Virtual Private Network* (VPN) to the Simula central site. SSH is used to access the Gatekeeper remotely, which is just a Linux server that is connected to the NORNET CORE network as well as the outside Internet. Any operating system with the appropriate tools (mainly SSH) can be used. To be able to use SSH on Windows machines, you can use *PutTY*, a free SSH and Telnet client for Windows [52]. An SSH connection to the Gatekeeper-server can then be established by the following command:

```
1 ssh [username]@gatekeeper.nntb.no
```

After accessing the Gatekeeper, we are presented with the display in figure 5.4

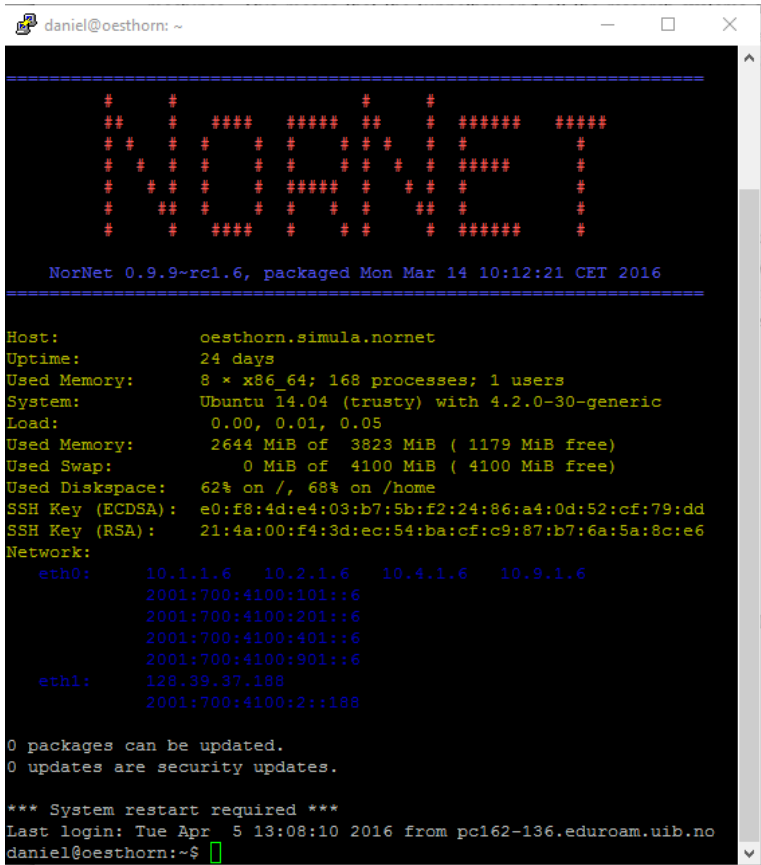


Figure 5.4: NORNET Gatekeeper

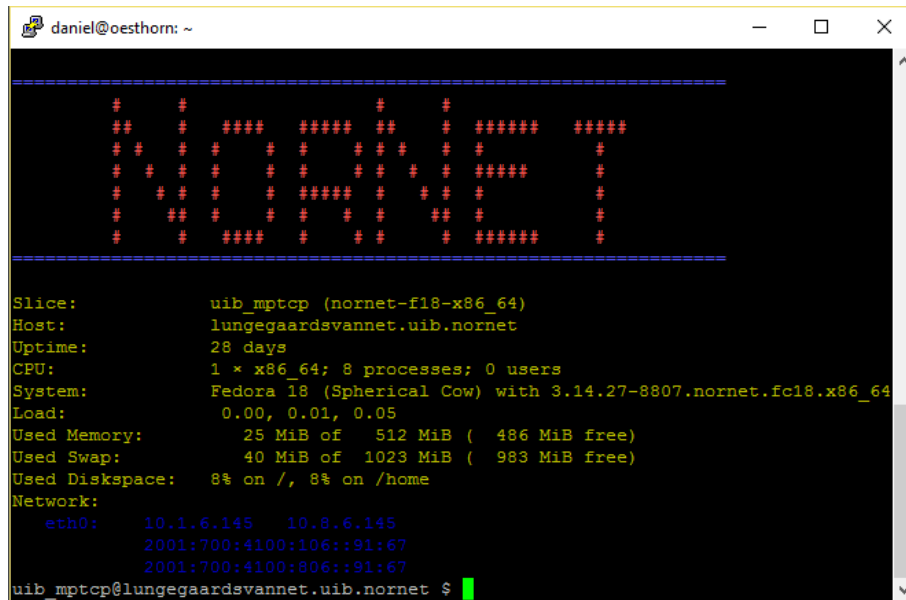
We can then connect to any of the slivers on a corresponding slice by an SSH command on the following format:

```
1 ssh -i [RSA-key] [slice]@[node].[site].nornet
```

For example, if we want to connect to one of the slivers on the node *lungegaardsvannet*, located at the University of Bergen, we can use the following command:

```
1 ssh -i ~/.ssh/id_rsa uib_mptcp@lungegaardsvannet.uib.nor-net
```

Inside a sliver, the user is presented with a Linux environment that can be configured as needed, e.g. by installing additional software, like standard software from the standard repository or custom software for research experiments. After accessing the sliver, the user is presented with the display in figure 5.5.



```
daniel@oesthorn: ~  
-----  
# # # # #  
## # #### ##### # # # ##### #  
# # # # # # # # # # # # # # #  
# # # # # ##### # # # # #  
# ## # # # # # # # # # # # # # #  
# # # # # # # # # # # # # # # #  
-----  
Slice:          uib_mptcp (nornet-fl8-x86_64)  
Host:          lungegaardsvannet.uib.nor-net  
Uptime:       28 days  
CPU:          1 * x86_64; 8 processes; 0 users  
System:       Fedora 18 (Spherical Cow) with 3.14.27-8807.nornet.fc18.x86_64  
Load:         0.00, 0.01, 0.05  
Used Memory:  25 MiB of 512 MiB ( 486 MiB free)  
Used Swap:    40 MiB of 1023 MiB ( 983 MiB free)  
Used Diskspace: 8% on /, 8% on /home  
Network:  
  eth0:       10.1.6.145  10.8.6.145  
              2001:700:4100:106::91:67  
              2001:700:4100:806::91:67  
uib_mptcp@lungegaardsvannet.uib.nor-net $
```

Figure 5.5: NORNET Sliver

As we can see in figure 5.5, a selected amount of information about the sliver is presented to the user when logging in. The user can then see which slice one is working on, the name of the node, the uptime of the node, which Linux distribution the sliver is running on and some memory statistics. Additionally, the network configuration of each sliver will show a single Ethernet interface, *eth0*, that has one or more logical IPv4 and IPv6 networks configured - one for each ISP it is connected to.

Earlier we mentioned that NORNET CORE implements server virtualization, meaning that all physical servers (the nodes) are split into one virtual Linux environment (the slivers) per slice. That means if there are 12 slices on a physical server, there are

also 12 slivers per physical server. In NORNET CORE, there will typically be one slice for each project that is using the testbed. Accordingly, in a setup with six physical servers at a site, there would need to be 12×6 slivers, per site. Of course, the server redundancy in NORNET is provided to handle server failures. Having multiple researchers sharing the same resources (without virtualization) can often provoke resource conflicts. Additionally, providing separate hardware for each sliver can become expensive when scaling.

There are multiple other reasons that the NORNET CORE implements virtual machines. The advantages of server virtualization are discussed in the next section.

5.2.6 Virtualization

The cost of wasted storage space, processing power and network utilization can be expensive, but server virtualization allows the flexibility to change the system resource allocation for the virtualized environments. Since these virtual servers can run on any machine, it means that the machine resources are fully shared between different users, which in turn ensures a high utilization level of the physical resources. Furthermore, if any of the physical resources are changed, it does not alter the virtualized servers. Server virtualization allows a growing, realistic testbed like NORNET to have a sufficient level of scalability and availability.

5.3 The NORNET MPTCP Implementation

The two most important MPTCP implementations are the FreeBSD-based implementation by the *Centre for Advanced Internet Architectures* (CAIA) at Swinburne University, as well as the *Linux Kernel MPTCP implementation* by the *Université catholique de Louvain* (UCL) [55]. While the first is still under development, the UCL implementation is by far the most widespread one. It is used by the IETF MPTCP working group as their reference implementation as well. Because of this, the UCL implementation is also the one used in the NORNET CORE testbed. In the following sections, installation and configuration of the UCL Linux Kernel MPTCP implementation are described.

5.3.1 Installation

First of all, the git-repository needs to be accessed and cloned with the following command:

```
1 git clone --depth=1 git://github.com/multipath-tcp/mptcp.git
```

When the repository is downloaded, the kernel is configured by doing `make xconfig` or `make menuconfig`. After this your kernel need to be compiled, installed and rebooted.

The following command will then enable MPTCP:

```
1 sysctl -w net.mptcp.mptcp_enabled=1
```

Since the UCL MPTCP implementation is used in NORNET, section 5.3.2 will describe how to configure the MPTCP options and system variables.

5.3.2 Configuration

The `sysctl` interface, where you modify kernel parameters at runtime, is used to configure the Linux Kernel Implementation of MPTCP, and its configurations are described next. To set a `sysctl` MPTCP-variable, the following format describes how:

```
1 sysctl -w net.mptcp.[name of the variable]=[value]
```

The following variables are available:

enabled

This option is used to enable or disable MPTCP support. MPTCP is enabled when the value is set to 1. In order to disable, the value is set to 0 (default value is 1).

mptcp_checksum

This option is used to enable or disable the MPTCP checksum. **Note:** Unless both hosts have disabled the checksum option, it will be used (default value is 1).

mptcp_syn_retries

This option specifies how many times a SYN with the MP_CAPABLE-option is retransmitted (default value is 3). If the number of retransmits reaches this number, the next SYN packet will not contain the MP_CAPABLE-option. The connection will then fall back to regular TCP. This is done because of middleboxes that drop SYNs containing unknown TCP options, as described in section 4.4.1.

For example, to switch from the default number of MPTCP SYN retries to five, use the following command:

```
1 sysctl -w net.mptcp.mptcp_syn_retries=5
```

5.3.2.1 Congestion Control

You can also configure different congestion controls with the `sysctl` command. You can change them with a command on the following format:

```
1 sysctl -w net.ipv4.tcp_congestion_control=[congestion control]
```

The congestion controls options available are:

Linked Increase Algorithm [lia] The *Linked Increase Algorithm* [40] (also known as the *Coupled Congestion Control*), described in detail in section 4.3.7, couples the congestion control algorithms running on different subflows by linking their additive increase functions, and dynamically controls the aggressiveness of the multipath flow. The result is a practical algorithm that is fair to TCP at bottlenecks while moving traffic away from congested links. This is chosen as the default congestion control algorithm for this MPTCP implementation.

Opportunistic Linked Increase Algorithm [olia] Similarly to LIA, the *Opportunistic Linked Increase Algorithm* (OLIA) couples the additive increases and uses unmodified TCP behavior in the case of loss. By measuring the number of transmitted bytes since the last loss, it reacts to events within the current window. By adapting the window increases as a function of RTTs, OLIA also compensates for different RTTs. OLIA provides fairness and theoretically optimal congestion balancing [29].

Delay-based Congestion Control for MPTCP [wvegas] The Delay-based Congestion Control for MPTCP is also known as *weighted Vegas* (wVegas), since it share characteristics with *TCP Vegas*. As LIA and OLIA are based on packet loss events, wVegas adopts packet queuing delay as the congestion signals. This is supposed to make wVegas more sensitive to the changes of network congestion, which in turn should achieve more timely traffic shifting [58].

Balanced Linked Adaption Congestion Control Algorithm [balia] The *Balanced Linked Adaption Congestion Control Algorithm* (Balial), is a window-based congestion control algorithm for MPTCP. The algorithm only applies to the AIMD (*Additive Increase | Multiplicative Decrease*) part of the congestion avoidance phase. The other parts, such as slow start, fast retransmit and fast recovery algorithms are the same as in regular TCP. The result is a MPTCP algorithm that strikes a good balance between friendliness and responsiveness. Balial is described in detail in [56].

For example, to change from the default *Linked Increase Algorithm* to the *Delay-based Congestion Control for MPTCP*, use the following command:

```
1 sysctl -w net.ipv4.tcp_congestion_control=wvegas
```

5.3.2.2 Path Manager

A *path-manager* controls the behaviour of an MPTCP-enabled host regarding the handling and creation of subflows. If a path-manager is not selected, a host will not attempt to create new subflows, nor will it inform the other host of alternative IP-addresses through the ADD_ADDR-option. To set a specific path-manager, use the following command:

```
1 sysctl -w net.mptcp.mptcp_path_manager=[path manager]
```

In the current implementation, four different path-managers are available:

default This path-manager is passive, as described above. It doesn't initiate the creation of subflows, nor does it advertise available IP-addresses. However, if the other host in the connection initiates more than one subflow, it will accept them.

fullmesh The *fullmesh* path-manager will create a full mesh of subflows on all available interfaces to all available IP-addresses at the other host. This means that a subflow will be created to every interface at the receiver, from each one of your interfaces. If both hosts have three interfaces, nine subflows will be created (3×3).

ndiffports The *ndiffports* path-manager will create a predetermined amount of subflows between a pair of IP-addresses (interfaces), transferring over different ports (hence the name). To configure the number of created subflows, a value ≥ 1 can be set with the following command:

```
1 sysctl -w net.mptcp.mptcp_ndiffports=[value]
```

binder This path-manager uses a model described in the paper *Binder: A System to Aggregate Multiple Internet Gateways in Community Networks* [10].

5.3.2.3 Scheduler

In section 4.3.4, we mentioned the *head-of-line blocking problem*, which can be a result of poor scheduling of packets. This can in turn affect the MPTCP performance. Accurately scheduling data across multiple paths, while avoiding the *head-of-line blocking problem* is difficult, but important. Therefore, it exists multiple implementations of schedulers, approaching the problem differently, and having different characteristics. You can select one of the pre-compiled schedulers by using the following command:

```
1 sysctl -w net.mptcp.mptcp_scheduler=[scheduler]
```

Currently, there exists multiple schedulers [37], but in the current implementation you have the choice between two of them:

default This scheduler is the default one, hence the name. It is also known as the *Lowest-RTT-First* (LowRTT). It will initially forward packets to the TCP subflows with the lowest RTT, until that subflow's congestion window is full. Thereafter, it will start transmitting packets on the subflow with the next lowest RTT, and so on. The default scheduler is currently the best performing, among of the ones available in this implementation [54].

roundrobin The *roundrobin* scheduler will transmit data according to the *round-robin*-principle - transmitting data on one subflow, then moving on to the next, with fixed intervals. It is possible to configure how many concurrent segments to be sent by modifying the value of `num_segments` in the `sysfs` - the default value is 1. In addition to this, it is also possible to set the boolean value `cwnd_limited`, specifying whether or not the scheduler attempts to fill the congestion window on all of the subflows (*true*), or if it prefers to have empty space in the congestion windows (*false*), in order to comply with the real *round-robin*, even if the different links are of varying capacities. This scheduler should not be used if you are unsure of the capacities of your links - then its performance would be very poor. This scheduler is mainly only interesting for academic- and testing purposes.

For example, to switch from the default scheduler to *roundrobin*, you can use the following command:

```
1 sysctl -w net.mptcp.mptcp_scheduler=roundrobin
```

Chapter 6

Methodology and Results

In the previous chapter, the testbed which forms the basis for our experiments was presented. The main objective of the NORNET testbed is to give us an environment where different aspects of performance can be evaluated under real network conditions at different multihomed sites. This chapter describes the experiments we are going to run, in order for us to analyze and evaluate MPTCP performance. Section 6.1 will give our expectations for MPTCP, and then section 6.2 will present the tools used for evaluation and our methods for gathering data. After examining the means of our experiments in section 6.4, we will conduct them in the NORNET CORE testbed. Results, comments, and possible explanations, will be given for each of the tests.

6.1 Expectations

Prior to having any actual results, we will present some expectations regarding the actual performance of MPTCP in real-life scenarios.

Throughput

We expect that the total throughput of an MPTCP connection is better, or at least equal to a single-path TCP connection running on the best available path. It would

be surprising if MPTCP provides a lower total throughput than a single-path connection. We have learned that the *Coupled Congestion Control Algorithm* acts less aggressive than the regular TCP congestion control algorithms, but can this result in an overall performance below our expectations?

Latency

Naturally, the latency on a given physical link should be more or less identical irrespective of which transport protocol used. But the latency can vary between links, and with MPTCP we have the opportunity of always benefiting from the link with the lowest latency. The exact behavior is difficult to predict, but it should be expected that overall MPTCP latency could be slightly higher compared to TCP over the best path. This can be explained by the fact that MPTCP has more overhead than TCP, in addition to the processing that has to be done in order to assemble the segments coming from each subflow in to a continuous byte-stream. However, we know that when links are congested, the latency often increases. We hope to see that MPTCP can balance traffic over the different paths in a way that will give benefits in form of low latency. As data are divided over different subflows, one unstable link can influence the performance as the host receives packets out-of-order and might have to wait for a retransmit - which most possibly will result in a delay.

Reliability

Reliability is probably the area where MPTCP really should excel, as its fundamental architecture is based on utilizing diverse paths connected to several interfaces simultaneously. Therefore, we expect that a seamless connection is provided even though one or more of the links get disconnected or doesn't respond.

6.2 Evaluation Tools

This section will give a description of the tools we used during our evaluations. These tools provided us with the experimental data needed, and helped us to analyze it.

Tcpdump

tcpdump is a packet analyzer which can output or store packets that are being transferred over the network(s) you are connected to through your interface(s). When

storing data, the *libpcap* (packet capture) API is used, which allows us to capture all network traffic and dump it to a local file. This file can later on be analyzed.

Wireshark

Similar to *tcpdump*, *Wireshark* is also a packet analyzer tool, but with a graphical user interface. In addition, it has some more advanced features for filtering and grouping of packets. *Wireshark* is used to study network communication protocols, network troubleshooting and for educational purposes.

NetPerfMeter

NetPerfMeter is a transport protocol evaluation tool. The software is open source, and currently support the use of TCP, UDP, MPTCP, SCTP and DCCP. *NetPerfMeter* measures the performance of network links, in terms of throughput and jitter. Data can be sent in each direction, between a passive- (server) and a active side (client). The data flows can be configured in a number of ways, including only using unidirectional or bidirectional flows and concurrent use of MPTCP and TCP. After running a performance test, flow statistics are recorded to scalar- and vector result files.

Ping

Ping is a statistical network software utility which is used to measure the reachability and RTT of a specified host connected to the Internet or on a local network. It operates by sending a ICMP (*Internet Control Message Protocol*) packet with an *Echo request* to the specified IP-address or host name, and then waits for the *Echo Reply* ICMP-packet. In addition to the minimum, maximum and average RTT, *Ping* also reports network errors and packet loss.

Traceroute

Traceroute is a network diagnostic utility, which shows you the route between two endpoints on a computer network. Unlike *Ping*, which only measures the delay to the destination, *traceroute* also measures the delay between the different hops on the route.

Tcptrack

Tcptrack is a monitoring utility which lists all TCP connections on the network interface given as an argument. The source- and destination addresses of each connection is given, in addition to the state and bandwidth usage.

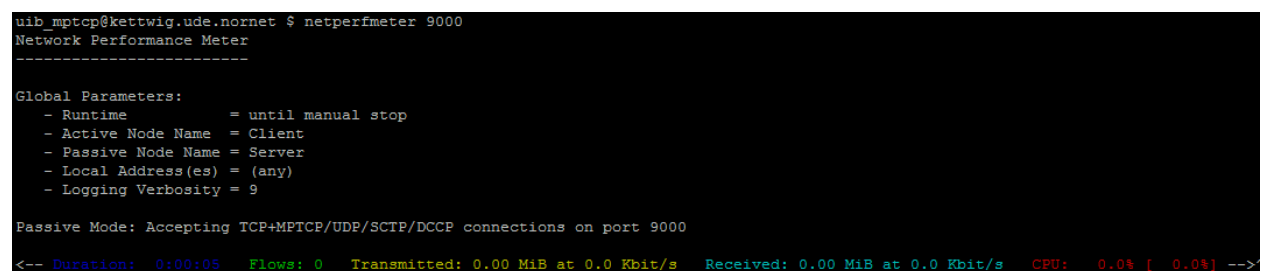
The combination of *tcpdump* and *Wireshark* allows us to study and analyze captured packet data from our testbed. In this way, we can also study MPTCP connection establishment, the creation of subflows, and look at the various TCP options that MPTCP make use of. Packet capture data can also be useful when analyzing subflows and the balancing of throughput between them, connection handovers and subflow latency. Given that MPTCP is still in the development process, and not yet fully deployed, there is not a large selection of performance evaluating tools with specific support for it. However, using the described tools, we are provided with the data we need in order to give an adequate analysis and evaluation.

6.2.1 Gathering Experimental Data

To collect experimental data, we first need to start a data transfer between two NORNET CORE endpoints (slivers). For this, we use *NetPerfMeter*, which we have to start at both the active side and the passive side. In section 5.2.5, we describe how you can access a sliver through *SSH*. We can easily access two slivers on the same computer by starting multiple shells.

After accessing the slivers, we can start a *NetPerfMeter* session. The passive side (server) has to be started first, which is done by the following command:

```
1 netperfmeter 9000
```



```
uib_mptcp@kettwig.ude.nornet $ netperfmeter 9000
Network Performance Meter
-----

Global Parameters:
- Runtime           = until manual stop
- Active Node Name  = Client
- Passive Node Name = Server
- Local Address(es) = (any)
- Logging Verbosity = 9

Passive Mode: Accepting TCP+MPTCP/UDP/SCTP/DCCP connections on port 9000
<-- Duration: 0:00:05  Flows: 0  Transmitted: 0.00 MiB at 0.0 Kbit/s  Received: 0.00 MiB at 0.0 Kbit/s  CPU: 0.0% [ 0.0%] -->
```

Figure 6.1: NetPerfMeter Passive Side

NetPerfMeter is now in server mode, waiting for connections on port 9000. Figure 6.1 shows *NetPerfMeter* listening at the passive side. The passive side will listen for con-

nections until it is manually stopped, which is desired in our setup. The receiving and sending rate is also shown in the figure. For a one-directional TCP flow, the passive side sending rate will naturally be zero. After the passive side is started, we can start the active side:

```
1 netperfmeter [IP address of passive side]:9000 \  
2 -runtime=60 \  
3 -tcp const0:const1460:const0:const0:cmt=off
```

This command will start a saturated TCP flow from the active to the passive side, transmitting for 60 seconds. The `-tcp` argument will open a TCP connection, and the following parameter will set some flow-specific variables. The first four variables, `outgoing_frame_rate`, `outgoing_frame_size`, `incoming_frame_rate`, `incoming_frame_size` (`const0:const1460:const0:const0`) may be substituted by the option "default", which will create a flow with default values.

The `cmt` argument configure the use of *Concurrent Multipath Transfer* (CMT). We use this option to switch between the use of MPTCP (`mptcp`) and TCP flows (`off`). To start an MPTCP flow between the active and passive side, with default flow-specific variables, we use the following command:

```
1 netperfmeter [IP address of passive side]:9000 \  
2 -runtime=60 \  
3 -tcp default:cmt=mptcp
```

The active side will now send data to the passive side using MPTCP, for 60 seconds. One of the most important features in *NetPerfMeter* is to easily get machine-readable result files. No statistic data from the connection will be stored unless we explicitly ask for it. The following command lets us save statistics from the connection in vector- and scalar files:

```
1 netperfmeter [IP address of passive side]:9000 \  
2 -runtime=60 \  
3 -scalar=scalars.sca \  
4 -vector=vectors.vec \  
5 -tcp default:cmt=mptcp \  
6
```

The performance of the data channels (like throughput and jitter) is evaluated and then stored on the passive side, as well as uploaded to the active side. We can then access these files using *Secure Copy* (SCP):

```
1 scp -i [RSA-key] [slice]@[node].[site].nornet: \  
2 [file name].[file format] [local destination]
```

Starting *NetPerfMeter* on the active side will show the following:

```
uib_mptcp@rennescoey.uis.nornet $ netperfmeter 10.30.42.138:9000 -runtime=60 -tcp default:cmt=mptcp
Network Performance Meter
-----

Active Mode:
- Measurement ID = 92e40377e7ecf6a7
- Remote Address = 10.30.42.138:9000
- Control Address = 10.30.42.138:9001 - connecting ... okay; sd=6

Flow #0: connecting MPTCP socket to 10.30.42.138:8999 ... okay <sd=7> <R1> <R2> <status=0> <R3> <R4> <status=0> okay

Global Parameters:
- Runtime = 60s
- Active Node Name = Client
- Passive Node Name = Server
- Local Address(es) = (any)
- Logging Verbosity = 9

+ MPTCP Flow (Flow ID #0 "Flow 0"):
- Receive Buffer Size: 233016
- Send Buffer Size: 233016
- Max. Message Size: 16000
- Defragment Timeout: 5000ms
- Outbound Frame Rate: 0.000000 (constant)
- Outbound Frame Size: 1440.000000 (constant)
- Inbound Frame Rate: 0.000000 (constant)
- Inbound Frame Size: 0.000000 (constant)
- On/Off: { *0 }
- Repeat On/Off: no
- Error on Abort: yes
- Debug: no
- No Delay: no
Congestion Control: default
Number of Diff. Ports: 4
Path Manager: fullmesh
Scheduler: default
- CMT: #4 (MPTCP)
- CCID: #0

Starting measurement ... <S1> <S2> <status=0> okay

<-- Duration: 0:00:11 Flows: 1 Transmitted: 7.83 MiB at 4850.4 Kbit/s Received: 0.00 MiB at 0.0 Kbit/s CPU: 2.1% [ 2.1% ] -->
```

Figure 6.2: NetPerfMeter Active Side

As you can see in figure 6.2, different connection data are provided. Just as in figure 6.1, sending- and receiving rate is shown, as well as duration, CPU usage and number of flows. The information given also includes some frame-rate and size information, as well as some MPTCP specifics, such as congestion control, path manager and scheduler. You can also see that both the congestion control and the scheduler are set to default, but are indeed switchable through *NetPerfMeter*. *NetPerfMeter* has the ability to set socket options before initiating an MPTCP connection, by specifying flow options as a parameter. We are

planning to do some experiments in respect to congestion controls and schedulers in the later sections.

To examine the experimental data which *NetPerfMeter* does not offer, we use *tcpdump* and *Wireshark*, as mentioned earlier. The NORNET CORE Linux distributions had *tcpdump* pre-installed, and it is easily used:

```
1 sudo tcpdump -ni eth0 -s0 -w [file name].pcap
```

This command generates a *pcap*-file containing all the packets transmitted or received over the *eth0* interface. In addition to *tcpdump* and *Wireshark*, we used *tcptrack* to provide us with some empirical data. We used *tcptrack* to monitor each subflow's throughput, and to ensure that MPTCP was working properly during our tests, by reading the source- and destination addresses, connection state, idle time and bandwidth usage for each subflow. *Tcptrack* as it is will not give us any statistics or the ability to plot data, but we used it solely for monitoring and observation reasons. It can be started at any time, as it just listens on the interface given as an argument. *Tcptrack* does not come pre-installed, but the following command will install it:

```
1 sudo dnf install tcptrack
```

The installation will have to be executed at each sliver where we want to use it. To start *tcptrack*, use the following command:

```
1 sudo tcptrack -i [interface]
```

Since the slivers have only one interface (with multiple logical addresses), we replace *[interface]* with *eth0*. *Tcptrack* will still be able to distinguish between the subflows. Figure 6.3 illustrates how *tcptrack* looks after starting an MPTCP connection:

Client	Server	State	Idle	A	Speed
10.1.2.146:39360	10.8.6.145:8999	ESTABLISHED	0s	*	1 MB/s
10.10.2.146:49178	10.1.6.145:8999	ESTABLISHED	1s		2 KB/s
10.9.2.146:36176	10.8.6.145:8999	ESTABLISHED	1s		8 KB/s
10.10.2.146:58150	10.8.6.145:8999	ESTABLISHED	1s		4 KB/s
10.1.2.146:33409	10.1.6.145:8999	ESTABLISHED	0s	*	16 MB/s
10.9.2.146:60106	10.1.6.145:8999	ESTABLISHED	1s		6 KB/s
TOTAL					17 MB/s
Connections 1-6 of 6			Unpaused	Unsorted	

Figure 6.3: Tcptrack During an MPTCP Transfer

More specifically, figure 6.3 shows how *tcptrack* provides us with per-flow information. Note that the number of flows equals the number of client ISPs \times the number of server ISPs (3×2). This substantiates what we mentioned in section 5.1, it is indeed a fully connected mesh.

6.3 Analysis of Packet Capture

In order to see how MPTCP connections operate in practice, the *tcpdump* tool was used to capture all packets during the initiation, data exchange, and closure parts of an MPTCP connection. The *Wireshark* tool was then used to examine the captured packet data. This section will present the most relevant options used for signaling, which are essential for connection handling. As described in section 4.3.1, the MPTCP signaling messages are sent in the *TCP Options* field, with the TCP Options number (*Kind*) 30. Screenshots from *Wireshark* are provided beneath, showing the MPTCP options in different scenarios.

The experimental data used in this case comes from an MPTCP transfer between the *rennesoey.uis.nor-net* node, located at the University of Stavanger with three connected ISPs, and the *nordberg.simula.nor-net* node, located at Simula Research Laboratory with four connected ISPs.

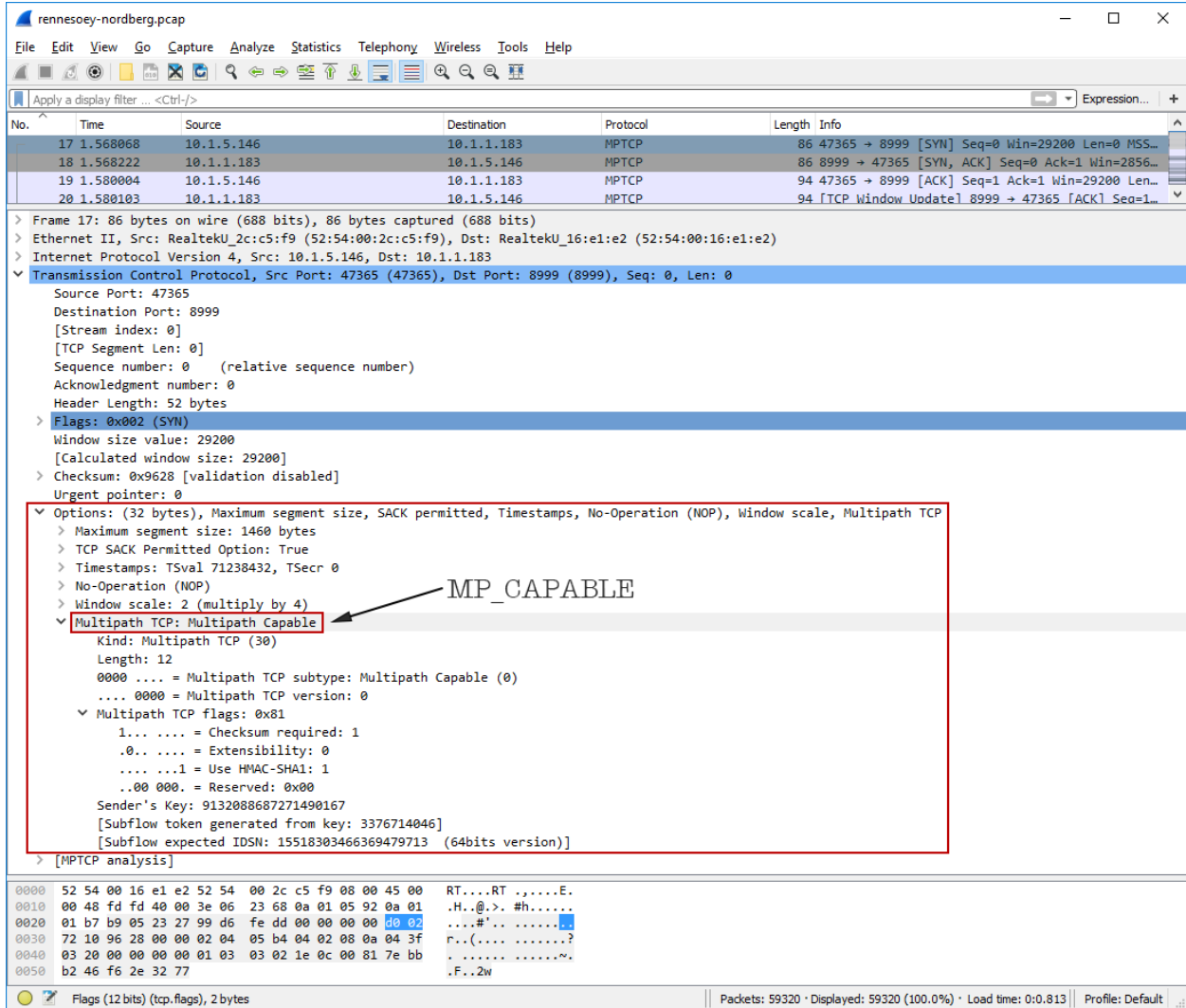


Figure 6.4: Wireshark screenshot - MP_CAPABLE signal

Figure 6.4 shows the establishment of a new MPTCP connection. The TCP Options of the selected packets in the screenshot can be seen inside the outer red rectangle. The connection starts with the first SYN packet sent from 10.1.5.146 (*rennesoey*) to 10.1.1.183 (*nordberg*), containing the MP_CAPABLE (MPTCP option value 0) option. In other words,

the *rennesoey* node asks the *nordberg* node if it is multipath capable. Further, we can see that the *nordberg* node replies with a SYN/ACK, and then the final ACK is sent from *rennesoey*. We can also see that this packet contains the *senders key*, and the *subflow token* generated from the key, as described in section 4.3.2. From the MPTCP flags, which is also sent through the TCP Options, we observe that the "Use HMAC-SHA1"-value is set to 1, which tells us that this is the agreed cryptographic algorithm to be used for the connection.

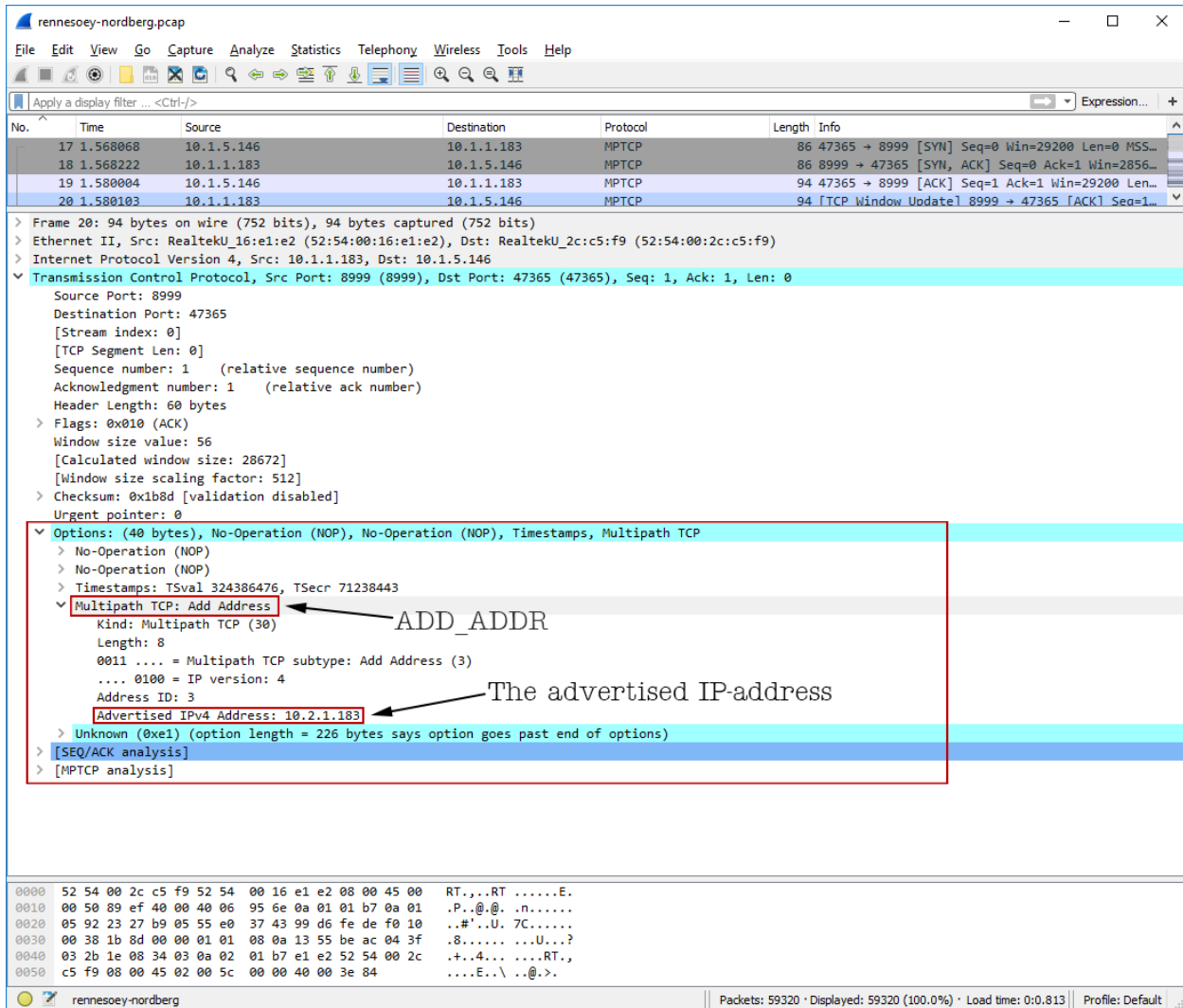


Figure 6.5: Wireshark screenshot - ADD_ADDR signal

Figure 6.5 shows us that *nordberg* sends a packet with the ADD_ADDR option to *rennesoey*,

where an additional IP-address is announced (10.2.1.183), as marked in the figure. This tells *rennesoey* that *nordberg* also can be reached at an additional address. Afterwards, as we can see in figure 6.6, *rennesoey* (10.1.5.146) sends a packet to the newly announced address 10.2.1.183 containing the MP_JOIN option (MPTCP option value 1), which means that it wants to create a subflow between the two IP-addresses. We can also see that the packet contains the receiver's *token*, the number needed in order for the receiver to know which MPTCP connection (the node might have several active connections) this subflow is supposed to be added to. The sender's *random number* is also provided.

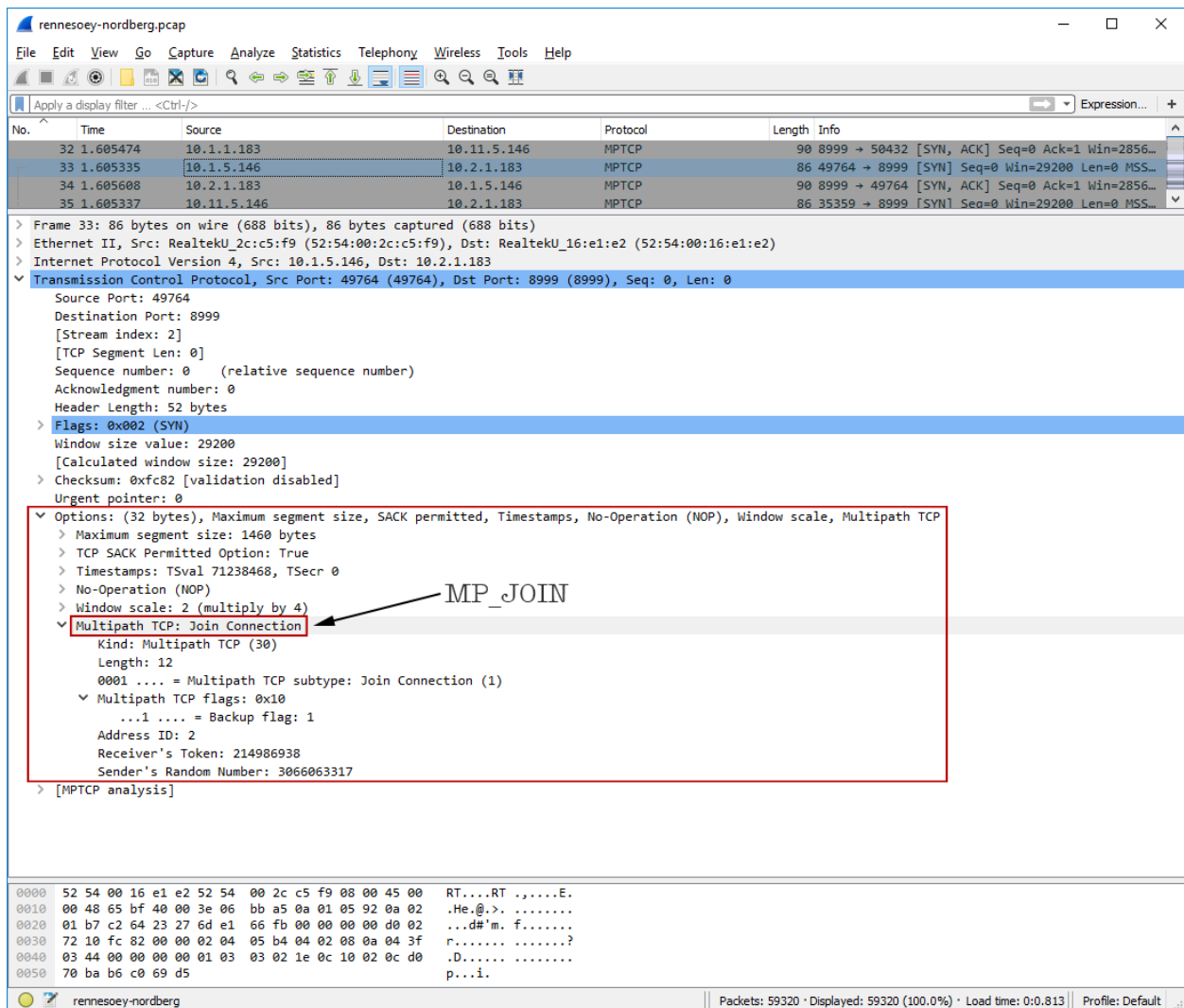


Figure 6.6: Wireshark screenshot - MP_JOIN signal

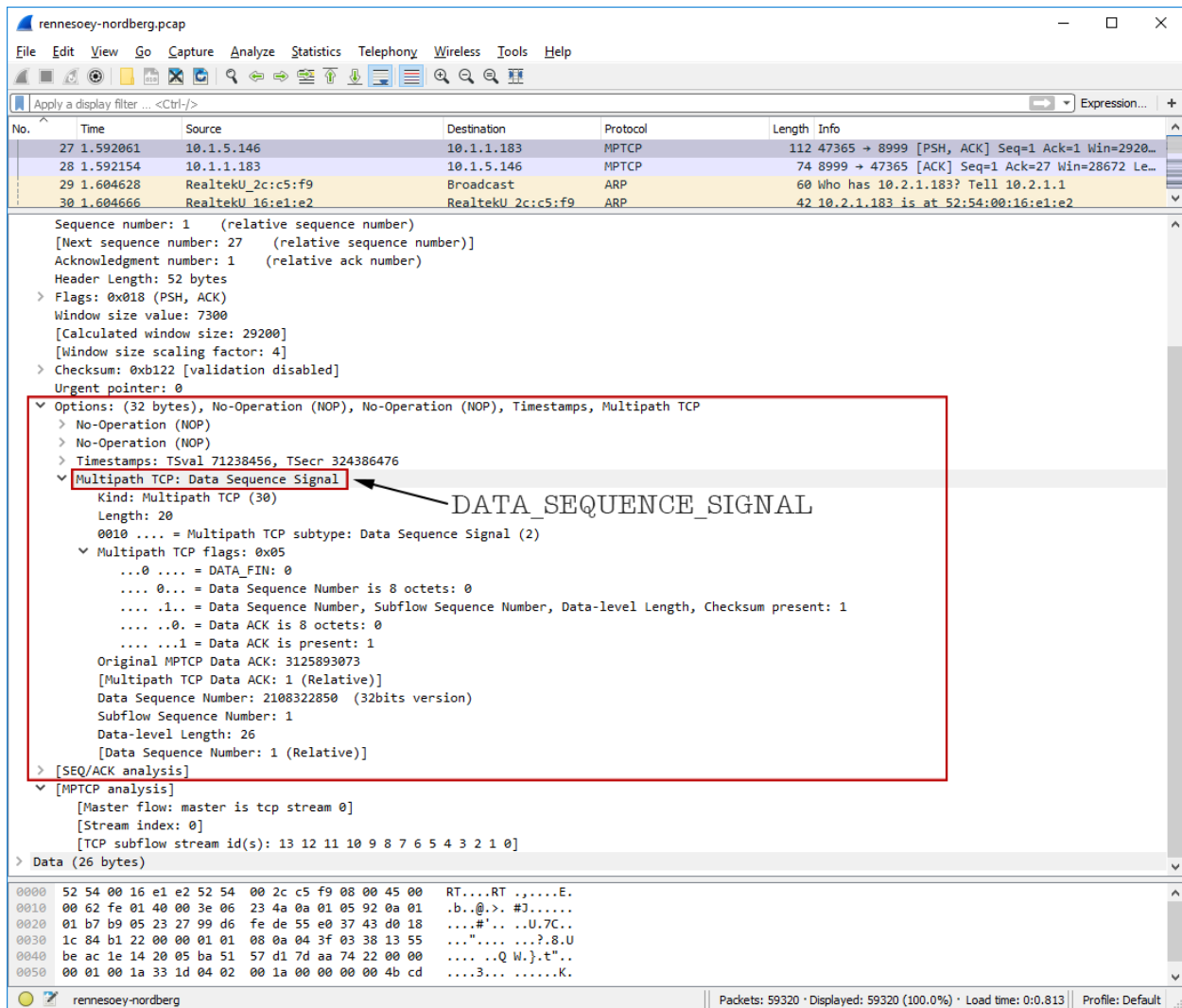


Figure 6.7: Wireshark screenshot - DSS signal

Figure 6.7 shows an MPTCP connection transmitting data - naturally with the `DATA_SEQUENCE_SIGNAL` option (MPTCP option value 2), giving the *data sequence number* and *subflow sequence number*, as described in section 4.3.4. In figure 6.8, the closure of the connection is showed. In the `DATA_SEQUENCE_SIGNAL` option, now the `DATA_FIN` flag is set, signaling the initiation of connection closure. As we also can observe from the figure, `FIN` is set in the TCP flags as well (marked in blue), to make sure that the connection is closed even if the TCP options are dropped. As mentioned in section 4.4.1, TCP Options can be dropped due to middleboxes.

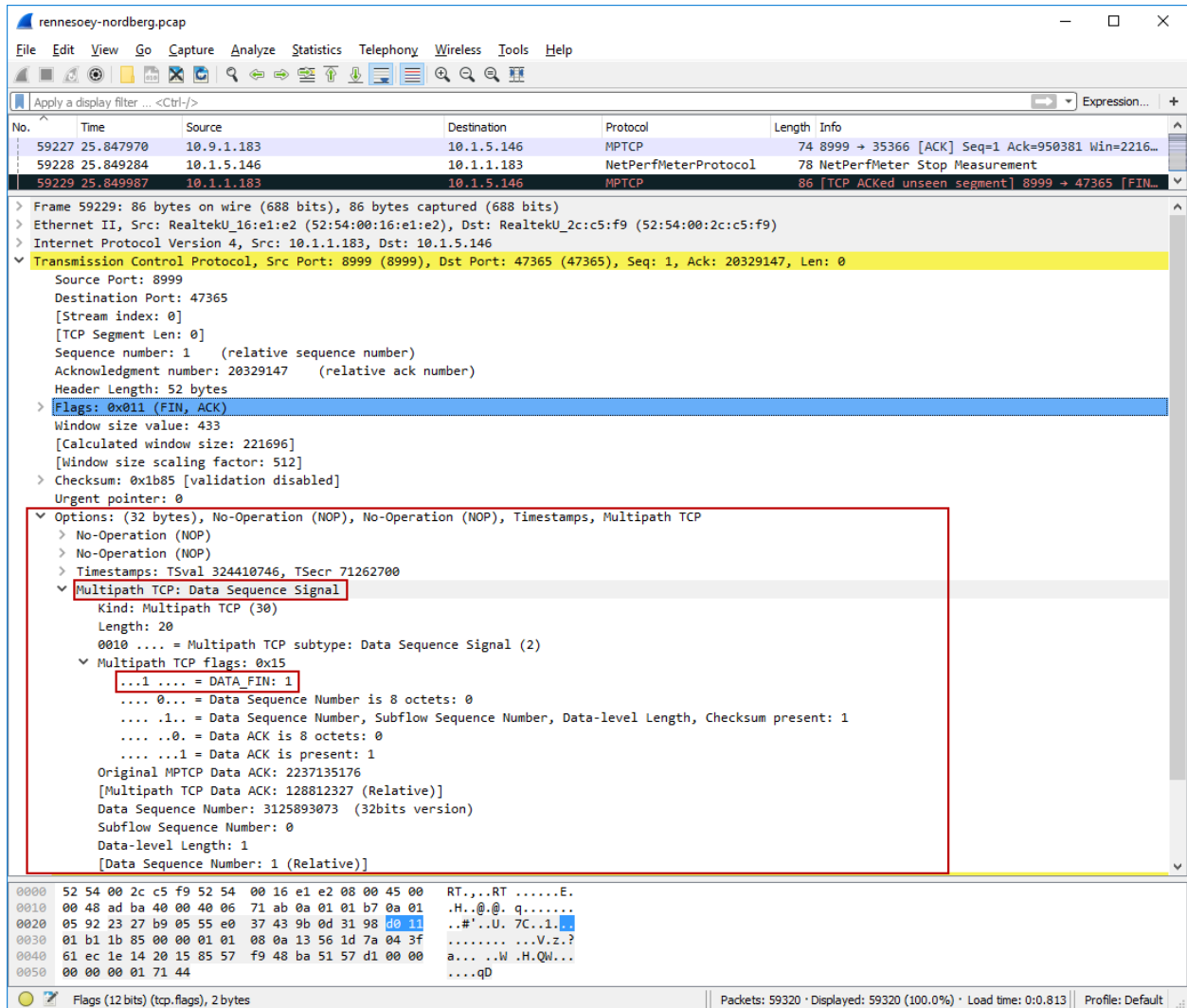


Figure 6.8: Wireshark screenshot - The Closure of a Connection

6.4 Experiments and Results

In the following subsections, different experiments will be presented and conducted in our testbed. Every test is run for **60 seconds**, between two specified nodes in NORNET CORE. We believe that 60 seconds will be enough for TCP to stabilize and give us sufficient data. A shorter run time may not show behaviour that develop over time, and a longer run time would result in too much experimental data for us to effectively analyze.

All throughput values stated are given in **Mbit/s**, and all latency values are given in **ms**. In the introduction part of each test, the ISPs of the involved nodes are given in parentheses after the node name. All MPTCP tests involving subflow analysis, throughput, latency and connection handover are run with the default MPTCP properties:

Path manager: fullmesh

Congestion Control: default (Coupled Congestion Control (*lia*))

Scheduler: default (Lowest-RTT-First (*LowRTT*))

6.4.1 MPTCP Subflow Analysis

Test 1: Subflow throughput: floeibanen (UiB) to bymarka (NTNU)

In this test, data are sent from **floeibanen** (*Uninett, BKK*), located at the University of Bergen, to **bymarka** (*Uninett, PowerTech*), located at the University of Trondheim (NTNU).

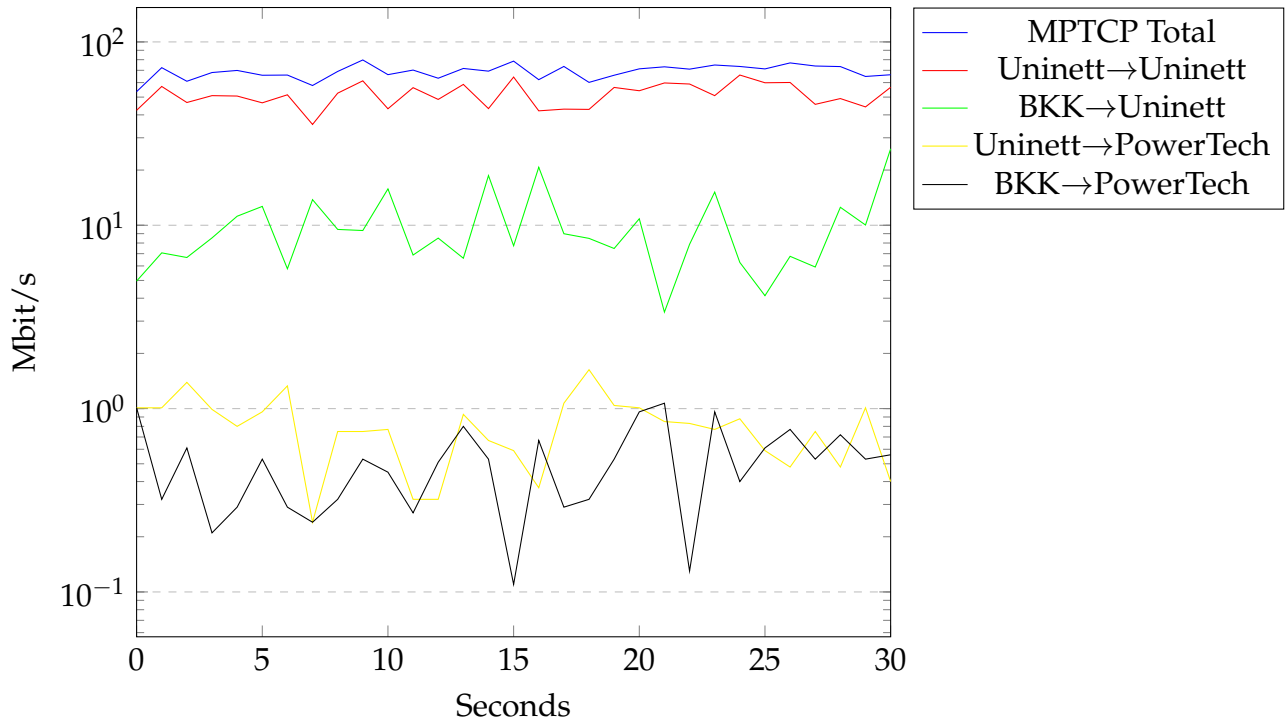


Figure 6.9: Subflow throughput analysis: floeibanen to bymarka

Subflow	From ISP	To ISP	Min	Max	Mean
MPTCP Total	Uninett, BKK	Uninett, PowerTech	57.83	79.73	69.36
Subflow 1	Uninett	Uninett	35.46	65.97	51.88
Subflow 2	BKK	Uninett	3.36	26.13	10.11
Subflow 3	Uninett	PowerTech	0.23	1.62	0.79
Subflow 4	BKK	PowerTech	0.10	1.06	0.50

Table 6.1: Subflow throughput analysis: floeibanen to bymarka

As the MPTCP API doesn't provide us with per-subflow statistics, we had to use *tcpdump* and *Wireshark*. In *Wireshark*, we added IP-address filters so that we could manually extract data from each of the subflows. As this was very time consuming, we chose to run this test for 30 seconds, seeing that this is mainly a proof of concept. As we expected from MPTCP, it consists of multiple TCP subflows, illustrated in figure 6.9. From the figure, we can see (2×2) different flows (excluding MPTCP total). This is because of the tunnels in a full mesh setup from each of *floeibanen's* ISPs to each of *bymarka's* ISPs. The flow

diversity is noteworthy, and is explained by each flow running over links of different characteristics, and therefore throughput and delay will vary. The throughput difference was of such extent, that we chose to plot the figure in a logarithmic scale to see how the weaker links behaved. The drawback of doing this is that the fluctuating characteristic of the total throughput is not that visible in the figure.

However, in table 6.1, each subflow's **min**, **max** and **mean** throughput values are given. We can see from the table that the *Uninett*→*Uninett* subflow dominates the others. This is explained by the fact that the *Uninett* links are high-bandwidth fiber links, as well as the behaviour of the default scheduler of MPTCP, *LowRTT*. The *LowRTT* scheduler might also explain the low throughputs to the *PowerTech*-links (ADSL), as it tries to avoid high-latency links. Also, as demonstrated in later tests, we will see that poor performance is a recurring issue regarding DSL-links. The differences in available link characteristics can therefore explain the huge differences in MPTCP subflow characteristics. The weaker links aren't necessarily boosting the throughput, but maintaining a constant connection between client and server if other links break down. Undoubtedly, in the case of an important Internet service running MPTCP, the weaker links keeping the service up and running can be the difference between failure or success.

6.4.1.1 Subflow Routing

When sending data out on links from different ISPs, the routing of the packets can vary. The paths may be disjoint, or they may meet at a point on the way. It would be interesting to look at the routing of packets for each of the subflows in figure 6.9 and the other tests, as the routing of packets can help us analyze the gathered data furthermore. As packets on different subflows may traverse different paths, the network may introduce different delays and bottlenecks to the subflows. If a subflow experiences high latency and low throughput, it may be partially explained by the subflow's routing. Additionally, it would be easier to identify possible bottlenecks in the network.

Normally, monitoring a packet's path through a network is straightforward using *traceroute*. Unfortunately, as described in section 5.1, due to the GRE-tunnel setup of NORNET CORE, *traceroute* will not work as desired. Instead of showing each hop of the path between any two nodes, *traceroute* will only show three hops, as demonstrated in

listing 6.1.

```
tracert to nordberg.simula.nor-net (10.1.1.104), 30 hops max, 60 byte packets
 1 stavanger.uninett.uis.nor-net (10.1.5.1) 0.772 ms 0.850 ms 0.813 ms
 2 uninett.uis.uninett.simula.nor-net (192.168.71.69) 11.542 ms 11.792 ms 11.759 ms
 3 nordberg.uninett.simula.nor-net (10.1.1.104) 14.399 ms 14.345 ms 14.333 ms
```

Listing 6.1: Example of *tracert* from *rennesoey* to *nordberg*

6.4.2 Throughput

6.4.2.1 MPTCP vs. TCP Throughput, Independent Flows

The following tests will compare total MPTCP throughput, using all available links between the specified nodes (*fullmesh*), to single-path TCP throughput using all combinations of links. All the flows run independently of each other, i.e. one 60-second test has been performed for each of the flows. As we will see, MPTCP is highly dependent on the combination of links. To demonstrate the various properties of MPTCP in the best possible way, we have chosen pairs of nodes with different link-characteristics. This is also done to cover different scenarios that is likely to take place in a real life setting. Latency values in the throughput tests are only given for the single-path TCP flows. Currently, it doesn't exist any tool that give an average latency value for an MPTCP transfer, nor does the MPTCP API provide this information. However, *Wireshark* is able to plot RTT pr. packet from captured packet data. As we will see in section 6.4.4, we will use this to evaluate latency in MPTCP.

Test 2: *lungegaardsvannet* (UiB) to *nordlys* (UNIS)

In this test, data are sent from **lungegaardsvannet** (*Uninett, BKK*), located at the University of Bergen, to **nordlys** (*Uninett, Telenor*), located at the University Centre in Svalbard.

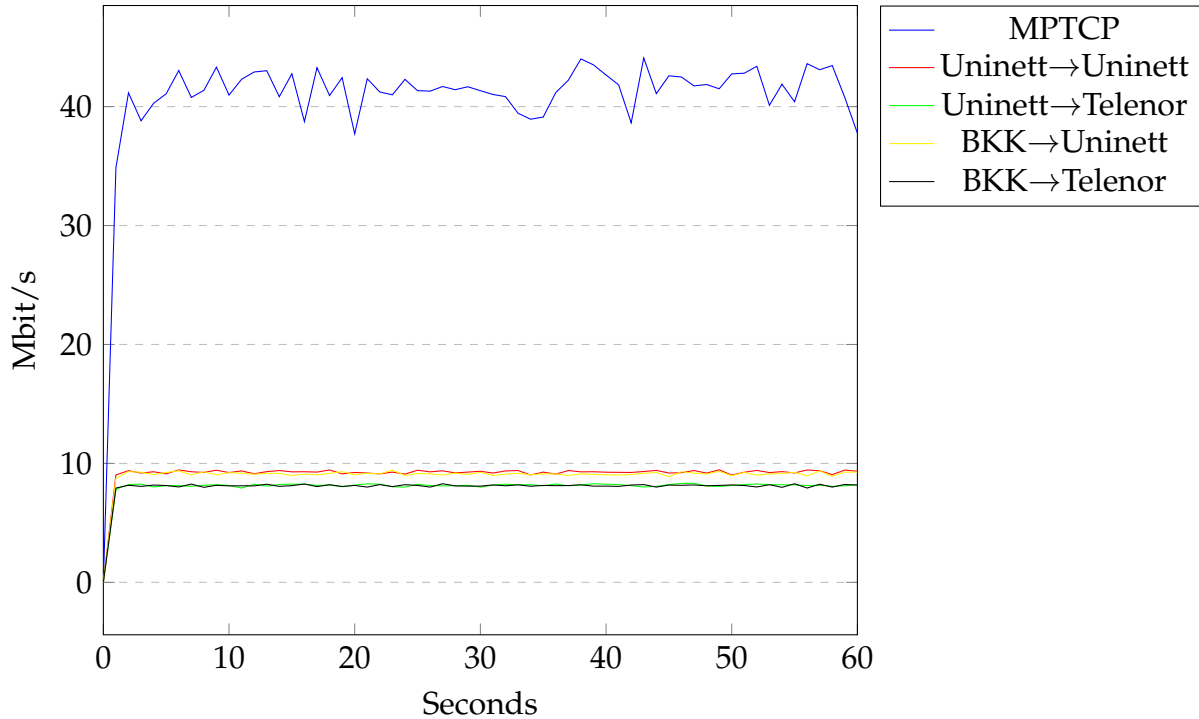


Figure 6.10: Independent throughput: lungegaardsvannet to nordlys

Protocol	From ISP	To ISP	Min	Max	Mean	Avg. Latency
MPTCP	Uninett, BKK	Uninett, Telenor	34.86	44.09	41.42	
TCP	Uninett	Uninett	9.02	9.46	9.28	39.83 ms
TCP	Uninett	Telenor	7.83	8.33	8.16	44.87 ms
TCP	BKK	Uninett	8.74	9.44	9.13	39.67 ms
TCP	BKK	Telenor	7.93	8.29	8.13	45.82 ms
Total TCP throughput (mean): 34.7 Mbit/s						

Table 6.2: Independent throughput: lungegaardsvannet to nordlys

The experimental data collected in this test are visualized in figure 6.10, and the most important metrics are presented in table 6.2. As we can observe in the comparison above, the throughput that MPTCP provide is actually a little higher compared to the total mean TCP throughput. These numbers are interesting, given that the MPTCP connection in practice consists of the four TCP flows listed above (as subflows). In section 4.3.7, we described that if an MPTCP subflow influences the link loss rate, it can possibly achieve slightly higher throughput than a TCP flow over the same link. This can help us explain why

MPTCP aggregate throughput is higher than the aggregate throughput of independent TCP flows over the same paths. Random factors like network congestion can potentially also affect our results. Since the flows are transmitting at different points in time, other network traffic may not have influenced each flow identically. Given the fact that both nodes in this test are connected with two fiber links, we observe from the plot that the throughput is relatively stable. As we know, fiber links provide stable throughput and low latency even when links are saturated, which can explain why the results from this test correspond well with theory. As we will see later on, stable throughput is not always the case.

Test 3: rennesoey (UiS) to floeibanen (UiB)

In this test, data are sent from **rennesoey** (*Uninett, Altibox, PowerTech*), located at the University of Stavanger, to **floeibanen** (*Uninett, BKK*), located at the University of Bergen.

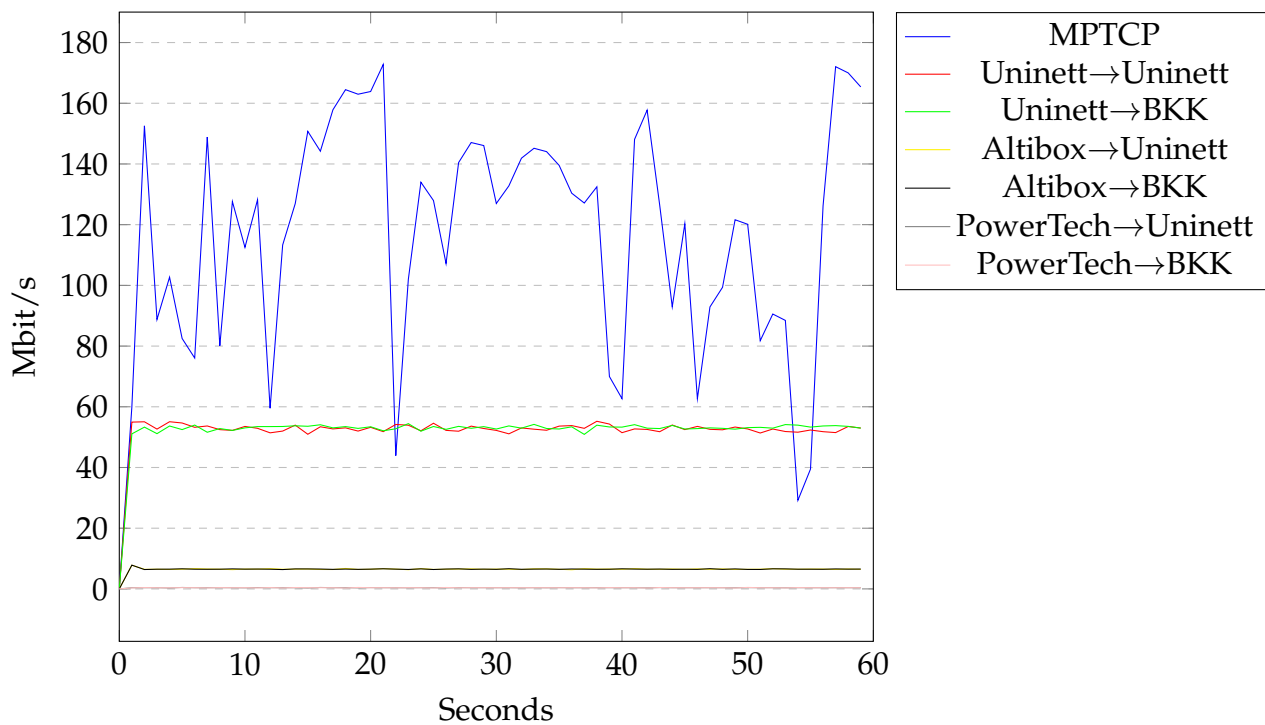


Figure 6.11: Independent throughput: rennesoey to floeibanen

Protocol	From ISP	To ISP	Min	Max	Mean	Avg. Latency
MPTCP	Uninett, Altibox, PowerTech	Uninett, BKK	29.21	172.76	113.12	
TCP	Uninett	Uninett	50.95	55.23	52.88	6.47 ms
TCP	Uninett	BKK	50.91	54.45	53.12	7.15 ms
TCP	Altibox	Uninett	6.31	6.74	6.53	19.96 ms
TCP	Altibox	BKK	6.37	6.66	6.53	18.41 ms
TCP	PowerTech	Uninett	0.29	0.48	0.30	147.80 ms
TCP	PowerTech	BKK	0.30	0.49	0.39	119.84 ms
Total TCP throughput (mean): 119.75 Mbit/s						

Table 6.3: Independent throughput: rennesoey to floeibanen

As it can be observed from table 6.3, the mean MPTCP throughput is close to the aggregate mean TCP throughput from the individual flows. But it can also be seen from figure 6.11 that the MPTCP throughput is relatively fluctuating. We believe that these frequent variations in bit rate is caused by **rennesoey's** *PowerTech*-link, which is actually an ADSL-link. As we know, ADSL-links have very weak upstream due to its asymmetric design, in addition to high latency. This can be confirmed by the data in table 6.3. We also experienced a considerable amount of packet loss on this link. We know that these factors combined lead to the *head-of-line blocking problem*, defined in section 4.3.4 - which again can lead to the sudden drops in throughput as experienced in this test.

Test 4: nordlys (UNIS) to kettwig (UDE)

In this test, data are sent from **nordlys** (*Uninett, Telenor*), located at the University of Svalbard, to **kettwig** (*DFN, Versatel*), located at the University of Duisburg-Essen.

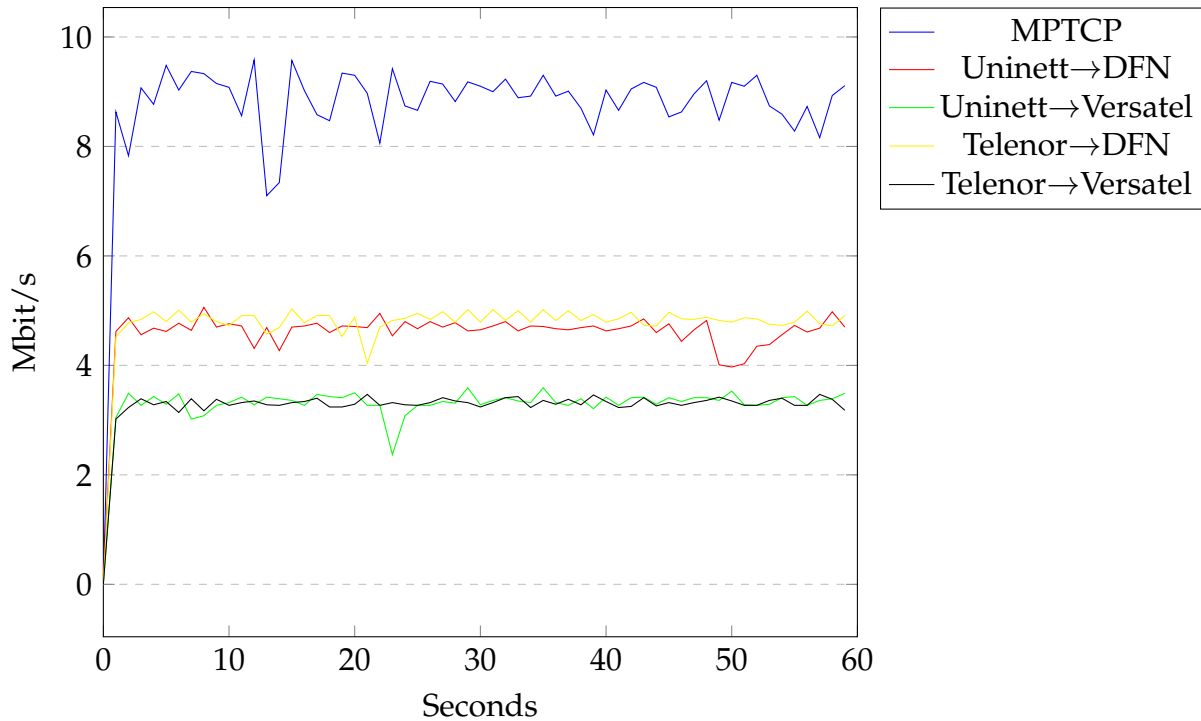


Figure 6.12: Independent throughput: nordlys to kettwig

Protocol	From ISP	To ISP	Min	Max	Mean	Avg. Latency
MPTCP	Uninett, Telenor	DFN, Versatel	7.1	9.58	8.86	
TCP	Uninett	DFN	3.97	5.06	4.64	79.53 ms
TCP	Uninett	Versatel	4.04	3.59	3.32	244.96 ms
TCP	Telenor	DFN	4.04	5.03	4.82	73.31 ms
TCP	Telenor	Versatel	3.02	3.47	3.31	286.95 ms
Total TCP throughput (mean): 16.09 Mbit/s						

Table 6.4: Independent throughput: nordlys to kettwig

In this test we see that the MPTCP throughput is approximately half of the sum of the TCP flows. This result tells us that even if the MPTCP cannot fully utilize each link, it can still perform according to the design goals. As long as MPTCP achieves no worse throughput than the best TCP flow, it does what it is supposed to do. From table 6.4, we can see that there are significant latency variations between the TCP flows. These variations can help us explain why MPTCP is not able to fully utilize each link's potential. The *resource pooling principle* is an obvious goal of MPTCP. To reach this goal, all available paths should

be considered a shared resource, i.e. behaving as the sum of the individual resources. A performance issue of MPTCP is that MPTCP may require larger receive buffers when subflows have different delays, as it does in this case. Another reason why MPTCP can experience performance issues, is when the network routing does not have completely disjoint paths between sender and receiver. The occurrence of shared bottlenecks between the MPTCP subflows may explain this reduced performance, as each of the subflows go through the same bottleneck, and act fair to each other.

6.4.3 Fairness

The fairness tests, which are essentially throughput tests with concurrent data flows, are performed with *NetPerfMeter*. Every test is run for 60 seconds, between two specified nodes in NORNET CORE. The MPTCP flows run with the default properties, as defined in the beginning of section 6.4.

6.4.3.1 MPTCP vs. TCP, Two Concurrent Flows

Test 5: rennesoey (UiS) to lungegaardsvannet (UiB) (2 concurrent flows)

In this test, data are sent from **rennesoey** (*Uninett, Altibox, PowerTech*), located at the University of Stavanger, to **lungegaardsvannet** (*Uninett, BKK*), located at the University of Bergen.

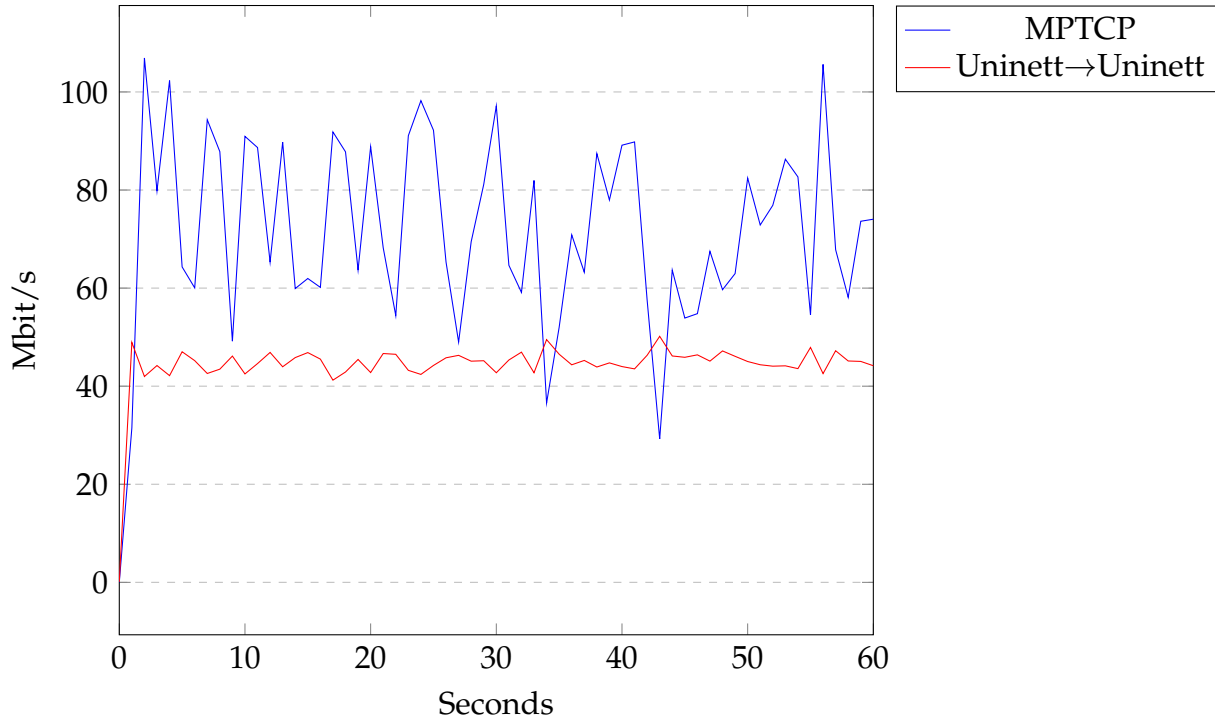


Figure 6.13: Fairness, two concurrent flows: rennesoey to lungegaardsvannet

Protocol	From ISP	To ISP	Min	Max	Mean
MPTCP	Uninett, Altibox, PowerTech	Uninett, BKK	29.24	106.93	72.47
TCP	Uninett	Uninett	41.23	50.17	45.03

Table 6.5: Fairness, two concurrent flows: rennesoey to lungegaardsvannet

Protocol	From ISP	To ISP	Min	Max	Mean
TCP	Uninett	Uninett	49.39	56.49	53.26
TCP	Uninett	Uninett	50.16	56.90	53.24
TCP	Uninett	BKK	43.02	48.53	46.82

Table 6.6: Fairness, three concurrent flows: rennesoey to lungegaardsvannet

In test 3, where the same links as in this test are used, we saw that the mean MPTCP throughput was 113.12 Mbit/s - here it is 72.47 Mbit/s, a decrease of 41 Mbit/s. On the other side, the *Uninett*→*Uninett* mean throughput was 52.88 Mbit/s - here it is 45.03

Mbit/s, a decrease of 7 Mbit/s. The only difference is that in this test, the flows transmit concurrently. We observe that MPTCP acts fair to the TCP flow, by not utilizing an unfair share of the *Uninett*-link, but rather decrease its own total throughput, compared to test 3 where the flows run independently of each other. In table 6.6, we are trying to reproduce the same amount of load that the *Uninett*-link gets when running one MPTCP-flow and one TCP-flow concurrently. This is done by performing a separate test with three concurrent TCP-flows, over the same combination of links as the MPTCP- and TCP-flow utilizes in this test. As we observe, the throughput values are a bit higher for each flow when running only TCP-flows concurrently.

Test 6: lungegaardsvannet (UiB) to kettwig (UDE) (2 concurrent flows)

In this test, data are sent from **lungegaardsvannet** (*Uninett, BKK*), located at the University of Bergen, to **kettwig** (*DFN, Versatel*), located at the University of Duisburg-Essen.

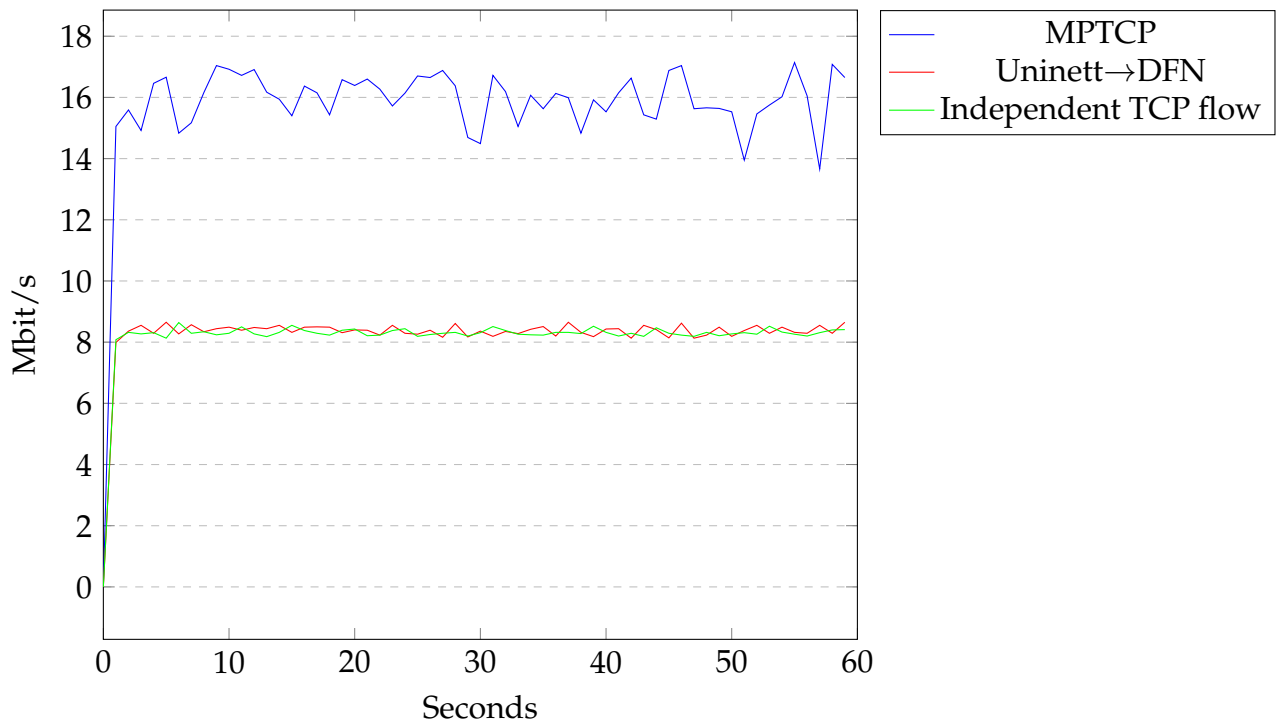


Figure 6.14: Fairness, two concurrent flows: nordberg to kettwig

In section 4.1.2.5, we described how the MPTCP should be fair to other MPTCP and TCP flows in the occurrence of shared bottlenecks. Because of this, we wanted to test the fairness of MPTCP from a domestic site to an international one. Our choice of nodes is based on the assumption that there is a higher possibility to encounter a shared bottleneck when transmitting data abroad. From figure 6.14, we can see that MPTCP does in fact have a higher throughput rate than TCP, which is expected. When running MPTCP and TCP (*Uninett*→*DFN*) concurrently, the throughput of the TCP-flow was approximately 8 Mbit/s, and 16 Mbit/s for the MPTCP-flow. To determine if MPTCP is fair to TCP or not, we also ran a TCP-flow independently after the initial test. As we can observe, the independent TCP flow achieved approximately the same throughput as when running concurrently with MPTCP. These observations can be explained by a couple of reasons. First and foremost, it can tell us that MPTCP does in fact not steal resources from other concurrent flows, that it is fair. It can also be explained by the fact that there might be no shared bottlenecks, i.e. the links are capable of handling all the flows without the flows fighting for resources. Since TCP will try to use all available bandwidth, for the latter to be true, it may suggest that every TCP flow in the network is limited to some fixed throughput rate.

6.4.3.2 MPTCP vs. TCP, Four Concurrent Flows

Test 7: rennesoey (UiS) to lungegaardsvannet (UiB) (4 concurrent flows)

In this test, data are sent from **rennesoey** (*Uninett, Altibox, PowerTech*), located at the University of Stavanger, to **lungegaardsvannet** (*Uninett, BKK*), located at the University of Bergen.

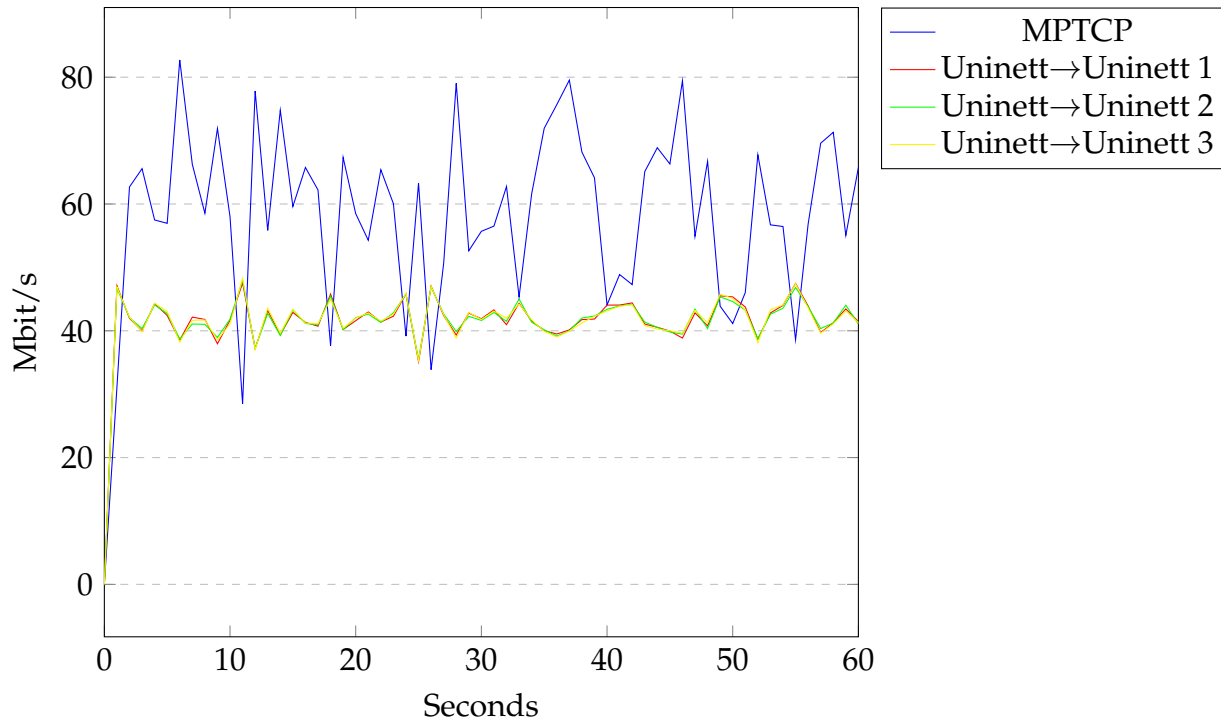


Figure 6.15: Fairness, four concurrent flows: rennesoey to lungegaardsvannet

Compared to test 5, where the same nodes run with two concurrent flows, we observe that the mean throughput of the TCP flows are approximately identical, but the MPTCP flow's throughput has decreased from a mean of 72.47 Mbit/s to a mean of 59.15 Mbit/s in this test. This observation shows us that MPTCP is trying to avoid taking up bandwidth from the TCP flows.

Test 8: nordberg (Simula) to kettwig (UDE) (4 concurrent flows)

In this test, data are sent from **nordberg** (*Uninett, Kvantel, PowerTech*), located at Simula Research Laboratory, to **kettwig** (*DFN, Versatel*), located at the University of Duisburg-Essen.

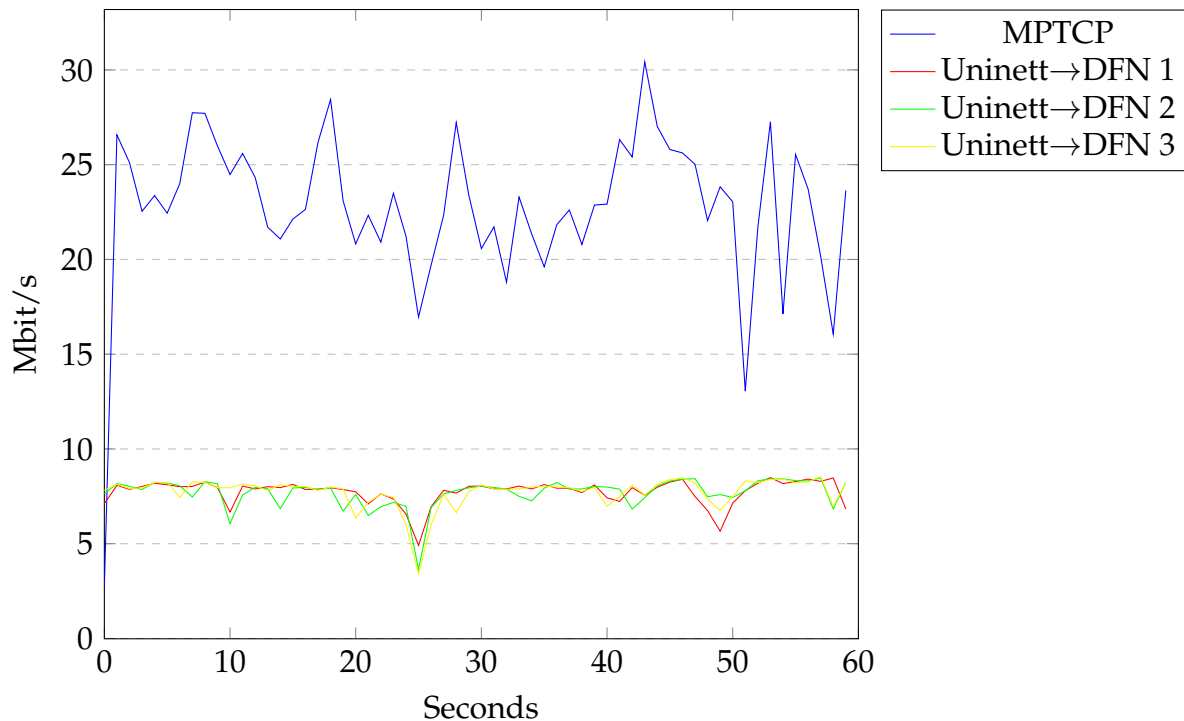


Figure 6.16: Fairness, four concurrent flows: lungegaardsvannet to kettwig

In this test, we are doing the same as in test 6, i.e. testing fairness from a domestic site, to an international site. Both tests show similar tendencies, MPTCP is dominant, while the TCP flows are at approximately 8 Mbit/s. This observation substantiates our statement of TCP flows being limited to a certain bit rate. We believe traffic shaping and rate limiting are the explanation of our observations of TCP having "constant" bit rate regardless of how many concurrent TCP flows we run (up to some point). We expect each TCP flow to have decreased throughput when increasing the number of TCP flows running concurrently.

When a link becomes saturated up to a significant level of contention, eventually bufferbloat will occur due to overly sized buffers in the network, substantially increasing the network latency. Traffic shaping is a technique to prevent these events from occurring and keeping latency in check. By rate limiting each TCP flow, it controls the volume of traffic being sent into a network. If this is the case, the constant bit rate of TCP is as expected, but at least we would also expect the MPTCP flow to behave like in test 6. However, in this test, MPTCP performs better than MPTCP in test 6. This is not as expected, as we would think that more traffic in the network would suggest lower throughput per

flow. Although, network traffic caused by others, and other random factors can explain this variation in throughput.

6.4.4 Latency

In this subsection, we will study MPTCP latency, and do comparisons to latency in single-path TCP flows. As the MPTCP API doesn't provide the end-user with statistical data, e.g. average latency in an MPTCP flow (or it's subflows), we had to study *pcap*-data which was captured during MPTCP data-transmission.

Test 9: Latency: ekeberg (UiO) to bymarka (NTNU)

In this test, data are sent from **ekeberg** (*Uninett, Broadnet, PowerTech*), located at the University of Oslo, to **bymarka** (*DFN, Versatel*), located at the Norwegian University of Science and Technology.

In the following figures, we have plotted the RTT for each packet during a data transmission using *NetPerfMeter* - marked as a dot in the scatter plot. Dots are color coded with respect to their RTT - the red dots naturally indicate packets with high RTT (and are mostly consisting of outliers). Blue and gray dots are representing values in the region around the average, whilst the yellow ones are in between.

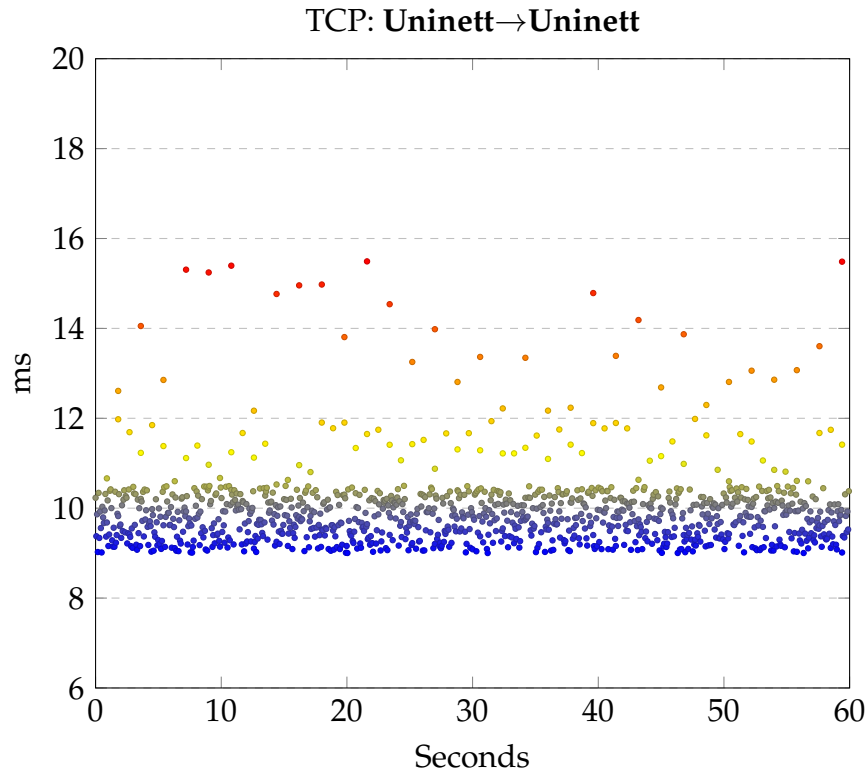


Figure 6.17: Latency: ekeberg to bymarka (TCP: Uninett→Uninett)

From the figure above we observe that the TCP-flow deliver fairly stable and constant latency, with RTT pr. packet mostly around 9 to 10.5 ms. We can also see that there are some outliers around 11-12 ms, and a few at 15 ms, but this is quite normal behavior.

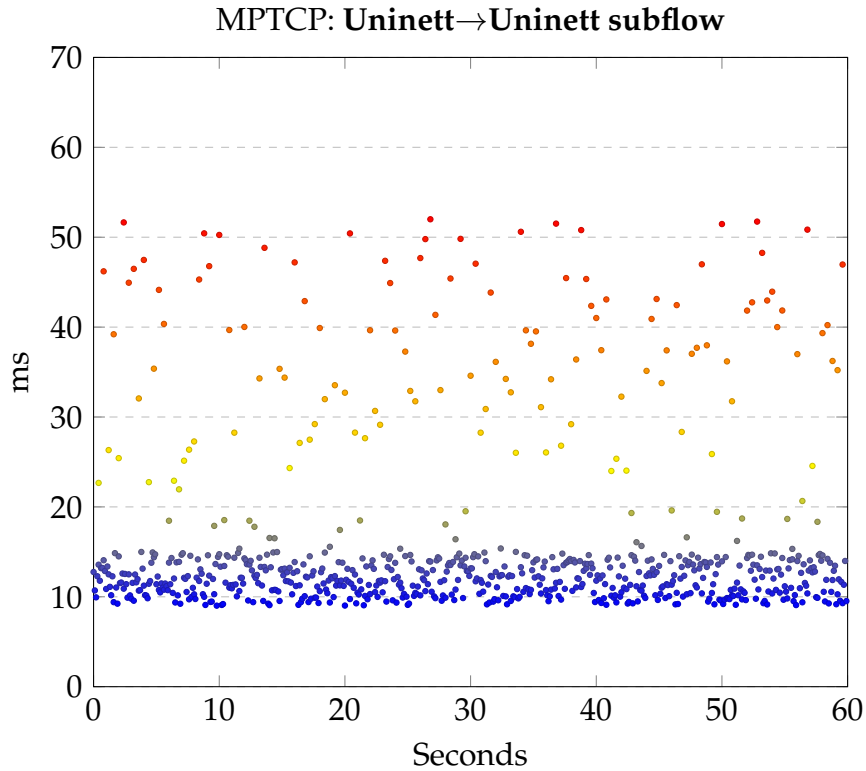


Figure 6.18: Latency: ekeberg to bymarka (MPTCP: Uninett→Uninett subflow)

The subflow shown in figure 6.18 is transmitting between the exact same links as the TCP-flow in figure 6.17, but in this case it transmits as a subflow of an MPTCP connection between *ekeberg* and *bymarka*. As we can see, the average RTT is a bit higher compared to the TCP-flow, in addition to several more outliers, spanning from ~ 15 to ~ 50 ms. This demonstrates that this subflow is indeed affected by the other subflows belonging to the MPTCP connection and potential MPTCP overhead, which results in a slightly higher average and a higher number of outliers. It is important to note that two of the links belonging to *ekeberg* are ADSL-links, which as we have seen can influence the performance of the overall flow.

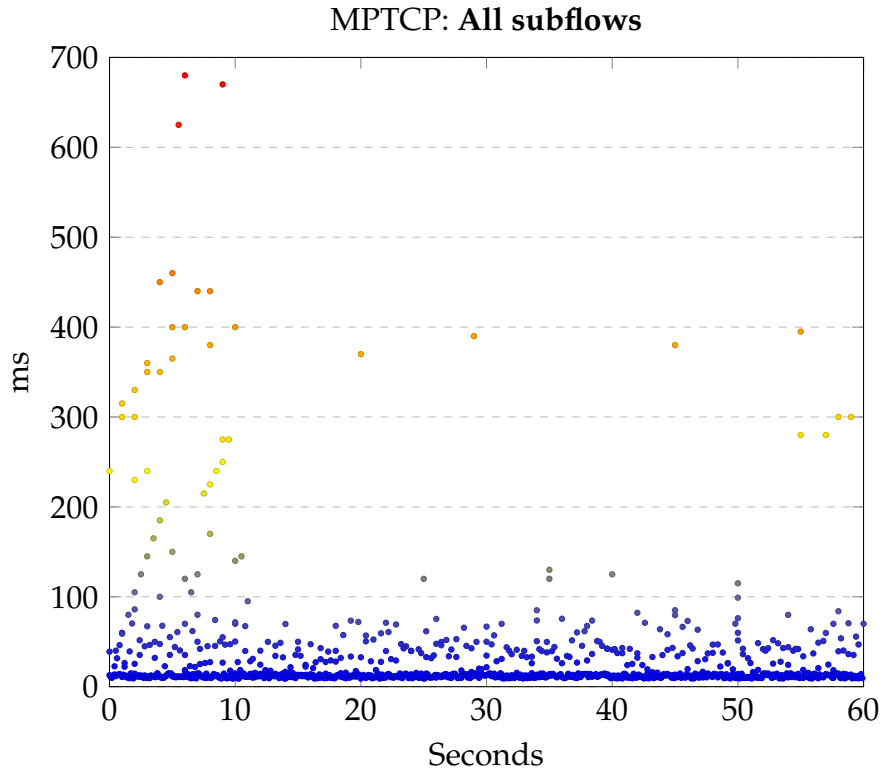


Figure 6.19: Latency: ekeberg to bymarka (MPTCP: All subflows)

Figure 6.19 consists of the same *Uninett*→*Uninett*-subflow data as in figure 6.18, but it also includes the RTT pr. packet of all the other subflows belonging to the MPTCP connection. Because of the high RTT of the outliers, the relatively low and stable RTT values of the *Uninett*→*Uninett*-subflow are hard to determine from the plot, but are seen as a solid line along the x-axis. In the beginning of the transmission, we experience numerous outliers - packets with extremely high RTT. Given the fact that we have already plotted the *Uninett*→*Uninett* subflow, we know that these outliers belong to the two ADSL-links of *ekeberg*. We believe that the reason to why most of the outliers disappear after the first 10 seconds, is because of the the *LowRTT*-scheduler. When the scheduler experiences the high RTT of the packets over the ADSL-links, it schedules less packets over those links, in accordance with the third goal of MPTCP, to balance the congestion in the best possible way.

Test 10: Latency: nordlys (UNIS) to lungegaardsvannet (UiB)

In this test, data are sent from **nordlys** (*Uninett, Telenor*), located at the University Centre in Svalbard, to **lungegaardsvannet** (*Uninett, BKK*), located at the University of Bergen.

Seeing that the ADSL-links caused some high latency in the previous test, in this test we have specifically chosen two nodes only connected with fiber-links. If links with extreme latency issues, like we experienced with the ADSL-links in some of our tests, are being utilized, all the packets on the faster links will have to wait for the slower ones to arrive.

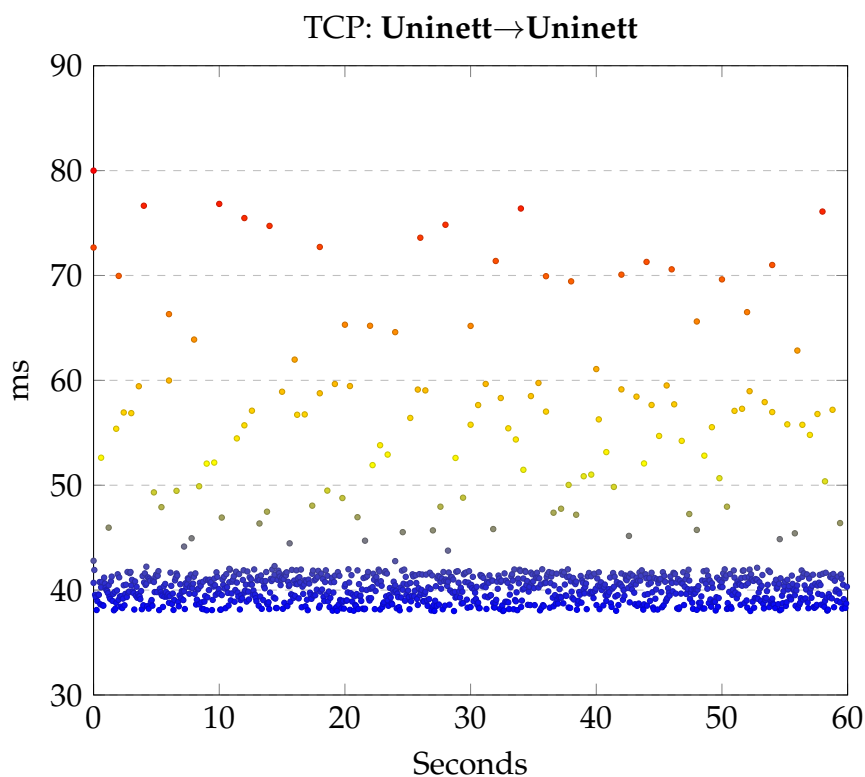


Figure 6.20: Latency: nordlys to lungegaardsvannet (TCP: Uninett→Uninett)

In figure 6.20, we observe that the RTT pr. packet is relatively stable around 40 ms, with an even spread of packets with some higher RTT, mainly in the range 30 to 80 ms. We note ourselves that due to the high geographically distance between the nodes in this test, some variation in the latency can be expected.

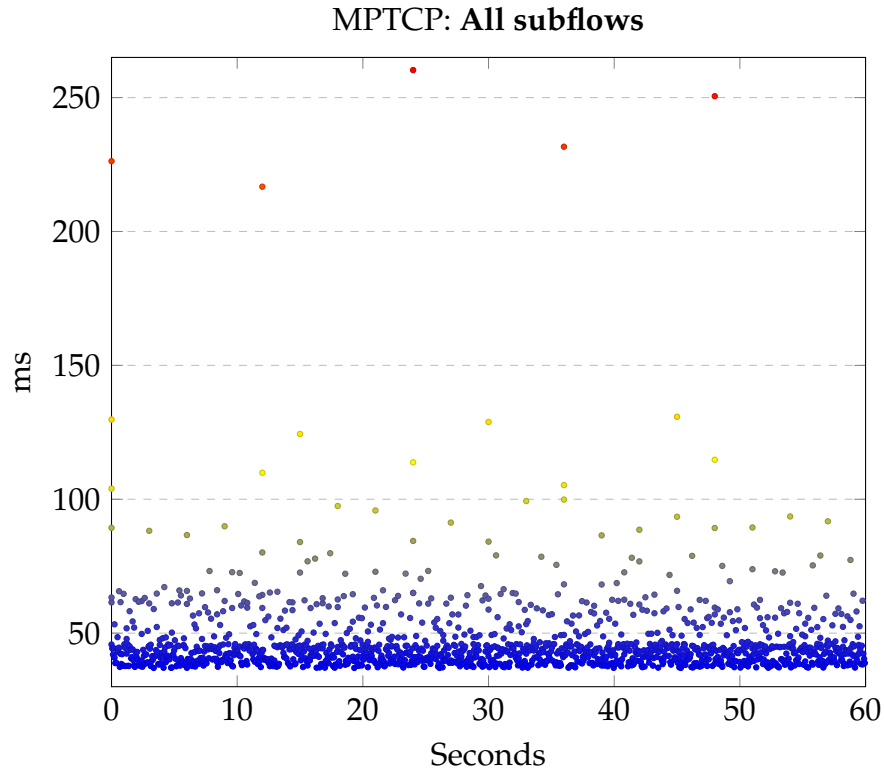


Figure 6.21: Latency: nordlys to lungegaardsvannet (MPTCP: All subflows)

In figure 6.21, we discover that the average RTT is slightly higher than in the previous figure. We also see that there are outliers with values above 200 ms, whereas the highest outliers in the previous figure which were around 70-80 ms. In addition, there is a higher number of packets with RTT values in the range 50-70 ms, compared with the TCP-flow. Overall, the MPTCP connection has a higher degree of varying latency. However, as we saw in test 2, the performance of the links of *nordlys* are quite similar, but still, some varying latency is observed. Anyhow, the results of this test show us that overall latency is improved when the available links are homogeneous, i.e. all links are of approximately same quality. Like we saw in test 9, the ADSL-links caused higher latency issues than in this test.

Test 11: Latency: rennesoey (UiS) to bymarka (NTNU) (independent flows)

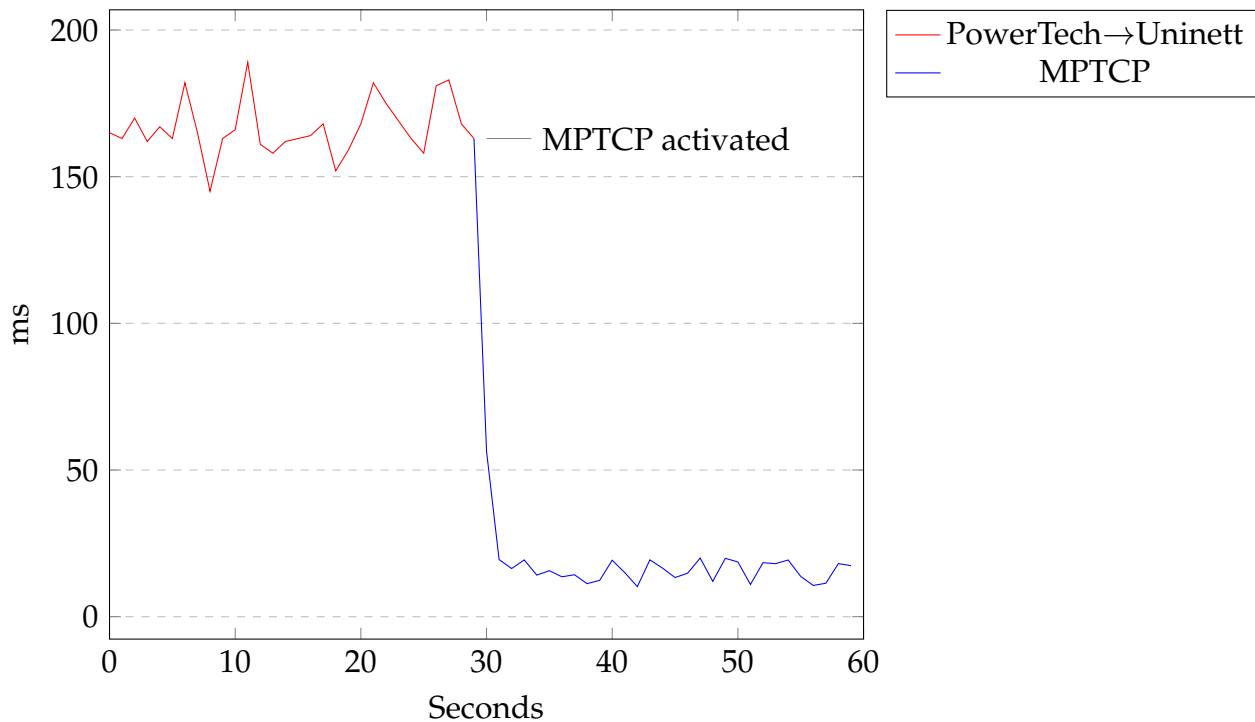


Figure 6.22: Latency: rennesoey to bymarka

In this test, the transmitting data flow starts as an ordinary TCP flow, transmitting from *PowerTech*, the ADSL-link at *rennesoey*, to *Uninett* at *bymarka*. For us to get average RTT values, we had to manually calculate them based on the *Wireshark* plot. This method will not give us exact values, but it gives us adequate estimates. Due to the limited upstream rate of the ADSL-link, figure 6.22 shows us that the latency is rather high and variable. However, MPTCP is enabled 30 seconds into the test, and as we observe, this decreases the average packet latency of the flow from over 160 ms to around 20 ms. When MPTCP is enabled, all links at *rennesoey* (*Uninett*, *Altibox*, *PowerTech*) is used, in addition to both links at *bymarka* (*Uninett*, *PowerTech*). The decrease in latency can naturally be explained by the fact that the fiber-links takes over most of the data transmission, because of the *LowRTT*-scheduler which schedules the data away from the slower *PowerTech*-link.

Test 12: Latency: rennesoey (UiS) to lungegaardsvannet (UiB)

In this test, data are sent from **rennesoey** (*Uninett, Altibox, PowerTech*), located at the University of Stavanger, to **lungegaardsvannet** (*Uninett, BKK*), located at the University of Bergen.

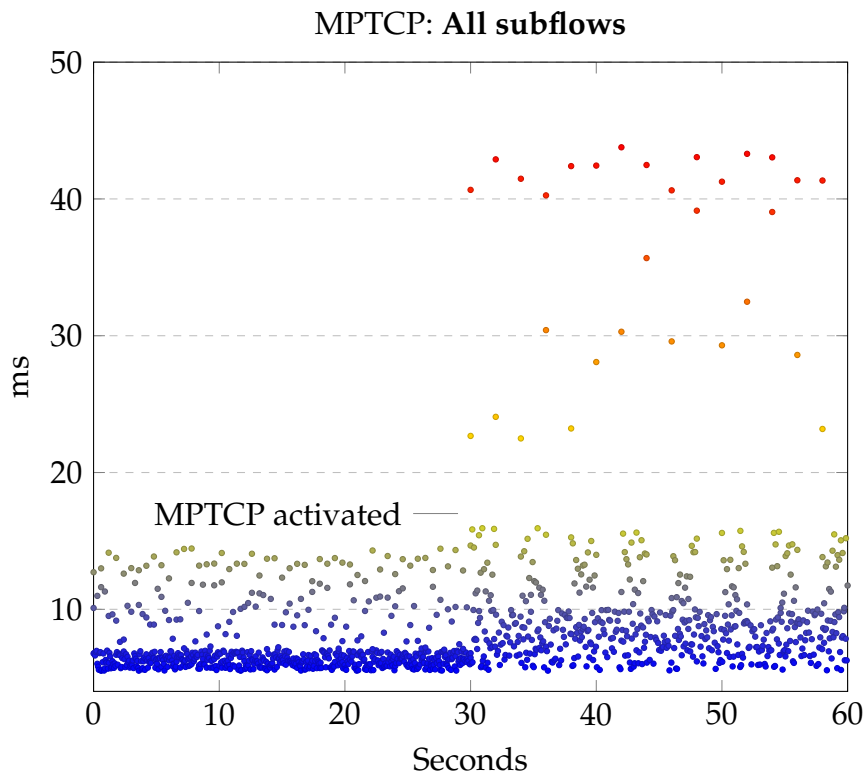


Figure 6.23: Latency: rennesoey to lungegaardsvannet (MPTCP: All subflows)

If a user is not using the best available link, he can suffer from unnecessary low throughput and high latency. In the previous test, we showed that MPTCP has the ability to lower the average latency by utilizing the additional low-latency links. To get a closer look on the latency difference between the best available link (*Uninett*→*Uninett*) and MPTCP, we again started a single TCP flow, and then activated MPTCP after 30 seconds. As in our other latency tests, we expect the MPTCP latency to be slightly higher than TCP. From figure 6.23, we can see that this expectation is accurate. In the TCP flow, the majority of the packets are in the 5.5-7 ms range, shown as a nearly solid line, with some outliers in the range 7-14 ms. MPTCP, on the other hand, has an average latency slightly above TCP,

in the 5.5-10 ms range. We also observed some small latency spikes in the 10-16 ms range. From these numbers, MPTCP doesn't seem too bad, but as you can see from the figure, the MPTCP flow unfortunately contains a noteworthy amount of outliers in the 20-45 ms range. We chose to not include the high-RTT packets caused by the *PowerTech*-link. Most of these packets are in the 100-300 ms range, but some of them are actually above 2000 ms. These packets are, as in the other tests, only experienced in the initial phase of the transmission.

6.4.5 Connection Handover

Together with improved throughput, resilience is probably the most interesting and useful aspect of MPTCP. This section will study the resilience of MPTCP, and the connection handover will be tested. Due to the fact that every link is virtualized behind one interface in the NORNET CORE nodes, we are not able to intentionally remove a link during a data transfer. By taking down an interface during a data transfer, we wish to test if MPTCP provides the resilience we expect - meaning that a data transfer doesn't get interrupted. In order to test this, we had to use a personal laptop. We used the operating system *Ubuntu 16.04 LTS*, with *MultiPath TCP v0.90* provided by the *Linux Kernel MultiPath TCP implementation* by UCL. The Linux kernel version in this implementation is *3.18.20-90-mptcp*. The laptop we used is equipped with an *802.11b/g/n* interface, and a USB-dongle provided additional 4G access to the *Telenor* cellular network.

Test 13: Handover with additional backup link

In this test, the Wi-Fi interface acts as the main link, with the 4G-link in *backup priority* mode. This means that it should take over if the main interface goes down or doesn't respond. To test this, it was started an HTTP file transfer from <http://multipath-tcp.org>, the home page of the MPTCP Linux Kernel implementation, which we know is MPTCP enabled. The download of a 121 MB file was started, but 10 seconds into the file transfer, the WiFi-interface (*wlan0*) was shut down with the following command:

```
1 ip link set dev wlan0 down
```

Afterwards, *pcap*-data from *tcpdump* was studied.

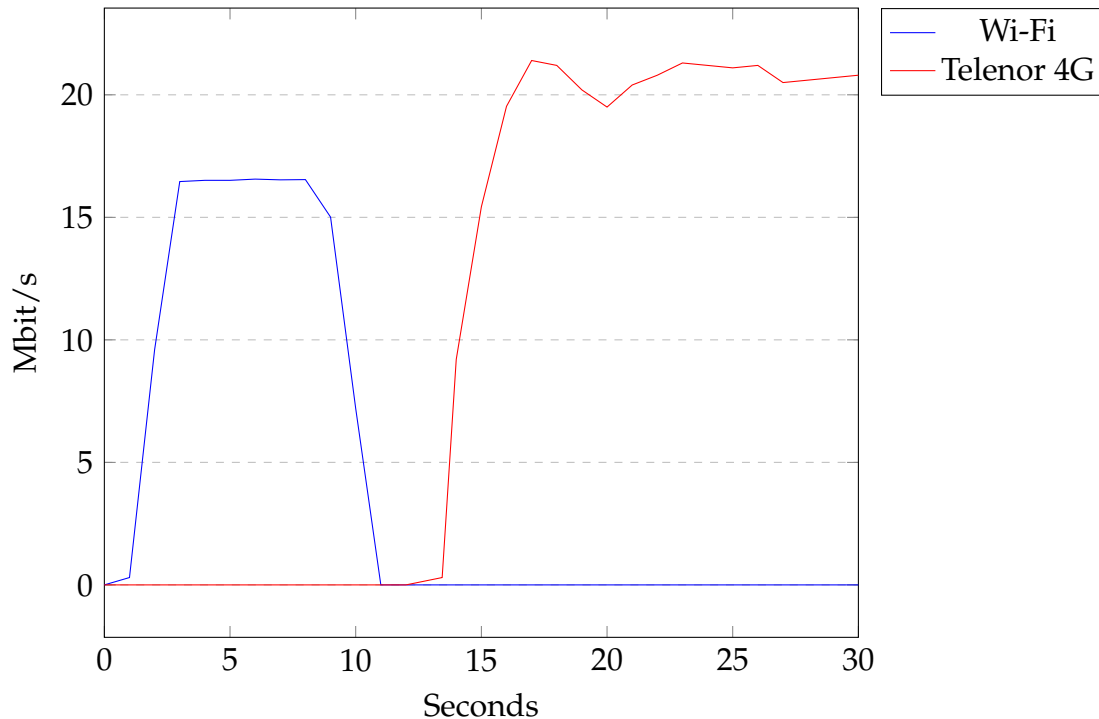


Figure 6.24: Connection Handover: One Backup Link

The outcome of this test was positive and expected, as the file transfer didn't get interrupted, but instead continued over the 4G-link. This confirms the resilience aspect of MPTCP. The exact handover time from the Wi-Fi interface was taken down, until data transfer began on the 4G-link, was **3.44 seconds**. However, the `MP_JOIN` signal was sent from the 4G-link to the web server only **1.73 seconds** after the Wi-Fi interface was taken down. In addition, the `REMOVE_ADDR` option was sent, telling the server that the initial address of the Wi-Fi interface should be removed from the connection.

6.4.6 Congestion Control Algorithms

In this section, we will examine and evaluate the different congestion controls algorithms that are available for MPTCP. In addition, we will also find out if there exist any difference in performance between the *coupled* and the *uncoupled* congestion controls, where the *uncoupled* ones mainly belong to TCP. The coupled congestion controls that are going

to be tested are the four described in section 5.3.2.1; *LIA*, *OLIA*, *Balia* and *Weighted Vegas* (*wVegas*). The uncoupled congestion controls that are also included for comparison, are the most known congestion controls for TCP, *Reno* and *Cubic*. In these experiments, data are sent from **floeibanen** (*Uninett*, *PowerTech*), located at the University of Bergen, to **ekeberg** (*Uninett*, *Broadnet*, *PowerTech*), located at the University of Oslo. The measurements are done using *NetPerfMeter*, with a transmission duration of 60 seconds. Each test with a given congestion control is repeated 10 times, in order to avoid outliers that can be caused by background traffic between the nodes. Finally, the average throughput is calculated and presented.

Test 14: Congestion Controls: floeibanen (UiB) to ekeberg (UiO)

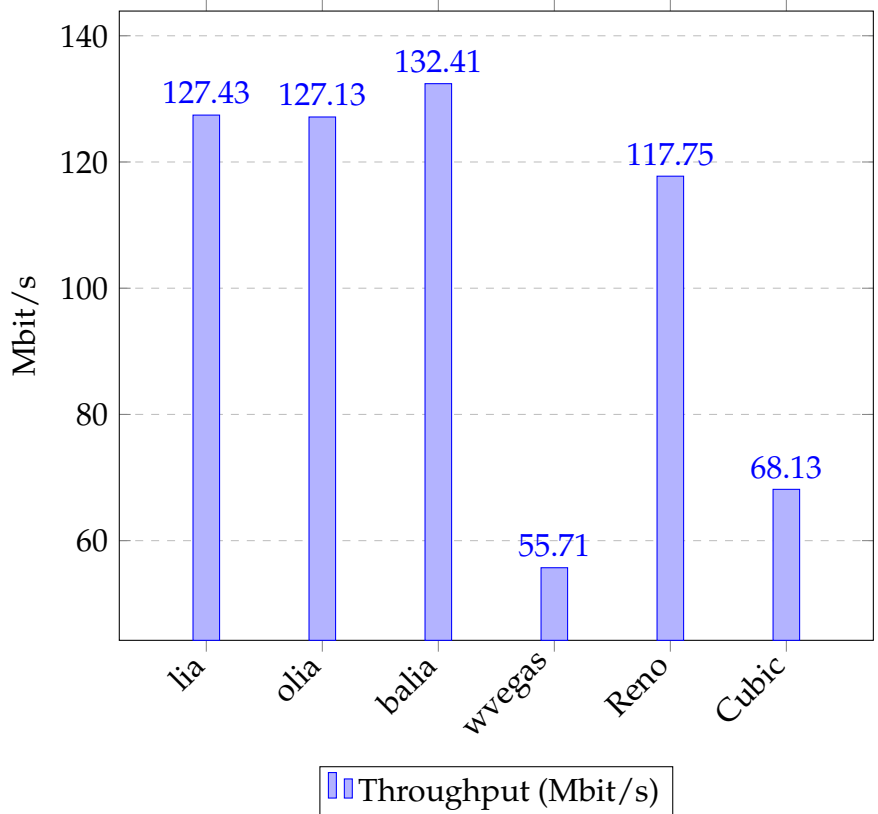


Figure 6.25: MPTCP Congestion Controls Performance

Figure 6.25 presents the results from the tests, where the average throughput are plotted for each of the congestion control algorithms. As we can observe, *wVegas* has the poorest

performance of all. However, the three other coupled congestion controls have pretty similar performance, without a clear stand out. But, as stated in section 4.1.2.5, only one of the three congestion control algorithm goals is to improve the throughput - the first goal. The second goal says that MPTCP should behave fairly, and the third goal is all about moving congestion away from congested paths or shared bottlenecks, in accordance with the *resource pooling principle*. Considering the current tunnel-configuration of NORNET CORE, it can be difficult to discover shared bottlenecks on the different paths taken.

6.4.7 Schedulers

As previously mentioned, there are only two schedulers available; the default one (*LowRTT*), and *roundrobin*. If you really want to take advantage of multipath transport, the *roundrobin* scheduler is not of great benefit, as it doesn't transmit concurrently over all available links, nor does it take into account that the available links may have very different qualities, considering throughput and latency. Regardless of the usefulness of *roundrobin*, we have tested it and compared it to the default scheduler, *LowRTT* (which is used in all other tests).

Test 15: Schedulers: ekeberg (UiO) to floeibanen (UiB)

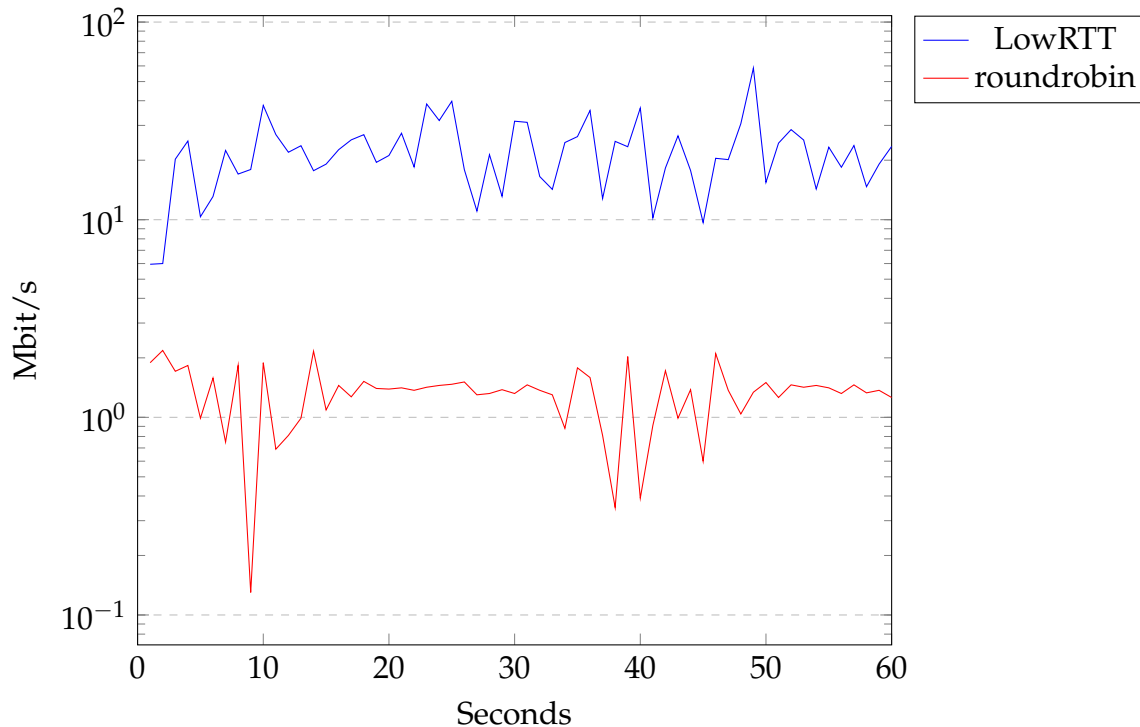


Figure 6.26: Scheduler: LowRTT vs. roundrobin

As expected, figure 6.26 shows that the use of the *roundrobin* scheduler results in a significant decrease in throughput compared to *LowRTT*. These results could frankly be predicted, given the design and behaviour of *roundrobin*, so no further explanation is really necessary. Due to the ADSL-link of *ekeberg*, the throughput is fluctuating for both schedulers, as experienced in earlier tests.

6.5 Summary and Evaluation

We started our experimentation with MPTCP by making sure that it actually utilized multipath transport by initiating the correct number of subflows (*fullmesh* between the available ISPs). This was confirmed in test 1, where we also saw that (some of) the nodes have links of different qualities, regarding throughput and latency. In test 2, 3 and 4,

we confirmed our expectations that MPTCP can provide a significant gain in throughput, compared to a single-path TCP-flow. We observed that between domestic sites, the MPTCP throughput was close to the aggregate throughput of independently run TCP-flows (*full mesh*). Although MPTCP provided a superior mean throughput, our tests show that MPTCP throughput can be highly fluctuating. This is especially the case for sites with heterogeneous link characteristics, such as the combination of fiber- and DSL-links. Test 4 showed that even though MPTCP cannot fully utilize each subflow compared to TCP, it still perform according to design goal one, *improve throughput*.

Test 5, 6, 7 and 8, running with two and four flows, we saw that MPTCP acted fairly to concurrently transmitting TCP-flows. However, as we don't have an overview or information about the NORNET CORE ISP interconnections, or the paths taken by our packets, it is difficult to identify possible shared bottlenecks. Based on our results, we suspect that TCP-flows (and then also MPTCP's subflows) are rate limited, to control the volume of traffic being sent on the network.

In test 9, 10, 11 and 12, we looked at latency in MPTCP connections, by studying the RTT of each packet. Our findings show that the average latency of an MPTCP connection is slightly higher compared to a TCP-flow. However, the MPTCP-flow had a higher number of outliers, and the RTT-value of the outliers were generally considerably higher. These observations are possibly caused by a combination of the overhead introduced by MPTCP, and the *head-of-line blocking problem*. In addition, MPTCP had some extremely high outliers, caused by the low quality ADSL-links. Here, we could also confirm that the *LowRTT*-scheduler in fact schedules data away from the slower links. The number of high latency outliers tells us that MPTCP running with variable link-quality can be unsuitable for latency-sensitive applications. In test 10, we studied latency between two sites with homogeneous link quality, and we observed that the number of extreme outliers decreased significantly.

Test 13 demonstrated the resilience aspect of MPTCP, and confirmed seamless connection handover to a backup link in cases where the main link goes down. The file transfer in the test didn't get interrupted, but it continued on the backup link.

Test 14 compared the available congestion control algorithms available to MPTCP, and compared them to the TCP congestion control algorithms *Reno* and *Cubic*.

Our observations show that *lia*, *olia* and *balia*, have very similar performance regarding throughput, unlike *wvegas* which showed a significant decrease in throughput. Finally, in test 15, we compared the default scheduler, *LowRTT*, to *roundrobin*, mainly just to show the differences.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The work carried out in this thesis have explored different aspects of the new and emerging *MultiPath TCP* (MPTCP) protocol. To obtain the best possible understanding of the underlying technologies of multipath protocols, it was natural to start this thesis by acquiring fundamental knowledge on the multihoming subject.

The first phase of this thesis presented recent research on multihoming, with various approaches for implementation at different network layers. Following, a thorough introduction to MPTCP and its operation was given - including the goals set by the *IETF* for the further development of this comparatively new multipath protocol. The second phase of this thesis embraced the NORNET testbed, gave implementation details, and presented our aims for testing the protocol in order to illuminate the most interesting fields.

We have successfully validated that the concepts of *MPTCP* works in a real-world setup, using only existing infrastructure between the test-nodes. We have seen that the MPTCP implementation actually is taking advantage of multipath transport, by concurrently transmitting data over all of the available paths, taking the qualities of the

links into consideration. We have also seen that the subflows appear as ordinary TCP flows to existing hardware and infrastructure. For a rational deployment of MPTCP in current network structure to be realistic, this part is particularly crucial. We have confirmed that the only modifications needed to be done are at the end-points, in order for MPTCP to function as supposed. After a wide variety of tests have been performed in the testbed, our experimental evaluations have shown that MPTCP can in fact provide users with a significant gain in throughput. We have confirmed that the path managers function as anticipated, by initiating the correct number of subflows, and we have presented the differences between the available congestion control algorithms. We have also demonstrated that MPTCP acts fairly to other TCP flows, in the best way possible.

Throughout these experiments, we have learned that the combination of links with very unlike qualities, e.g. business-grade fiber-links and weaker DSL-links, can result in fluctuating throughput and latency. This behaviour doesn't come unexpected - as we learned in the *application compatibility goals* in section 4.1.2.1. Here we saw that MPTCP may not be able to provide the same level of consistency of throughput and latency as a single TCP connection. However, more importantly, this does not affect the resilience aspect of multipath transport - if you loose one link, you will still be connected through another link.

We believe that our work has proved the main concepts of MPTCP, by successfully having performed tests in an operational implementation of MPTCP in the NORNET CORE testbed.

7.2 Future Work

We do also believe there is room for future work. In order to do thoroughly testing of fairness regarding shared bottlenecks, we need an overview and a detailed analysis of the paths and interconnections of the different ISPs. Due to the tunneling configuration of NORNET CORE, it is currently not possible to study the paths taken by the different data flows. It would also be interesting to test the impact of variable *maximum segment size* (MSS) and window size.

We have seen that the current MPTCP scheduler tries to avoid using links with poor performance, which naturally is important if the end-user is running *real-time applications* (RTA), such as video conferencing, *voice over IP* (VoIP) and online gaming. However, our experiments show that the first phases of data transfers often contain high-RTT packets from the ADSL-links - this might be avoided by implementing a smarter scheduler that has knowledge of the qualities of the available links prior to a data transmission. It can e.g. use the *bandwidth* \times *delay* product and latency to find the optimal *window size* and *MSS* for each subflow.

Another interesting approach to improve MPTCP further, could be to use *network coding*. As we experienced throughout our testing, MPTCP was negatively affected by high packet loss due to poor channel conditions on the ADSL-links. Network coding could be able to overcome the majority of packet loss and maintain a stable throughput, as the *head-of-line blocking problem* will not occur as frequently [15]. In addition, network coding can make for easier packet scheduling - if a packet is lost, other packets sent on different subflows can be used to recover the lost packet.

MultiPath TCP is surely an exciting technology, and it will be of great interest to follow the further development. We are confident that this protocol has the potential to provide improved throughput and superior resilience compared to the current transport layer protocols.

Bibliography

- [1] J. Ahrenholz and T. Henderson. Shim6 manual. <http://openhip.sourceforge.net/docs/shim6.pdf>, 2007.
- [2] K. E. Aldatay. Mobile IP Handover Delay Reduction Using Seamless Handover Architecture. Master's thesis, Blekinge Institution of Technology, August 2009.
- [3] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681, Internet Engineering Task Force, September 2009.
- [4] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581, Internet Engineering Task Force, April 1999.
- [5] J. Arkko and I. van Beijnum. Failure Detection and Locator Pair Exploration Protocol for IPv6 Multihoming. RFC 5534, Internet Engineering Task Force, June 2009.
- [6] S. Barré. LinShim6. <http://inl.info.ucl.ac.be/software/linshim6>, August 2007.
- [7] S. Barré. *Implementation and Assessment of Modern Host-based Multipath Solutions*. PhD Thesis, Louvain School of Engineering, October 2011.
- [8] M. Belshe, R. Peon, and E. M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, Internet Engineering Task Force, May 2015.
- [9] E. Blanton and M. Allman. On Making TCP More Robust to Packet Reordering. Article, Ohio University, BBN Technologies/NASA GRC, January 2002.
- [10] L. Boccassi, M. M. Fayed, and M. K. Marina. Binder: A System to Aggregate Multiple Internet Gateways in Community Networks. Article, LCDNet '13, August 2013.

- [11] Ed. C. Perkins. IP Mobility Support for IPv4. RFC 3344, Internet Engineering Task Force, August 2002.
- [12] Ed. C. Perkins and D. Johnson. Mobility Support in IPv6. RFC 6275, Internet Engineering Task Force, July 2011.
- [13] Cisco Systems Inc. *IEEE 802.3ad Link Bundling*, February 2007.
- [14] Cisco Systems Inc. *Link Aggregation Control Protocol (LACP) (802.3ad)*, March 2007.
- [15] J. Cloud, F. d. Pin Calmon, W. Zeng, G. Pau, L. M. Zeger, and M. Médard. Multi-Path TCP with Network Coding for Mobile Devices in Heterogeneous Networks. Article, Internet Engineering Task Force, 2013.
- [16] B. Cohen. The BitTorrent Protocol Specification. http://www.bittorrent.org/beps/bep_0003.html, October 2011.
- [17] S. E. Deering and R. M. Hinden. Internet Protocol, Version 6 (IPv6). RFC 2460, Internet Engineering Task Force, December 1998.
- [18] T. Dreibholz. The NorNet Core Handbook. Technical report, Simula Research Laboratory, August 2015.
- [19] F5 Networks Inc. *BIG-IP Link Controller*. Datasheet specification.
- [20] A. Ford, C. Raiciu, M. Handley, S. Barré, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. RFC 6182, Internet Engineering Task Force, March 2011.
- [21] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, Internet Engineering Task Force, January 2013.
- [22] N. Freed. Behavior of and Requirements for Internet Firewalls. RFC 2979, Internet Engineering Task Force, October 2000.
- [23] E. G. Gran, T. Dreibholz, and A. Kvalbein. NorNet Core - A Multi-homed Research Testbed. Article, Simula Research Laboratory, January 2014.
- [24] M. Handley, V. Jacobsen, and C. Perkins. SDP: Session Description Protocol. RFC 4566, Internet Engineering Task Force, July 2006.

- [25] Internet Assigned Numbers Authority (IANA). Transmission Control Protocol (TCP) Parameters. <http://www.iana.org/assignments/tcp-parameters/tcp-parameters.xml>, 2015. [Online: accessed 25-Feb-2016].
- [26] University of Southern California Information Sciences Institute. Internet Protocol. RFC 760, Information Sciences Institute, University of Southern California, January 1980.
- [27] University of Southern California Information Sciences Institute. Transmission Control Protocol. RFC 793, Information Sciences Institute, University of Southern California, September 1981.
- [28] J. R. Iyengar, Paul D. Amer, and R. Stewart. Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths. Technical report, Internet Engineering Task Force, October 2006.
- [29] R. Khalili, N. Gast, and J-Y Le. Boudec. Opportunistic Linked-Increases Congestion Control Algorithm for MPTCP. Article, Internet Engineering Task Force, July 2014.
- [30] J. Klensin. Simple Mail Transfer Protocol. RFC 2821, Internet Engineering Task Force, April 2001.
- [31] M. Komu, M. Sethi, and N. Beijar. A survey of identifier-locator split addressing architectures. *Elsevier*, May 2015.
- [32] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach*. Pearson Education, fourth edition, 2008.
- [33] Link Aggregation Task Force. IEEE 802.3ad. <http://www.ieee802.org/3/axay/index.html>, January 2009.
- [34] R. Moskowitz and P. Nikander. Host Identity Protocol (HIP) Architecture. RFC 4433, Internet Engineering Task Force, May 2006.
- [35] R. Moskowitz and P. Nikander. End-Host Mobility and Multihoming with the Host Identity Protocol. RFC 5206, Internet Engineering Task Force, April 2008.
- [36] E. Nordmark and M. Bagnulo. Shim6: Level 3 Multihoming Shim Protocol for IPv6. RFC 5533, Internet Engineering Task Force, June 2009.

- [37] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure. Experimental Evaluation of Multipath TCP Schedulers. In *ACM SIGCOMM Capacity Sharing Workshop (CSWS)*. ACM, 2014.
- [38] J. Postel. User Datagram Protocol. RFC 768, Internet Engineering Task Force, August 1980.
- [39] J. Postel and J. Reynolds. File Transfer Protocol (FTP). RFC 959, Internet Engineering Task Force, October 1985.
- [40] C. Raiciu, M. Handley, and D. Wischik. Coupled Congestion Control for Multipath Transport Protocols. RFC 6356, Internet Engineering Task Force, October 2011.
- [41] C. Raiciu, M. Handley, and D. Wischik. Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6181, Internet Engineering Task Force, March 2011.
- [42] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, Internet Engineering Task Force, June 2002.
- [43] M. Scharf and A. Ford. Multipath TCP (MPTCP) Application Interface Considerations. RFC 6897, Internet Engineering Task Force, March 2013.
- [44] M. Scharf and S. Kiesel. Quantifying Head-of-line Blocking in TCP and SCTP. Internet Draft, TCPM, July 2013.
- [45] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobsen. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, Internet Engineering Task Force, July 2003.
- [46] V. Singh, J. Ott, and S. Ahsan. MPRTCP: Multipath Consideration for Real-time Media. Article, Aalto University, March 2013.
- [47] V. Singh, J. Ott, T. Karkkainen, S. Ahsan, and L. Eggert. Multipath RTP (MPRTCP). Internet-draft, Aalto University, July 2014.
- [48] V. Singh, J. Ott, T. Karkkainen, R. Globisch, and T. Schierl. Multipath RTP (MPRTCP) attribute in Session Description Protocol. Internet-draft, MMUSIC Working Group, July 2012.

- [49] T. Socolofsky and C. Kale. A TCP/IP Tutorial. RFC 1180, Internet Engineering Task Force, January 1991.
- [50] R. Stewart. Stream Control Transmission Protocol. RFC 4960, Internet Engineering Task Force, September 2007.
- [51] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, Internet Engineering Task Force, October 2000.
- [52] S. Tatham. PuTTY. <http://www.putty.org/>.
- [53] R. H. Tse. TCP Fairness in Multipath Transport Protocols. Bachelor thesis, Brown University, May 2006.
- [54] Université catholique de Louvain. Configure MPTCP. <http://multipath-tcp.org/pmwiki.php/Users/ConfigureMPTCP>, September 2015.
- [55] Université catholique de Louvain. MPTCP v0.90 Release. <http://multipath-tcp.org/pmwiki.php?n=Main.Release90>, September 2015.
- [56] A. Walid, Q. Peng, J. Hwang, and S. Low. Balanced Linked Adaptation Congestion Control Algorithm for MPTCP. Internet Draft, Internet Engineering Task Force, January 2016.
- [57] D. Wischik, M. Handley, and M. B. Braun. The Resource Pooling Principle. *ACM SIGCOMM Computer Communication Review*, 2008.
- [58] M. Xu, Y. Cao, and E. Dong. Delay-based Congestion Control for MPTCP. Internet Draft, Internet Engineering Task Force, January 2016.
- [59] C. Åhlund and A. Zaslavsky. Multihoming with Mobile IP, 2003.

Appendices

Appendix A

NetPerfMeter Vector Parser

To easily extract performance data from the vector files provided by *NetPerfMeter*, we created a simple Java program. The program simply reads multiple vector files from a directory, calculates the bit-rate, and then combines them into a text file. As our project files were synchronized using *OneDrive* (i.e. same absolute path), we could both easily access the test files by entering our name into the program. The output files allow us to more effectively create tables and figures from the extracted data.

```
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.*;

public class NPM_Vector_Parser {
    public static void main(String[] args){
        // Prompting the user
        String name = "";
        Scanner in = new Scanner(System.in);
        System.out.println("Please type your name: (krist = Kristian) (daniel =
            Daniel)");
        name = in.next();

        System.out.println("Please choose the parent directory of the .vec files
```

```

        (integer):");

// Listing directories that the user can choose from
File[] directories = listChildDirectories(new File("C:\\Users\\" + name
    + "\\OneDrive\\INF399\\Testdata"));

File testDirectory = directories[in.nextInt()];

// Reading from .vec files in chosen directory
ArrayList<ArrayList<String>> data = readFromFile(name, testDirectory);

// Writing to new .txt file
writeToFile(data, testDirectory);
in.close();
}

// Listing all child directories of a directory
public static File[] listChildDirectories(File file) {
    File[] dirs = file.listFiles(new FileFilter() {

        @Override
        public boolean accept(File path) {
            return path.isDirectory();
        }
    });
    for (int i = 0; i < dirs.length; i++) {
        System.out.println(i + ": " + dirs[i].getName());
    }
    return dirs;
}

// Writing data to a .txt file
public static void writeToFile(ArrayList<ArrayList<String>> data, File
    testDirectory) {
    PrintWriter writer = null;
    try {
        writer = new PrintWriter(testDirectory + "\\\" +
            testDirectory.getName() + ".txt");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

```

int count = 0;
for (int i = 0; i < 61; i++) {
    writer.print(count + "\t");
    for (int j = 0; j < data.size(); j++) {
        writer.print(data.get(j).get(i) + "\t");
    }
    count++;
    writer.println();
}
writer.close();
System.out.print("The following file was created:\n" + testDirectory +
    "\\\" + testDirectory.getName() + ".txt");
}

// Calculating the throughput in Mbit/s
public static String calculateMbits(int bytes_last, int bytes_now) {
    double speed = bytes_now - bytes_last;
    speed = (speed * 8) / 1000000;
    return String.format("%.2f", speed).replace(",", ".");
}

// Reading from a collection of .vec files
public static ArrayList<ArrayList<String>> readFromFile(String name, File
    testDirectory) {
    ArrayList<ArrayList<String>> listOfLists = new
        ArrayList<ArrayList<String>>();
    try {
        for (File file : testDirectory.listFiles(new FilenameFilter() {

            @Override
            public boolean accept(File dir, String name) {
                return name.endsWith("passive.vec");
            }
        })) {
            BufferedReader in = new BufferedReader(new FileReader(file));

            ArrayList<String> oldList = (ArrayList<String>)
                Files.readAllLines(Paths.get(file.getPath()));
            ArrayList<String> newList = new ArrayList<>();

```

```
int bytes_last = 0;
for (int j = 1; j < oldList.size(); j++) {
    if(oldList.get(j).contains("\Received\")) {
        String[] tokens = oldList.get(j).split("\\s+");
        String flow_ID = tokens[4];
        String absBytes = tokens[8];
        if(Integer.parseInt(flow_ID) == 0) {
            newList.add(calculateMbits(bytes_last,
                Integer.parseInt(absBytes)));
            bytes_last = Integer.parseInt(absBytes);
        }
    }
}
listOfLists.add(newList);
in.close();
}
} catch (Exception e) {
    e.printStackTrace();
}
return listOfLists;
}
}
```
