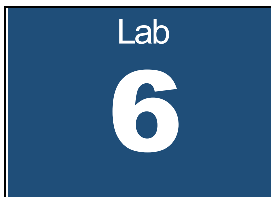


Transport Layer Protocols

What you will learn in this lab:

- The differences between data transfers with UDP and with TCP
- Impact of IP Fragmentation on TCP and UDP
- How to analyze measurements of a TCP connection
- The difference between interactive and bulk data transfers in TCP
- How TCP performs retransmissions
- How TCP congestion control works

Updated: March 2021

Table of Content

STUDY MATERIAL FOR LAB 6 .... **3**

PRELAB 6 4

LAB 6 – TRANSPORT LAYER PROTOCOLS: UDP & TCP **5**

PART 1. TCP TRAFFIC AT A WEB SERVER 6
Exercise 1-a. Network setup *7*
Exercise 1-b. Web access *8*
PART 2. WORKING WITH *IPERF* 10
Exercise 2-a. Transmitting data with UDP *11*
Exercise 2-b. Transmitting data with TCP *12*
PART 3. IP FRAGMENTATION OF UDP AND TCP TRAFFIC 14
Exercise 3-a. Fragmentation of UDP traffic *14*
Exercise 3-b. Avoiding fragmentation in TCP *16*
Exercise 3-c. Fragmentation of UDP traffic in IPv6 .... *18*
PART 4. TCP CONNECTION MANAGEMENT 20
Exercise 4-a. Opening and Closing a TCP Connection *20*
Exercise 4.b- Requesting a connection to non-existing host *22*
Exercise 4.c- Requesting a connection to a non-existing port *22*
Exercise 4-d. Configuration parameters of TCP *23*
PART 5. TCP DATA EXCHANGE – INTERACTIVE APPLICATIONS 25
Exercise 5-a. Interactive Applications (low latency, high data rate) *26*
Exercise 5-c. Interactive Applications (high latency) *28*
PART 6. TCP DATA EXCHANGE – BULK DATA TRANSFER 30
Exercise 6-a. TCP Data Transfer – Fast link *31*
Exercise 6-b. Generating graphs of TCP data transfer *32*
Exercise 6-c. TCP Data Transfer – Small receiver windows *35*
Exercise 6-d. TCP Data Transfer – Slow link *37*
PART 7. RETRANSMISSIONS IN TCP 39
Exercise 7-a. TCP Retransmissions *40*
Exercise 7-b. TCP performance with sporadic packet losses *43*
Exercise 7-c. TCP retransmissions at an overloaded link *45*
PART 8. TCP CONGESTION CONTROL 47
Exercise 8-a. Network setup and congestion parameters *48*
Exercise 8-b. Observing TCP congestion control *50*
Exercise 8-c. Fairness of congestion control *52*
Exercise 8-d. TCP competing with UDP traffic *54*

Study Material for Lab 6

1. ***iPerf***: Read about the *iPerf* tool for diagnosing network performance at
   iperf - The Easy Tutorial

2. **Path MTU discovery**: Refer to the Wikipedia entry for Path MTU Discovery.
   https://en.wikipedia.org/wiki/Path_MTU_Discovery

3. **TCP and UDP**: Read the Wiki pages on TCP and UDP available at
   https://en.wikipedia.org/wiki/Transmission_Control_Protocol

   https://en.wikipedia.org/wiki/User_Datagram_Protocol

   More detailed descriptions can be found in Chapters 11-12 of

   http://intronetworks.cs.luc.edu/1/html/

4. **Network emulation (netem).** Read about the network emulation (*netem*) functionality in Linux at

   https://wiki.linuxfoundation.org/networking/netem

5. **TCP Congestion Control**: Read the Wiki page on TCP Congestion control at
   https://en.wikipedia.org/wiki/TCP_congestion_control

   For a detailed description of TCP congestion control, refer to Chapters 13-16 of

   http://intronetworks.cs.luc.edu/1/html/

   The standard TCP congestion control algorithm is specified in RFC 5681

   https://tools.ietf.org/html/rfc5681

Prelab 6

Answer the questions in the space provided below. Use extra sheets if needed and attach them to this document.

1. Explain the role of port numbers in TCP and UDP.
2. Provide the syntax of the *iPerf* command for both the sender and receiver, which executes the following scenario:
   A TCP server has IP address 10.0.2.6 and a TCP client has IP address 10.0.2.7. The TCP server is waiting on port number 2222 for a connection request. The client connects to the server and transmits 2000 bytes to the server, which are sent as 4 write operations of 500 bytes each.
3. Answer the following questions on Path MTU Discovery:
   a. How does TCP decide the maximum size of a TCP segment?
   b. How does UDP decide the maximum size of a UDP datagram?
   c. What is the ICMP error generated by a router when it needs to fragment a datagram with the DF bit set? Is the MTU of the interface that caused the fragmentation also returned?
   d. Explain why a TCP connection over an Ethernet segment never runs into problems with fragmentation.
4. Assume a TCP sender receives an acknowledgement (ACK), that is, a TCP segment with the ACK flag set, where the acknowledgement number is set to 34567 and the window size is set to 2048. Which sequence numbers can the sender transmit?
5. What is the purpose of selective acknowledgements in TCP? Describe a scenario where a TCP receiver sends a selective acknowledgement.
6. Briefly describe the following heuristics used in TCP and explain why they are used:
   a. Nagle's algorithm
   b. Karn's Algorithm
7. Answer the following questions about TCP acknowledgements:
   a. What is a *delayed acknowledgement*?
   b. What is a *duplicate acknowledgement*?
8. Describe how the retransmission timeout (RTO) value is determined in TCP.
9. Answer the following questions on TCP congestion control:
   a. Describe the concepts of *slow start* and *congestion avoidance* in TCP.
   b. Explain the concept of *fast retransmit* and *fast recovery* in TCP.

Lab 6 – Transport Layer Protocols: UDP & TCP

This lab explores the operation of the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP), the two main transport protocols of the Internet protocol architecture.

UDP is a simple protocol for exchanging messages from a sending application to a receiving application. UDP adds a small header to the message, and the resulting data unit is called a *UDP datagram*. When a UDP datagram is transmitted, the datagram is encapsulated in an IP header and delivered to its destination. There is one UDP datagram for each application message.

The operation of TCP is more complex. First, TCP is a connection-oriented protocol, where a TCP client establishes a logical connection to a TCP server, before data transmission can take place. Once a connection is established, data transfer can proceed in both directions. The data unit of TCP, called a *TCP segment*, consists of a TCP header and payload which contains application data. A sending application submits data to TCP as a single stream of bytes without indicating message boundaries in the byte stream. The TCP sender decides how many bytes are put into a segment.

TCP ensures reliable delivery of data, and uses checksums, sequence numbers, acknowledgements, and timers to detect damaged or lost segments. The TCP receiver acknowledges the receipt of data by sending an acknowledgement segment (ACK). Multiple TCP segments can be acknowledged in a single ACK. When a TCP sender does not receive an ACK, the data is assumed lost, and is retransmitted.

TCP has two mechanisms that control the amount of data that a TCP sender can transmit. First, TCP receiver informs the TCP sender how much data the TCP sender can transmit. This is called *flow control*. Second, when the network is overloaded and TCP segments are lost, the TCP sender reduces the rate at which it transmits traffic. This is called *congestion control*.

This lab covers the main features of UDP and TCP. Part 1 explores the exchange of TCP segments during a Web access. Part 2 inspects the performance of data transmissions in TCP and UDP. Part 3 explores how TCP and UDP deal with IP fragmentation. The remaining parts address important components of TCP. Part 4 explores connection management, Parts 5 and 6 look at flow control and acknowledgements, Part 7 explores retransmissions, and Part 8 is devoted to congestion control.

Part 1. TCP traffic at a web server

As a first exercise, you set up a web server and a web client and observe the TCP traffic generated by a web access. The network topology is as shown in Figure 6.1, where *PC1* runs the web client, *PC2* runs a web server, and *PC3* is set up as an IP router. The address configuration is shown in Tables 6.1 and 6.2.

The network topology in Figure 6.1 is used in Parts 1-6.

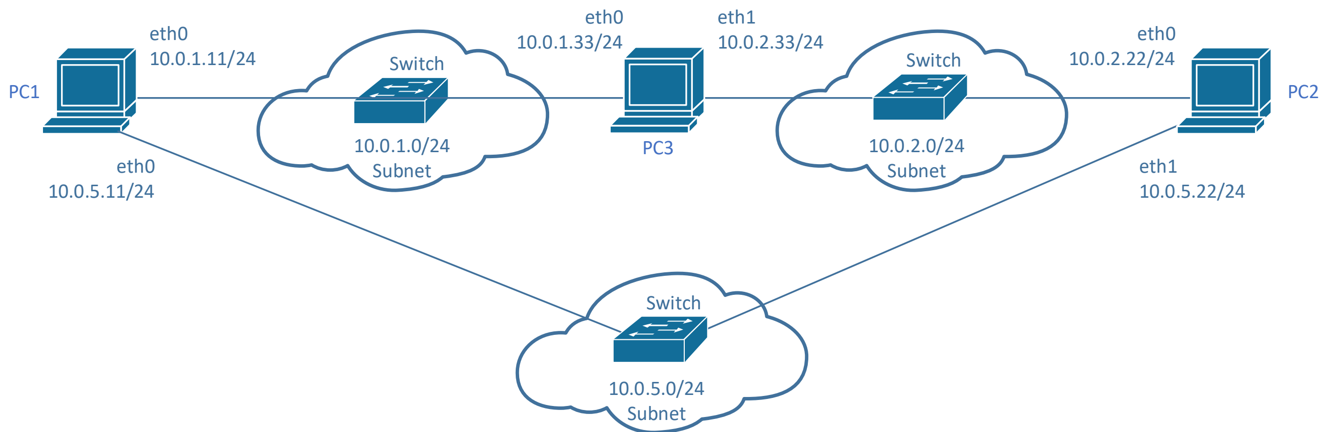


Figure 6.1. Network topology.

Table 6.1. IPv4 Addresses of the Linux PCs.

Linux PC	Ethernet interface *eth0*	Ethernet interface *eth1*	Default gateway
PC1	10.0.1.11/24	10.0.5.11/24	10.0.1.33
PC2	10.0.2.22/24	10.0.5.22/24	10.0.2.33
PC3	10.0.1.33/24	10.0.2.33/24	–

Table 6.2. IPv6 Addresses of the Linux PCs.

Linux PC	Ethernet Interface *eth0*	Ethernet Interface *eth1*	Default gateway
PC1	fd01:0:0:1::11/64	fd01:0:0:5::11/64	fd01:0:0:1::33

PC2	fd01:0:0:2::22/64	fd01:0:0:5::22/64	fd01:0:0:2::33
PC3	fd01:0:0:1::33/64	fd01:0:0:2::33/64	–

Exercise 1-a. Network setup

Connect the Ethernet interfaces of the Linux PCs as shown in Figure 6.1. Configure the IP addresses of the interfaces as given in Tables 6.1 and 6.2.

Step 1: Create a network topology for the Linux PCs as shown in Figure 6.1. Configure the IPv4 addresses of the interfaces as given in Table 6.1, and the IPv6 addresses as shown in Table 6.2. For reference, the commands for the *eth0* interface of *PC1* are

```
PC1$ sudo ip addr add 10.0.1.11/24 dev eth0
PC1$ sudo ip addr add fd01:0:0:1::11/64 dev eth0
```

Step 2: All PCs must have IPv6 enabled. On *PC1*, IPv6 is enabled with the command

```
PC1$ sudo sysctl -w net.ipv6.conf.all.disable_ipv6=0
PC1$ sudo sysctl -w net.ipv6.conf.default.disable_ipv6=0
```

Step 3: *PC3* is configures as an IPv4 and IPv6 router, and *PC1* and *PC2* should be set up as hosts. To query the status of IPv4 and IPv6 forwarding on *PC3* issue the commands

```
PC3$ sysctl net.ipv4.ip_forward
PC3$ sysctl  net.ipv6.conf.all.forwarding
```

IPv4 and IPv6 forwarding are enabled when the displayed value is "1" and disabled when the value is "0".
If necessary, enable IPv4 and IPv6 forwarding on *PC3* with the commands

```
PC3$ sudo sysctl -w net.ipv4.ip_forward=1
PC3$ sudo sysctl -w net.ipv6.conf.all.forwarding=1
```

You may also want to make sure that IPv4 and IPv6 forwarding are disabled on *PC1* and *PC2* by setting the above system parameters to "0".

Step 4: Add default routes to the routing tables of *PC1* and *PC2*, so that *PC3* is the default gateway. For *PC1*, the commands are

```
PC1$ sudo ip route add default via 10.0.1.33
PC1$ sudo ip route add default via fd01:0:0:1::33
```

Step 5: Verify that the setup is correct by issuing a *ping* command from *PC1* to *PC2* over both paths:

```
PC1$ ping –c2 10.0.2.22
PC1$ ping –c2 10.0.5.22
PC1$ ping6 –c2 fd01:0:0:2::22
PC1$ ping6 –c2 fd01:0:0:5::22
```

Exercise 1-b. Web access

Next you set up a web (HTTP) server on *PC2* and run a web (HTTP) client on *PC1*. The web server is a simple Python HTTP server, which serves web pages from the directory where the web server is started. For the web client, you use the command `wget` instead of a web browser with a graphical user interface.

Step 1: On *PC2*, go to the home directory of the labuser account. Use a text editor to write a short HTML file with name *"index.html".* Here is a sample of a simple HTML page:

```
<HTML>
<HEAD>
<TITLE>Your Title Here</TITLE>
</HEAD>
<BODY>
<H1> Large Header</H1>
<H2> Medium Header</H2>
<P> This is a paragraph. </P>
<HR>
</BODY>
</HTML>
```

Step 2: On *PC2*, start an HTTP server with the command

```
PC2$ sudo python3 -m http.server 80
```

This starts a web server that listens on port 80 of all its network interfaces for a web request by a web client.

Step 3: Start a *Wireshark* session to capture traffic on the *eth0* interface of *PC1*. Set a display filter to capture only TCP traffic ("tcp").

Step 4: On *PC1*, issue a web access for the HTML file on *PC2* with the command

```
PC1$ wget http://10.0.2.22/index.html
```

Step 5: Observe the TCP traffic captured by *Wireshark* after the web access:

- Explore the packets involved in the three-way handshake for opening a TCP connection. Determine the flags that are set in the involved packets.
- Explore the TCP header options that appear in the three-way handshake and describe the role they play in configuring the TCP connection.
- Determine the window size of sliding window flow control (in bytes) for both directions of the TCP connection.
- How many packets with a TCP payload are sent by the client? How many are sent by the server?

- Is the TCP connection closed at the end of the web access? Explain the outcome. Explore the relevant packets and determine the flags that are set in these packets.

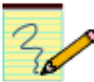
Step 6: *Wireshark* has features for tracking the packets of a TCP connections.

- **View the transmitted TCP payload:** Select one of the TCP packets captured by *Wireshark.* Then select "Analyze→ Follow→TCP Stream". Take a screen snapshot of the displayed window.

- **View the Flow Graph (time-sequence diagram):** In the menu bar of *Wireshark,* select "Statistics→ Flow Graph". In the displayed window, under "Flow Type" select "TCP Flows". Take a screen snapshot of the displayed window.

Step 7: Save the captured TCP packets, as you may need them for the lab report.

Step 8: Terminate the web server on *PC2* with `Ctrl-C` and terminate the *Wireshark* session.



Lab Question/Report

1. Answer the questions in Step 5. Support the answers with the *Wireshark* data saved in Step 8.
2. Provide the screen snapshots from Step 6 with a brief description.

Part 2. Working with *iPerf*

The *iPerf* utility is a Linux tool for generating synthetic traffic loads with the transport protocols TCP and UDP. It can be used for measuring packet transmissions between a client and a server. Together with *ping* and *traceroute*, *iPerf* is an essential utility program for debugging problems in IP networks. Running *iPerf* consists of setting up an *iPerf* server (or receiver) on one host and an *iPerf* client (or sender) on another host. Once the *iPerf* client is started, it sends data to the *iPerf* server.

By default, *iPerf* transmits data over a TCP connection. The *iPerf* client (sender) opens a TCP connection to the *iPerf* receiver, transmits data as fast as is possible and then closes the connection. The *iPerf* server must be running when the *iPerf* client is started. UDP data transfer is specified with the -u option. Since UDP is a connectionless protocol, the *iPerf* client (sender) starts immediately sending UDP datagrams at a specified rate, even if there is no *iPerf* server.

Syntax of the *iPerf* command

An *iPerf* server is started with the command

`iperf -s [-u] [-p port] [-B iface] -p port`

An *iPerf* client is started with the command

`iperf -c host [-u] [-p port] [-B iface] [-b Bandw] [-l Buflength] [-t Time] [-L port]`

The options of the command are:

-u Uses UDP instead of TCP. By default, *iPerf* uses TCP to send data.

-V Uses IPv6 addresses.

-p *port* Connects with or expects data packets on the specified port (default 5001).

-c *host* Starts *iPerf* in client mode, where *host* specifies the IP address of the server.

-b *n [KM]* Limits transmission rate of the client when using UDP (default is 1 Mbps) and sets a target rate when using TCP (default is as fast as possible). Here, `n K' limits the rate to n kbps and `n M' limits the rate to n Mbps.

-l *n [KM]* Sets the buffer size (TCP) or datagram length (UDP) to *n* (TCP default 128K, UDP default 1470).

-n *num [KM]* Number of bytes to be transmitted by the client (overrides the –t option). Note: *`-n 1'* sends a single byte of payload.

-M num Sets the maximum segment size for TCP transmissions

-t time Sets the duration of the data exchange in seconds (default 10 seconds).

Exercise 2-a. Transmitting data with UDP

This exercise consists of setting up a UDP data transfer between *PC1* and *PC2* and observe the UDP traffic.

Step 1: On *PC1*, start *Wireshark* to capture packets on interface *eth1* between *PC1* to *PC2*. Set a display filter to UDP traffic ("udp").

Step 2: On *PC2*, start an *iPerf* server that receives UDP traffic with the command

```
PC2$ iperf -s -u
```

Step 3: On *PC1*, start an *iPerf* client that transmits UDP traffic to *PC2* by typing

```
PC1$ iperf -c 10.0.5.22 -l 1000 -n 10000 -u
```

Observe the traffic captured by *Wireshark*.

a. How many packets are exchanged in the data transfer? How many packets are transmitted for each UDP datagram? What is the size of the UDP payload of these packets?
b. Compare the total number of bytes transmitted including Ethernet, IP, and UDP headers, to the amount of application data transmitted.
c. Inspect the fields in the UDP headers. Which fields in the headers do not change in different packets?
d. Observe the port numbers in the UDP header. How did the *iPerf* client select the source port number?

Take screen snapshots from *Wireshark* (for the lab report) that support your answers above.

Step 4: Create a flow graph of the UDP data exchange (by following Step 6 in *Exercise 1-b*). To limit the graph to UDP packets, select "*Limit to display filter"* in the flow graph window. Take a snapshot of the flow graph.

Step 5: Terminate *iPerf* on *PC2* by entering `Ctrl-C`.

Step 6: Repeat the *iPerf* experiment, but use IPv6 this time. For this you start the *iPerf* server on *PC2* with

```
PC2$ iperf -s -u -V
```

and the *iPerf* client on *PC1* with

```
PC1$ iperf -c fd01:0:0:5::22 -l 1000 -n 10000 -u -V
```

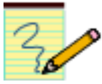
Here, the option -V forces *iPerf* to use IPv6.

Observe the traffic captured by *Wireshark*.

- Comment on the similarities and differences of the data exchange using IPv4 (in Steps 2 and 3) and IPv6. Take one or more screen snapshot from *Wireshark* to support your answer(s).

- Create a flow graph of the data exchange as in Step 4. To limit the flow to the IPv6 packets with UDP payload, set the display filter in *Wireshark* to *"UDP && IPv6"* and "*Limit to display filter"* in the flow graph window. Take a screen snapshot.

Step 7: Terminate the *iPerf* server on *PC2* with `Ctrl-C`.



Lab Question/Report:

1. Provide the answers in Step 3 and Step 5, and support your answers with the screen snapshots.
2. Include the flow graphs from Step 4 and Step 6.

Exercise 2-b. Transmitting data with TCP

Here, you repeat the previous exercise using TCP as transport protocol.

Step 1: Set the display filter in the *Wireshark* window to *"TCP"*.
Step 2: Start an *iPerf* server on *PC2* with

```
PC2$ iperf -s
```

Also, start an *iPerf* client on *PC1* that transmits packets to *PC2* with

```
PC1$ iperf -c 10.0.5.22 -n 10000
```

Observe the traffic captured by *Wireshark*.

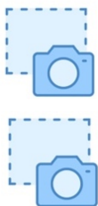
a. How many packets are exchanged in the data transfer? What are the sizes of the TCP segments?

b. Describe the order of packets with TCP payload (sent by *PC1*) and the acknowledgements for these packets (sent by *PC2*). What can explain this sequence.

c. How many packets do not carry a payload, that is, how many packets are control packets?

d. Inspect the TCP headers. Which packets contain flags in the TCP header? Which types of flags do you observe?

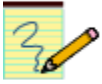
e. Observe the packets involved in closing the TCP connection. Do you observe a difference to the closing of the TCP connection in *Exercise 1-b.*



Take screen snapshots from *Wireshark* (for the lab report) that support your answers above.

Step 3: Create a flow graph of the TCP data exchange (see Step 7 in *Exercise 1-b*). To limit the graph to TCP packets, select "*Limit to display filter"* in the flow graph window. Take a snapshot of the flow graph.

Step 4: Terminate the *iPerf* server on *PC2* by hitting *CTRL-C* and terminate the *Wireshark* session.



Lab Question/Report:

1. Provide the answers in Step 2 and support your answers with the screen snapshots.
2. Include the flow graph from Step 3, and compare it with the flow graph of the UDP data exchange.
3. Compare the amount of data transmitted in the TCP and the UDP data transfers.

Part 3. IP Fragmentation of UDP and TCP traffic

In this part of the lab, you observe the effect of IP Fragmentation on UDP and TCP traffic. Fragmentation occurs when the transport layer sends a packet of data to the IP layer that exceeds the Maximum Transmission Unit (MTU) of the underlying data link network. For example, in Ethernet networks, the MTU is 1500 bytes. If an IPv4 datagram exceeds the MTU size, the datagram is fragmented into multiple datagrams, or, if the *Don't fragment (DF)* flag is set in the IPv4 header, the datagram is discarded.

When an IP datagram is fragmented, its payload is split into multiple IP datagrams, each satisfying the limit imposed by the MTU. Each fragment is an independent IP datagram, and is routed in the network independently from the other fragments. In IPv4, fragmentation can occur at the sending host or at intermediate IP routers. Fragments are reassembled only at the destination host.

Even though IP fragmentation provides flexibility that can hide differences of data link technologies to higher layers, it incurs considerable overhead, and, therefore, should be avoided. Transport protocols seek to avoid fragmentation with a *Path MTU Discovery* scheme that determines the smallest MTU found on any interface on the path from the source to the destination, referred to as Path MTU.

In IPv6, fragmentation only occurs at the source host, but is not performed at intermediate routers. Here, performing Path MTU Discovery is crucial, otherwise, payloads in packets that are too big for some intermediate router are never delivered to the destination.

You will explore the issues with IP fragmentation of TCP and UDP transmissions in the network from Figure 6.1, with *PC1* as sending host, *PC2* as receiving host, and *PC3* as intermediate IP router.

Exercise 3-a. Fragmentation of UDP traffic

In this exercise you observe fragmentation of UDP traffic. In the following exercise, *iPerf* is used to generate UDP traffic between *PC1* and *PC2* via IP router *PC3*. You gradually increase the size of UDP datagrams until fragmentation occurs. You can observe that IP headers do not have the *DF* bit set for UDP payloads.

Step 1: Verify that the network is configured as shown in Figure 6.1 and Table 6.1. The PCs should be configured as described in *Exercise 1-a.*

Step 2: Check the MTU value on all interfaces of all PCs. The MTU is displayed when you show the IP configuration of an interface, e.g,

```
PC2$ ip addr show eth0
```

Step 3: Start *Wireshark* to capture traffic on the *eth0* interfaces of both *PC1* and *PC2*.

Step 4: Use *iPerf* to generate UDP traffic between *PC1* and *PC2*. The connection parameters are selected so that IP Fragmentation does not occur initially.

- On *PC2*, execute the following command:

```
PC2$ iperf -s -u
```

- On *PC1*, execute the command:

```
PC1$ iperf -c 10.0.2.22 -l 1000 -n 5000 -u
```

Step 5: Vary the size of the UDP datagrams by increasing and decreasing the argument given with the "-l" option.

- Determine the size of the largest IPv4 datagram (which includes the IP header) which is not fragmented at *PC1*? Compare this to the MTU value of the *eth0* interface at *PC1*.

Take screen snapshots from *Wireshark* (for the lab report) that support your answer.

Step 6: Change the MTU value of the *eth1* interface of *PC3* to 600 bytes with the command

```
PC3$ sudo ip link set dev eth1 mtu 600
```

Step 7: Rerun the *iPerf* client on *PC1* as given in Step 4 (with the option "-l 1000").

Step 8: Observe the traffic that is captured in both *Wireshark* sessions.

a) Explore the first datagram sent by *PC1* and the reaction of *PC3*.
   - Determine the size of the datagram.
   - Check the value of the DF flag in the datagram.

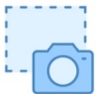
b) In the ICMP Destination Unreachable message that is sent by *PC3*, identify the MTU value at *PC3*.

c) When *PC1* receives the ICMP message, it changes the Path MTU that it uses for destination 10.0.2.22. This is stored in the routing cache of *PC1*, which you can retrieve with the command

```
PC1$ ip route show cache
```

d) Explore the next datagrams sent by *PC1* and how it is processed by *PC3*.
   - Determine the size of the datagrams.
   - Check the value of the DF flags in the datagram.

e) Where does fragmentation occur? Explain.



Take multiple screen snapshots from *Wireshark* (for the lab report) that document your observations in (a)–(e).

Resetting the route cache
The MTU value in the route cache is kept for a few minutes (default: 10 minutes). You can reset this by deleting the route cache with the command

```
PC1$ sudo ip route flush cache
```

Minimum MTU enforcement in IPv4
If you set the MTU value of *PC3* (*eth1*) to 400 and then repeat Step 6, the outcome (under recent Linux distributions), you observe fragmentation at *PC1* to an IP datagram size of 552, and at *PC3* to an IP datagram size of 400.
As background, IPv4 requires the MTU to be set to at least 576 bytes. While this limit is generally not enforced by hosts or routers, Linux seems to enforce a smallest MTU of 552 byte. This raises two questions: (1) Why does Linux not enforce the minimum MTU of 576 byte? (2) Why does Linux accept a command that sets the MTU to a value below the limit.

1. Provide your answer to Step 5 and add the supporting screen captures.
2. Report your observations in Step 8, to create a timeline of the events at *PC1* and *PC3* for dealing with the reduced MTU value at *PC3*. Use screenshots to support your observations.

Exercise 3-b. Avoiding fragmentation in TCP

TCP tries to completely avoid fragmentation with the following two mechanisms:

1. When a TCP connection is established, it negotiates a maximum segment size (MSS). Both the TCP client and the TCP server send the MSS in an option that is attached to the TCP header of the first transmitted TCP segment. Each side sets the MSS so that no fragmentation occurs at the outgoing network interface, when it transmits segments. The smaller value is adopted as the MSS value for the connection.
2. The exchange of the MSS only addresses MTU constraints at the hosts, but not at the intermediate routers. To determine the smallest MTU on the path from the sender to the receiver, TCP employs a method which is known as *Path MTU Discovery*, and which works as follows. The sender always sets the DF (Don't Fragment) bit in all IP datagrams. When a router needs to fragment an IP packet with the DF bit set, it discards the packet and generates an ICMP error message of type "*Destination unreachable; Fragmentation needed"*. Upon receiving such an ICMP error message, the TCP sender reduces the segment size. This continues until a segment size is determined which does not trigger an ICMP error message.

Step 1: Verify that the MTU all interfaces of the topology in Figure 6.1 are set to 1500 bytes. You can view the MTU by typing, for example for the *eth1* interface of *PC3*,

```
PC3$ ip addr show eth1
```

To set the MTU value of interface *eth1* on *PC3* to 1500 bytes, use the *ip link* command as follows:

```
PC3$ sudo ip link set dev eth1 mtu 1500
```

Step 2: Start *Wireshark* traffic captures on the *eth0* interfaces of *PC1* and *PC2*.

Step 3: Start an *iPerf* receiver on *PC2*, and a *iPerf* sender on *PC1* that generate TCP traffic with the commands

```
PC2$ iperf -s
PC1$ iperf -c 10.0.2.22 -n 4000
```

Observe the output of *Wireshark*, where you focus on the TCP segments with a payload that are sent by *PC1*.

a. Check the DF flags in the TCP segments with a payload.

b. Then, check the sizes of the TCP segments with a payload that are sent by *PC1*. Compare the sizes to the MTU limit.

Step 4: Change the MTU of the *eth0* interface of *PC2* to 600 bytes with the command

```
PC2$ sudo ip link set dev eth0 mtu 600
```

Then rerun the *iPerf* client on *PC1* with

```
PC1$ iperf -c 10.0.2.22 -n 4000
```

In *Wireshark*, check the sizes of TCP segments sent by *PC1* that carry a payload. Observe that segment sizes are such they fit the MTU limit at *PC2*.

a. Are there any ICMP messages sent to *PC1*?

b. Determine how *PC1* learns that the MTU limit of *PC2* has changed and that it should send smaller segments.

Take screen snapshots from *Wireshark* (for the lab report) that support your answers above.

Step 5: Now change the MTU size of interface *eth0* on *PC2* back to 1500 bytes. Then, change the MTU size on interface *eth1* of *PC3* to 600 bytes.

Step 6: Repeat the *iPerf* transmission in Step 4.
Observe the traffic captured by *Wireshark*, where you should focus on the differences of the *iPerf* transmissions from Step 4

a. Do you observe fragmentation? If so, where does it occur and what are the consequences. Explain your observation.

b. Describe how the observed ICMP messages are used for *Path MTU Discovery*.

Take screen snapshots of both *Wireshark* sessions showing the packet list (top pane) of all TCP packets. For one observed ICMP message,

Step 7: Terminate the *iPerf* server on *PC2*. If you do not continue with *Exercise 3-c*, terminate the *Wireshark* sessions, and reset all MTU values that you have changed to 1500.

Lab Question/Report:

1. In Step 4, you explored how TCP determines the Path MTU if the smallest MTU is at the destination. Provide the answers to the questions in Step 4 and use the screen snapshots to support your answers.

2. In Step 6, you observed how TCP determines the Path MTU if the smallest MTU is found at an intermediate router. Provide the answers to the questions in Step 6 and use the screen snapshots to support your answers.

Exercise 3-c. Fragmentation of UDP traffic in IPv6

In IPv6, routers never perform fragmentation. Fragmentation is limited to the source host. The following explores how IPv6 handles UDP datagrams whose size exceeds the MTU at an intermediate router.

Minimum MTU enforcement in IPv6
In IPv6, the minimum MTU of an interface is 1280 bytes. Linux systems enforce the MTU limit of IPv6 better than they do for IPv4. This is done by disallowing the configuration of an IPv6 address if the MTU is less than 1280 bytes. If an IPv6 address has been configured for an interface and the MTU is set to a value below the minimum size, the configured address is deleted.

Step 1: On *PC3*, check the IP addresses and MTU of interface *eth1* at *PC1* with the command

```
PC3$ ip addr show eth1
```

You should see that the MTU is set to 600 bytes (as configured in *Exercise 3-b*), that the IPv4 address is as it was configured in Part 1, but that the configured IPv6 address is no longer present.

Step 2: Unless the MTU is increased to at least 1280 bytes, Linux refuses to accept the configuration. So, set the MTU of the *eth1* interface to the minimum value with

```
PC3$ sudo ip link set dev eth0 mtu 1280
```

Step 3: Make sure that *Wireshark* traffic captures on the *eth0* interfaces of *PC1* and *PC2* are running. If necessary, restart the traffic captures.

- As in *Exercise 3-a*, run *iPerf* to generate UDP traffic over IPv6 between *PC1* and *PC2*. On *PC2*, execute the following command:

```
PC2$ iperf -s -u -V
```

- On *PC1*, send UDP datagrams of with a payload of 1400 bytes with the *iPerf* client command

```
PC1$ iperf -c fd01:0:0:2::22 -l 1400 -n 5000 -u -V
```

Observe the traffic that is captured in both *Wireshark* sessions, where you should focus on the differences to the observations in the UDP/IPv4 experiment from *Exercise 3-a*.

a. Check the first datagram sent by *PC1* and the reaction of *PC3*.

   - Determine the size of the datagram.
   - Obviously, there is no DF flag set. This is implicit for each IPv6 datagram.

b. *PC3* sends an ICMPv6 Packet Too Big message to *PC1*, which includes the MTU value set at *PC3*.

c. When *PC1* receives the ICMPv6 message, it changes the Path MTU that it uses for destination *fd01:0:0:2::22*. Different from IPv4, the Path MTU is not stored routing cache of *PC1*. You can access the stored MTU value by querying the routing table lookup for the given destination. The command is

   PC1$ **ip route get to fd01:0:0:2::22**

d. The following datagrams sent by *PC1* are fragmented by *PC1*. The information about the fragments is sent in a so-called fragment header, which follows the IPv6 header.

   - Explore the fragment headers of the UDP datagram. Match the fields in the fragment header with the corresponding fields in the IPv4 header.

e. Check the traffic that is forwarded by *PC3* to *PC2*, to verify that there is no additional fragmentation taking place at intermediate router *PC3*.

Take screen snapshots from *Wireshark* (for the lab report) that support your answers above.

Step 5: Terminate the *iPerf* server on *PC2* by hitting *CTRL-C* and terminate the *Wireshark* sessions. Also, reset all MTU values that you have changed in Part 3 to 1500.

Lab Question/Report:

1. Provide the answers to the questions in Step 3. Support your answers with the screen snapshots.

Part 4. TCP connection management

TCP is a connection-oriented protocol. The establishment of a TCP connection is initiated when a TCP client sends a request for a connection to a TCP server. The TCP server must be running when the connection request is issued.

TCP requires three packets to open a connection. This procedure is called *three-way handshake.* During the handshake, the TCP client and TCP server negotiate essential parameters of the TCP connection, including the initial sequence numbers, the maximum segment size, and the size of the windows for the sliding window flow control. TCP requires three or four packets to close a connection. Each end of the connection is closed separately, and each part of the closing is called a *half-close.*

TCP does not have separate control packets for opening and closing connections. Instead, TCP uses bit flags in the TCP header to indicate that a TCP header carries control information. The flags involved in the opening and the closing of a connection are: SYN, ACK, and FIN.

Here, you use *Telnet* to set up a TCP connection and observe the control packets that establish and terminate a TCP connection. The experiments involve *PC1* and *PC2* in the network shown in Figure 6.1.

Exercise 4-a. Opening and Closing a TCP Connection

Here, you establish a *Telnet* session, which create a TCP connection. You will inspect the packets that open and close the TCP connection, and determine the parameters of the connection that are negotiated between the TCP client and the TCP server.

Step 1: Verify that the network is configured as shown in Figure 6.1 and Table 6.1. The PCs should be configured as described in *Exercise 1-a*.

Check that that the MTU values of all interfaces of *PC1* and *PC2* are set to 1500 bytes, which is the default MTU for Ethernet interfaces. Running `ip addr show' on a Linux PC displays the MTU values of all interfaces.

Step 2: Start *Wireshark* on the *eth1* interface of *PC1*. Set the display filter to `tcp'.

Step 3: **Establishing a TCP connection**: Start a *Telnet* server on *PC2* with the command

```
PC2$ sudo service xinetd start
```

Telnet
Telnet is a remote terminal program that operates over a TCP connection. Since *Telnet* does not encrypt the payload, it is not widely used anymore, and has largely been replaced by *ssh*. TCP servers bind to the well-known TCP port 23.
Telnet performs a login on the remote system. On the Linux PCs, the username and password are both `*labuser*'. After the login, you can enter commands on the remote system.
To terminate a *Telnet* session, type exit or Ctrl-d.

Step 4: On *PC1*, establish a set of *Telnet* session by starting a *Telnet* client on *PC1* with the command

```
PC1$ telnet 10.0.5.22
```

The *Telnet* client now establishes a TCP connection to the Telnet server. Do not complete the login, when so prompted.

Observe the TCP segments of the packets that are transmitted:

a. Identify the packets of the three-way handshake. Which flags are set in the TCP headers? Explain how these flags are interpreted by the receiving TCP server or TCP client.

b. During the connection setup, the TCP client and TCP server tell each other the first sequence number they will use for data transmission. What is the initial sequence number of the TCP client and the TCP server?

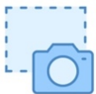
c. Determine the configurations of the TCP connection that are set during the three-way handshake.

d. The TCP client and TCP server exchange window sizes to get the maximum amount of data that the other side can sent at any time. Given the configuration of the TCP connection, determine the values of the window sizes for the TCP client and the TCP server. With this, determine the range of sequence numbers that client and server can send to each other, once the three-way handshake is completed.

e. Compare the exchanged windows sizes as they are displayed by *Wireshark* with the actual hexadecimal values in the TCP header fields. Describe the difference.

f. Identify the first packet that contains application data? Compare the sequence number used in the first byte of application data sent by the TCP client to the TCP server and compare it to the initial sequence number.

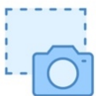
g. How long does it take to open the TCP connection, i.e., how much time elapses before the three-way handshake is completed?



Take multiple screen snapshots from *Wireshark* (for the lab report) that document your observations in (a)–(f).

Step 5: **Closing a TCP connection (initiated by server)**: If the TCP client does not complete the login, the *Telnet* session is terminated by the server, and the TCP server closes the established TCP connection. A message is displayed at the *Telnet* client application, that the session is closed.

Identify the packets that are involved in closing the TCP connection. Which flags are set in these packets? Explain how these flags are interpreted by the receiving TCP client and TCP server.



Take screen snapshots of the TCP headers in *Wireshark* of those TCP segments that are involved in closing the TCP connection.

Exercise 4.b- Requesting a connection to non-existing host

Here you observe how often a TCP client tries to establish a connection to a host that does not exist, before it gives up.

Step 1: Start a new traffic capture with *Wireshark* on interface *eth1* of *PC1*.

Step 2: Set a static entry in the neighbor cache for the non-existing IP address 10.0.5.100. Note that neither the IP address nor the MAC address exist in the network.

```
PC1$ sudo ip neigh add 10.0.5.100 lladdr 01:02:03:04:05:06 dev eth1
```

Display the neighbor cache to make sure the entry is correct

```
PC1$ ip neigh show
```

Step 3: From *PC1*, establish a *Telnet* session to the non-existing host with the command

```
PC1$ telnet 10.0.5.100
```

Observe the TCP segments that are transmitted. The TCP client retransmits the SYN segments if there is no reply. Each retransmission is a retry to open the connection.

a. How often does the TCP client try to establish a connection? How much time elapses between repeated attempts to open a connection?

b. Does the TCP client terminate or reset the connection, when it gives up with trying to establish a connection?

c. Why does this experiment require to set a static neighbor cache entry?

Take a screen snapshot in *Wireshark* of the packet list, showing all attempts by *PC1* to establish the TCP connection.

Step 4: Remove the neighbor cache entry that you added in Step 2.

```
PC1$ sudo ip neigh del 10.0.5.100 dev eth1
```

Exercise 4.c- Requesting a connection to a non-existing port

When a host tries to establish a TCP connection to a port at a remote server, and no TCP server is listening on that port, the remote host terminates the TCP connection. This is observed in the following exercise.

Step 1: Make sure the *Telnet* server on *PC2* is still running, and re-start it if necessary. Establish a TCP connection to port 80 of *PC2* by typing

```
PC1$ telnet 10.0.5.22 80
```

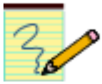
Observe the TCP segments of the packets that are transmitted:

- How does TCP at the remote host (*PC2*) terminate this connection? How is this different from closing the connection in *Exercise 4-a* and from what you observed in *Exercise 4-b.*

Take a screen snapshot in *Wireshark* of the packet list, showing all TCP segments that are exchanged between *PC1* and *PC4*.

Step 2: Terminate the *Wireshark* session.

Lab Question/Report:



- Answer the questions in *Exercise 4-a* (Steps 4 and 5), *Exercise 4-b* (Step 3), and *Exercise 4-c* (Step 1). Include *Wireshark* screen snapshots to support your answers.

Exercise 4-d. Configuration parameters of TCP

TCP is a complex protocol with a large number of tunable parameters. On Linux systems, some, but far from all, parameters can be globally modified through system parameters. A few system parameters can be tuned with the `ip route` command by attaching parameter settings to a routing table entry. Application programs can modify TCP parameters individually for each stream socket, where each stream socket establishes one TCP connection.

In addition to tunable parameters, TCP connections maintain numerous internal variables. For active TCP connections the current value of these variables can be displayed with the command *"ss -i"*.

Step 1: On *PC1*, list the Linux system parameters that are available to modify TCP configuration settings with the command

```
PC1$ sudo sysctl -a | grep tcp | less
```

and view the output.

Save the output to a file, e.g., with the command

```
PC1$ sudo sysctl -a | grep tcp > myfile.txt
```

Step 2: Next, set up a TCP connection between *PC1* and *PC2* using the *Telnet* application. Start a *Telnet* server on *PC2* with the command

```
PC2$ sudo service xinetd restart
```

Then, start a *Telnet* client on *PC1* with the command

```
PC1$ telnet 10.0.5.22 80
```

Complete the login with the "*labuser*" account.

Step 3: On *PC2*, display the current values of TCP variables and configuration parameters with the *socket statistic (ss)* command

```
PC2$ ss -i
```

The command displays the state of configuration parameters (which remain unchanged during a data transfer) as well as variables (which change during a data transfer).

- Using your knowledge of TCP, identify state information that is relevant to flow control, congestion control, and error control.

- Take a snapshot of the displayed information.

Step 4: **(Optional)** To distinguish immutable configuration parameters from variables, you can initiate a data transfer and repeatedly run *"ss -i"* on *PC2*. This can be done by creating a large file in the *Telnet* session on *PC1*, and then displaying the file.

On *PC1*, create a large text file with the command and then display the text file with the commands

```
PC2$  yes "Put some text here" | head -n 1000000 > tmp.txt
PC2$  cat tmp.txt
```

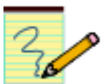
Since *PC1* is logged into *PC2*, it shows the prompt of *PC2*.

While the display is running, switch to *PC2* and repeatedly run the socket statistic command

```
PC2$ ss -i
```

Observe that some information changes and some remains the same. The former are the configuration parameters and the latter are the variables.

Lab Question/Report:



- Provide the list of system parameters from Step 1. For 5 of the listed parameters, explain their meaning.
- Go to the Linux man page for the TCP protocol at
  https://man7.org/linux/man-pages/man7/tcp.7.html
  which lists TCP parameters than be accessed as Linux system parameters, as well as TCP parameters that are accessed by setting options of a stream socket (using `setsockopt`). Identify parameters that can only be set via `setsockopt`.
- Provide the snapshot from Step 3. Identify and explain the configuration parameters and time-varying variables that are relevant to flow control, error control, and congestion control. You may want to refer to the Linux man page
  https://man7.org/linux/man-pages/man8/ss.8.html
  for an explanation of the parameters.

Part 5. TCP data exchange – Interactive applications

In Parts 5 and 6 you study acknowledgments and flow control in TCP. In TCP, the receiver of data segments acknowledges the receipt by sending segments that have the ACK flag set in the TCP header. These segments are called *acknowledgements* or ACKs. In TCP, each transmitted byte of application data has a sequence number. The sender of a segment writes the sequence number of the first byte of transmitted application data in the sequence number field of the TCP header. When a receiver sends an ACK, it writes a sequence number in the acknowledgement number field of the TCP header. The acknowledgement number is by one larger than the highest sequence number that the receiver wants to acknowledge. A TCP receiver can acknowledge multiple segments in a single ACK. This is called *cumulative acknowledgements* or *cumulative ACKs*.

With cumulative ACKs, the receiver can acknowledge a sequence number only if all bytes with a smaller sequence number have been received. If a TCP segment has been lost, but later TCP segments were delivered, cumulative ACKs do not allow the receiver to acknowledge these later TCP segments. TCP has an (optional) feature, referred to as *selective ACKs,* that allows the receiver to acknowledge ranges of sequence numbers that are smaller than the last correctly received sequence numbers. Selective ACKs are carried as options in a TCP header.

In TCP, different mechanisms are at play for sending acknowledgements of interactive applications and bulk data transfers. This part of the lab addresses acknowledgments interactive applications, such as *Telnet.* Interactive applications typically generate a small volume of data, e.g., one byte at a time. Since interactive applications are generally delay sensitive, a TCP sender does not wait until the application data fills a complete TCP segment, and, instead, TCP sends data as soon as it arrives from the application. This, however, results in an inefficient use of bandwidth where small segments with only one byte of payload mainly consist of protocol headers.

TCP has mechanisms that keep the number of segments with a small payload small. One such mechanism, called *delayed acknowledgements*, requires that the receiver of data waits for a certain amount of time before sending an ACK. If, during this delay, the receiver has data for the sender, the ACK can be piggybacked to the data, thereby saving the transmission of a segment. Another such mechanism, called *Nagle's algorithm*, limits the number of small segments that a TCP sender can transmit without waiting for an ACK.

Today, most TCP connections deal with bulk data transfers. Also, the waste of bandwidth consumed by interactive applications appears negligible in the light of high data rates available to even end users. For this reason, the relevance of TCP mechanisms for interactive applications is diminished, and many operating systems disable these mechanisms by default or allow users to disable them.

The interactions between TCP client and TCP server depend on the delay as well as the available bandwidth rate between client and server. The maximum amount of data that can be in transit (in one direction) between client and server is the product of the available bandwidth and delay, referred to *bandwidth-delay product*. To test a network under a variety of network environments, the experiments in this lab increase the bandwidth-delay product by configuring a rate limit as well as a delay on *PC3*, which is the IPv4 router between *PC1* and *PC2* in Figure 6.1. The rate limit and delay are configured on the

network interfaces of *PC3* by attaching a network emulation (*netem*) queueing discipline (*qdisc*) to the network interfaces.

<u>**Configuring a *netem* qdisc**</u>

`tc qdisc show`
`tc qdisc show eth0`

Displays the current configuration of scheduling components on all interfaces and on interface *eth0*. The command `tc`, which stands for *traffic control*, provides commands for modifying packet classification and scheduling methods. In Linux, `qdisc` is an acronym for queuing discipline, which refers to a packet scheduling or shaping method.

`sudo tc qdisc add dev eth0 root netem delay 30ms rate 10Mbit`

The command adds a *network emulation qdisc* to interface *eth0* that imposes a delay of 30ms and a rate limit of 10 Mbps. All traffic that is sent on this interface experiences the configured delay and rate limit.

`sudo tc qdisc change dev eth0 root netem delay 30ms`
`sudo tc qdisc change dev eth0 root netem rate 10Mbit`

Changes an existing network emulation on interface *eth0*. Here, the change is to impose a delay without a rate limit, and a rate limit without a delay.

`sudo tc qdisc change dev eth0 root netem rate 10Mbit limit 100`

Changes an existing network emulation on interface *eth0* to set a rate limit of 10 Mbps and to limit the buffer size to hold at most 100 packets.

`tc qdisc add dev eth0 root netem loss 0.1%`
`tc qdisc change dev eth0 root netem loss 0.1%`

Adds a network emulation, that drops packets with a probability of 0.1 %. The change option must be used in case a *netem qdisc* is already configured.

`sudo tc qdisc del dev eth0 root netem`

Deletes a configured network emulation.
There are many additional configuration options available for network emulation.

Exercise 5-a. Interactive Applications (low latency, high data rate)

Here you observe interactive data transfer in TCP by establishing a TCP connection from *PC1* to *PC2* across *PC3* as IP router.

Step 1: Continue with the network setup from Part 1. Make sure that the IP configuration is as given in *Exercise 1-a*.

Step 2: Start or restart a *Telnet* server on *PC2* with the command

```
PC2$ sudo service xinetd restart
```

Step 3: Start a traffic capture with *Wireshark* for interface *eth1* of *PC1* for interface *eth1.* Limit the displayed packets to TCP traffic by setting a display filter to *"tcp"*.

Step 4: **Disable *Telnet* display in *Wireshark***. As seen earlier, the traffic of a *Telnet* session uses a single TCP connection. However, *Wireshark* displays TCP segments that carry *Telnet* data differently than TCP segments that only has control information, e.g., SYN, ACK, etc. This can be disabled by deselecting *Telnet* as a recognized protocol. To do this, select *"Analyze → Enabled Protocols",* then unselect the protocol "*TELNET"*. Note that this setting is permanent and remains when you restart *Wireshark*.

Step 5: On *PC1*, establish a *Telnet* session to *PC2* by typing:

```
PC1$ telnet 10.0.2.22
```

Complete the login with username and password "*labuser*".

Step 6: On *PC1*, type a few characters in the window with the *Telnet* session. The *Telnet* client sends each typed character in a separate TCP segment to the *Telnet* server, which, in turn, echoes the character back to the client. Including ACKs, one would expect to see four packets for each typed character. However, due to delayed acknowledgments, this is not the case.

Observe the output of *Wireshark*:

- Observe the number of packets exchanged between the PCs for each keystroke. Describe the payload of the packets. Use your knowledge of delayed acknowledgements to explain the sequence of segment transmissions. Explain why you do not see four packets per typed character.
- Which flags, if any, are set in the TCP segments that carry typed characters as payload? Explain the role of these flags.
- Why do segments that have an empty payload carry a sequence number? Why does this not result in confusion at the TCP receiver?
- Create a flow graph that shows the transmission pattern of packets and the timing. When the TCP client (*PC1*) receives the echo of a character, it waits a certain time before sending the ACK. How long is this delay? How much does the delay vary?

Step 7: Type characters in the *Telnet* client program as fast as you can, e.g., by pressing a key and holding it down. Do you observe a difference in the transmission of segment payloads and ACKs?

Step 8: Take screenshots from the *Wireshark* display and the flow graph that support your answers in Steps 6 and 7.

Step 9: If the observed three packets per typed character are due to delayed acknowledgements, then disabling delayed acknowledgements should lead to different observations.
Delayed acknowledgements are disabled by setting the QUICKACK option. If enabled at the receiver, the receiver immediately acknowledges every arriving TCP segment. The QUICKACK option can be

modified via the routing table by setting the option for a particular routing table entry. To enable QUICKACK at the TCP server on *PC2* for a client on *PC1*, type the command

```
PC2$ sudo ip route change default via 10.0.2.33 quickack 1
```

This command applies QUICKACK to all outgoing packets that use the default gateway. Run the command `*ip route show*' to verify the setting.

Step 10: As in Step 6, type a few characters in the window with the *Telnet* session, and observe the output in *Wireshark*.
Create a flow graph that shows the transmission pattern after you changed the QUICKACK setting.

- Compare the flow graph to the previous flow graph.
- Take a snapshot of the flow graph. The graph should show the packets resulting from a few character strokes on *PC1*.

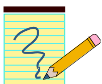
Step 11: Re-enable delayed acknowledgments by disabling QUICKACK on *PC2* for the default route. This is done with the command

```
PC2$ sudo ip route change default via 10.0.2.33 quickack 0
```

Now, the packet transmissions when typing characters at the *Telnet* client on *PC1* should return to the pattern observed in Step 5. Verify this by observing the captured traffic on *Wireshark*, and, possibly, creating another flow graph.

Step 12: You may leave the *Telnet* session on *PC2* in place for the next exercise. Also, you may keep the *Wireshark* traffic capture running.

Lab Question/Report:



1. Include your answers to the questions in Steps 6 and 7. Include screenshots from Step 8 that support your answers.
2. For one character typed at the Telnet client, include a drawing that shows the transmission of TCP segments between *PC1* and *PC2* due to this character.
3. Describe your observations from Step 10 and compare them to the earlier observations. Use the screen capture of the flow graph to support your observation. Comment on the validity of the conclusion that QUICKACK disables delayed acknowledgements.

Exercise 5-c. Interactive Applications (high latency)

This exercise repeats the previous exercise, but establishes a data connection over link with a latency of 50ms. Due to the long delay, one would expect that the TCP sender transmits multiple segments, each carrying a payload of one typed character. However, this is not always the case. A heuristic in TCP, called *Nagle's algorithm*, forces the sender to wait for an ACK after transmitting a small segment that carries

only one byte of payload, even if the window size would allow the transmission of multiple segments. Therefore, when sending small TCP segments, you should only observe one TCP segment in transmission at a time, no matter how slow or fast you type.

Step 1: On *PC3*, impose a delay of 50ms on both interfaces with the commands

```
PC3$ sudo tc qdisc add dev eth0 root netem delay 50ms
PC3$ sudo tc qdisc add dev eth1 root netem delay 50ms
```

Step 2: Check that the *Wireshark* traffic capture on interface *eth0* of *PC1* is running. If necessary, restart the traffic capture. The display filter should be set to "tcp".

Step 3: On *PC1*, establish a *Telnet* session to *PC2* by typing

```
PC1$ telnet 10.0.2.22
```

As done earlier, complete a login as *labuser*.

Step 4: As in the previous exercise, type a few characters in the console window with the *Telnet* session. Vary the rate at which you type characters in the *Telnet* client program.

Observe the output of *Wireshark*. Also, create a flow graph of the data exchange.

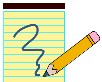
a. Observe the number of packets that are exchanged between the PCs for each keystroke? Observe how the transmission of packets changes when you type characters more quickly.

b. Do you observe delayed acknowledgments? Why is the outcome expected?

c. If you type very quickly, i.e., if you hold a key down, you should observe that multiple characters are transmitted in the payload of a segment. Explain this outcome.

Step 5: Take screenshots from the *Wireshark* display and the flow graph that support your answers in Step 4. Include a screen snapshot or an excerpt from the flow graph showing that Nagle's algorithm is used by the TCP sender.

Step 6: Terminate the *Telnet* session by typing *exit* and stop the traffic capture with *Wireshark.*



Lab Question/Report:

1. Include your answers to the questions in Step 4. Use the *Wireshark* screen captures and the flow graph to support your answers.

2. Include the screen snapshot or flow graph showing that Nagle's algorithm is used by the TCP sender. Provide an explanation.

Part 6. TCP data exchange – Bulk data transfer

Flow control enables a receiver of data to control the transmission rate of the sender of data in order to avoid getting overwhelmed with data. Flow control is not an issue with interactive applications, since the traffic volume of these applications is small, but plays an important role in bulk data transfers.

In TCP, the receiver controls the amount of data that the sender can transmit using a *sliding window flow control* scheme, with the acknowledgement and the window size fields in an ACK segment. The number of bytes that the receiver is willing to accept is written in the window size field. An ACK that has values *(250, 100)* for the acknowledgement number and the window size is interpreted by the TCP sender, that the sender is allowed to transmit data with sequence numbers *250, 251, …, 349*. The TCP sender may have already transmitted some data in that range.

The amount of data that can be in transit in each direction of a TCP connection can be increased up to the bandwidth-delay product between TCP client and TCP server, marking the point where the efficiency of a network is maxima. With a 16-bit long window size field, the maximum data in transit allowed by window flow control is 65,535 bytes, which is well below the bandwidth-delay product in modern networks (Note that a 300 km long link operating at 1 Gbps already has a bandwidth-delay product of at least 1 Mb). For this reason, TCP uses *window scaling*, where the window flow control field is multiplied by a power of 2. The value of the power of 2, referred to as *scaling factor*, is negotiated between TCP client and server in TCP options during the connection establishment phase.

In current Linux systems, the default scaling factor of the window size field is set to 7 (or a multiplying factor of 2^7=128), which permits up to ~8.4 MB in transit. With window sizes up to this range, the mechanisms of sliding window flow control can be observed only when transmitting massive amounts of data and when transmitting at a very high data rate. By disabling window scaling, we will make the mechanisms of window flow control observable at lower rates and data volumes. We can further limit the sliding windows by configuring limits on the maximum window size.

In this part of the lab, you observe acknowledgements and flow control for bulk data transfers, where traffic is generated with the *iPerf* tool. Not that bulk data transfers generally transmit full segments, that is, segments whose size is equal to the MSS. To observe the bulk data transfer, we introduce a feature of *Wireshark* that allows you to view the data of a TCP connection in a graph.

Observations depend on network equipment

The outcomes of the experiments in this part depend on the maximum data rate between *PC1* and *PC2*, which is determined by the network interface cards, and intermediate switches and routers. If the configured network is run as an emulation, the data rate is limited by the CPU where the emulation is running.

Exercise 6-a. TCP Data Transfer – Fast link

The purpose of this exercise is to observe the operation of the sliding window flow control scheme in a bulk data transfer, where *PC1* sends a large number of segments to *PC2* using the *iPerf* traffic generation tool.

Step 1: Continue with the network setup from Part 1. Make sure that the IP configuration is as given in *Exercise 1-a.*

Step 2: Start *Wireshark* on *PC1* for interface *eth1*, and start to capture traffic. Do not set any display filters.

Step 3: Use *iPerf* to generate TCP traffic between *PC1* and *PC2*.

- On *PC2*, start a *iPerf* receiving process by typing:

```
PC2$ iperf -s
```

- On *PC1*, start a *iPerf* sender process that sends 500 kB of application data by typing:

```
PC1$ iperf -c 10.0.5.22 -n 500K
```

  By using 10.0.5.22 as destination address, traffic will go over through the direct Ethernet link between *PC1* and *PC2*.

Step 4: From the output of *Wireshark* on *PC1*, observe the sliding window flow control scheme. The sender transmits data up to the window size advertised by the receiver and then waits for ACKs.

1. Display the flow graph of the TCP connection (by selecting a packet in *Wireshark* and then selecting Statistics→FlowGraph). Observe the transmission of TCP segments and ACKs. How frequently does the receiver send ACKs? Is there an ACK sent for each TCP segment, or less often. Can you determine the rule used by TCP to send ACKs? Can you explain this rule?

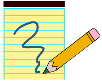
2. How much data (measured in bytes) does the receiver acknowledge in a typical ACK? What is the most data that is acknowledged in a single ACK?

3. What is the range of the window sizes advertised by the receiver? How does the window size vary during the lifetime of the TCP connection?
   Note: You can use a display filter, e.g., "*ip.src==10.0.5.22*", to only show packets sent by *PC2* to *PC1*.

4. Select an arbitrary ACK packet in *Wireshark* sent by *PC2* to *PC1*. Locate the acknowledgement number in the TCP header. Now relate this ACK to a segment sent by *PC1*. Identify this segment in the *Wireshark* output. How long did it take from the transmission of the segment, until the ACK arrives at *PC1*?

5. Determine whether, or not, the TCP sender generally transmits the maximum amount of data allowed by the advertised window. Explain your answer.

Step 5: Take screen snapshots *Wireshark* windows and excerpts of the flow graph so that you can support your answers above.



Lab Question/Report:

1. Include your answers to the questions in Step 4. Include screen captures from Step 5 to support your answers.

Exercise 6-b. Generating graphs of TCP data transfer

In addition to the flow graphs*, Wireshark* can generate graphs that plot details of the transmissions on a TCP connection. This exercise familiarizes you with these graphing capabilities of *Wireshark* and shows how you can extract information from these graphs.

Step 1: Select a TCP connection: In the *Wireshark* packet list, select a packet from the TCP connection of the *iPerf* measurement in *Exercise 6-a*.

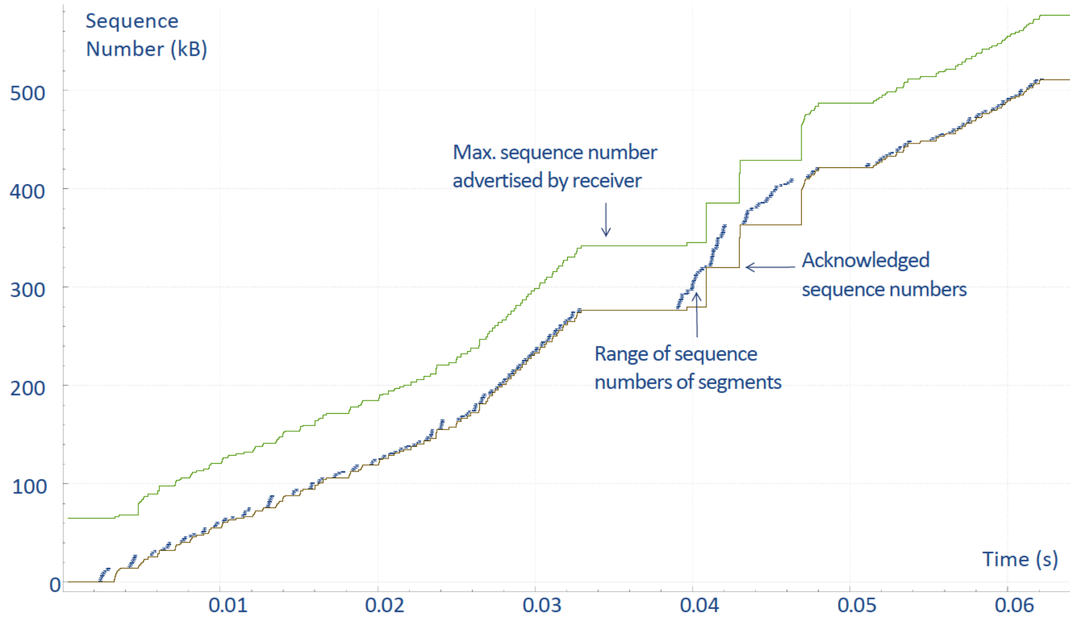
Step 2: Select the type of graph: From the *Wireshark* menu, select *Statistics→TCP Stream Analysis* in the pull-down menu. This show the available graphs for selections, which are all plotted as functions of time.

- *Time-Sequence (Stevens):* Displays the transmission of sequence numbers, with one data point for each transmission of a TCP segment.
- *Time-Sequence (tcptrace):* In addition to showing the transmissions of TCP segments, the graph also includes acknowledgements, the window size advertised by the receiver, and selective acknowledgements.
- *Throughput:* Displays the average throughput of data transmissions.
- *Round Trip Time:* Displays the roundtrip time (RTT) as a function of time.
- *Window Scaling:* Displays the advertised window size (received by the sender) and the number of transmitted but unacknowledged bytes (transmitted by the sender). The latter can be thought of as the bytes in transit, and are also referred to as the "*bytes in flight*".

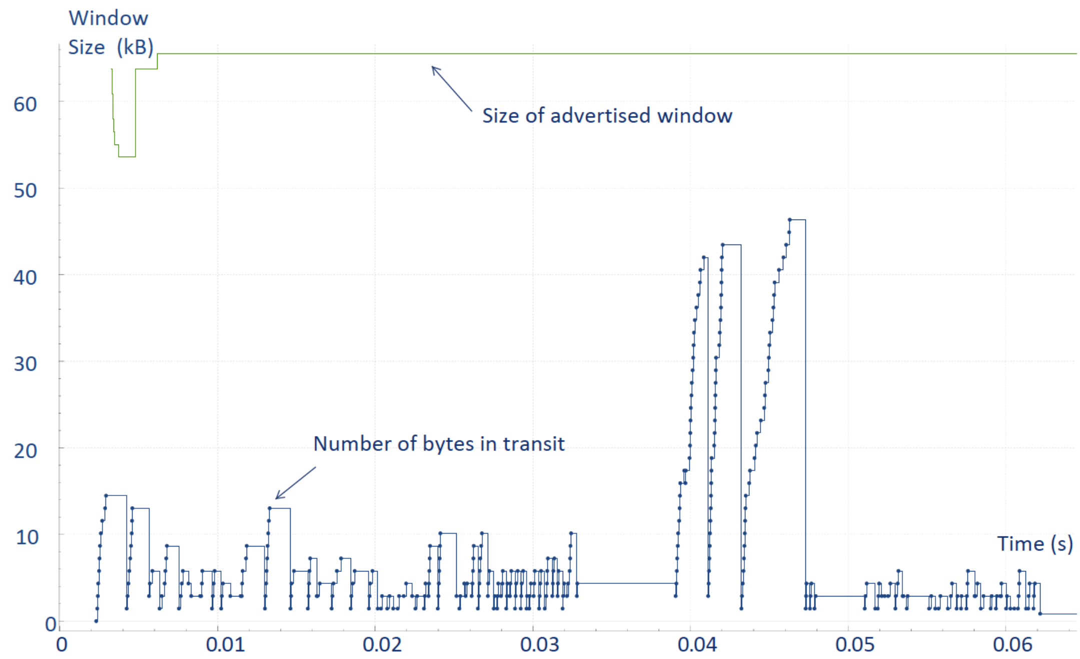
Stream analysis graphs

- You can switch between the different types of graphs by selecting the pull-down menu shown with the graphs.
- For each TCP connection, you get different graphs for each direction of data transmission. Once a graph is displayed, selecting "Switch Direction" displays the graph for the other direction.
  For all *iPerf* measurements in this lab, select the direction that shows the transmissions from the *iPerf* client to the *iPerf* server.

This lab uses the graphs for *Time-Sequence (tcptrace) and Window Scaling,* which are shown in Figure 6.2.



(a) Time-Sequence (tcptrace)



(b) Window Scaling.

Figure 6.2. *Wireshark* graphs for TCP connection.

The time-sequence graph uses colors to distinguish different types of data:

Color codes of time-sequence graphs:	
Blue vertical lines	Range of sequence numbers of transmitted TCP segments
Green line	Max. sequence number that permitted by window flow control
Brown line	Sequence number of the maximum contiguous data that has been acknowledged (not accounting for selective acknowledgements)
Red vertical lines	Range of sequence numbers that are acknowledged via selective acknowledgments

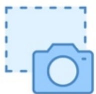
The colors used in the window scaling graph is as follows:

Color codes of window scaling graphs:	
Green line	Size of the advertised window (received from the other end of the connection)
Blue line	Number of bytes in transit, i.e., bytes that are transmitted but have not been acknowledged. Dots on the line indicate transmissions of TCP segments.

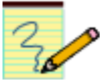
Step 3: ***Navigate the graphs*****:** Create a time-sequence graph for the *iPerf* transmissions from the previous exercise. Learn how to navigate the graphs generated by *Wireshark*, to display certain ranges of the displayed data. Options for navigating these graphs are as follows:

- A menu at the bottom of the graph allows to select "drag" or "zooms". When selecting "drag", holding the left mouse button in the displayed graphs allows to move the displayed data. When "zooms" is selected, selecting an area in the graph zooms while holding the left mouse button enlarges that section of the graph.
- Selecting the right mouse button (when the mouse is located on the graph) displays all available options and shortcuts.
- The displayed data is reset to the original graph by selecting "Reset" or typing the shortcut "0".
- The displayed graph can be saved to a file by selecting the "Save as …" button.

Step 4: **Interpret the graphs:** Explore the graphs for time-sequence (tcptrace) and window scaling for the TCP connection of the *iPerf* transmissions from *PC1* to *PC2* in Step 3 of *Exercise 6-a.*



1. Take screen snapshots of the graphs or save the graphs as PDF files.
2. Review the questions in Step 4 of *Exercise 6-a* and determine which answers you can obtain directly from the graphs, possibly by using the navigation features to focus on certain parts of the graph.



Lab Question/Report:

1. Include the graphs from Step 4.
2. Which of the questions in Step 4 of *Exercise 6-a* (if any) cannot be provided easily from the two graphs. Justify your answer.

Exercise 6-c. TCP Data Transfer – Small receiver windows

This experiment repeats the *iPerf* measurements with a smaller window size at *PC2*.

Step 1: Terminate the *iPerf* server on *PC2* with Ctrl-C. Then disable TCP window scaling on *PC2* with the command

```
PC2$ sudo sysctl -w net.ipv4.tcp_window_scaling=0
```

Then, restart the *iPerf* server.

Step 6: Check that the *Wireshark* session for *PC1* (*eth1*) is still active and that the *iPerf* server on *PC2* is still running. If not, restart the *Wireshark* traffic capture and start an *iPerf* server by typing

```
PC2$ iperf -s
```

Step 2: On *PC1*, start an *iPerf* client with the command

```
PC1$ iperf –c 10.0.5.22 -n 500K
```

Step 3: Create graphs for *time-sequence (tcptrace)* and window scaling for the TCP connection of the data transfer and take snapshots of the graphs.

Compare the graphs to those created in *Exercise 6-b*:

1. Does the *iPerf* client (on *PC1*) saturate the advertised window of the *iPerf* server (on *PC2*)?
2. How does the pattern of data segments and ACK change, as compared to the fast link in the previous exercise? Does the frequency of ACKs sent by *PC2* change?
3. How does the range of window sizes advertised by the receiver differ from the range seen in *Exercise 6-a*?
4. Does the TCP sender generally transmit the maximum amount of data allowed by the advertised window? Explain your answer.

Step 4: The maximum size of the advertised window can be reduced further by imposing a limit on the maximum window advertisement. This can be done by limiting the amount of allocated memory for the receive buffer for a TCP connection, which is controlled by the system parameter *net.ipv4.tcp_rmem*. First terminate the running *iPerf* server. Then view the current setting of the parameter on *PC2* with the command

```
PC2$ sudo sysctl net.ipv4.tcp_rmem
```

The parameter stores three values: (1) the minimum size (in bytes) of the receive buffer for each TCP connection, (2) the default size, and (3) the maximum size.

Step 5: On *PC2*, change the size of the receive buffer to 20 kB with the command

```
PC2$ sudo sysctl -w net.ipv4.tcp_rmem =“4096 20000 20000”
```

Since TCP advertises roughly one half of the available buffer space to the sender, this setting sets the maximum advertised window to around 10kB.

Step 6: Restart the *iPerf* server on *PC2* with

```
PC2$ iperf -s
```

and start another *iPerf* client on *PC1* with

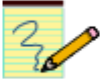
```
PC1$ iperf –c 10.0.5.22 -n 500K
```

Step 7: Create graphs for *time-sequence (tcptrace)* and *window scaling* for the TCP connection of the data transfer and take snapshots of the graphs.

Compare the graphs to the graphs in Step 3:

1. Determine the range of window sizes advertised by *PC2*. What is the maximum window size?
2. The *iPerf* client (on *PC1*) should now saturate the advertised window of the *iPerf* server (on *PC2*). Can you confirm this? (Hint: The packet list in *Wireshark* displays when the TCP sender saturates the window.)
3. Does the limit on the window size have an impact on the throughput or the completion time of the *iPerf* client. Note that, for each *iPerf* client, the *iPerf* server displays the throughput of once the transmission is completed. The completion time of transmissions can be obtained from the x-axis of the graphs.

Step 8: Terminate the *Wireshark* traffic capture. On *PC2*, terminate the *iPerf* server on *PC2* and reset *net.ipv4.tcp_rmem* to the original default values. If you do not recall the default values, run the command of Step 4 on another PC.



Lab Question/Report:

1. Include the graphs from Step 3 and use them to answer the questions of Step 3.
2. Include the graphs from Step 7 and use them to answer the questions of Step 7.

Exercise 6-d. TCP Data Transfer – Slow link

The last data transfer in this part considers data transmissions across a slow link. This is done by configuring the IP router on *PC3* bandwidth throttle.

Step 1: Start two traffic captures with *Wireshark* for the *eth0* interfaces of *PC1* and *PC2*.

Step 2: On *PC3*, impose a rate limit of 100 kbps on both interfaces with the commands

```
PC3$ sudo tc qdisc change dev eth0 root netem rate 100kbit
PC3$ sudo tc qdisc change dev eth1 root netem rate 100kbit
```

Note that the "change" command assumes that a *qdisc* has been previously configured on *PC3*. If this is not the case, use the "add" command instead.

Step 3: On *PC2*, ensure that window scaling is disabled.

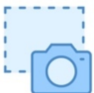
Step 4: Start an *iPerf* session that traverses *PC3*.

- On *PC2*, start a *iPerf* receiving process by typing

```
PC2$ iperf -s
```

- On *PC1*, start a *iPerf* sender with the command

```
PC1$ iperf -c 10.0.2.22 -n 500K
```



Step 5: Create graphs for *time-sequence (tcptrace)* for the TCP connection of the data transfer for both traffic captures at *PC1* (*eth0*) and *PC2* (*eth0*).

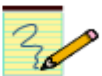
- First compare the graph from *PC1* (*eth0*) to the previously constructed graphs:

  1. How does the pattern of data segments and acknowledgements change, as compared to the previous data transfers?
  2. Is the range of window sizes advertised by the receiver different from what you have observed before?
  3. What would be the impact of changing the maximum window size at *PC2* (e.g., by enabling window scaling or by setting the advertised window to a small value as in *Exercise 6-c*) on the outcome of the experiment ?

- Next, compare the time-sequence graph for the traffic captures at *PC1* (*eth0*) and *PC2* (*eth0*). Observe that one graph shows the transmissions of segments hugging the upper bound (green line), while the other graph is very close to the lower bound (brown line).

    4. Explain the source of the differences of the graphs, and explain why this result should be expected.

Step 6: On *PC2*, terminate the *iPerf* server on *PC2*. Also, terminate the *Wireshark* traffic captures.



Lab Question/Report:

1. Include the graphs from Step 5 and use them to answer the questions of Step 5.

Part 7. Retransmissions in TCP

Next you observe retransmissions in TCP. TCP uses ACKs and timers to trigger retransmissions of lost segments. A TCP sender retransmits a segment when it assumes that the segment has been lost. This occurs in two situations:

1. *No ACK has been received for a segment*. Each TCP sender maintains a retransmission timer for the connection. When the timer expires, the TCP sender retransmits the earliest segment that has not been acknowledged. The timer is started when a segment with a payload is transmitted and the timer is not running, when an ACK arrives that acknowledges new data, and when a segment is retransmitted. The timer is stopped when all outstanding data has been acknowledged.

   The retransmission timer is set to a retransmission timeout (RTO) value, which adapts to the current network delays between the sender and the receiver. A TCP connection performs round-trip measurements by calculating the delay between the transmission of a segment and the receipt of the acknowledgement for that segment. The RTO value is calculated based on these round-trip measurements. Following a heuristic which is called *Karn's algorithm*, measurements are not taken for retransmitted segments. Instead, when a retransmission occurs, the current RTO value is simply doubled.

2. *Multiple ACKs have been received for the same segment*. A duplicate acknowledgment for a segment can be caused by an out-of-order delivery of a segment or by a lost packet. A TCP sender takes multiple, in most cases three, duplicates as an indication that a packet has been lost. In this case, the TCP sender does not wait until the timer expires, but immediately retransmits the segment that is presumed lost. This mechanism is known as *fast retransmit*. The TCP receiver expedites a fast retransmit by sending an ACK for each packet that is received out-of-order.

A disadvantage of cumulative acknowledgements in TCP is that a TCP receiver cannot request the retransmission of specific segments. For example, if the receiver has obtained segments 1, 2, 3, 5, 6, 7 cumulative acknowledgements only permit to send an ACK for segments 1, 2, 3 but not for the other correctly received segments. This may result in an unnecessary retransmission of segments 5, 6, and 7. The problem can be remedied with an optional feature of TCP, which is known as *selective acknowledgement (SACKs)*. Here, in addition to acknowledging the highest sequence number of contiguous data that has been received correctly, a receiver can acknowledge additional blocks of sequence numbers. The range of these blocks is included in TCP header options. Whether SACKs are used or not, is negotiated in TCP header options when the TCP connection is created.

The network topology for Part 7 is shown in Figure 6.3, which adds *PC4* to the network. *PC4* will be configured as an IP router and *PC2* will use this router as the default gateway.

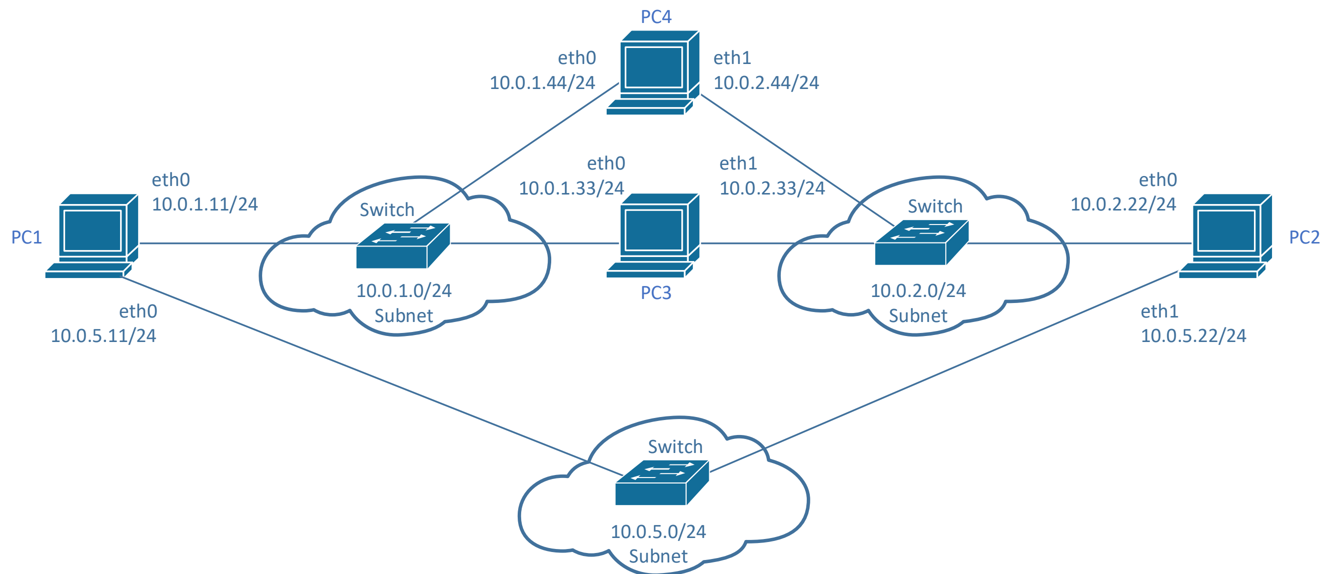


Figure 6.3. Network topology for Part 7.

Table 6.3. IPv4 Addresses of *PC4*.

Linux PC	Ethernet Interface *eth0*	Ethernet Interface *eth1*
PC4	10.0.1.44/24	10.0.2.44/24

Exercise 7-a. TCP Retransmissions

The purpose of this exercise is to observe when TCP retransmissions occur. As before, you transmit data from *PC1* to *PC2* via *PC3*. As in *Exercise 6-d*, the link rates at *PC3* are set to 100 kbps. The path from *PC2* to *PC1* will go through the newly added *PC4*.

During the data transfer, *PC4* will be disconnected so that ACKs cannot reach *PC1*. As a result, a timeout occurs and *PC1* performs retransmissions.

Step 1: Add *PC4* to the network topology as shown in Figure 6.3. Configure *PC4* IPv4 addresses as shown in Table 6.3. Set *PC4* up as an IPv4 router with

```
PC4$ sudo sysctl -w net.ipv4.ip_forward=1
```

Step 2: Check and modify the configurations of *PC1*, *PC2*, and *PC3*.

- On *PC2*, change the address of the default gateway to 10.0.2.44 by typing

```
PC2$ sudo ip route add default via 10.0.2.44
```

- Disable window scaling on *PC1* and *PC2*. For *PC1*, the command is

```
PC1$ sudo sysctl -w net.ipv4.tcp_window_scaling=0
```

- On *PC3*, make sure that as in *Exercise 6-d*, a rate limit of 100 kbps is running on both interfaces. You can check the status with the command
  ```
  PC3$ sudo tc qdisc show
  ```
  If necessary, configure the rate limit with the commands from *Exercise 6-d* (If a *Netem* is already configured, but with a different configuration, replace the “add” in the command with “change”).

Step 3: Start a *Wireshark* traffic capture for interface *eth0* of *PC1*. Set a display filter to TCP traffic (“tcp”).

Step 4: Start an *iPerf* receiving process on *PC2* with the command

```
PC2$ iperf -s
```

Step 5: Start an *iPerf* sender on *PC1* with

```
PC1$ iperf –c 10.0.2.22 -n 500K -b 1M
```

This command sends 500 kB at a target rate of 1Mbps to the *iPerf* server on *PC2*. With this, the experiment should last around 40s.

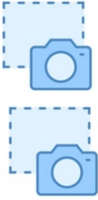
- Create a graphs *time-sequence (tcptrace)* graph or the TCP connection to get an overview of the transmissions. You should see that the rate of data transmissions (blue lines) is faster than the rate at which acknowledgements return (brown line). With window scaling enabled, you should see that the data transmissions do not fill up the advertised window (green line).

Step 6: Now, repeat Step 5. After about 10 seconds, disconnect the link from the *eth1* interface of *PC3* to the switch. Wait until the *iPerf* client on *PC1* terminates and shows a command prompt. This can take more than 10 minutes.

Observe TCP retransmissions from *PC1* in the output of *Wireshark*.

- For the first observed retransmission determine the elapsed time since the original transmission.
- Observe the time instants when retransmissions take place. How many packets are retransmitted at one time?
- Try to derive the algorithm that sets the time when a packet is retransmitted. (Repeat the experiment, if necessary). Is there a maximum time interval between retransmissions?

- After how many retransmissions, if at all, does the TCP sender give up with retransmitting the segment? Describe your observations.
- Create a graphs *time-sequence (tcptrace)* for this TCP connection. The graph shows the time between retransmission events.



- Take a snapshot of the *Wireshark* packet list pane, showing the retransmission events.
- Take a screen snapshot of the *time-sequence (tcptrace)* graph or save the graph to a file.

Locating retransmissions in *Wireshark*

- You can set a display filter to only show retransmissions. The filter is "tcp.analysis.retransmission". With "tcp.analysis.fast_retransmission" the display filter displays only fast retransmits.
- In the *Wireshark* window, selecting "Analyze→Expert Information", displays certain events that are observed by *Wireshark*, including retransmissions.

  Note: Since *Wireshark* only observes traffic it can only infer the reason for a retransmission. For example, when a repeated duplicate ACK is followed by a retransmission, *Wireshark* marks this event as a fast retransmit.

Step 7: Reconnect the *eth1* interface of *PC3*. Then, repeat Step 6, by disconnecting the link of the *eth1* interface of *PC3* to the switch after 10 seconds. After 60 seconds, reconnect the link. You observe that the *iPerf* client resumes. Wait until the transmissions are completed.

Observe the TCP retransmissions from *PC1* in the output of *Wireshark*.

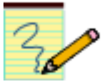
- Create a graphs *time-sequence (tcptrace)* for this TCP connection. Take a screen snapshot of the graph or save the graph to a file.
- Observe which packets are retransmitted when the link is disconnected, and which packets are retransmitted when the link is reconnected. Does *PC1* at any time retransmit multiple packets at a time or does it wait for an acknowledgement before retransmitting the next packet?
- Do you observe selective acknowledgements (indicated by red vertical lines)? If so, explain why selective acknowledgements occur.

Step 8: This is another repetition of the same experiment. Here, after 10 seconds, disconnect the link that connects the *eth0* interface of *PC4*. Reconnect the link after 60 seconds. Wait until the transmissions are completed.
Again, observe the TCP retransmissions from *PC1* in the output of *Wireshark*.

- Create a graphs *time-sequence (tcptrace)* for this TCP connection. Take a screen snapshot of the *time-sequence (tcptrace)* graph or save the graph to a file.
- Describe the events that are captured when the link is reconnected.
- Compare the retransmission events to those in Step 7. Explain the different outcomes.

Step 9: Terminate the *Wireshark* traffic captures and close the *time-sequence (tcptrace)* graphs.

Lab Question/Report:



1. Include the answers to the questions from Steps 6, 7 and 8. Include the graphs and other screen captures to *time-sequence (tcptrace)* your answers.
   Annotate the tcptrace graphs to emphasize your answers. In particular, indicate all retransmissions by arrows.

Exercise 7-b. TCP performance with sporadic packet losses

The next experiment explores retransmission events when an intermediate router sporadically drops packets. This is done by configuring a fixed loss probability at *PC3*. You also observe how TCP performance degrades when the loss probability is increased.

Step 1: Change the network emulator on both interfaces of *PC3* so that it drops packets with a probability of 5%. The commands for *PC3* are

```
PC3$ sudo tc qdisc change dev eth0 root netem rate 100kbit loss 5%
PC3$ sudo tc qdisc change dev eth1 root netem rate 100kbit loss 5%
```

Step 2: Start a *Wireshark* traffic capture for interface *eth0* of *PC2*. Set a display filter to TCP traffic ("tcp"). By capturing traffic at *PC2*, you can observe which packets are dropped by *PC3*.

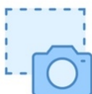
Step 3: Start an *iPerf* receiving process on *PC2* with the command

```
PC2$ iperf -s
```

Step 4: Start an *iPerf* sender on *PC1* with

```
PC1$ iperf -c 10.0.2.22 -n 500K
```

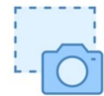
- Identify the retransmissions that are captured by *Wireshark* by setting a display filter to "tcp.analysis.retransmission". In the *Wireshark* window, select "Analyze→Expert Information",



and expand the retransmission events. Determine the type of retransmissions (fast retransmit, retransmission after an RTO timeout). Take a screenshot of the retransmission events for reference.

- Create *time-sequence (tcptrace)* graph or the TCP connection to get an overview of the transmissions. Take a screenshot of the entire graph.
- For each fast retransmission event determine (and record) how many duplicate acknowledgements (DUP ACKs) are observed before the retransmission occurs.
- Relate the observed duplicate acknowledgements with transmissions of selective acknowledgements (red vertical lines).

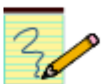
- Try to identify duplicate acknowledgments that acknowledge multiple ranges of sequence numbers. In the time-sequence graph, zoom into such an area and take another screen snapshot. (If you do not find multiple SACK ranges, find an interesting area with a single SACK range).



- Select one packet containing a selective acknowledgement (ideally, a packet that acknowledges multiple ranges of sequence numbers) check the details of the packet in *Wireshark*. The selective acknowledgement is in an option of the TCP header. Take a snapshot of the TCP header of this packet showing the details of the TCP header options.

Step 5: Terminate the *Wireshark* traffic captures and close the *time-sequence (tcptrace)* graphs. Also, terminate the *iPerf* server on *PC2*.

Lab Question/Report:



1. Describe the type of retransmissions observed in this experiment. (There is no need to include the screen shot of the “Expert Information” window.)
2. Include the *time-sequence (tcptrace)* graph of the complete data exchange from *PC1* to *PC2*. Annotate the screenshot to answer the following questions from Step 4:
   a. How many duplicate acknowledgements (DUP ACKs) are observed before a fast retransmit.
   b. Indicate in the graph where duplicate acknowledgements are sent with an selective acknowledgement.
3. Use the detailed (zoom in) screenshot of a retransmission event with one or more SACK ranges. Which event stopped the transmission of SACKs. Indicate this in the figure.
4. Include the snapshot of the TCP header containing a selective acknowledgement. Describe which data is acknowledged in the ACK field and the sequence number range of the selective acknowledgement(s).

Exercise 7-c. TCP retransmissions at an overloaded link

Next you observe loss events at an overloaded link. This is done by limiting the data rate of *PC3* and introducing an overload which results in losses and retransmissions.

In this exercise, *PC4* is used as an *iPerf* client that sends UDP traffic to *PC2* via *PC3*.

Step 1: On *PC3*, change the network emulation so that it has a rate limit of 100 kbps and a buffer size limit of 30 packets on each interface, using the commands

```
PC3$ sudo tc qdisc change dev eth0 root netem rate 100kbit limit 30
PC3$ sudo tc qdisc change dev eth1 root netem rate 100kbit limit 30
```

With the low rate and small buffer size, we expect to see a buffer overflow when *PC1* sends data to *PC2*.

Step 2: On *PC2*, change the default route to 10.0.2.33 with the command

```
PC2$ sudo ip route change default via 10.0.2.33
```

Then, start an *iPerf* server process with

```
PC2$ iperf -s
```

Step 3: Start a *Wireshark* traffic capture for interface *eth0* of *PC1* and *PC2*. Set a display filter to TCP traffic ("tcp").

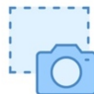
Step 4: Start an *iPerf* client on *PC1* with

```
PC1$ iperf –c 10.0.2.22 -n 1M
```

The number of packets is set large to ensure that an overflow occurs at *PC3*. When the data transfer is completed check the throughput that is reported by the *iPerf* server on *PC2*. It will be slightly below 100 kbps.

- Identify the retransmissions that are captured by *Wireshark*. Determine the type of retransmissions.

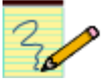
- Create *time-sequence (tcptrace)* graphs of the TCP connection for both traffic captures at *PC1* and *PC2*. Take screen snapshots of the entire graph.



- You will observe multiple time periods in the graphs where you see duplicate ACKS, SACKs, and retransmissions. These time periods follow a buffer overflow event. Zoom in on the first such time period (in both graphs) and take screen snapshots.
- Describe the differences observed of the occurrences of retransmissions if compared to *Exercise 7-b*.

Step 6: Terminate the *Wireshark* traffic captures and close the *time-sequence (tcptrace)* graphs. Also, terminate the *iPerf* server on *PC2*.

Lab Question/Report:



1. Describe the type of retransmissions observed in this experiment. Describe the differences observed of the occurrences of retransmissions if compared to *Exercise 7-b.*

2. Include the *time-sequence (tcptrace)* graphs of the complete data exchange from *PC1* to *PC2*. Use the graphs to confirm that the retransmissions follow a buffer overflow.

3. Provide the detailed (zoom in) screenshots of the time-*sequence (tcptrace)* from both *PC1* (*eth0*) and *PC2* (*eth0*). Annotate the graphs and describe the events that you observe in the graphs (e.g., buffer size limit is exceeded, retransmission, duplicate ACK, SACK, etc.)

Part 8. TCP Congestion Control

TCP congestion control adapts the sending rate of a TCP sender to the current conditions in the network. When the network is highly loaded (congested), the TCP sender reduces its rate. When the network is not congested, the TCP sender increases its sending rate. Each TCP sender maintains a *congestion window* that limits the number of segments that can be sent without waiting for an acknowledgement. The actual number of segments that can be sent is the smaller of the congestion window and the flow control window that is advertised by the receiver.

In standard TCP congestion control (TCP Reno or TCP NewReno), the TCP sender keeps two variables, a *congestion window* (*cwnd*) and a *slow-start threshold* (*ssthresh*). The initial value of *cwnd* is set to one or a few segments of maximal size (MSS), and that of *ssthresh* is set to the maximum window size. The congestion control algorithm operates in two phases, called *slow start* and *congestion avoidance*. The sender is in the slow start phase when *cwnd* ≤ *ssthresh*. Here, *cwnd* is increased by one segment for each received ACK that confirms receipt of previously unacknowledged data. This results in a doubling of *cwnd* for each roundtrip time. When *cwnd* > *ssthresh*, the TCP sender is in the congestion avoidance phase. Here, the *cwnd* is incremented by one segment only after *cwnd* ACKs of new data.

The TCP sender assumes that the network is congested when a segment is lost, where a segment loss is inferred when the retransmission timer has a timeout or when a third duplicate ACK has arrived. When a timeout occurs, the TCP sender sets *ssthresh* to half the current value of *cwnd* and then sets *cwnd* to one. This puts the TCP sender in slow start mode. When a third duplicate ACK arrives, the TCP sender performs what is called a *fast recovery*. Here, *ssthresh* is set to half the current value of *cwnd*, and *cwnd* is set to the new value of *ssthresh*. The full congestion control algorithm has additional tweaks, and details differ slightly from the above description.

Standard TCP congestion control is not ideal for all connections. If a TCP connection is short-lived, the congestion window remains small, hence limiting the rate of TCP data exchanges. Also, for fast networks with a large bandwidth-delay product, TCP congestion avoidance increases too slowly to exploit the available network bandwidth efficiently. Today there many alternative congestion control algorithms, which all seek to improve TCP performance for specific settings.

The earlier encountered *"ss -i"* Linux command displays the current values of *cwnd* and *ssthresh*. However, there is no convenient method to track these variables during a data transfer. Therefore, in can only infer the *cwnd* values from observing transmissions. Note, however, that the observed traffic results from both the congestion window and the advertised window.

Table 6.4. IPv4 addresses and default routes of *PC1* and *PC2*.

Linux PC	Ethernet Interface *eth0*	Default gateway
PC1	10.0.1.11/24	10.0.1.44
PC2	10.0.2.22/24	10.0.2.33

Exercise 8-a. Network setup and congestion parameters

For the experiments in this part the routing table entries are set so that traffic from *PC2* to *PC1* traverses the path *PC2*→*PC3*→*PC1*, and the reverse path is *PC1*→*PC4*→*PC2*.

Step 1: Continue with the network from Part 7.

Step 2: Change the routing tables of *PC1* and *PC2* so that the paths between the two PCs are *PC1*→*PC4*→*PC2* and *PC2*→*PC3*→*PC1*. This can be done by adding routing table entries for the subnets of the PCs with the commands

```
PC1$ sudo ip route add 10.0.2.0/24 via 10.0.1.44
PC2$ sudo ip route add 10.0.1.0/24 via 10.0.2.33
```

Use *traceroute* to verify that traffic from *PC1* to *PC2* passes through *PC4*, and traffic from *PC2* to *PC1* passes through *PC3*.

Step 3: **Selecting the congestion control algorithm**. Linux supports a large number of congestion control algorithms. The congestion control algorithm can be changed globally or locally for a specific routing table entry (except the default route).

- The default congestion control algorithm is kept in `net.ipv4.tcp_congestion_control.` Display the current congestion control at *PC1* with the command

```
PC1$ sysctl net.ipv4.tcp_congestion_control
```

Supported congestion control algorithms
Current Linux distributions support a long list of congestion control algorithms. Here is a small election of prominent algorithms:

- `reno` – TCP Reno/New Reno. This algorithm has been standardized by the IETF and was earlier referred to as Standard TCP congestion control.
- `cubic` – TCP Cubic is currently the default TCP congestion control algorithm in Linux systems.
- `bbr` – TCP BBR stands for bottleneck bandwidth and round-trip propagation time. It is a recent addition that currently enjoys considerable buzz.

The acronyms of other algorithms, some of which are specialized for certain network conditions are `bic, westwood, vegas, illinois, veno,` and `hybla.`
Note:

- Since the algorithm is configured on a local system, the two sides of a TCP congestion may run different congestion control algorithms.
- On Linux systems, TCP congestion control cannot be disabled.

- Change the congestion control algorithm at *PC1* to TCP Reno by typing

```
PC1$ sudo sysctl -w net.ipv4.tcp_congestion_control=reno
```

- The congestion control algorithms and some parameters of congestion control can be configured for specific routes. To change the algorithm to TCP Reno on *PC2* with an initial congestion window of one segment for the route to PC 1, run the command

```
PC1$ sudo ip route change 10.0.2.0/24 via 10.0.1.44 congctl reno initcwnd 1
```

If the route does not already exist, replace “change” by “add”. Then, verify the route-specific change by displaying the routing table with

```
PC1$ ip route show
```

Step 4: Viewing state variables of congestion control. The Linux command “*ss -i*” (see *Exercise 4-d*) displays state information of currently active TCP connections, which includes information about congestion control. Here, you will explore the TCP connection of a Telnet session.

- Start a Telnet server on *PC1* with the command

```
PC1$ sudo service xinetd start
```

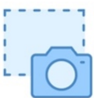
- On *PC2*, establish a set of Telnet session to *PC1* with the command

```
PC2$ telnet 10.0.1.11
```

Complete the login as *labuser*.

- After the login is completed, on *PC1*, type

```
PC1$ ss -i
```



Explore the output and parse it for information on the congestion control. Relevant fields are the congestion control algorithm, the congestion window (*cwnd*), and the slow start threshold (*ssthresh*). Take a screen snapshot of the output.

- Next, initiate a data transfer from *PC2* to *PC1*. This can be done by displaying a lot of data on the console. For example, the command

```
PC(via telnet)$  yes "Put some text here" | head -n 1000000 > tmp.txt
PC(via telnet)$  cat tmp.txt
```

creates a large file and then displays it. The prompt `*PC1(via telnet)$*' indicates that this is the *Telnet* client at *PC2* that has logged into *PC1*.

- On *PC1*, after the file listing is completed, repeat the socket statistic command

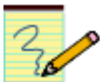
```
PC1$ ss -i
```

Check how the information on congestion control has changed. Take another snapshot.

Step 5: On *PC2*, terminate the Telnet session with the command

```
PC1(via telnet)$ exit
```

Lab Question/Report:



1. Include the snapshot from Step 4, and highlight the variables that relate to congestion control. Can you determine if the TCP connection is in slow start or in congestion avoidance?

Results may depend on data rates of the network
The outcomes of some of the following experiments depend on the data rate of the Ethernet interfaces of the PCs. If the Lab is run in a virtual environment, the data rate may be variable.

Exercise 8-b. Observing TCP congestion control

To observe the growth of the congestion window during data transmissions, *PC2* will send data to *PC1* via *PC3*. The ACKs from *PC1* to *PC2* travel on the path *PC1→PC3→PC2*, where we introduce a delay at *PC3*. When *PC2* sends data to *PC1*, data segments can be transmitted quickly to *PC1*, but ACKs only return to *PC2* with a considerable delay. The sender will therefore transmit a full congestion window worth of packets up to the threshold of the congestion window, and then be forced to wait for ACKs before transmitting the next batch of packets.

Step 1: On *PC3*, change the network emulation so that a delay of 100 ms is imposed on the traffic on both interfaces. The commands are

```
PC3$ sudo tc qdisc change dev eth0 root netem delay 100ms limit 1000
PC3$ sudo tc qdisc change dev eth1 root netem delay 100ms limit 1000
```

The commands also reset the buffer size to the default limit of 1000 packets. If *PC3* has been rebooted since Part 7, replace "change" with "add".

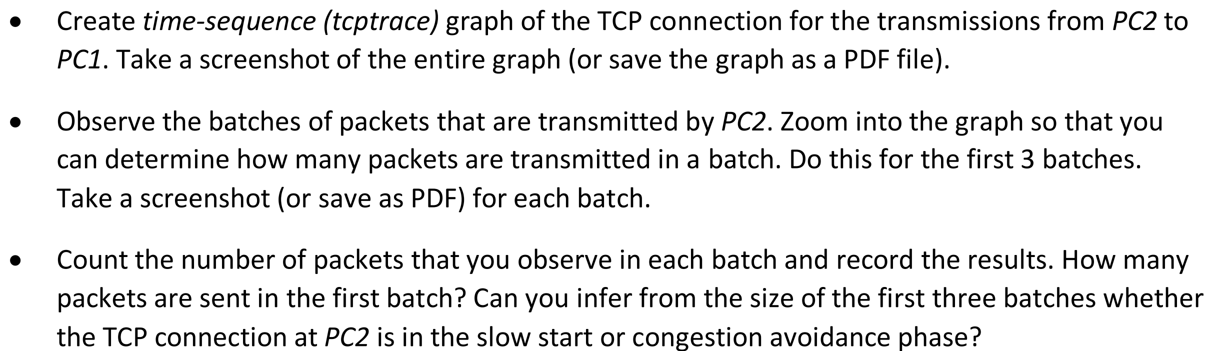
Step 2: Start a *Wireshark* traffic capture on interface *eth0* of *PC2*. Set a display filter to TCP traffic.

Step 3: Start an *iPerf* server on *PC1* with the command

```
PC1$ iperf -s
```

Step 4: Start an *iPerf* client process on *PC2* that transmits 1 MB of data at transmission rate of 10 Mbps with

```
PC2$ iperf –c 10.0.1.11 -n 1M -b 10Mbit
```



- Create *time-sequence (tcptrace)* graph of the TCP connection for the transmissions from *PC2* to *PC1*. Take a screenshot of the entire graph (or save the graph as a PDF file).
- Observe the batches of packets that are transmitted by *PC2*. Zoom into the graph so that you can determine how many packets are transmitted in a batch. Do this for the first 3 batches. Take a screenshot (or save as PDF) for each batch.
- Count the number of packets that you observe in each batch and record the results. How many packets are sent in the first batch? Can you infer from the size of the first three batches whether the TCP connection at *PC2* is in the slow start or congestion avoidance phase?

Displaying *cwnd* and *ssthresh*
Since the socket statistic command command displays the values of cwnd and ssthresh for active connections only, the command must be run while the *iPerf* client is still transmitting. The command can be issued in an additional console window. Alternatively, the command can be issued together with *iPerf* in one console window. For example,

```
PC1$ iperf –c 10.0.2.22 & ss -i
```

Runs the *iPerf* client as a background process and then starts *"ss-i"*. To prevent that the *"ss -i"* command completes before the data transfer is started, a delay can be added. For example, typing

```
PC1$ iperf –c 10.0.2.22 & sleep 0.2; ss -i
```

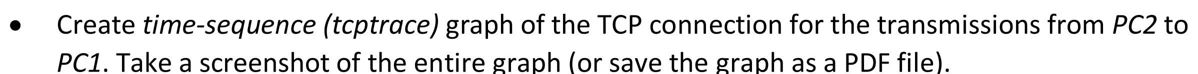
Delays the socket statistic command by 0.2 seconds.

Step 5: Next, change the size of the initial congestion window at *PC2* for destination 10.0.1.0/24 to one segment by typing

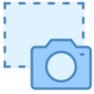
```
PC2$ sudo ip route change 10.0.1.0/24 via 10.0.2.33 initcwnd 1
```

Step 6: Then, repeat Step 4 and explore the tcptrace graphs. Specifically, start an *iPerf* client with

```
PC2$ iperf –c 10.0.1.11 -n 1M -b 10Mbit
```



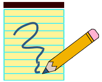
- Create *time-sequence (tcptrace)* graph of the TCP connection for the transmissions from *PC2* to *PC1*. Take a screenshot of the entire graph (or save the graph as a PDF file).



- Zoom into the graph to determine how many packets are transmitted in a batch. Do this for the first 3 batches. Take a screenshot (or save as PDF) for each batch.
- Count the number of packets that you observe in each batch and record the results. How many packets are sent in the first batch? Can you infer from the size of the first three batches whether the TCP connection at *PC2* is in the slow start or congestion avoidance phase?
- Compare the duration of the transmissions for the *iPerf* commands in Step 4 and Step 5.

Step 7: Terminate the *iPerf* server on *PC1*. (You may leave the *Wireshark* capture on PC (*eth0*) running.)

Lab Question/Report:



1. Include the screenshots from Steps 4 and 5, and discuss the graphs. Provide answers to the questions in both steps.

Exercise 8-c. Fairness of congestion control

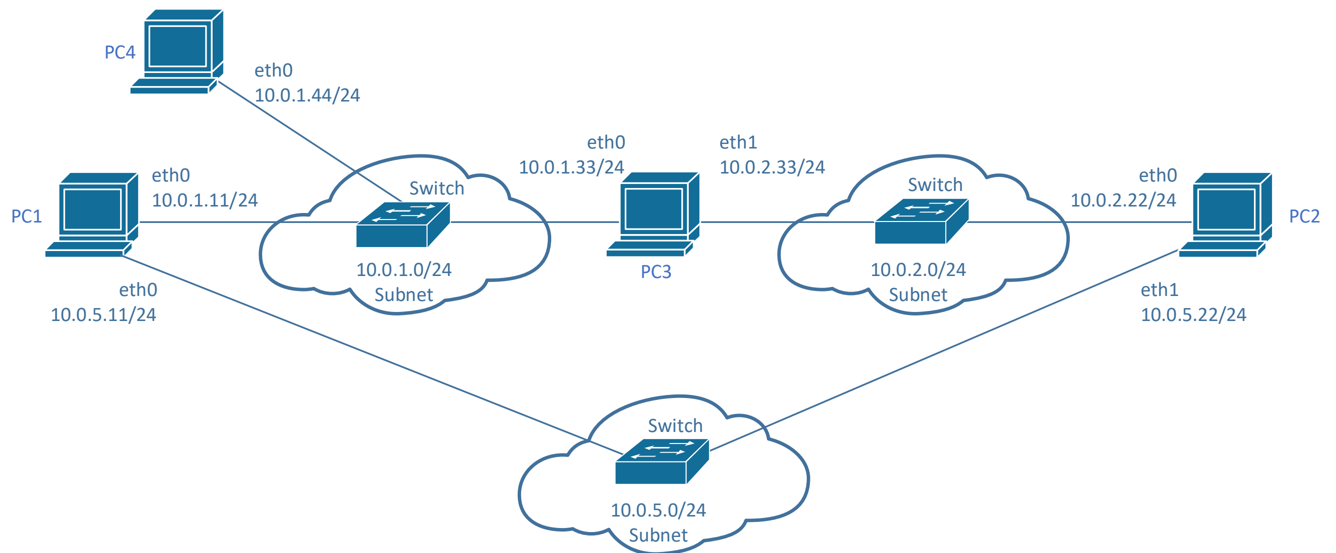


Figure 6.4. Network topology for Part 8.

A goal of congestion control is to ensure that all traffic flows sharing a bottleneck resource obtain the same transmission rate at that resource. Here, you test this having two *iPerf* clients running on *PC1* and *PC4* compete for resources at *PC3*.

Step 1: On *PC4*, disable interface *eth1* and configure *PC3* as the default gateway with the commands

```
PC4$ sudo ip link set dev eth0 down
PC4$ sudo ip route add default via 10.0.1.33
```

Step 2: On *PC1* and *PC2*, disable the routes that were added in *Exercise 8-a* with

```
PC1$ sudo ip route del 10.0.2.0/24 via 10.0.1.44
PC2$ sudo ip route del 10.0.1.0/24 via 10.0.2.33
```

Check the congestion control algorithms on *PC1* and *PC2*. Set the algorithm on both PCs to the default Cubic algorithm.

With this, all traffic between *PC1* and *PC2*, and between *PC4* and *PC2* goes through *PC3*.

Step 3: On *PC3*, impose a rate limit of 10 Mbps and a buffer limit of 100 packets on both interfaces of *PC3* by typing

```
PC3$ sudo tc qdisc change dev eth0 root netem rate 10Mbit limit 100
PC3$ sudo tc qdisc change dev eth1 root netem rate 10Mbit limit 100
```

Step 4: Check if the *Wireshark* traffic capture on interface *eth0* of *PC2* is still active. If necessary, start a new traffic capture.

Step 5: Start an *iPerf* server on *PC2* with

```
PC2$ iperf -s
```

Step 6: On *PC1*, run an *iPerf* client that transmits a large amount of data (10 MB) at a rate that saturates the link rates of *PC3* with the command

```
PC1$ iperf –c 10.0.2.22 -n 20M -b 10Mbit
```

Step 7: After about 5 seconds, start an *iPerf* client on *PC4* that also saturates *PC3* by typing

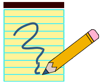
```
PC4$ iperf –c 10.0.2.22 -n 10M -b 10Mbit
```

When *PC4* starts transmitting, its traffic will compete with that from *PC1* for the 10 Mbps link at *PC3*. Ideally, the available link rate is split evenly. Your task is to determine how fairly the bandwidth is divided between *PC1* and *PC4*.

Step 8: Once the *iPerf* clients have completed, create *time-sequence (tcptrace)* graphs of the TCP connections of the *iPerf* clients. Create throughout graphs by selecting “Throughput” in the type window.

- Take screenshots of the tcptrace graphs and the throughput graphs (or save the graphs as PDF files). For the throughput graphs, unselect "Segment Length". Also, the averaging window ("MA window") should be set to 1 second.
- Use the graphs to explain how the bandwidth is split between the transmissions. What is the throughput that each TCP connection obtains when both connections are active.
- In the tcptrace graphs you can observe the retransmission events that occur in the experiment, which point to loss events at *PC3*.
- Take this into consideration when determining the throughput observed of the two TCP connections.

Lab Question/Report:



1. Include the screenshots from Steps 4 and 5, and discuss the graphs. Provide answers to the questions in both steps.

Exercise 8-d. TCP competing with UDP traffic

UDP does not perform congestion control. Therefore, if a TCP traffic flow competes with a UDP flow at a bottleneck resource, the TCP flow will reduce its traffic rate if a loss is experienced, while the UDP flow continues sending. This exercise creates such a scenario. It repeats the same transmission scenario as *Exercise 8-c*, with the difference that the *iPerf* client at *PC4* sends UDP traffic.

Step 1: Check that the network setup is the same as in *Exercise 8-c* (Steps 1–3). Make sure that there is a *Wireshark* traffic capture for the *eth0* interface of *PC2*.

Step 2: Check that the *iPerf* server is running on *PC2*. If necessary, start a new *iPerf* server with

```
PC2$ iperf -s
```

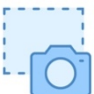
Step 3: On *PC1*, run an *iPerf* client that transmits a large amount of data (10 MB) at a rate that saturates the link rates of *PC3* with the command

```
PC1$ iperf –c 10.0.2.22 -n 20M -b 10Mbit
```

Step 4: After about 5 seconds, start an *iPerf* client on *PC4* that sends UDP traffic which saturates *PC3* by typing

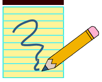
```
PC4$ iperf –c 10.0.2.22 -n 10M -b 10Mbit -u
```

Step 5: Once the *iPerf* clients have completed, create *Throughput* graph of the TCP connections of the *iPerf* client at *PC1*.



- Describe the throughput graph for the duration of the experiment. What happens to the traffic from the *iPerf* client on *PC1*, when the UDP traffic at *PC4* is activated.
- Take screenshots of the Throughput graphs where you unselect “Segment Length” and set the averaging window (“MA window”) to 1 second.

Lab Question/Report:



1. Include the screenshots from Steps 4 and 5, and discuss the graphs. Provide answers to the questions in both steps.