Based on
Mastering Networks - An Internet Lab Manual
by Jörg Liebeherr and Magda Al Zarki

Adapted for
'Labo Computernetwerken'
by Johan Bergs, Nicolas Letor, Michael Voorhaen and Kurt Smolderen

Completed by
Josse Coen Armin Halilovic Jonas Vanden Branden Group 2

April 22, 2016

Lab 5

Transport Layer Protocols: UDP and TCP

What you will learn in this lab:

- The differences between data transfers with UDP and with TCP
- What effect IP Fragmentation has 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

5.1 Prelab 5

TCP and UDP

Use the following resources to prepare yourself for this lab session:

1. TCP and UDP: Read the overview of TCP and UDP available at http://en.wikipedia.org/wiki/Transmission_Control_Protocol and http://en.wikipedia.org/wiki/User_Datagram_Protocol.
2. IP Fragmentation: Refer to the website http://www.tcpipguide.com/free/t_IPMessageFragmentationProcess.htm for information on IP Fragmentation and Path MTU Discovery.
3. TCP Retransmissions: Refer to RFC 2988, which is available at http://tools.ietf.org/html/rfc2988, and read about TCP retransmissions.
4. TCP Congestion Control: Refer RFC 2001, which is available at http://tools.ietf.org/html/rfc2001, and read about TCP congestion control.

Prelab Questions

Question 1)

Explain the role of port numbers in TCP and UDP.

The port numbers are used to identify communication channels using TCP/UDP. Applications can subscribe to those channels, and listen to the packets that arrive.

Question 2)

Provide the syntax of the `ttcp` 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.

Receiver: ttcp -rs -l500 -n4 -p2222 Sender: ttcp -ts -l500 -n4 -p2222 10.0.2.6

Question 3.a)

How does TCP decide the maximum size of a TCP segment?

This is stored in the Maximum Segment Size parameter (MSS) he default TCP MSS is 536 bytes. Where a host wishes to set the MSS to another value, MSS is specified as a TCP option, in the first or second packet of the TCP handshake, depending on which side wants to make the change.

The maximum transmission unit (MTU) at the host is also brought into consideration: TCP will choose its MSS such that the packet size won't exceed the host's MTU, in a measure to avoid fragmentation.

Question 3.b)

How does UDP decide the maximum size of a UDP datagram?

UDP doesn't have a maximum segment size. The maximum size is limited by the the maximum size an IP packet can be (65536 - 8 byte UDP header - 20 byte IP header).

Question 3.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?

"*ICMP Fragmentation needed and DF set*"

According to RFC1191:

To support the Path MTU Discovery technique specified in this memo, the router MUST include the MTU of that next-hop network in the low-order 16 bits of the ICMP header field that is labelled "unused" in the ICMP specification.

So, yes, the MTU of the interface that caused the fragmentation is also returned.

Question 3.d)

Explain why a TCP connection over an Ethernet segment never runs into problems with fragmentation.

Typically the MSS is announced by each side using the MSS option during the handshake, in which case it is derived from the maximum transmission unit (MTU) size of the data link layer (ETHERNET in this case) of the host.

For intermediary hops, TCP uses Path MTU Discovery: TCP always sets the Don't Fragment flag, causing an intermediary router to return an ICMP "Fragmentation needed" error message. This message includes the MTU needed for a packet to get through, and TCP will adjust its MSS accordingly.

Question 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?

It means that the receiver has received and ACKed the data up to 34567. According to the window size, up to seqnr. 36615 (= 34567 + 2048) may be sent upon receiving this ACK.

Question 5.a)

Describe Nagle's algorithm and explain why it is used in TCP

Nagle's algorithm reduces the amount of packets sent over a network, especially when there are many small packets to send. This reduces the overhead on the network.
It waits with sending of packets on the outgoing buffer until an ACK is received OR when the MSS is reached with all the data in the buffer.

Question 5.b)

Describe Karn's Algorithm and explain why it is used in TCP

Karn's algorithm is used to get more accurate RTT values when using TCP. Retransmitted segments are ignored when calculating the RTT, only using ACKs for packets that were succesfully delivered without a retransmission.
However this may result in failure as the RTT will never be updated when all packets sent are retransmissions, possibly resulting in endless retransmissions.

Question 6.a)

What is a delayed acknowledgement in TCP?

Several ACK responses may be combined together into a single response, reducing protocol overhead.

Question 6.b)

What is a piggybacked acknowledgement in TCP?

An acknowledgement sent together with another datapacket, as to reduce overhead and optimizing resource usage.

Question 7)

Describe how the retransmission timeout (RTO) value is determined in TCP.

According to RFC 6298:
- Until a round-trip time (RTT) measurement has been made for a segment sent between the sender and receiver, the sender SHOULD set RTO <- 1 second.
- When the first RTT measurement R is made, the host MUST set:
— SRTT <- R
— RTTVAR <- R/2
— RTO <- SRTT + max (G, 4*RTTVAR)
- When a subsequent RTT measurement R' is made, a host MUST set:
— RTTVAR <- (0.75) * RTTVAR + 0.25 * |SRTT - R'|
— SRTT <- (0.875) * SRTT + 0.125 * R'
— RTO <- SRTT + max (G, 4*RTTVAR)
- When RTO < 1 second => RTO is set to 1 second.
With G the clock granularity, defaulted on 6 seconds. Finer granularities (in the range of dozens of milliseconds) appear to perform better, at the cost of more overhead.

Question 8.a)

Describe the sliding window flow control mechanism used in TCP .

Both sender and receiver have a sequence number and a window size.
Their sequence numbers increase for each byte received and acknowledged by the other side.
The receiver can send data up to its current sequence number + its window size.

Question 8.b)

Describe the concepts of slow start and congestion avoidance in TCP.

Congestion avoidance tries to avoid having the medium congested by all of its data, taking other streams into account as well (avoiding TCP global synchronisation).

Slow start starts with sending a few packets, defined by the congestion window: the amount of packets it may send before having to stop to avoid congestion. This window is increased by 1 for each ACK received, resulting in an exponential growth.
When loss is detected, half of the current window size is stored as threshold value and slow start is restarted from the initial value. When the previous threshold value is reached, linear growth is used, increasing the window by 1 / (congestion window) for each ACK received.
When the receivers' advertised window size is reached, the congestion window stops growing.

Question 8.c)

Explain the concept of fast retransmit and fast recovery in TCP.
Fast Retransmit reduces the time a sender waits before retransmitting a lost segment. When the ACK of the previously sent packet is received a certain number of times, we can assume that the subsequent packet was lost and retransmit it, as opposed to waiting for the timout timer to indicate the loss of a packet.
This increases the performance and works when there is a continuous flow of packets.

5.2 Lab 5

This lab explores the operation of the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP), the two 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.

The lab covers the main features of UDP and TCP. Parts 1 and 2 compare 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.

This lab has two different network topologies. The topology for Parts 1-4 is shown in Figure 5.1. In this configuration, PC1 and PC2 are used as hosts, and PC3 is set up as an IP router. The network configuration for Parts 5-8 is shown in Figure 5.2. Here, the four Cisco routers interconnected via Ethernet, where one of the links is configured to emulate a slow serial link, as show in Figure 5.2.

Part 1. Learning how to use nttcp

The `nttcp` command is a Linux tool used to generate synthetic UDP and TCP traffic loads. Together with `ping` and `traceroute`, `nttcp` is an essential utility program for debugging problems in IP networks. Running `nttcp` tool consists of setting up a nttcp receiver on one hosts and then a nttcp sender on another host. Once the nttcp sender is started, it blasts the specified amount of data as fast as possible to the nttcp receiver.

Some useful `nttcp` commands:

`nttcp -u -i -rs -lbuflen -nnumbufs -pport`
: *Start a nttcp receiver process*

`nttcp -u -ts -lbuflen -nnumbufs -pport -D IPaddress`
: *Start a nttcp sender process*

List of important `nttcp` options:

`-t`
: *Specifies the transmit mode.*

`-r`
: *Specifies the receive mode*

`-u`
: *Specifies to use UDP instead of TCP. By default, nttcp uses TCP to send data. Make sure this is the first option*

`-s`
: *Sends a character string as payload of the transmitted packets. Without the -s option, the default is to transmit data from the terminal window (stdin) of the sender and to print the received data to the terminal window (stdout) at the receiver*

`-nnumbufs`
: *Number of blocks of application data to be transmitted (Default value is 2048)*

`-lbuflen`
: *Length of the application data blocks that are passed to UDP or TCP in bytes (default 8192). When UDP is used, this value is the number of data bytes in UDP datagram*

`-D`
: *Disables buffering of data in TCP and forces immediate transmission of the data at the nttcp sender. Only used in the context of TCP*

`-pport`
: *Port number to send to or listen on. The port number must be identical at the sender and at the receiver. The default value is 5000*

`IPaddress`
: *IP address of the nttcp receiver*

By default, nttcp transmits data over a TCP connection. The nttcp sender opens a TCP connection to a nttcp receiver, transmits data and then closes the connection. The nttcp receiver must be running when the nttcp sender is started. UDP data transfer is specified with the `-u` option. Since UDP is a connectionless protocol, the nttcp sender starts immediately sending UDP datagrams, regardless if there is a nttcp receiver established or not.

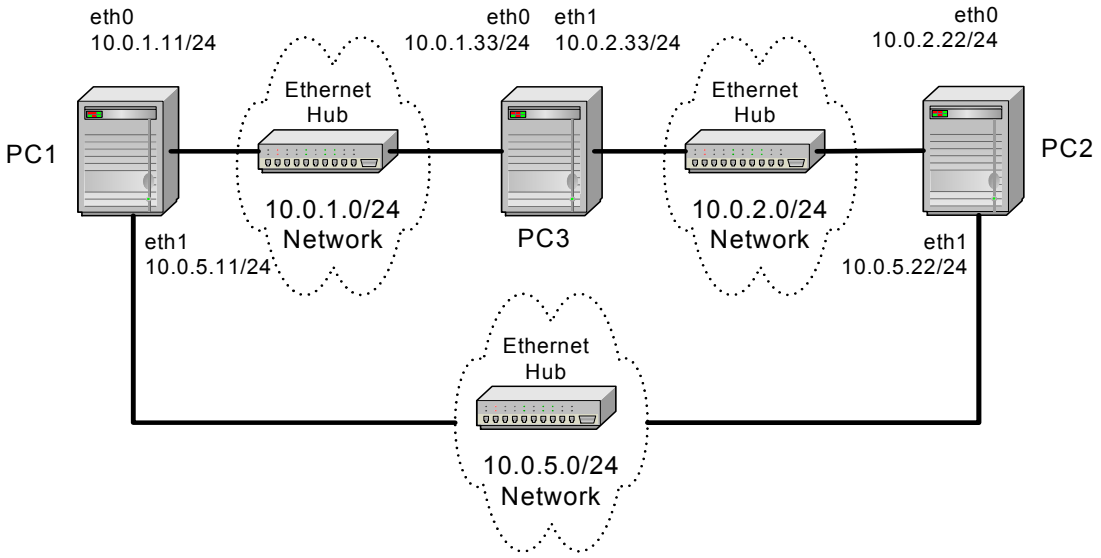


Figure 5.1: Network Topology for Parts 1-4.

Linux PC	Ethernet Interface eth0	Ethernet Interface eth1
PC1	10.0.1.11/24	10.0.5.11/24
PC2	10.0.2.22/24	10.0.5.22/24
PC3	10.0.1.33/24	10.0.2.33/24

Table 5.1: IP Addresses of the Linux PCs.

Exercise 1-a. Network setup

1. Connect the Ethernet interfaces of the Linux PCs as shown in Figure 5.1. Configure the IP addresses of the interfaces as given in Table 5.1.

2. PC1 and PC2 are set up as hosts and IP forwarding should be disabled. On PC1, this is done with the command

```
PC1% echo "0" > /proc/sys/net/ipv4/ip_forward
```

3. PC3 is set up as an IP router. Enable IP forwarding on PC3 with the command

```
PC3% echo "1" > /proc/sys/net/ipv4/ip_forward
```

4. Add default routes to the routing tables of PC1 and PC2, so that PC3 is the default gateway. For PC1 the command is as follows:

```
PC1% route add default gw 10.0.1.33
```

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

```
PC1% ping 10.0.2.22
PC1% ping 10.0.5.22
```

Exercise 1-b. Transmitting data with UDP

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

1. On PC1, start Wireshark to capture packets on interface *eth1* between PC1 to PC2.

   ```
   PC1% wireshark -i eth1 -f 'host 10.0.5.22'
   ```

   This sets a capture filter to packets that include IP address 10.0.5.22 in the IP header. Start to capture traffic.

2. On PC2, start a nttcp receiver that receives UDP traffic with the following command:

   ```
   PC2% nttcp -u -i -rs -l1024 -n10 -p4444
   ```

3. On PC1, start a nttcp sender that transmits UDP traffic by typing:

   ```
   PC1% nttcp -u -ts -l1024 -n10 -p4444 10.0.5.22
   ```

   Observe the captured traffic captured by Wireshark.

4. Stop the Wireshark capture on PC1 and save the captured traffic.

Use the data captured with Wireshark to answer the questions in Step 3. Support your answers with the saved Wireshark data.

Question 1.B.1.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?
We can count 14 UDP packets in the data transfer. (We only consider UDP packets, since only they are part of the actual file transfer.) Only one packet per UDP datagram was transmitted; there was no fragmentation. We see three different UDP datagrams, their difference lies in the payload:

Amount of packets	Payload length
1	4
10	1024
3	4

Question 1.B.1.b)
Compare the total number of bytes transmitted, in both directions, including Ethernet, IP, and UDP headers, to the amount of application data transmitted.
We count 10676 bytes transmitted in total, 10256 bytes of UDP payload data. That means there was approximately 5% of overhead data.

Question 1.B.1.c)
Inspect the fields in the UDP headers. Which fields in the headers do not change in different packets?
The source port and destination port fields never change in different packets.

Question 1.B.1.d)

Observe the port numbers in the UDP header. How did the nttcp sender select the source port number?
source port: 40951, destination port: 5038

We notice that port 4444 is not in these packets. That port is used by the TCP packets sent by nttcp to set up the connection. For the transmitted UDP packets, we believe that a random source port is selected randomly from a range of ports that are not assinged to a particular application layer protocol. In fact, the port that was used, 40951, is indeed unassigned according to this web page: http://whatismyipaddress.com/port-list

Exercise 1-c. Transmitting data with TCP

Here, you repeat the previous exercise, but use TCP for data transfer.

1. On PC1, start Wireshark and capture packets on interface *eth1* between PC1 to PC2:

```
PC1% wireshark -i eth1 -f 'host 10.0.5.22'
```

2. Start a nttcp receiver on PC2 that receives packets sent by PC1:

```
PC2% nttcp -i -rs -l1024 -n10 -p4444
```

3. Start a nttcp sender on PC1o that transmits packets from PC1 to PC2:

```
PC1% nttcp -ts -l1024 -n10 -p4444 -D 10.0.5.22
```

Use the data captured with Wireshark to answer the questions in Step 3. Support your answers by including data from the saved Wireshark data captures.

Question 1.C.1.a)

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

We count 16 packets in the data transfer.

packet numbers	packet length
10, 15 - 27	66
8, 9	74
11	1090
12	1514
13	2962
14	4938

Question 1.C.1.b)

What is the range of the sequence numbers?
Lowest sequence number: 0
Highest sequence number: 10242 (packet no. 27)

Question 1.C.1.c)
How many packets are transmitted by PC1 and how many packets are transmitted by PC2?

Packets transmitted by PC1: 7
Packets transmitted by PC2: 9

Question 1.C.1.d)
How many packets do not carry a payload, that is, how many packets are control packets?

12 of the packets are control packets.
Their packet numbers are: 8, 9, 10, 15 - 21, 26, 27

Question 1.C.1.e)
Compare the total number of bytes transmitted, in both directions, including Ethernet, IP, and UDP headers, to the amount of application data transmitted.
From PC1 to PC2: 7 packets, total packet length: 10710 bytes, total application data length: 10240 bytes

From PC2 to PC1: 9 packets, total packet length: 602 bytes, total application data length: 0 bytes

We see that only PC1 has sent application data, and more specifically has sent 10 * 1024 bytes of data (flags -l1024 -n10 in nttcp).

Question 1.C.1.f)
Inspect the TCP headers. Which packets contain flags in the TCP header? Which types of flags do you observe?

packet numbers	flag
8, 9	SYN
9 - 21, 26, 27	ACK
11, 14	PSH
14, 26	FIN

4. Stop the wireshark capture on PC1, and save the captured traffic to files. Save both the summary and detail output in the Print menu.

Question 1.C.2)
Compare the amount of data transmitted in the TCP and the UDP data transfers.
Total:
UDP transfer: 10676 B; TCP transfer: 11312 B
As is expected, the total amount of data transferred over TCP is greater than the amount of data transferred via UDP, since the TCP header is larger and the UDP received does not send any acknowledgements. Thus, using TCP, there was an increase of approximately 6%.
Payload data:
The TCP transfer had exactly the amount of payload data we'd expect: 10 * 1024 B = 10240 B. The UDP transfer had 16 B more, spread over four packets each carrying 4 B of payload data. We don't know why that's the case.

Question 1.C.3)

Take the biggest UDP datagram and the biggest TCP segment that you observed, and compare the amount of application data that is transmitted in the UDP datagram and the TCP segment.

The biggest UDP datagram had a length of 1066 bytes and a payload of 1024 bytes. The biggest TCP segment had a length of 4938 bytes and payload of 4872 bytes.

We see that the TCP relatively holds much more application data here.

Part 2. File Transfers using TCP and UDP

Here you compare the throughput of a file transfer with TCP and UDP, using the application programs `ftp` and `tftp`.

The File Transfer Protocol (FTP) for copying files between hosts, as described in the Introduction, employs TCP as its transport protocol, thereby ensuring a reliable transfer of transmitted data. Two TCP connections are established for each FTP session: A control connection for exchanging commands, and a data connection for the file transfer.

The Trivial File Transfer Protocol (TFTP) is a minimal protocol for transferring files without authentication. TFTP employs UDP for data transport. A TFTP session is initiated when a TFTP client sends a request to upload or download a file to UDP port 69 of a TFTP server. When the request is received, the TFTP server picks a free UDP port and uses this port to communicate with the TFTP client. Since UDP does not recover lost or corrupted data, TFTP is responsible for maintaining the integrity of the data exchange. TFTP transfers data in blocks of 512 bytes. A block must be acknowledged before the next block can be sent. When an acknowledgment is not received before a timer expires, the block is retransmitted.

The purpose of the following exercises is to observe that, despite the overhead of maintaining a TCP connection, file transfers with FTP generally outperform those with UDP.

Exercise 2. Comparison of FTP and TFTP

Study the performance of FTP and TFTP file transfers for a large file.

1. Create a large file. On PC1, create a file with name `large.d` in directory `/tftpboot` by using the `dd` tool. Create the file and change its access permissions as follows:

   ```
   PC1% dd if=/dev/urandom of=/tftpboot/large.d bs=1024 count=1024
   PC1% chmod 644 /tftpboot/large.d
   ```

   Use the command `ls -l` to check the length of the file.

2. Start Wireshark: Invoke Wireshark on interface *eth1* of PC1 with a capture filter set for PC2, and start to capture traffic:

   ```
   PC1% wireshark -i eth1 -f âĂIJhost 10.0.5.22âĂİ
   ```

3. FTP file transfer: On PC2, perform the following steps:

   - Change the current directory to `/labdata`.
   - Start the FTP server on PC1 by typing

     ```
     PC1% service vsftpd start
     ```

   - Invoke an FTP session to PC1 by typing

     ```
     PC2% ftp 10.0.5.11
     ```

     Log in as the root user.

   - Transfer the file large.d from PC1 to directory `/labdata` on PC2 by typing

     ```
     ftp> cd /tftpboot
     ftp> get large.d
     ftp> quit
     ```

4. TFTP file transfer: On PC2 perform the following tasks:
   - Start the TFTP server on PC1 by typing
     ```
     PC1% service tftpd-hpa start
     ```
   - Start a TFTP session to PC1 by typing
     ```
     PC2% tftp 10.0.5.11
     ```
   - Transfer the file `large.d` from PC1 to PC2.
     ```
     tftp> get large.d
     tftp> quit
     ```
     By default, TFTP copies data from the directory `/tftpboot`.
   - Observe the output of the TFTP session and save the output to a file.
5. Analysis of outcome: On PC1, stop the Wireshark output.

Include the answers to the questions in Step 5.

Question 2.A.a)

From the timestamps recorded by wireshark, obtain the times it took to transfer the file with FTP and with TFTP? Use your knowledge of FTP, TFTP, TCP, and UDP to explain the outcome.

From "/Lab 5/traces/2.A.pcap":

(protocol: packet number -> packet number (time -> time) => time difference)

FTP: 28 -> 789 (22.95 -> 24.13) => 1.18 seconds

TFTP: 824 -> 4952 (406.71 -> 408.37) => 1.66 seconds
TFTP: 4955 -> 9083 (415.80 -> 417.44) => 1.64 seconds

We see that, indeed, FTP outperforms TFTP. The size of the data in FTP data packets ranges from 1448 bytes to 7240 bytes. In the TFTP packets this is 512 bytes (and 350 bytes for the last packet).

Since TCP operates a sliding window for acknowledgements, the FTP sender doesn't have to wait for every packet's ACK before sending the next packet. Only when the delay of ACKs is large enough (determined by the sliding window parameters) the sender will start retransmitting packets.
With TFTP, the sender *does* have to wait for the ACK of every packet, so, every time a packet is sent after the receiver has received the packet, no data is sent while the ACK is traveling back to the sender.

Question 2.A.b)

Identify the TCP connections that are created in the FTP session, and record the port numbers at the source and at the destination.

Source ports used: 21 for ftp settings, 20 for sending ftp data
Destination ports used: 60889 for ftp settings, 42752 for sending ftp data

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 IP datagram exceeds the MTU size, the IP datagram is fragmented into multiple IP datagrams, or, if the DonâĂŹt fragment (DF) flag is set in the IP header, the IP 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. 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. TCP tries to avoid fragmentation with an Path MTU Discovery scheme that determines a Maximum Segment Size (MSS) which does not result in fragmentation.

You explore the issues with IP fragmentation of TCP and UDP transmissions in the network configuration shown in Figure 5.1, with PC1 as sending host, PC2 as receiving host, and PC3 as intermediate IP router.

Exercise 3-a. UDP and Fragmentation

In this exercise you observe IP fragmentation of UDP traffic. In the following exercise, use `nttcp` to generate UDP traffic between PC1 and PC2, across IP router PC3, and gradually increase the size of UDP datagrams until fragmentation occurs. You can observe that IP headers do not set the DF bit for UDP payloads.

1. Verify that the network is configured as shown in Figure 5.1 and Table 5.1. The PCs should be configured as described in Exercise 1-a.
2. Start Wireshark on the *eth0* interfaces of both PC1 and PC2, and start to capture traffic. Do not set any filters.
3. Use `nttcp` 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% nttcp -u -i -rs -l1024 -n12 -p4444
     ```
   - On PC1, execute the command:
     ```
     PC1% nttcp -u -ts -l1024 -n12 -p4444 10.0.2.22
     ```
4. Increment the size of the UDP datagrams, by increasing the argument given with the `-l` option.
   - Determine the exact UDP datagram size at which fragmentation occurs.
   - Determine the maximum size of the UDP datagram that the system can send and receive, regardless of fragmentation, i.e., fragmentation of data segments occurs until a point beyond which the segment size is too large for to be handled by IP.
5. Stop the traffic capture on PC1 and PC2, and save the Wireshark output.

Question 3.A.1)

From the saved Wireshark data, select one IP datagram that is fragmented. Include the complete datagram before fragmentation and include all fragments after fragmentation. For each fragment of this datagram, determine the values of the fields in the IP header that are used for fragmentation (Identification, Fragment Offset, DonâĂŹt Fragment Bit, More Fragments Bit).

Taken from “Lab 5/traces/3.A.PC1.pcap”:

As the packets are fragmented at the host, there is no “before” and “after” fragmentation.

Here’s an example of a datagram that was fragmented into 2 fragments.

```
No.     Time        Source                Destination           Protocol Length
    Info
     72 8.015696    10.0.1.11             10.0.2.22             UDP      35
         Source port: 39068  Destination port: 5038

Frame 72: 35 bytes on wire (280 bits), 35 bytes captured (280 bits)
    Encapsulation type: Ethernet (1)
    Arrival Time: Mar 15, 2016 11:35:52.642696000 CET
    [Time shift for this packet: 0.000000000 seconds]
    Epoch Time: 1458038152.642696000 seconds
    [Time delta from previous captured frame: 0.000002000 seconds]
    [Time delta from previous displayed frame: 0.000002000 seconds]
    [Time since reference or first frame: 8.015696000 seconds]
    Frame Number: 72
    Frame Length: 35 bytes (280 bits)
    Capture Length: 35 bytes (280 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:ip:udp:data]
    [Coloring Rule Name: UDP]
    [Coloring Rule String: udp]
Ethernet II, Src: IntelCor_1a:7c:70 (68:05:ca:1a:7c:70), Dst: IntelCor_1a:80:18
    (68:05:ca:1a:80:18)
    Destination: IntelCor_1a:80:18 (68:05:ca:1a:80:18)
        Address: IntelCor_1a:80:18 (68:05:ca:1a:80:18)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
            default)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Source: IntelCor_1a:7c:70 (68:05:ca:1a:7c:70)
        Address: IntelCor_1a:7c:70 (68:05:ca:1a:7c:70)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
            default)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.0.1.11 (10.0.1.11), Dst: 10.0.2.22 (10.0.2.22)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (
        Not ECN-Capable Transport))
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..00 = Explicit Congestion Notification: Not-ECT (Not ECN-Capable
            Transport) (0x00)
    Total Length: 21
    Identification: 0x07ef (2031)
    Flags: 0x00
        0... .... = Reserved bit: Not set
        .0.. .... = Don't fragment: Not set
        ..0. .... = More fragments: Not set
    Fragment offset: 1480
    Time to live: 64
    Protocol: UDP (17)
    Header checksum: 0x5b10 [validation disabled]
        [Good: False]
        [Bad: False]
    Source: 10.0.1.11 (10.0.1.11)
```

```
49      Destination: 10.0.2.22 (10.0.2.22)
        [Source GeoIP: Unknown]
51      [Destination GeoIP: Unknown]
        [2 IPv4 Fragments (1481 bytes): #71(1480), #72(1)]
53          [Frame: 71, payload: 0-1479 (1480 bytes)]
            [Frame: 72, payload: 1480-1480 (1 byte)]
55          [Fragment count: 2]
            [Reassembled IPv4 length: 1481]
57          [Reassembled IPv4 data: 989c13ae05c9462e1472baa5935b4c7b3df37df0ae953907
                ...]
    User Datagram Protocol, Src Port: 39068 (39068), Dst Port: 5038 (5038)
59      Source port: 39068 (39068)
        Destination port: 5038 (5038)
61      Length: 1481
        Checksum: 0x462e [validation disabled]
63          [Good Checksum: False]
            [Bad Checksum: False]
65  Data (1473 bytes)

67  0000  14 72 ba a5 93 5b 4c 7b 3d f3 7d f0 ae 95 39 07   .r...[L{=.}...9.
    (...) [90 lines]
69  05b0  d9 99 f5 95 1c 59 9c 55 7d 2e 4e 87 2f 46 08 a8   .....Y.U}.N./F..
    05c0  e3                                                .
71      Data: 1472baa5935b4c7b3df37df0ae953907c78076113b82f6e7...
        [Length: 1473]
```

```
    No.     Time        Source                Destination           Protocol Length
        Info
2       71 8.015694     10.0.1.11             10.0.2.22             IPv4     1514
            Fragmented IP protocol (proto=UDP 17, off=0, ID=07ef) [Reassembled in #72]

4   Frame 71: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits)
        Encapsulation type: Ethernet (1)
6       Arrival Time: Mar 15, 2016 11:35:52.642694000 CET
        [Time shift for this packet: 0.000000000 seconds]
8       Epoch Time: 1458038152.642694000 seconds
        [Time delta from previous captured frame: 0.000016000 seconds]
10      [Time delta from previous displayed frame: 0.000016000 seconds]
        [Time since reference or first frame: 8.015694000 seconds]
12      Frame Number: 71
        Frame Length: 1514 bytes (12112 bits)
14      Capture Length: 1514 bytes (12112 bits)
        [Frame is marked: False]
16      [Frame is ignored: False]
        [Protocols in frame: eth:ip:data]
18  Ethernet II, Src: IntelCor_1a:7c:70 (68:05:ca:1a:7c:70), Dst: IntelCor_1a:80:18
        (68:05:ca:1a:80:18)
        Destination: IntelCor_1a:80:18 (68:05:ca:1a:80:18)
20          Address: IntelCor_1a:80:18 (68:05:ca:1a:80:18)
            .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
                default)
22          .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        Source: IntelCor_1a:7c:70 (68:05:ca:1a:7c:70)
24          Address: IntelCor_1a:7c:70 (68:05:ca:1a:7c:70)
            .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
                default)
26          .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        Type: IP (0x0800)
28  Internet Protocol Version 4, Src: 10.0.1.11 (10.0.1.11), Dst: 10.0.2.22 (10.0.2.22)
        Version: 4
30      Header length: 20 bytes
        Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (
            Not ECN-Capable Transport))
32          0000 00.. = Differentiated Services Codepoint: Default (0x00)
```

```
        .... ..00 = Explicit Congestion Notification: Not-ECT (Not ECN-Capable
            Transport) (0x00)
    Total Length: 1500
    Identification: 0x07ef (2031)
    Flags: 0x01 (More Fragments)
        0... .... = Reserved bit: Not set
        .0.. .... = Don't fragment: Not set
        ..1. .... = More fragments: Set
    Fragment offset: 0
    Time to live: 64
    Protocol: UDP (17)
    Header checksum: 0x3602 [validation disabled]
        [Good: False]
        [Bad: False]
    Source: 10.0.1.11 (10.0.1.11)
    Destination: 10.0.2.22 (10.0.2.22)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
    Reassembled IPv4 in frame: 72
Data (1480 bytes)

0000  98 9c 13 ae 05 c9 46 2e 14 72 ba a5 93 5b 4c 7b   ......F..r...[L{
(...) [90 lines]
05b0  8c b0 c2 6f 3f 06 ef c8 d9 99 f5 95 1c 59 9c 55   ...o?........Y.U
05c0  7d 2e 4e 87 2f 46 08 a8                           }.N./F..
    Data: 989c13ae05c9462e1472baa5935b4c7b3df37df0ae953907...
    [Length: 1480]
```

Determine the values of the fields in the IP header that are used for fragmentation:

Packet 71 (IPv4):
Identification: 2031
Don't Fragment Bit: Not set
More Fragments Bit: Set
Fragment Offset: 0

Packet 72 (UDP):
Identification: 2031
Don't Fragment Bit: Not set
More Fragments Bit: Not set
Fragment Offset: 1480

In packet 71, the More Fragments bit is set to show that this packet belongs to a fragment of a datagram. The Identification is set to be the same as the UDP packet 72 to indicate that packet 71 is a fragment for packet 72. The fragment offset value is used to reassemble the data in the correct order.

Question 3.A.2)

Include the outcome of the experiment in Step 4. Indicate the UDP datagram size at which fragmentation occurs. Also, determine the maximum size of the UDP datagram that the system can send.

At -l1473, fragmentation starts, and continues to happen until -l65507.
After -l65508, there is no fragmentation, as the packet is not sent at all anymore.
This happens because the limit for the data length for IPv4 is 65507 bytes (= 64kB - 8 byte UDP header - 20 byte IP header).

In “Lab 5/traces/3.A.PC1.pcap”, we captured the transmissions in following order: -l1024, -l1072. -l1073, -l65507, -l65508

Exercise 3-b. TCP and Fragmentation

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

- When a TCP connection is established, it negotiates the 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.
- 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 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 âĂIJDestination unreachable; Fragmentation neededâĂİ. 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.

1. Modify the MTU of the interfaces with the values as shown in Table 5.2. In Linux, you can

Linux PC	MTU size of Ethernet Interface eth0	MTU size of Ethernet Interface eth1
PC1	1500	not used
PC2	500	not used
PC3	1500	1500

Table 5.2: MTU Sizes

view the MTU values of all interfaces in the output of the `ifconfig` command. For example, on PC2, you type:

```
PC2% ifconfig
```

The same command is used to modify the MTU value. For example, to set the MTU value of interface *eth0* on PC2 to 500 bytes, use the `ifconfig` command as follows:

```
PC2% ifconfig eth0 mtu 500
```

2. Start Wireshark on the *eth0* interfaces of both PC1 and PC3, and start to capture traffic with no filters set.
3. Start a nttcp receiver on PC2, and a nttcp sender on PC1 and generate TCP traffic with the following commands:

```
PC2% nttcp -i -rs -l1024 -n2 -p4444
PC1% nttcp -ts -l1024 -n2 -p4444 -D 10.0.2.22
```

Observe the output of Wireshark:

Question 3.B.1.3.a)
Do you observe fragmentation? If so, where does it occur? Explain your observation.
We do not observe fragmentation, TCP avoids fragmentation by setting the DF bit.

Additionally:
The MSS is negotiated in the three way handshake and determined by the minimum MTU value of the two pcs. We see in this case that the MSS is 460.
This size is respected by all of the packets in the nttcp transmission, so no fragmentation is needed.

Question 3.B.1.3.b)
Explain why there is no ICMP error message generated in the first part of the experiment (Step 3). Is the DF bit set in the IP datagrams?
Only when an intermediary hop is causing a "bottleneck" for a packet to be forwarded due to its MTU being exceeded will there be any ICMP error messages: the hosts themselves have knowledge of their own MTUs and based on those, the MSS is negotiated in the three way handshake. We see in this case that the MSS is 460.

The DF-bit is set in the IP datagrams.

4. Now change the MTU size on interface eth1 of PC3 to 500 bytes. Change the MTU size of interface eth0 on PC2 to 1500 bytes.

5. Repeat the nttcp transmission in Step 3.

Question 3.B.1.5.a)
Do you observe fragmentation? If so, where does it occur? Explain your observation.
We do not observe fragmentation, TCP avoids fragmentation by setting the DF bit.

Question 3.B.1.5.b)
If you observe ICMP error messages, describe how they are used for Path MTU Discovery.

The ICMP error messages are used for Path MTU Discovery to reduce the TCP transmission's MSS value and send smaller packets.

We see this in packets 45, 47 and 48 in the file "Lab 5/traces/3.B.PC1.pcap":

- packet 45 is a TCP packet sent by PC1 with a payload of 1024 bytes, which exceeds PC3's MTU.
- PC1 then receives packet 47 from PC3, which is the ICMP error message.
- packet 48 is a TCP packet sent by PC1 with a smaller payload of 500 bytes.

So, we see that ICMP error messages are used for Path MTU Discovery. The size of the packets has been decreased. However, the decreased size is still too big. We see that the payload is now 500 bytes, but the total packet size is what should be 500 bytes. We're not sure what's going wrong here, but suspect it was caused our nttcp configuration.

6. Save all Wireshark output (Select the Print details option).

Question 3.B.2)
If you observed ICMP error messages, include one such message in the report. Also

include the first TCP segment that is sent after PC1 has received the ICMP error message.

If you observed ICMP error messages, include one such message in the report:

```
No.     Time        Source                Destination           Protocol Length
    Info
     47 537.418549  10.0.1.33             10.0.1.11             ICMP     590
        Destination unreachable (Fragmentation needed)

Frame 47: 590 bytes on wire (4720 bits), 590 bytes captured (4720 bits)
    Encapsulation type: Ethernet (1)
    Arrival Time: Apr 11, 2016 10:13:49.771318000 CEST
    [Time shift for this packet: 0.000000000 seconds]
    Epoch Time: 1460362429.771318000 seconds
    [Time delta from previous captured frame: 0.002373000 seconds]
    [Time delta from previous displayed frame: 0.002373000 seconds]
    [Time since reference or first frame: 537.418549000 seconds]
    Frame Number: 47
    Frame Length: 590 bytes (4720 bits)
    Capture Length: 590 bytes (4720 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:ip:icmp:ip:tcp:data]
    [Coloring Rule Name: ICMP errors]
    [Coloring Rule String: icmp.type eq 3 || icmp.type eq 4 || icmp.type eq 5 ||
        icmp.type eq 11 || icmpv6.type eq 1 || icmpv6.type eq 2 || icmpv6.type eq 3
         || icmpv6.type eq 4]
Ethernet II, Src: IntelCor_36:39:c7 (68:05:ca:36:39:c7), Dst: IntelCor_36:33:a0
    (68:05:ca:36:33:a0)
    Destination: IntelCor_36:33:a0 (68:05:ca:36:33:a0)
        Address: IntelCor_36:33:a0 (68:05:ca:36:33:a0)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
            default)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Source: IntelCor_36:39:c7 (68:05:ca:36:39:c7)
        Address: IntelCor_36:39:c7 (68:05:ca:36:39:c7)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
            default)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.0.1.33 (10.0.1.33), Dst: 10.0.1.11 (10.0.1.11)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0xc0 (DSCP 0x30: Class Selector 6; ECN: 0x00:
        Not-ECT (Not ECN-Capable Transport))
        1100 00.. = Differentiated Services Codepoint: Class Selector 6 (0x30)
        .... ..00 = Explicit Congestion Notification: Not-ECT (Not ECN-Capable
            Transport) (0x00)
    Total Length: 576
    Identification: 0x8814 (34836)
    Flags: 0x00
        0... .... = Reserved bit: Not set
        .0.. .... = Don't fragment: Not set
        ..0. .... = More fragments: Not set
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
    Header checksum: 0xd9bd [validation disabled]
        [Good: False]
        [Bad: False]
    Source: 10.0.1.33 (10.0.1.33)
    Destination: 10.0.1.11 (10.0.1.11)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
Internet Control Message Protocol
    Type: 3 (Destination unreachable)
    Code: 4 (Fragmentation needed)
```

```
    Checksum: 0x164f [correct]
56  MTU of next hop: 500
    Internet Protocol Version 4, Src: 10.0.1.11 (10.0.1.11), Dst: 10.0.2.22
        (10.0.2.22)
58      Version: 4
        Header length: 20 bytes
60      Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT
            (Not ECN-Capable Transport))
            0000 00.. = Differentiated Services Codepoint: Default (0x00)
62          .... ..00 = Explicit Congestion Notification: Not-ECT (Not ECN-Capable
                Transport) (0x00)
        Total Length: 1076
64      Identification: 0x985b (39003)
        Flags: 0x02 (Don't Fragment)
66          0... .... = Reserved bit: Not set
            .1.. .... = Don't fragment: Set
68          ..0. .... = More fragments: Not set
        Fragment offset: 0
70      Time to live: 64
        Protocol: TCP (6)
72      Header checksum: 0x8748 [validation disabled]
            [Good: False]
74          [Bad: False]
        Source: 10.0.1.11 (10.0.1.11)
76      Destination: 10.0.2.22 (10.0.2.22)
        [Source GeoIP: Unknown]
78      [Destination GeoIP: Unknown]
    Transmission Control Protocol, Src Port: 43803 (43803), Dst Port: 5038 (5038),
        Seq: 3445317143, Ack: 468857916
80      Source port: 43803 (43803)
        Destination port: 5038 (5038)
82      Sequence number: 3445317143
        [Stream index: 3]
84      Sequence number: 3445317143    (relative sequence number)
        Acknowledgment number: 468857916    (relative ack number)
86      Header length: 32 bytes
        Flags: 0x018 (PSH, ACK)
88          000. .... .... = Reserved: Not set
            ...0 .... .... = Nonce: Not set
90          .... 0... .... = Congestion Window Reduced (CWR): Not set
            .... .0.. .... = ECN-Echo: Not set
92          .... ..0. .... = Urgent: Not set
            .... ...1 .... = Acknowledgment: Set
94          .... .... 1... = Push: Set
            .... .... .0.. = Reset: Not set
96          .... .... ..0. = Syn: Not set
            .... .... ...0 = Fin: Not set
98      Window size value: 229
        [Calculated window size: 229]
100     [Window size scaling factor: 128]
        Checksum: 0x8f0f [validation disabled]
102         [Good Checksum: False]
            [Bad Checksum: False]
104     Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
            No-Operation (NOP)
106             Type: 1
                    0... .... = Copy on fragmentation: No
108                 .00. .... = Class: Control (0)
                    ...0 0001 = Number: No-Operation (NOP) (1)
110         No-Operation (NOP)
                Type: 1
112                 0... .... = Copy on fragmentation: No
                    .00. .... = Class: Control (0)
114                 ...0 0001 = Number: No-Operation (NOP) (1)
            Timestamps: TSval 61392463, TSecr 61391765
116             Kind: Timestamp (8)
                Length: 10
```

```
118                 Timestamp value: 61392463
                    Timestamp echo reply: 61391765
120     Data (496 bytes)

122 0000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
124 0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
126 0040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    0050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
128 0060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    0070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
130 0080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    0090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
132 00a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    00b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
134 00c0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    00d0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
136 00e0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    00f0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
138 0100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    0110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
140 0120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    0130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
142 0140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    0150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
144 0160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    0170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
146 0180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    0190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
148 01a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    01b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
150 01c0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
    01d0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
152 01e0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
          Data: 000000000000000000000000000000000000000000000000...
154       [Length: 496]
```

The first TCP segment that is sent after PC1 has received the ICMP error message:

```
    No.     Time        Source                Destination           Protocol Length
        Info
2       48 537.418590   10.0.1.11             10.0.2.22             TCP      566    [
            TCP Out-Of-Order] 43803 > 5038 [ACK] Seq=1 Ack=1 Win=29312 Len=500 TSval
            =61392464 TSecr=61391765

4   Frame 48: 566 bytes on wire (4528 bits), 566 bytes captured (4528 bits)
        Encapsulation type: Ethernet (1)
6       Arrival Time: Apr 11, 2016 10:13:49.771359000 CEST
        [Time shift for this packet: 0.000000000 seconds]
8       Epoch Time: 1460362429.771359000 seconds
        [Time delta from previous captured frame: 0.000041000 seconds]
10      [Time delta from previous displayed frame: 0.000041000 seconds]
        [Time since reference or first frame: 537.418590000 seconds]
12      Frame Number: 48
        Frame Length: 566 bytes (4528 bits)
14      Capture Length: 566 bytes (4528 bits)
        [Frame is marked: False]
16      [Frame is ignored: False]
        [Protocols in frame: eth:ip:tcp:data]
18      [Coloring Rule Name: Bad TCP]
        [Coloring Rule String: tcp.analysis.flags && !tcp.analysis.window_update]
20  Ethernet II, Src: IntelCor_36:33:a0 (68:05:ca:36:33:a0), Dst: IntelCor_36:39:c7
        (68:05:ca:36:39:c7)
        Destination: IntelCor_36:39:c7 (68:05:ca:36:39:c7)
22          Address: IntelCor_36:39:c7 (68:05:ca:36:39:c7)
```

```
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
            default)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Source: IntelCor_36:33:a0 (68:05:ca:36:33:a0)
        Address: IntelCor_36:33:a0 (68:05:ca:36:33:a0)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
            default)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.0.1.11 (10.0.1.11), Dst: 10.0.2.22 (10.0.2.22)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (
        Not ECN-Capable Transport))
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..00 = Explicit Congestion Notification: Not-ECT (Not ECN-Capable
            Transport) (0x00)
    Total Length: 552
    Identification: 0x985d (39005)
    Flags: 0x02 (Don't Fragment)
        0... .... = Reserved bit: Not set
        .1.. .... = Don't fragment: Set
        ..0. .... = More fragments: Not set
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0x8952 [validation disabled]
        [Good: False]
        [Bad: False]
    Source: 10.0.1.11 (10.0.1.11)
    Destination: 10.0.2.22 (10.0.2.22)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
Transmission Control Protocol, Src Port: 43803 (43803), Dst Port: 5038 (5038), Seq:
     1, Ack: 1, Len: 500
    Source port: 43803 (43803)
    Destination port: 5038 (5038)
    [Stream index: 3]
    Sequence number: 1    (relative sequence number)
    [Next sequence number: 501    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    Header length: 32 bytes
    Flags: 0x010 (ACK)
        000. .... .... = Reserved: Not set
        ...0 .... .... = Nonce: Not set
        .... 0... .... = Congestion Window Reduced (CWR): Not set
        .... .0.. .... = ECN-Echo: Not set
        .... ..0. .... = Urgent: Not set
        .... ...1 .... = Acknowledgment: Set
        .... .... 0... = Push: Not set
        .... .... .0.. = Reset: Not set
        .... .... ..0. = Syn: Not set
        .... .... ...0 = Fin: Not set
    Window size value: 229
    [Calculated window size: 29312]
    [Window size scaling factor: 128]
    Checksum: 0x193b [validation disabled]
        [Good Checksum: False]
        [Bad Checksum: False]
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
        No-Operation (NOP)
            Type: 1
                0... .... = Copy on fragmentation: No
                .00. .... = Class: Control (0)
                ...0 0001 = Number: No-Operation (NOP) (1)
        No-Operation (NOP)
            Type: 1
```

```
                0... .... = Copy on fragmentation: No
                .00. .... = Class: Control (0)
                ...0 0001 = Number: No-Operation (NOP) (1)
        Timestamps: TSval 61392464, TSecr 61391765
            Kind: Timestamp (8)
            Length: 10
            Timestamp value: 61392464
            Timestamp echo reply: 61391765
    [SEQ/ACK analysis]
        [Bytes in flight: 2049]
        [TCP Analysis Flags]
            [This frame is a (suspected) out-of-order segment]
                [Expert Info (Warn/Sequence): Out-Of-Order segment]
                    [Message: Out-Of-Order segment]
                    [Severity level: Warn]
                    [Group: Sequence]
Data (500 bytes)

0000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00c0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00d0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00e0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00f0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
01a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
01b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
01c0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
01d0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
01e0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
01f0  00 00 00 00                                       ....
    Data: 00000000000000000000000000000000000000000000000000...
    [Length: 500]
```

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 a 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 5.1.

Answer the questions in Exercise 4-a (Steps 3, 4, and 5), Exercise 4-b (Step 3), and Exercise 4-c (Step 2). **For each answer, include Wireshark data to support your answer.**

Exercise 4-a. Opening and Closing a TCP Connection

Set up a TCP connection and observe the packets that open and close the connection. Determine how the parameters of a TCP connection are negotiated between the TCP client and the TCP server.s

1. This part of the lab only uses PC1 and PC2 in the network configuration in Figure 5.1. If the network is not set up, follow the instructions of Exercise 1-a. Verify that the MTU values of all interfaces of PC1 and PC2 are set to 1500 bytes, which is the default MTU for Ethernet networks.
2. Start Wireshark on the *eth1* interface of PC1 to capture traffic of the Telnet connection. Do not set any filters.
3. Establishing a TCP connection: Establish a Telnet session from PC1 to PC2 as follows:

   ```
   PC1% telnet 10.0.5.22
   ```

   Observe the TCP segments of the packets that are transmitted:

Question 4.A.3.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.
The three-way handshake packets are packets no. 1, 2 and 3 in “Lab 5/traces/4.A.PC1.pcap”.

The first packet (from client (PC1) to server (PC2)) has the SYN flag set. PC2 recognizes this as the first packet of the three-way handshake. SYN is for synchronizing the sequence numbers.

Say PC1 sent sequence number i, then PC2 knows that the next packet from PC1 will have sequence number i + 1. In this case, because the SYN flag is set, even though no application data was sent, the sequence number increases by 1.

The next packet (from server to client) has the SYN and ACK flags set. PC1 interprets this as the second packet in the three-way handshake. Say PC2 sent sequence number j, then PC1 knows the next packet from PC2 will have sequence number j + 1. This packet also acknowledges PC1's initial packet by sending acknowledgement number i + 1 (an ACK always has the sequence number of the first byte the host expects to receive next).

Then PC1 concludes the three-way handshake by sending one more packet with sequence number i + 1 and an acknowledgement of PC2's packet with sequence number j + 1.

```
1 0.000000  10.0.5.11 10.0.5.22 TCP 74  48613âĘŠ23 [SYN] Seq=2495102103 Win=29200
    Len=0 MSS=1460 SACK_PERM=1 TSval=4215509 TSecr=0 WS=128
2 0.000621  10.0.5.22 10.0.5.11 TCP 74  23âĘŠ48613 [SYN, ACK] Seq=3213249774 Ack
    =2495102104 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=4214383 TSecr=4215509 WS
    =128
3 0.000660  10.0.5.11 10.0.5.22 TCP 66  48613âĘŠ23 [ACK] Seq=2495102104 Ack
    =3213249775 Win=29312 Len=0 TSval=4215509 TSecr=4214383
```

Question 4.A.3.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?

PC1 started with sequence number 2495102103 and PC2 started with sequence number 3213249774.

In Wireshark we see the relative seq-number (they both start with a relative seq-number of 0) but on inspection of the value in the packet we get the real (hexadecimal) value sent over the wire.

In the Wireshark preferences, relative sequence numbers can be turned off, which we did for the output in the next questions.

The absolute seq-number may be between 0 and 4,294,967,295.

```
1 1 0.000000  10.0.5.11 10.0.5.22 TCP 74  48613âĘŠ23 [SYN] Seq=2495102103 Win=29200
    Len=0 MSS=1460 SACK_PERM=1 TSval=4215509 TSecr=0 WS=128
  2 0.000621  10.0.5.22 10.0.5.11 TCP 74  23âĘŠ48613 [SYN, ACK] Seq=3213249774 Ack
    =2495102104 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=4214383 TSecr=4215509 WS
    =128
```

Question 4.A.3.c)

Identify the first packet that contains application data? What is the sequence number used in the first byte of application data sent from the TCP client to the TCP server?

Packet no. 4, the first packet after the three-way handshake, contains Telnet application data.

The sequence number of the first byte from client to server is 2495102104, which is the same sequence number PC1 used in the final message of the three-way handshake.

The relative sequence number is 1.

```
Frame 4: 93 bytes on wire (744 bits), 93 bytes captured (744 bits)
Ethernet II, Src: IntelCor_1a:80:15 (68:05:ca:1a:80:15), Dst: IntelCor_1a:7c:75
    (68:05:ca:1a:7c:75)
Internet Protocol Version 4, Src: 10.0.5.11 (10.0.5.11), Dst: 10.0.5.22 (10.0.5.22)
Transmission Control Protocol, Src Port: 48613 (48613), Dst Port: 23 (23), Seq:
    2495102104, Ack: 3213249775, Len: 27
    Source Port: 48613 (48613)
    Destination Port: 23 (23)
    [Stream index: 0]
    [TCP Segment Len: 27]
    Sequence number: 2495102104
    [Next sequence number: 2495102131]
    Acknowledgment number: 3213249775
    Header Length: 32 bytes
    .... 0000 0001 1000 = Flags: 0x018 (PSH, ACK)
    Window size value: 229
    [Calculated window size: 29312]
    [Window size scaling factor: 128]
    Checksum: 0x1e62 [validation disabled]
    Urgent pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
        No-Operation (NOP)
        No-Operation (NOP)
        Timestamps: TSval 4215509, TSecr 4214383
    [SEQ/ACK analysis]
Telnet
    Do Suppress Go Ahead
    Will Terminal Type
    Will Negotiate About Window Size
    Will Terminal Speed
    Will Remote Flow Control
    Will Linemode
    Will New Environment Option
    Do Status
    Will X Display Location
```

Question 4.A.3.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. Determine the values of the window sizes for the TCP client and the TCP server.

In the first two packets of the three-way handshake, the window sizes are rather large: 29200 for PC1 and 28960 for PC2.

After this, the data is sent with a window size fluctuating between 227 and 229, with a scaling factor of 128, so respectively 29056 and 29312.

This window scale is also agreed upon in the handshake.

```
Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
Ethernet II, Src: IntelCor_1a:80:15 (68:05:ca:1a:80:15), Dst: IntelCor_1a:7c:75
    (68:05:ca:1a:7c:75)
Internet Protocol Version 4, Src: 10.0.5.11 (10.0.5.11), Dst: 10.0.5.22 (10.0.5.22)
Transmission Control Protocol, Src Port: 48613 (48613), Dst Port: 23 (23), Seq:
    2495102103, Len: 0
    Source Port: 48613 (48613)
    Destination Port: 23 (23)
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 2495102103
    Acknowledgment number: 0
    Header Length: 40 bytes
```

```
    .... 0000 0000 0010 = Flags: 0x002 (SYN)
    Window size value: 29200
    [Calculated window size: 29200]
    Checksum: 0x1e4f [validation disabled]
    Urgent pointer: 0
    Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-
        Operation (NOP), Window scale
Frame 2: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
Ethernet II, Src: IntelCor_1a:7c:75 (68:05:ca:1a:7c:75), Dst: IntelCor_1a:80:15
    (68:05:ca:1a:80:15)
Internet Protocol Version 4, Src: 10.0.5.22 (10.0.5.22), Dst: 10.0.5.11 (10.0.5.11)
Transmission Control Protocol, Src Port: 23 (23), Dst Port: 48613 (48613), Seq:
    3213249774, Ack: 2495102104, Len: 0
    Source Port: 23 (23)
    Destination Port: 48613 (48613)
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 3213249774
    Acknowledgment number: 2495102104
    Header Length: 40 bytes
    .... 0000 0001 0010 = Flags: 0x012 (SYN, ACK)
    Window size value: 28960
    [Calculated window size: 28960]
    Checksum: 0x7b28 [validation disabled]
    Urgent pointer: 0
    Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-
        Operation (NOP), Window scale
    [SEQ/ACK analysis]
Frame 3: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
Ethernet II, Src: IntelCor_1a:80:15 (68:05:ca:1a:80:15), Dst: IntelCor_1a:7c:75
    (68:05:ca:1a:7c:75)
Internet Protocol Version 4, Src: 10.0.5.11 (10.0.5.11), Dst: 10.0.5.22 (10.0.5.22)
Transmission Control Protocol, Src Port: 48613 (48613), Dst Port: 23 (23), Seq:
    2495102104, Ack: 3213249775, Len: 0
    Source Port: 48613 (48613)
    Destination Port: 23 (23)
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 2495102104
    Acknowledgment number: 3213249775
    Header Length: 32 bytes
    .... 0000 0001 0000 = Flags: 0x010 (ACK)
    Window size value: 229
    [Calculated window size: 29312]
    [Window size scaling factor: 128]
    Checksum: 0x1e47 [validation disabled]
    Urgent pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    [SEQ/ACK analysis]
Frame 4: 93 bytes on wire (744 bits), 93 bytes captured (744 bits)
Ethernet II, Src: IntelCor_1a:80:15 (68:05:ca:1a:80:15), Dst: IntelCor_1a:7c:75
    (68:05:ca:1a:7c:75)
Internet Protocol Version 4, Src: 10.0.5.11 (10.0.5.11), Dst: 10.0.5.22 (10.0.5.22)
Transmission Control Protocol, Src Port: 48613 (48613), Dst Port: 23 (23), Seq:
    2495102104, Ack: 3213249775, Len: 27
    Source Port: 48613 (48613)
    Destination Port: 23 (23)
    [Stream index: 0]
    [TCP Segment Len: 27]
    Sequence number: 2495102104
    [Next sequence number: 2495102131]
    Acknowledgment number: 3213249775
    Header Length: 32 bytes
    .... 0000 0001 1000 = Flags: 0x018 (PSH, ACK)
    Window size value: 229
    [Calculated window size: 29312]
    [Window size scaling factor: 128]
```

```
71      Checksum: 0x1e62 [validation disabled]
        Urgent pointer: 0
73      Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
        [SEQ/ACK analysis]
75  Telnet
```

4.A.3.e) What is the MSS value that is negotiated between the TCP client and the TCP server?
PC1's initial packet includes a MSS of 1460, as does PC2's initial packet. They happened to both send the same values, although both directions could have different values for MSS. Only the packets where SYN is set include values for MSS.

```
Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
Ethernet II, Src: IntelCor_1a:80:15 (68:05:ca:1a:80:15), Dst: IntelCor_1a:7
    c:75 (68:05:ca:1a:7c:75)
Internet Protocol Version 4, Src: 10.0.5.11 (10.0.5.11), Dst: 10.0.5.22
    (10.0.5.22)
Transmission Control Protocol, Src Port: 48613 (48613), Dst Port: 23 (23),
    Seq: 2495102103, Len: 0
    Source Port: 48613 (48613)
    Destination Port: 23 (23)
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 2495102103
    Acknowledgment number: 0
    Header Length: 40 bytes
    .... 0000 0000 0010 = Flags: 0x002 (SYN)
    Window size value: 29200
    [Calculated window size: 29200]
    Checksum: 0x1e4f [validation disabled]
    Urgent pointer: 0
    Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps,
        No-Operation (NOP), Window scale
        Maximum segment size: 1460 bytes
            Kind: Maximum Segment Size (2)
            Length: 4
            MSS Value: 1460
        TCP SACK Permitted Option: True
        Timestamps: TSval 4215509, TSecr 0
        No-Operation (NOP)
        Window scale: 7 (multiply by 128)
            Kind: Window Scale (3)
            Length: 3
            Shift count: 7
            [Multiplier: 128]
Frame 2: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
Ethernet II, Src: IntelCor_1a:7c:75 (68:05:ca:1a:7c:75), Dst: IntelCor_1a
    :80:15 (68:05:ca:1a:80:15)
Internet Protocol Version 4, Src: 10.0.5.22 (10.0.5.22), Dst: 10.0.5.11
    (10.0.5.11)
Transmission Control Protocol, Src Port: 23 (23), Dst Port: 48613 (48613),
    Seq: 3213249774, Ack: 2495102104, Len: 0
    Source Port: 23 (23)
    Destination Port: 48613 (48613)
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 3213249774
    Acknowledgment number: 2495102104
    Header Length: 40 bytes
    .... 0000 0001 0010 = Flags: 0x012 (SYN, ACK)
    Window size value: 28960
    [Calculated window size: 28960]
    Checksum: 0x7b28 [validation disabled]
    Urgent pointer: 0
    Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps,
        No-Operation (NOP), Window scale
        Maximum segment size: 1460 bytes
            Kind: Maximum Segment Size (2)
            Length: 4
            MSS Value: 1460
        TCP SACK Permitted Option: True
        Timestamps: TSval 4214383, TSecr 4215509
        No-Operation (NOP)
        Window scale: 7 (multiply by 128)
            Kind: Window Scale (3)
            Length: 3
            Shift count: 7
            [Multiplier: 128]
    [SEQ/ACK analysis]
```

4.A.3.f) How long does it take to open a TCP connection?
If we consider the connection to be "open" as soon as the first packet with application data is sent, in this case it took 0.843 ms. However, in practice this largely depends on the network conditions.

```
1 1 0.000000   10.0.5.11 10.0.5.22 TCP 74   48613âĘŠ23 [SYN] Seq=2495102103 Win
    =29200 Len=0 MSS=1460 SACK_PERM=1 TSval=4215509 TSecr=0 WS=128
  2 0.000621   10.0.5.22 10.0.5.11 TCP 74   23âĘŠ48613 [SYN, ACK] Seq
    =3213249774 Ack=2495102104 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval
    =4214383 TSecr=4215509 WS=128
3 3 0.000660   10.0.5.11 10.0.5.22 TCP 66   48613âĘŠ23 [ACK] Seq=2495102104 Ack
    =3213249775 Win=29312 Len=0 TSval=4215509 TSecr=4214383
  4 0.000843   10.0.5.11 10.0.5.22 TELNET  93   Telnet Data ...
```

4. Closing a TCP connection (initiated by client): On PC1, type `Ctrl-]` at the Telnet prompt and type quit, to terminate the connection. (If the Telnet session is no longer running, first create a new session). In the output of Wireshark, observe the TCP segments of the packets that are transmitted:

Question 4.A.4)
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 server or TCP client.

In "Lab 5/traces/4.A.PC1.pcap" the packets that are involved in closing the TCP connection are packets with no. 76, 77, 78. They both have the FIN flag set.
When a sender sets the FIN flag, it indicates that they won't send any more data. PC1 (client) sends one last ACK after sending a packet with FIN set, acknowledging the receipt of PC2's FIN message.

```
76  76.193351 10.0.5.11 10.0.5.22 TCP 66   48613âĘŠ23 [FIN, ACK] Seq=2495102305 Ack
    =3213250523 Win=30336 Len=0 TSval=4234558 TSecr=4216062
2 77  76.194236 10.0.5.22 10.0.5.11 TCP 66   23âĘŠ48613 [FIN, ACK] Seq=3213250523 Ack
    =2495102306 Win=29056 Len=0 TSval=4233431 TSecr=4234558
  78  76.194264 10.0.5.11 10.0.5.22 TCP 66   48613âĘŠ23 [ACK] Seq=2495102306 Ack
    =3213250524 Win=30336 Len=0 TSval=4234558 TSecr=4233431
```

5. , Closing a TCP connection (initiated by server): The closing of a connection can also be initiated by the server application, as seen next. Establish a Telnet session on PC1 to PC2 as follows:

```
PC1% telnet 10.0.5.22
```

Do not type anything. After a while, the connection will be closed by the TCP server, and a message is displayed at the Telnet client application.

Question 4.A.5.a)
Describe how the closing of the connection is different from Step 4.
The first packet with the FIN flag set is from PC2 (server), contrary to when the client closes the connection, then the client replies and the server acknowledges.

```
1 348 209.621366  10.0.5.22 10.0.5.11 TCP 66  23âĘŠ48615 [FIN, ACK] Seq=419429485 Ack
      =3956140201 Win=29056 Len=0 TSval=4266788 TSecr=4252924
  349 209.621460  10.0.5.11 10.0.5.22 TCP 66  48615âĘŠ23 [FIN, ACK] Seq=3956140201
      Ack=419429486 Win=29312 Len=0 TSval=4267915 TSecr=4266788
3 350 209.621939  10.0.5.22 10.0.5.11 TCP 66  23âĘŠ48615 [ACK] Seq=419429486 Ack
      =3956140202 Win=29056 Len=0 TSval=4266788 TSecr=4267915
```

Question 4.A.5.b)

How long does the Telnet server wait until it closes the TCP connection?
60 seconds

6. Save the Wireshark output.

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.

1. Start a new traffic capture with Wireshark on interface *eth1* of PC1.
2. Set a static entry in the ARP table for the non-existing IP address 10.0.5.100. Note that the IP address does not exist.

   ```
   PC1% arp -s 10.0.5.100 00:01:02:03:04:05
   ```

3. From PC1, establish a Telnet session to the non-existing host:

   ```
   PC1% telnet 10.0.5.100
   ```

   Observe the TCP segments that are transmitted.

Question 4.B.3.a)

How often does the TCP client try to establish a connection? How much time elapses between repeated attempts to open a connection?
We observed 7 attempts. Each try there is more time between retries.
1 second, 2 seconds, 4, 8, 16, 32.
So 2^{i-1} seconds for the i^{th} retry.

```
1 1 0.000000  10.0.5.11 10.0.5.100  TCP 74  33347âĘŠ23 [SYN] Seq=2556468869 Win=29200
      Len=0 MSS=1460 SACK_PERM=1 TSval=4441845 TSecr=0 WS=128
  2 0.999586  10.0.5.11 10.0.5.100  TCP 74  [TCP Retransmission] 33347âĘŠ23 [SYN] Seq
      =2556468869 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=4442095 TSecr=0 WS=128
3 3 3.003578  10.0.5.11 10.0.5.100  TCP 74  [TCP Retransmission] 33347âĘŠ23 [SYN] Seq
      =2556468869 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=4442596 TSecr=0 WS=128
  4 7.011578  10.0.5.11 10.0.5.100  TCP 74  [TCP Retransmission] 33347âĘŠ23 [SYN] Seq
      =2556468869 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=4443598 TSecr=0 WS=128
5 5 15.019585 10.0.5.11 10.0.5.100  TCP 74  [TCP Retransmission] 33347âĘŠ23 [SYN] Seq
      =2556468869 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=4445600 TSecr=0 WS=128
  6 31.051586 10.0.5.11 10.0.5.100  TCP 74  [TCP Retransmission] 33347âĘŠ23 [SYN] Seq
      =2556468869 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=4449608 TSecr=0 WS=128
7 7 63.147583 10.0.5.11 10.0.5.100  TCP 74  [TCP Retransmission] 33347âĘŠ23 [SYN] Seq
      =2556468869 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=4457632 TSecr=0 WS=128
```

Question 4.B.3.b)

Does the TCP client terminate or reset the connection, when it gives up with trying to establish a connection?
No it does not try, it just stops sending. Terminating would be pointless, as the connection has already stopped by the timeout.

Question 4.B.3.c)

Why does this experiment require to set a static ARP table entry?
If no ARP entry was set for the non-existing IP, an ARP request would first be sent out. The ARP request would fail, since no-one has that IP, and consequently no TCP packets would be sent out.

4. Save the Wireshark output.

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.

1. Start a new traffic capture with Wireshark on interface *eth1* of PC1.
2. Establish a TCP connection to port 80 of PC2.

```
PC1% telnet 10.0.5.22 80
```

There should not be a TCP server running on PC2 that is listening at this port number. Observe the TCP segments of the packets that are transmitted:

Question 4.C.2)

How does TCP at the remote host close this connection? How long does the process of ending the connection take?
A TCP packet with [RST,ACK] is returned by the receiving server. This means that the port is closed.

```
1 1 0.000000  10.0.5.11  10.0.5.22  TCP 74  49306âĘŠ80 [SYN] Seq=4067677396 Win=29200
    Len=0 MSS=1460 SACK_PERM=1 TSval=4578023 TSecr=0 WS=128
  2 0.000546  10.0.5.22  10.0.5.11  TCP 60  80âĘŠ49306 [RST, ACK] Seq=0 Ack=4067677397
    Win=0 Len=0
```

3. Save the Wireshark output.

Part 5. TCP data exchange - Interactive applications

In Parts 5 and 6 you study acknowledgements and flow control in TCP. The receiver of TCP data acknowledges the receipt of data in segments that have the ACK flag set. 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. Whenever possible, a TCP receiver sends an ACK in a segment that carries a payload. This is called piggybacking. A TCP receiver can acknowledge multiple segments in a single ACK. This is called cumulative acknowledgements.

In this lab, you study acknowledgements separately for interactive applications, such as Telnet, and for bulk transfer applications, such as file transfers. You will observe that different TCP mechanisms play a role for these different types of applications. In this part, you study the data transfer of interactive applications.

Interactive applications typically generate a small volume of data. 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 since small segments mainly consist of protocol headers. Here, 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 NagleâĂŹs algorithm, limits the number of small segments that a TCP sender can transmit without waiting for an ACK.

The network configuration is shown in Figure 5.2. The network connects two Linux PCs, PC1 and PC2, such that there are three paths between the PCs. One route goes over three Ethernet links (with either 10 Mbps or 100 Mbps), and one route goes over a serial WAN link (which will be set to 125 kbps), and one route goes over a direct Ethernet link (also with 10 Mbps or 100 Mbps).

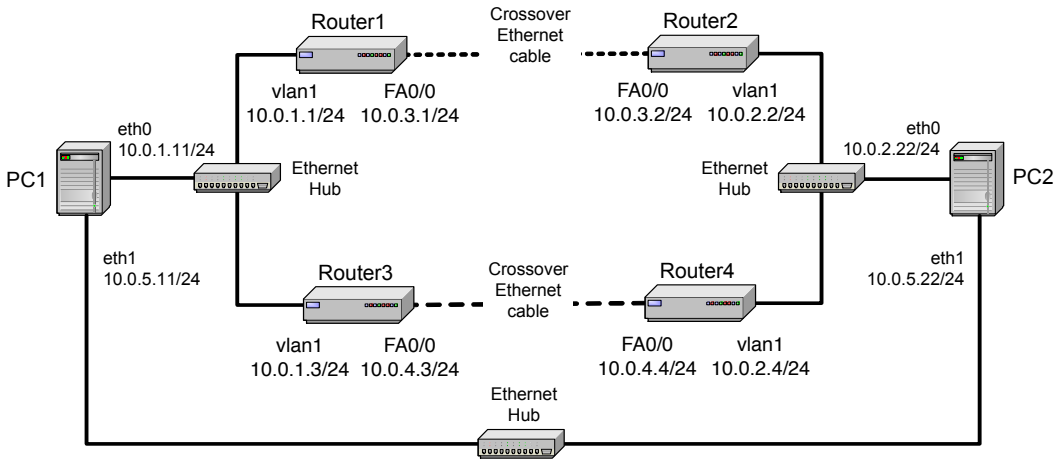


Figure 5.2: Network Topology for Parts 1-4.

Linux PC	Interface eth0	Interface eth1
PC1	10.0.1.11/24	10.0.5.11/24
PC2	10.0.2.22/24	10.0.5.22/24
Cisco Router	**Interface FastEthernet0/0**	**Interface vlan1**
Router1	10.0.3.1/24	10.0.1.1/24
Router2	10.0.3.2/24	10.0.2.2/24
Cisco Router	**Interface FastEthernet0/0**	**Interface vlan1**
Router3	10.0.4.3/24	10.0.1.3/24
Router4	10.0.4.4/24	10.0.2.4/24

Table 5.3: IP Addresses of Linux PCs and Cisco routers.

Linux PC	Routing Table Entries	
	Destination	**Next Hop**
PC1	default gateway	10.0.1.3
PC2	default gateway	10.0.2.4
Linux PC	**Routing Table Entries**	
	Destination	**Next Hop**
Router1	network 10.0.4.0/24	10.0.1.3
Router1	default gateway	10.0.3.2
Router2	network 10.0.4.0/24	10.0.2.4
Router2	default gateway	10.0.3.1
Router3	network 10.0.3.0/24	10.0.1.1
Router3	default gateway	10.0.4.4
Router4	network 10.0.3.0/24	10.0.2.2
Router4	default gateway	10.0.4.3

Table 5.4: Routing table entries for Part 5-8.

Exercise 5-a. Network Setup

The following network configuration is used in the remaining parts of the lab.

1. Set up the Ethernet connections as shown in Figure 5.2. *Note: connect two routers through a hub or switch if you don't have a crossover cable available.*

2. Configure the IP addresses and the routing tables of the PCs as shown in Tables 5.3 and 5.4.

   - IP forwarding should be disabled on both PC1 and PC2.
   - Set a new default gateways on PC1 and PC2. Remove the default gateway entry that was set in Part 1 of the lab. Changing the default gateway on PC1 is done with the following commands:

     ```
     PC1% route del default gw 10.0.1.33
     PC1% route add default gw 10.0.1.3
     ```

   Repeat the steps on PC2. Use the command `netstat -rn`, to verify that there are no other static routing entries.

3. Verify that the PCs are connected to the console ports of routers. PC1 should be connected to Router1, PC2 to Router 2, and so on. On each PC, establish a `minicom` session to the connected router.

4. Configure the IP addresses and routing table entries of the routers. The commands for Router4 are as follows:

```
Router4> enable
Password: <enable secret>
Router4# configure terminal
Router4(config)# no ip routing
Router4(config)# ip routing
Router4(config)# ip route 0.0.0.0 0.0.0.0 10.0.4.3
Router4(config)# ip route 10.0.3.0 255.255.255.0 10.0.2.2
Router4(config)# interface FastEthernet0/0
Router4(config-if)# no shutdown
Router4(config-if)# ip address 10.0.2.4 255.255.255.0
Router4(config-if)# interface FastEthernet0/1
Router4(config-if)# no shutdown
Router4(config-if)# interface vlan1
Router4(config-if)# no shutdown
Router4(config-if)# ip address 10.0.4.4 255.255.255.0
Router4(config-if)# end
```

   Use the commands `show ip route` and show interfaces to verify that the routing table and the interfaces are set correctly.

5. Emulate a slow serial link between Router3 and Router4 using traffic-shaping:

```
> interface FastEthernet0/0
> ip address 10.10.1.80 255.255.255.0
> fair-queue
> traffic-shape rate 64000 5000 5000 1000
```

   This will simulate a link of 64kbps on the packets going out of *FE0/0*. You can change the parameters of the “traffic-shape” command to control the rate of the link. DonâĂŹt forget that shaping only works on the outgoing packets of the interface, so you will have to configure this in both Router3 and Router4! Also note that you can not apply the traffic shaping to a virtual interface like *Vlan1*. You can use

```
> show running-config
```

   to check if the traffic shaping is correctly configured to 64kbps.

6. Test the network connectivity by issuing ping commands between PC1 and PC2. Verify the route taken by traffic between the PCs by issuing `traceroute` commands.

```
PC1% traceroute 10.0.2.22 PC1% traceroute 10.0.5.22
PC2% traceroute 10.0.1.11 PC2% traceroute 10.0.5.11
```

   Also, you should be able to issue ping commands between the routers. If the commands are not successful, use the commands `traceroute` (on Linux) or `trace` (on IOS) and the content of the routing tables to locate configuration problems.

If all commands are successful, then you are ready to continue.

Exercise 5-b. TCP Data Transfer - Interactive Applications over a fast link

Here you observe interactive data transfer in TCP, by establishing a TCP connection from PC1 to PC2 over the Ethernet link between the PCs. Dependent on the type of hub, the Ethernet link has a maximum data rate of 10 Mbps or 100 Mbps.

1. Start Wireshark on PC1 for interface *eth1*, and start to capture traffic. Do not set any filters.
2. On PC1, establish a Telnet session to PC2 by typing:

   ```
   PC1% telnet 10.0.5.22
   ```

   Log in as root user.
3. Now start to type a few characters in the window which contains 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:

Include your answers to the following questions. Include examples from the saved Wireshark data to support your answers.

Question 5.B.1.a)

Observe the number of packets exchanged between the Linux 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.

There are 3 packets per keystroke. The client sends one packet with the typed character. The server echoes the character, sending one packet back, upon which it also piggybacks the acknowledgement of the receival of the client's packet. The server then sends an acknowledgement of the server's echoed character.

If there was no piggybacking of acknowledgements, we would see four packets per typed character.

Here's an example from our trace file, from traces/5.B.pcap.

```
Frame 29: 67 bytes on wire (536 bits), 67 bytes captured (536 bits)
Ethernet II, Src: IntelCor_39:cc:79 (68:05:ca:39:cc:79), Dst: IntelCor_39:e1:36
    (68:05:ca:39:e1:36)
Internet Protocol Version 4, Src: 10.0.5.11 (10.0.5.11), Dst: 10.0.5.22 (10.0.5.22)
Transmission Control Protocol, Src Port: 32872 (32872), Dst Port: 23 (23), Seq:
    3586174827, Ack: 679939142, Len: 1
Telnet
    Data: t
Frame 30: 68 bytes on wire (544 bits), 68 bytes captured (544 bits)
Ethernet II, Src: IntelCor_39:e1:36 (68:05:ca:39:e1:36), Dst: IntelCor_39:cc:79
    (68:05:ca:39:cc:79)
Internet Protocol Version 4, Src: 10.0.5.22 (10.0.5.22), Dst: 10.0.5.11 (10.0.5.11)
Transmission Control Protocol, Src Port: 23 (23), Dst Port: 32872 (32872), Seq:
    679939142, Ack: 3586174828, Len: 2
Telnet
    Data:
Frame 31: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
Ethernet II, Src: IntelCor_39:cc:79 (68:05:ca:39:cc:79), Dst: IntelCor_39:e1:36
    (68:05:ca:39:e1:36)
Internet Protocol Version 4, Src: 10.0.5.11 (10.0.5.11), Dst: 10.0.5.22 (10.0.5.22)
Transmission Control Protocol, Src Port: 32872 (32872), Dst Port: 23 (23), Seq:
    3586174828, Ack: 679939144, Len: 0
```

Question 5.B.1.b)

When the TCP client receives the echo of a character, it waits a certain time before sending the ACK. Why does the TCP client delay? How long is this delay? How much does the delay vary?

The delay is very small, around $2.0 \times 10^{-5}s$ on average.
We have found intervals ranging from $1.3 \times 10^{-5}s$ to $44 \times 10^{-5}s$.

Question 5.B.1.c)
What is the time delay associated with the transmission of ACKs from the Telnet server on PC3?
The time delay will be as small as possible, because the server wants to send the echo reply immediately upon receival of a typed character and the ack can be biggypacked on the reply.
We don't know exactly how long the server took to respond since we weren't instructed to trace the server's network activity.

Question 5.B.1.d)
Which flags, if any, are set in the TCP segments that carry typed characters as payload? Explain the meaning of these flags.
The PSH and ACK flags are set.
The PSH is the push-flag and is used to tell the receiver to *push* the data straight on the receiving socket, without waiting for the buffer to fill further.
The ACK is an acknowledgement of a previously received TCP packet, piggybacked on the telnet data packet.

Question 5.B.1.e)
Why do segments that have an empty payload carry a sequence number? Why does this not result in confusion at the TCP receiver?
They carry a sequence number to specify for which packet it ACKs.
With a payload length of 0, the sequence number is not increased, so there can be no confusion.

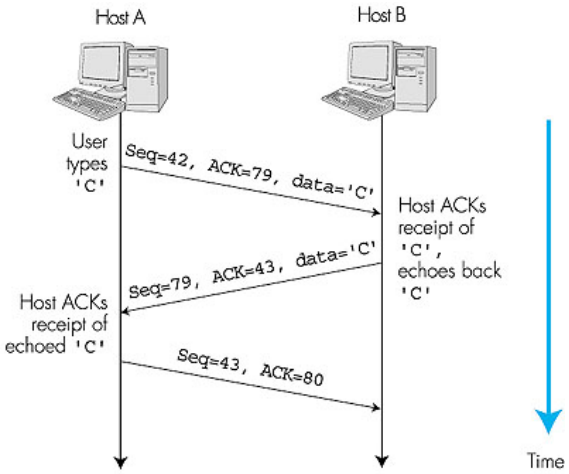
Question 5.B.1.f)
What is the window size that is advertised by the Telnet client and the Telnet server? How does the value of the window size field vary as the connection progresses?
It is alternating 227 and 229. (with scaling factor 128)
It grows very steadily to 235 and even 254.

Question 5.B.1.g)
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?
We see that now, the client is often also able to piggyback acknowledgements on packets that carry a payload, since it has data ready to be sent when a packet from the server comes in.

4. Terminate the Telnet session by typing exit.

5. Stop the traffic capture with Wireshark and save the captured packets.

Question 5.B.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.
We found this image on $http://netlab.ulusofona.pt/rc/book/3-transport/3_05/03-30.jpg$



Exercise 5-c. TCP Data Transfer - Interactive Applications over a slow link

This exercise repeats the previous exercise, but establishes a data connection over the emulated slow link. The rate of this link should be set to 9600bps. This low rate introduces significant delays between PC1 and PC2. 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 the case. A heuristic in TCP, called NagleâĂŹs algorithm, forces the sender to wait for an ACK after transmitting a small segment, even if the window size would allow the transmission of multiple segments. Therefore, no matter how slow or fast you type, you should only observe one TCP segment in transmission at a time, when the TCP segments are small.

1. Start to capture traffic with Wireshark on interface *eth0* of PC1. Do not set any display filters.
2. On PC1, establish a Telnet session to PC2 by typing:

   ```
   PC1% telnet 10.0.2.22
   ```

   Log in as root user. Note With the above IP address the route between PC1 to PC2 passes through the emulated serial link between Router3 and Router4.
3. As, in the previous exercise, type a few characters in the window that contains the Telnet session. Vary the rate at which you type characters in the Telnet client program. Observe the output of Wireshark:

Include your answers to the following questions. For each answer, include Wireshark data to support your answer.

Question 5.C.1.a)

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

We still observe 3 packets per keystroke.
When we type characters more quickly, some are combined in a single packet, and also some acknowledgements for the server are piggybacked onto the client's packets.
Here's an example of a packet (found in traces/5.C.pcap) where two characters were combined into a single packet and which also piggybacked an acknowledgement for the server.

```
Frame 225: 68 bytes on wire (544 bits), 68 bytes captured (544 bits)
Ethernet II, Src: IntelCor_36:33:a0 (68:05:ca:36:33:a0), Dst: Cisco_ef:eb:24 (00:0d
    :bc:ef:eb:24)
Internet Protocol Version 4, Src: 10.0.1.11 (10.0.1.11), Dst: 10.0.2.22 (10.0.2.22)
Transmission Control Protocol, Src Port: 51889 (51889), Dst Port: 23 (23), Seq:
    1236438828, Ack: 1914573200, Len: 2
    Source Port: 51889 (51889)
    Destination Port: 23 (23)
    [Stream index: 0]
    [TCP Segment Len: 2]
    Sequence number: 1236438828
    [Next sequence number: 1236438830]
    Acknowledgment number: 1914573200
    Header Length: 32 bytes
    .... 0000 0001 1000 = Flags: 0x018 (PSH, ACK)
    Window size value: 237
    [Calculated window size: 30336]
    [Window size scaling factor: 128]
    Checksum: 0x1749 [validation disabled]
    Urgent pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    [SEQ/ACK analysis]
Telnet
    Data: kl
```

Question 5.C.1.b)

Do you observe delayed acknowledgements? Why is the outcome expected?
We do see delayed acknowledgements, which is not abnormal.
It could become a problem however, if a delayed ACK would be kept waiting until all data is received, but Nagle's algorithm does not send out all the data yet, waiting for an ACK, resulting in a deadlock which resumes when a timeout is reached.
But we have not been able to reproduce this situation.

Question 5.C.1.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.
Nagle's algorithm waits until an ACK is received for the previous character sent, if this does not arrive in time, the following characters are kept in buffer till the ACK arrives or the buffer reaches a certain size.
When the ACK arrives, the characters in buffer are sent in a single packet.

4. Terminate the Telnet session by typing exit.
5. Stop the traffic capture with Wireshark and save the captured packets.

Question 5.C.2)

Include an example from the saved Wireshark data, which shows that NagleâĂŹs algorithm is used by the TCP sender.

```
    No.     Time        Source                Destination           Protocol Length
        Info
    294 24.417583    10.0.1.11             10.0.2.22             TELNET   69
        Telnet Data ...

Frame 294: 69 bytes on wire (552 bits), 69 bytes captured (552 bits)
```

```
    Encapsulation type: Ethernet (1)
    Arrival Time: Mar 18, 2016 14:31:40.785202000 CET
    [Time shift for this packet: 0.000000000 seconds]
    Epoch Time: 1458307900.785202000 seconds
    [Time delta from previous captured frame: 0.000022000 seconds]
    [Time delta from previous displayed frame: 0.000022000 seconds]
    [Time since reference or first frame: 24.417583000 seconds]
    Frame Number: 294
    Frame Length: 69 bytes (552 bits)
    Capture Length: 69 bytes (552 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:ip:tcp:telnet]
    [Coloring Rule Name: TCP]
    [Coloring Rule String: tcp]
Ethernet II, Src: IntelCor_36:33:a0 (68:05:ca:36:33:a0), Dst: Cisco_ef:eb:24 (00:0d
    :bc:ef:eb:24)
    Destination: Cisco_ef:eb:24 (00:0d:bc:ef:eb:24)
        Address: Cisco_ef:eb:24 (00:0d:bc:ef:eb:24)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
            default)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Source: IntelCor_36:33:a0 (68:05:ca:36:33:a0)
        Address: IntelCor_36:33:a0 (68:05:ca:36:33:a0)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory
            default)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.0.1.11 (10.0.1.11), Dst: 10.0.2.22 (10.0.2.22)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x10 (DSCP 0x04: Unknown DSCP; ECN: 0x00: Not-
        ECT (Not ECN-Capable Transport))
        0001 00.. = Differentiated Services Codepoint: Unknown (0x04)
        .... ..00 = Explicit Congestion Notification: Not-ECT (Not ECN-Capable
            Transport) (0x00)
    Total Length: 55
    Identification: 0x75c0 (30144)
    Flags: 0x02 (Don't Fragment)
        0... .... = Reserved bit: Not set
        .1.. .... = Don't fragment: Set
        ..0. .... = More fragments: Not set
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0xadd0 [correct]
        [Good: True]
        [Bad: False]
    Source: 10.0.1.11 (10.0.1.11)
    Destination: 10.0.2.22 (10.0.2.22)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
Transmission Control Protocol, Src Port: 51889 (51889), Dst Port: telnet (23), Seq:
     280, Ack: 897, Len: 3
    Source port: 51889 (51889)
    Destination port: telnet (23)
    [Stream index: 0]
    Sequence number: 280    (relative sequence number)
    [Next sequence number: 283    (relative sequence number)]
    Acknowledgment number: 897    (relative ack number)
    Header length: 32 bytes
    Flags: 0x018 (PSH, ACK)
        000. .... .... = Reserved: Not set
        ...0 .... .... = Nonce: Not set
        .... 0... .... = Congestion Window Reduced (CWR): Not set
        .... .0.. .... = ECN-Echo: Not set
        .... ..0. .... = Urgent: Not set
```

```
66          .... ...1 .... = Acknowledgment: Set
            .... .... 1... = Push: Set
68          .... .... .0.. = Reset: Not set
            .... .... ..0. = Syn: Not set
70          .... .... ...0 = Fin: Not set
        Window size value: 237
72      [Calculated window size: 30336]
        [Window size scaling factor: 128]
74      Checksum: 0x174a [validation disabled]
            [Good Checksum: False]
76          [Bad Checksum: False]
        Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
78          No-Operation (NOP)
                Type: 1
80                  0... .... = Copy on fragmentation: No
                    .00. .... = Class: Control (0)
82                  ...0 0001 = Number: No-Operation (NOP) (1)
            No-Operation (NOP)
84              Type: 1
                    0... .... = Copy on fragmentation: No
86                  .00. .... = Class: Control (0)
                    ...0 0001 = Number: No-Operation (NOP) (1)
88          Timestamps: TSval 986467, TSecr 985898
                Kind: Timestamp (8)
90              Length: 10
                Timestamp value: 986467
92              Timestamp echo reply: 985898
        [SEQ/ACK analysis]
94          [This is an ACK to the segment in frame: 293]
            [The RTT to ACK the segment was: 0.000022000 seconds]
96          [Bytes in flight: 3]
Telnet
98      Data: fka
```

Part 6. TCP data exchange - Bulk data transfer

The TCP receiver can use acknowledgements to control the transmission rate at the TCP sender. This is called flow control. Flow control is not an issue for interactive applications, since the traffic volume of these applications is small, but plays an important role in bulk transfer applications.

Bulk data transfers generally transmit full segments. In TCP, the receiver controls the amount of data that the sender can transmit using a sliding window flow control scheme. This prevents that the receiver gets overwhelmed with data. 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,..., 359. The TCP sender may have already transmitted some data in that range.

In this part of the lab, you observe acknowledgements and flow control for bulk data transfers, where traffic is generated with the nttcp tool. 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. This is done in Exercise 6-c. We also show how to save the graphs to a file.

All exercises are done with the network configuration from Figure 5.2.

Exercise 6-a. TCP Data Transfer - Bulk 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 nttcp traffic generation tool.

1. The network configuration is the same as in Part 5. If the network is not set up accordingly, then follow the instructions in Exercise 5-a.
2. Start Wireshark on PC1 for interface *eth1*, and start to capture traffic. Do not set any display filters.
3. Use nttcp to generate TCP traffic between PC1 and PC2.
   - On PC2, start a nttcp receiving process by typing:
     ```
     PC2% nttcp -i -rs -l1000 -n500 -p4444
     ```
   - On PC1, start a nttcp sender process that sends 500 blocks of application data by typing:
     ```
     PC1% nttcp -ts -l1000 -n500 -p4444 -D 10.0.5.22
     ```

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

The outcome of this experiment is dependent on the data rate of the Ethernet link between PC1 and PC2. If PC1 and PC2 are connected directly by an Ethernet . crossover cable or by a dual-speed hub, they will most likely exchange traffic at a data rate of 100 Mbps. If the Linux PCs are connected by a 10 Mbps Ethernet hub, the data rate is limited accordingly. The rate of the connection has a big impact on the outcome of the experiment.

Include your answers to the following questions. Include captured traffic to support your answers.

Question 6.A.a)

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?
There is not an ACK replied for every single data packet received.
Acknowledgements are sent according to the TCP window size, which is the amount of data can be sent before an ACK is required.
Note that sending an earlier ACK is permitted, as the window size only specifies the size after which it has to be sent.

Question 6.A.b)

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?
There is typically something like 2000 bytes ACKed.
The maximum amount is equal to the window size, which is around 29000 for the sending host.

Question 6.A.c)

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?
The window size increases rather rapidly on the receivers side, starting at 29000 and going up to 266000.
The senders' window size doesn't change very much.

Question 6.A.d)

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?
Packet 314: at time 0.375 has acknr. 231233.
Corresponds with:

Packet 281: at time 0.324 with seqnr. 231233
So the time elapsed is around 50 msec.

Question 6.A.e)

Determine whether, or not, the TCP sender generally transmits the maximum amount of data allowed by the advertised window. Explain your answer.
It generally sends up the maximum amount of data as it has periods it has to stop and wait for the ACKs before proceeding.

Question 6.A.f)

When the nttcp sender has transmitted all its data, it closes the connection, but acknowledgements from PC2 still trickle in. What does PC2 do when it has sent all ACKs?
It also ACK's & FIN 's the teardown request and closes the connection.

5. Stop the traffic capture with Wireshark and save the Wireshark output.

Exercise 6-b. TCP Data Transfer - Bulk transfer (Slow Link)

This exercise repeats the previous experiment, with the exception that traffic is sent over the emulated serial link.

1. Set the data rate of the serial link to 125 kbps. As in Exercixe-5b, this is done by using the traffic-shaping option.
2. Create a new Wireshark session on PC1 for interface *eth0*, and start to capture traffic. Do not set any display filters.
3. Use nttcp to generate TCP traffic between PC1 and PC2.
   - On PC2, start a nttcp receiving process by typing:

     ```
     PC2% nttcp -i -rs -l1000 -n500 -p4444
     ```
   - On PC1, start a nttcp sender process that sends 500 blocks of application data by typing:

     ```
     PC1% nttcp -ts -l1000 -n500 -p4444 -D 10.0.2.22
     ```

   With the given destination address, traffic will go through the slow serial link.
4. Observe the differences to the data transmission in the previous exercise.
5. Stop the traffic capture with wireshark and save the wireshark output.

Include your answers to the following questions. Emphasize the differences to the observations made in Exercise 6-a.

Question 6.B.a)
How does the pattern of data segments and ACK change, as compared to the fast Ethernet link?
ACKs are more frequently sent, relatively, or the sender waits more until it receives its ACKs.

Question 6.B.b)
Does the frequency of ACKs change?
During the transmission, the ACKs are sent rather constantly, without much variation in their frequency.

Question 6.B.c)
Is the range of window sizes advertised by the receiver different from those in 6-a?
The window size advertised by the receiver grows larger, even after a collapse around 25 seconds as a result of many retransmissions and their duplicate ACKs. It grows up to 350000+ bytes.

Question 6.B.d)
Does the TCP sender generally transmit the maximum amount of data allowed by the advertised window? Explain your answer.
He does, as this limit is reached resulting in retransmissions and their subsequent duplicate ACKs, because of the unnecessary retransmissions.

Exercise 6-c. View a graph of TCP data transfer

Wireshark can generate graphs that illustrate the transmissions of segments on a PC connection. This exercise familiarizes you with the graphing capabilities of Wireshark, and shows how you can extract information from the graphs.

1. **Select a TCP connection:** In the Wireshark main window, select a packet from the TCP connection for which you want to build a graph.
   - Here, select a TCP packet sent from PC1 to PC2 in Exercise 6-b.
2. **Select the type of graph:** Select the Statistics menu from the Wireshark main window, and then select TCP Stream Analysis in the pull down menu, as shown in Figure 5.3. This displays the plotting functions available in Wireshark:

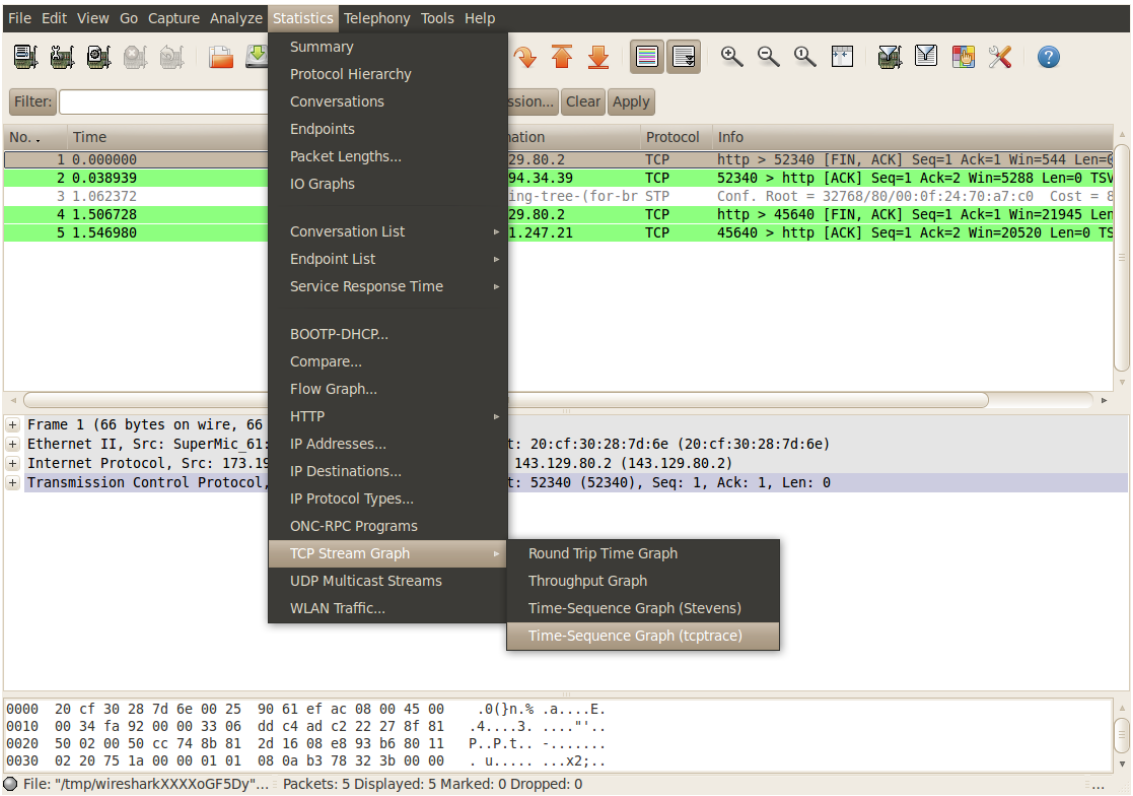


Figure 5.3: Selecting the type of graph for a TCP connection.

Time-Sequence Graph (Stevens) Plots the transmission of sequence numbers as a function of time. There is one data point for each transmission of a TCP segment.

Time-Sequence Graph (tcptrace) Generates a plot as shown in Figure 5.4. The graph is similar to the previous one, but additional information is included on the state of the sliding window.

Throughput Graph Shows the rate of data transmission as a function of time.

RTT Graph Shows the roundtrip time (RTT) as a function of time.

Try out each of the graphs for the TCP connection from Exercise 6-b. Make sure that you select a packet with TCP payload from PC1 to PC2 in the Wireshark main window, before you generate a graph.

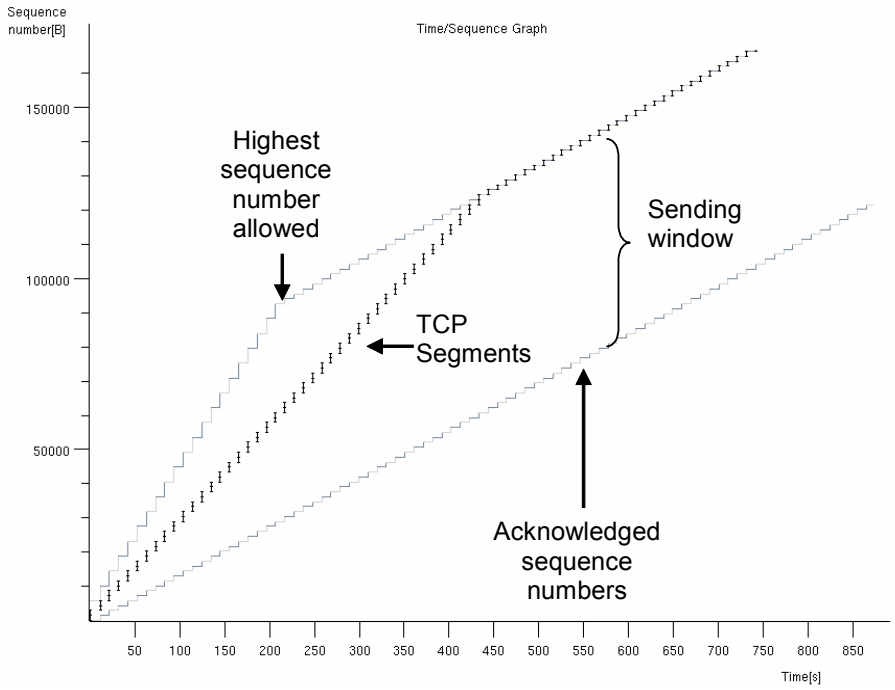


Figure 5.4: Time-Sequence Graph (tcptrace).

3. **Navigating the graphs:** It is possible to navigate the graphs generated by Wireshark. For example, you can zoom into a graph to display an area of interest at a greater level of detail. Here is the complete set of available options:

Left mouse button	Selecting a data point highlights the corresponding segment in the Wireshark main window.
Middle mouse button	Zooms to a selected part of the graph.
Shift + Middle mouse button	Zooms out.
Right mouse button	Holding this button down and moving the mouse, moves the displayed section of the graph. This only works if the graph is zoomed in.
Ctrl + Right Button	Magnifies a small portion of the graph.
Space	Toggles between showing and not showing crosshairs.
s	Toggles between relative and absolute sequence numbers.
t	Toggles the display of the x-axis.

Table 5.5: **Navigating graphs of TCP connection in Wireshark:**

4. **Interpreting the Time-Sequence Graph (tcptrace):** A lot of information can be extracted from the Time-Sequence Graph (tcptrace). Refer to Figure 5.4 of a TCP connection. This graph shows the transmission of TCP segments and acknowledgements. The short vertical bars indicate the transmission of TCP segments. Each short bar represents one TCP segment, and the length of a bar corresponds to the length of a segment. The figure also shows two step curves. The top step curve shows the highest allowed sequence number, and the bottom step curve shows the acknowledged sequence numbers. These functions are determined from the acknowledgement number and the window size fields of received

ACKs. The vertical distance between the two step curves shows the open part of the sliding window, that is, the sequence numbers that the TCP sender may transmit.
From Figure 5.4, you can see that most of the time, TCP segments are transmitted in groups of two segments. An inspection of the vertical plot shows that no segments are retransmitted. The figure shows that the sequence numbers of transmitted segments are close to the upper step curve. This indicates that the TCP sender utilizes the entire sliding window, and that the transmissions by the TCP sender are triggered by arrivals of ACKs from the TCP receiver.

- Study the Time-Sequence Graph (tcptrace) of the TCP connections in Exercise 6-a and Exercise 6-b. Review the questions in Step 4 of Exercise 6-a and Step 3 in Exercise 6-b, and try to determine the answers to the questions directly from the graphs.

5. **Saving graphs to a file:** Unfortunately, Wireshark does not allow you to save the graphs for a TCP connection. However, there is a simple method in Linux to save a window on the desktop to a file. Suppose you have constructed a TCP graph, similar to that of Figure 5.4, on PC1 and want to save it as a TIFF file. This is done by typing

```
PC1% import lab6c.tif
```

and then clicking on the window with the TCP graph. This saves the graph to a TIFF file with name `lab6c.tif`. If you use a different file extension, the file is saved to a different image format. Select a file format that you can use in your lab report and that has sufficient image quality. The import command supports numerous file formats, including those in Table 5.5. We recommend that you use the TIFF file format, which offers the highest quality image. The size of the file can be reduced to less than 20 kB if you compress the file following the instructions below.

File extension	Format	approximate size of resulting file
.jpeg	JPEG	30kB
.eps	Encapsulated Postscript	3.5 MB
.gif	GIF	300kB
.tif	TIFF	3.5MB

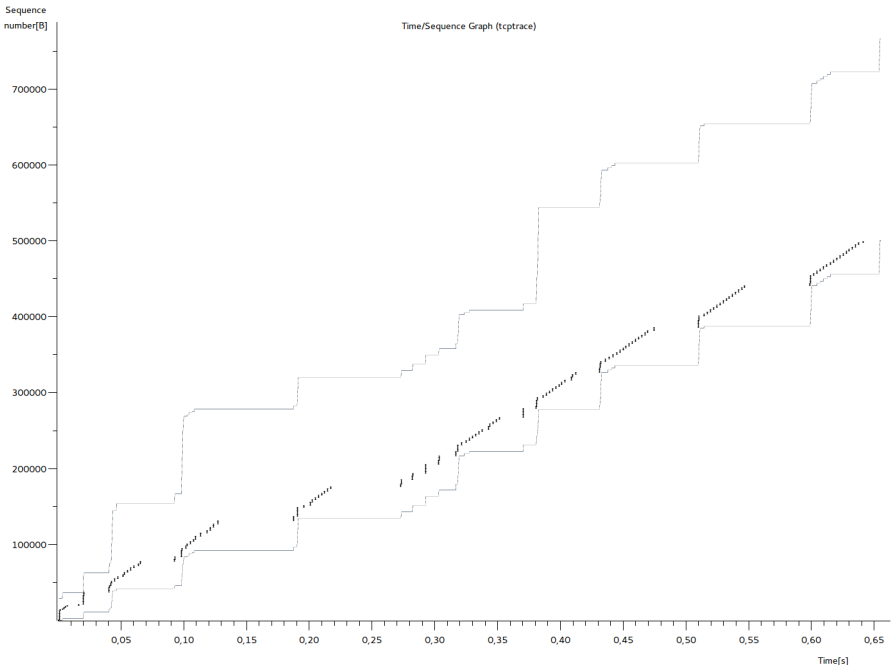
Table 5.6: File formats for import command.

- Save the Time-Sequence Graph (`tcptrace`) that you created for the TCP connections in Exercise 6-a and Exercise 6-b. Select a file format that you can use in your lab report. If you want to include detailed areas from the graphs, you may want to save multiple files for each graph.
- Verify the file has been correctly saved.

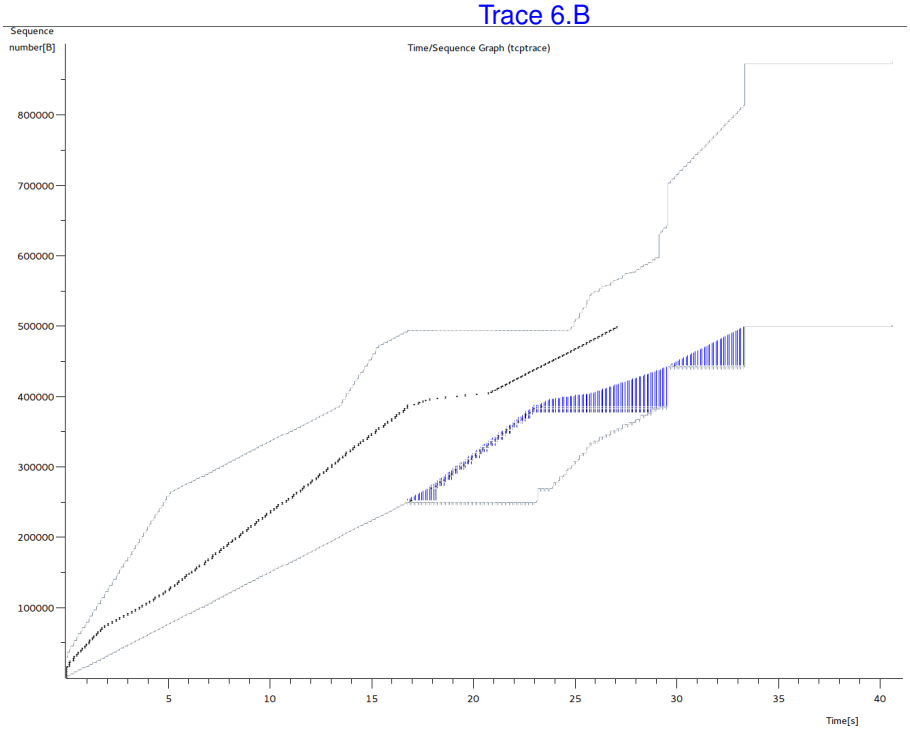
Question 6.C)

Include the Time-Sequence Graph (`tcptrace`) graphs that you saved. You may also use these graphs for your answers to the lab report questions for Exercise 6-a and Exercise 6-b.

Trace 6.A


Sequence number[B]
Time/Sequence Graph (tcptrace)
700000
600000
500000
400000
300000
200000
100000
0,05
0,10
0,15
0,20
0,25
0,30
0,35
0,40
0,45
0,50
0,55
0,60
0,65
Time[s]

Trace 6.B


Sequence number[B]
Time/Sequence Graph (tcptrace)
800000
700000
600000
500000
400000
300000
200000
100000
5
10
15
20
25
30
35
40
Time[s]

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 one 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 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 (see RFC 2988 from the prelab). Following a heuristic which is called KarnâĂŹs 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 acknowledgement 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 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 headers as an option. Whether SACKs are used or not, is negotiated in TCP header options when the TCP connection is created.

The exercises in this part explore aspects of TCP retransmissions that do not require access to internal timers. Unfortunately, the roundtrip time measurements and the RTO values are difficult to observe, and are, therefore, not included in this lab.

The network configuration for this part is the network shown in Figure 5.2.

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. Here, data is sent over the serial link, which is set to 125 kbps. When you disconnect one of the cables of the network, ACKs cannot reach the sending host. As a result, a timeout occurs and the sender performs retransmissions.

1. The network configuration is the same as in Part 5. If the network is not setup accordingly, then follow the instructions in Exercise 5-a.

2. Set the data rate of the emulated serial link to 125 kbps. If you continue from Part 6, this is the current value of the data rate of the link. If not, you proceed as in Exercise 5-a by setting the the traffic-shaping.

3. Start Wireshark on PC1 and capture traffic on interface *eth0*. Set a display filter to TCP traffic. This is done by typing `tcp` in the window at the bottom of the main window of Wireshark, next to the label Filter.

4. Start a nttcp receiving process on PC2:

```
PC2% nttcp -i -rs -l1000 -n500 -p4444
```

5. Start a nttcp sending process on PC1:

```
PC1% nttcp -ts -l1000 -n500 -p4444 -D 10.0.2.22
```

Question 7.A.5)

When the connection is created do the TCP sender and TCP receiver negotiate to permit SACKs? Describe the process of the negotiation.
In the three way handshake there is a 'TCP SACK permitted option' field.
When one of the two hosts denie permission, it is set to false and not used, otherwise true is used and SACKs are permitted.

6. Once Wireshark has transmitted at least one hundred packets, disconnect the cable that connects *FastEthernet0/0* of Router3 to the Ethernet hub. Disconnect the cable at the hub. Wait at least five minutes before you reconnect the cable. Observe TCP retransmissions from PC1 in the output of Wireshark.

Question 7.A.6.a)

Observe the time instants when retransmissions take place. How many packets retransmitted at one time?
There are 6 retransmissions sent over a period, all of the same packet, presumeably the first packet which should be retransmitted, if not the only.

Question 7.A.6.b)

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?
The waiting period grows rapidly, waiting respectively 5, 10, 20, 40, 300 and 100 seconds between each retransmission, where the last retransmission seems to be succesfull and resulting in the resuming of the connection .
The retransmissions stop after the 4th retransmission, and resume on reconnecting the cable.

Question 7.A.6.c)

After how many retransmissions, if at all, does the TCP sender give up with retransmitting the segment? Describe your observations.
After 3-4 retransmissions it stops. 3 as observed in 7.A.third.pcap,
4 as observed in 7.A.first- & -second.pcap
It resumes when reconnecting the cable.

7. Now reconnect the cable, and wait until the transmission resumes (This may take some time). Now quickly disconnect and reconnect the cable that connects the interface *FastEthernet0/0* of Router3 to the Ethernet hub. Repeat this procedure a number of times, by varying the length of time that the cable is disconnected. Now observe the retransmissions from PC1.

Question 7.A.7)

Are the retransmissions different from those in Step 6? Specifically, do you observe fast retransmits and/or SACKs?

On reconnecting, the first ACK received is da SACK, but this was also the case when reconnecting after waiting a longer time.

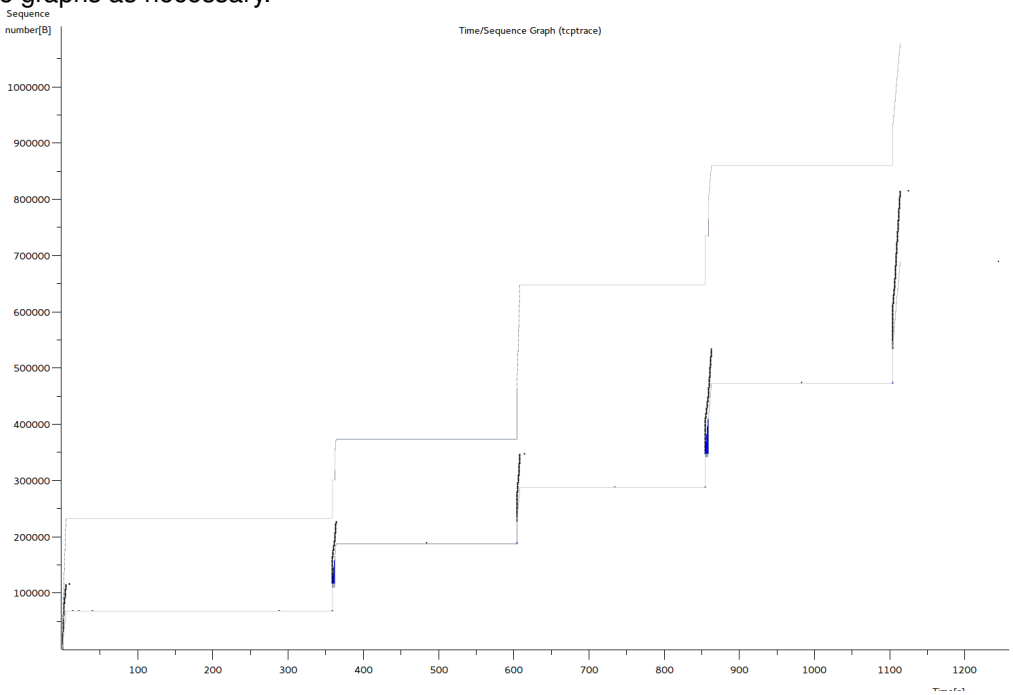
Fast retransmits show only when the cable was reconnected after a longer time.

8. Use the instructions from Exercise 6-c to build a Time-Sequence Graph (tcptrace) in Wireshark for the TCP connection. Study the output of the graph and use the graph to provide answers to the questions above. Use the navigating features to zoom in to parts of the graph that are of interest.

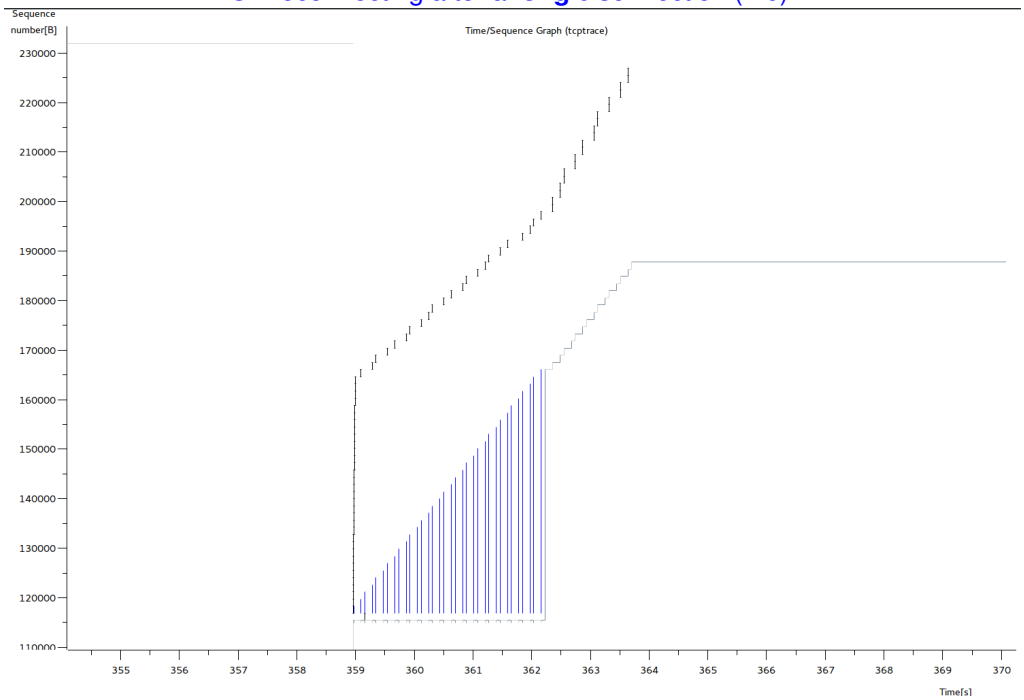
9. Follow the instructions from Exercise 6-c to generate image files for the Time- Sequence Graph (tcptrace). Save enough images so that you can use the graphs to answer the above questions in your lab report. Generate images that show clearly the retransmission attempts in Step 6 and Step 7.

Question 7.A.9)

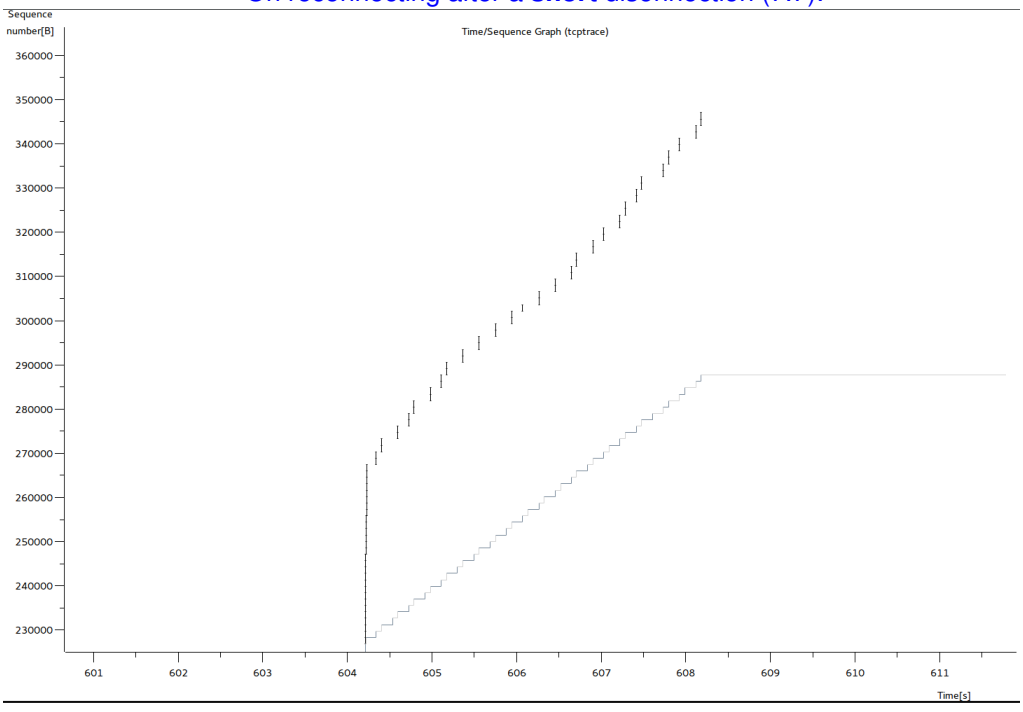
Include the image files saved in Step 9, and use them to support your answers. Annotate the graphs as necessary.



On reconnecting after a **long** disonnection (7.6):



On reconnecting after a **short** disonnection (7.7):



Exercise 7-b. TCP performance at an overloaded link

Next you perform an experiment, where you overload the emulated serial link between Router3 and Router4, and cause losses and retransmissions due to buffer overflows at Router3.

As in Exercise 7-a, you set up a TCP connection from PC1 to PC2. Here, however, you flood

Router 3 with ICMP Echo Request messages. The purpose of this exercise is to observe how a TCP connection performs when a router is overloaded.

1. Set the data rate of the serial link to 64 kbps.

2. Start Wireshark on PC1 for interface *eth0* interface, and start to capture traffic. Set a display filter to TCP traffic.

3. Start a nttcp receiving process on PC2:

```
PC2% nttcp -i -rs -l1000 -n500 -p4444
```

4. Start a nttcp sending process on PC1:

```
PC1% nttcp -ts -l1000 -n500 -p4444 -D 10.0.2.22
```

5. Once Wireshark has transmitted at least one hundred TCP packets, start to flood ICMP Echo Request messages by typing on PC1

```
PC1% ping -f 10.0.2.22
```

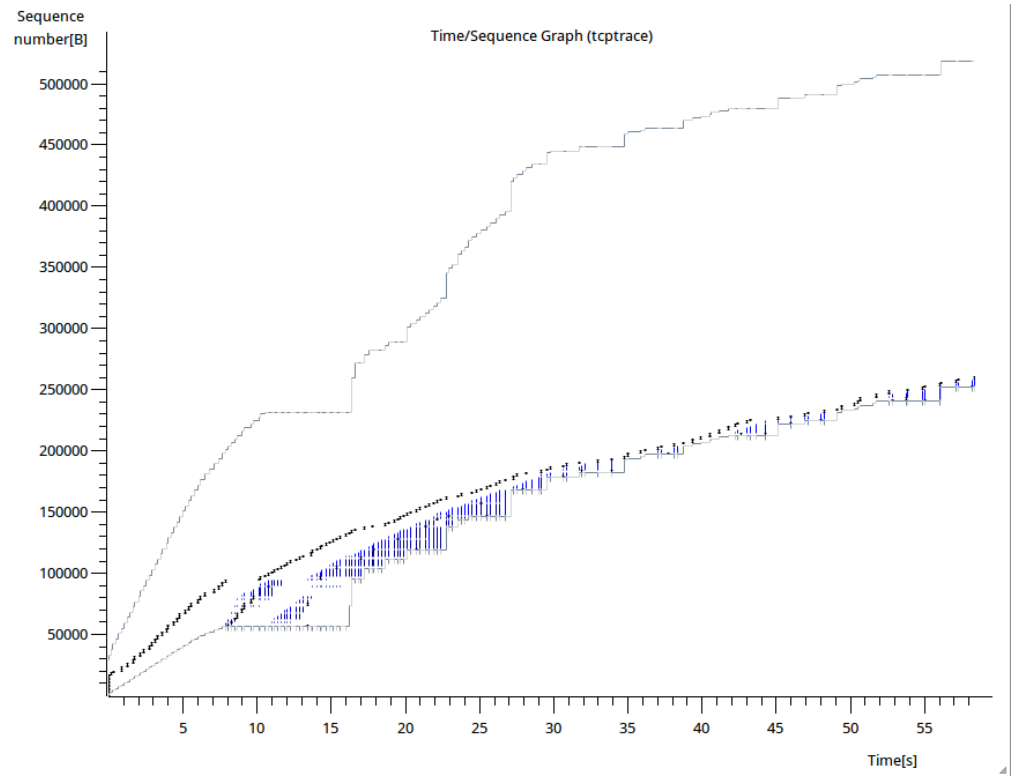
   Recall that with the `-f` option, PC2 sends ICMP Echo Request packets as fast as possible. The ICMP traffic sent from PC1 to PC2 will overflow the buffers of Router3 at the serial link.

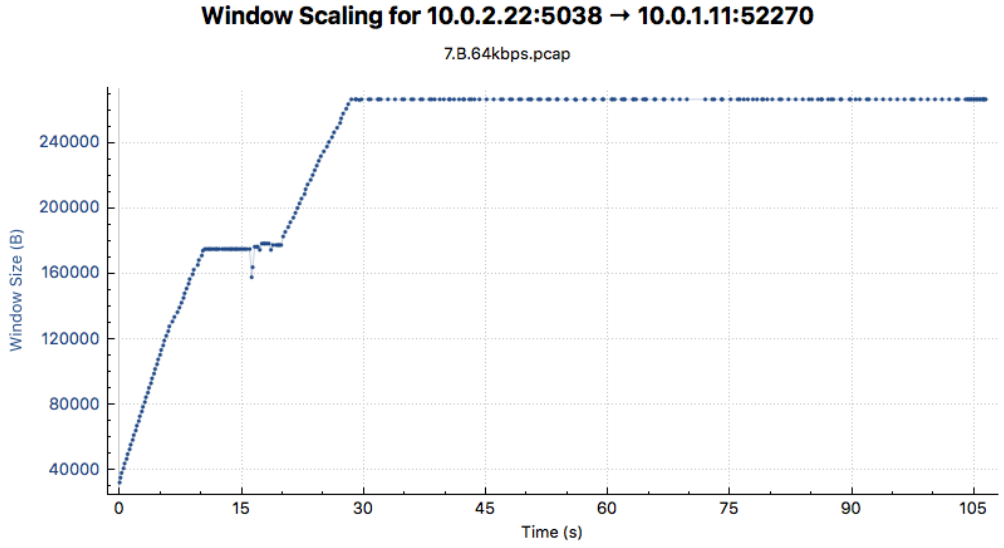
6. Follow the instructions from Exercise 6-c to build a Time-Sequence Graph (tcptrace) in Wireshark for the TCP connection. If you do not observe any retransmissions, reduce the data rate of the serial link. As long as the nttcp sender is transmitting packets, rerun the construction of the graph to observe how the graph changes as time progresses. In the graph, observe the progress of the TCP connection:

7. Follow the instructions from Exercise 6-c to generate image files for the Time- Sequence Graph (tcptrace) and Throughput Graph. Save enough images so that you can use the graphs to answer the above questions in your lab report. Generate an image that shows in detail the loss events that occur right after the ping command is started.

Question 7.B)

Include your answers to the questions in Step 6. Included the saved image files from step 7. Annotate and describe the plots to support your answers.





Question 7.B.6.a)

Describe the losses that occur in the graph when the ping command is started. Do losses occur in regular intervals or irregularly?

On the tcptrace graph, the lower gray line shows the ACKs received. A loss is indicated when this line stays horizontal. Losses seem to occur irregularly.

Question 7.B.6.b)

From the graph, describe the size of the advertised window changes when the flooding ping is started.

On the tcptrace graph, the window size can be derived from the distance between the upper gray line and the lower gray line. The upper gray line shows the greatest sequence number

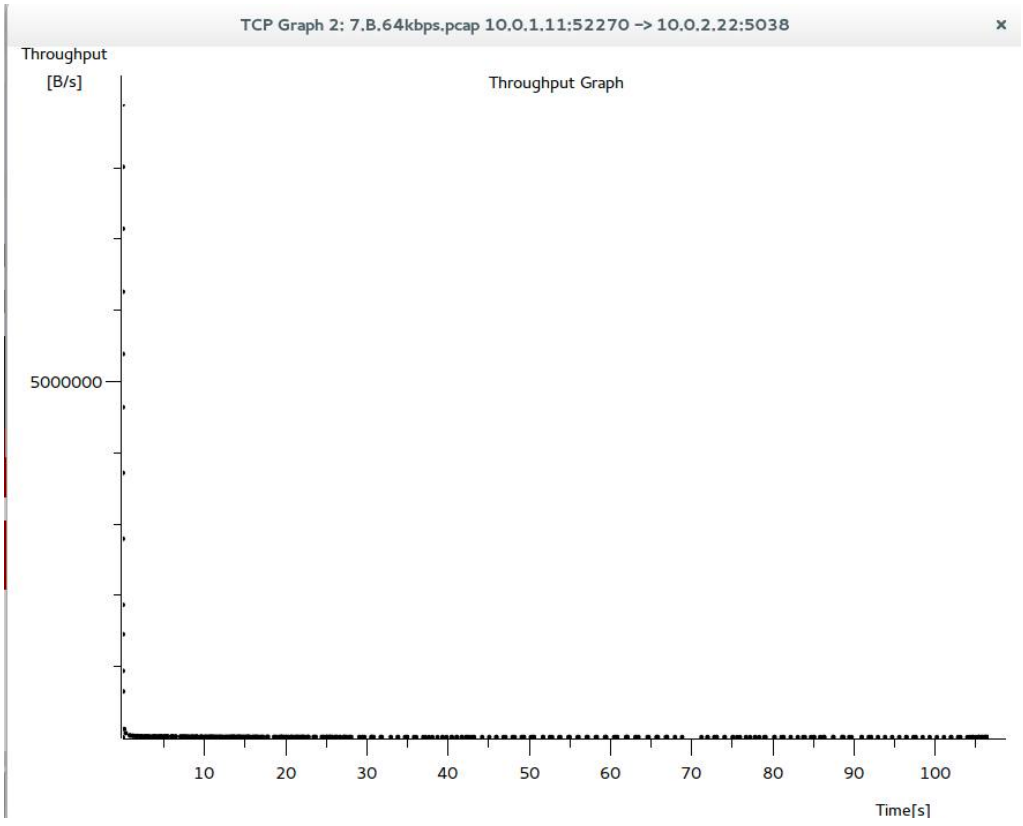
the sender is allowed to send, given the ACKs it has received. The lower gray line shows the ACK number the sender has received, as we noted before.
We did, however, also include a graph of the window size itself, where we can clearly see that it stops increasing when the effect of the pings starts to kick in.

Question 7.B.6.c)

Try to determine if retransmissions occur due to fast retransmit or due to timeouts of the timers. How can you determine which type of retransmissions you observe?
In the trace file ("Lab 5/traces/7.B.64kbps.pcap") we see that packets sometimes are being retransmitted right after a duplicate acknowledgement, which is an indication that packet loss has occurred. Thus it doesn't seem unreasonable to assume that the sender performed a fast retransmit. Wireshark also indicates when it suspects a retransmisstion was a fast retransmit, and we do indeed see this happening.

Question 7.B.6.d)

Generate a Throughput Graph to view the data rate of the TCP connection. How does the throughput change after the flood of pings is started?



We see the throughput drop drastically once the pings start.

Part 8. TCP Congestion Control

TCP congestion control consists of a set of algorithms that adapt the sending rate of a TCP sender to the current conditions in the network. When the network is not congested, the TCP sender is allowed to increase its sending rate, and when the network is congested, the TCP sender reduces its rate. The TCP sender maintains a congestion window which 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 minimum of the congestion window and the window size sent by the receiver.

For congestion control, each TCP sender keeps two variables, the congestion window (cwnd) and the slow-start threshold (ssthresh). The initial values are set to one segment for cwnd and 65535 bytes for ssthresh. TCP congestion control operates in two phases, called slow start and congestion avoidance. The sender is in the slow start phase when cwnd $\leq$ ssthresh. Here, cwnd is increased by one for each arrived ACK. 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 only after cwnd ACKs. This is done by incrementing cwnd by a fraction of a segment when an ACK arrives.

The TCP sender assumes that the network is congested when a segment is lost, that is, when the retransmission timer has a timeout or when three duplicate ACKs arrive. 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 goal of this part of the lab is to observe the development of the congestion window. Since the number of the segments that can be transmitted by a TCP sender is the result of the congestion window as well as the advertised window, and since data segments and returning ACKs interleave, the size of the congestion window is not derived by observing traffic.

Exercise 8-a. Network Setup

The network configuration used is that in Figure 5.2. To observe the slow start features, change the routing table entries so that traffic from PC1 to PC2 traverses the path PC1 $\rightarrow$ R1 $\rightarrow$ R2 $\rightarrow$ PC2, and the reverse path is PC2 $\rightarrow$ R4 $\rightarrow$ R3 $\rightarrow$ PC1. When PC1 sends data to PC2, data segments can be transmitted quickly to PC2, but ACKs only slowly return to PC1. The sender will therefore transmit a full window of packets up to the threshold of the congestion window, and then be forced to wait to receive the ACKs before transmitting the next batch of packets.

1. The network configuration is similar to that in Parts 5-7. If the network is not setup accordingly, then follow the instructions in Exercise 5-a. The following two steps are modifications to the setup of Exercise 5-a.

2. Set a new default gateway of PC1 to Router1. If the default gateway from Table 5.4 is still set, you must first delete the existing entry. Use the command `netstat -rn` to see if a default gateway is configured. Assuming that the configuration from Table 5.4, you must enter the following commands:

```
PC1% route del default gw 10.0.1.3
PC1% route add default gw 10.0.1.1
```

The default gateway of PC2 remains unchanged and should be as shown in Table 5.4. With this modification, traffic from PC1 to PC2 passes through Router1 and Router2, and traffic from PC2 to PC1 passes through Router4 and Router3. Verify that this is the case.

3. Set the data rate of the emulated serial link to 1 Mbps.

Exercise 8-b. Observing TCP congestion control

This exercise is similar to Exercise 6-a, that is, PC1 transmits TCP segments to PC2.

1. Start Wireshark for interface *eth0* on PC1, and start to capture traffic. Set a display filter to TCP traffic.

2. Start a nttcp receiving process on PC2:

```
PC2% nttcp -i -rs -l1000 -n5000 -p4444
```

3. Start a nttcp sending process on PC1 that transmits 5000 blocks of data, each with 1000 Bytes:

```
PC1% nttcp -ts -l1000 -n5000 -p4444 -D 10.0.2.22
```

4. Once Wireshark has transmitted at least one hundred TCP packets, disconnect the cable that connects *FastEthernet0/0* of Router1 to the Ethernet hub. Disconnect the cable at the hub. Now reconnect the cable, and wait until the transmission resumes. Repeat this for a few times, varying the durations when the cable is disconnected.

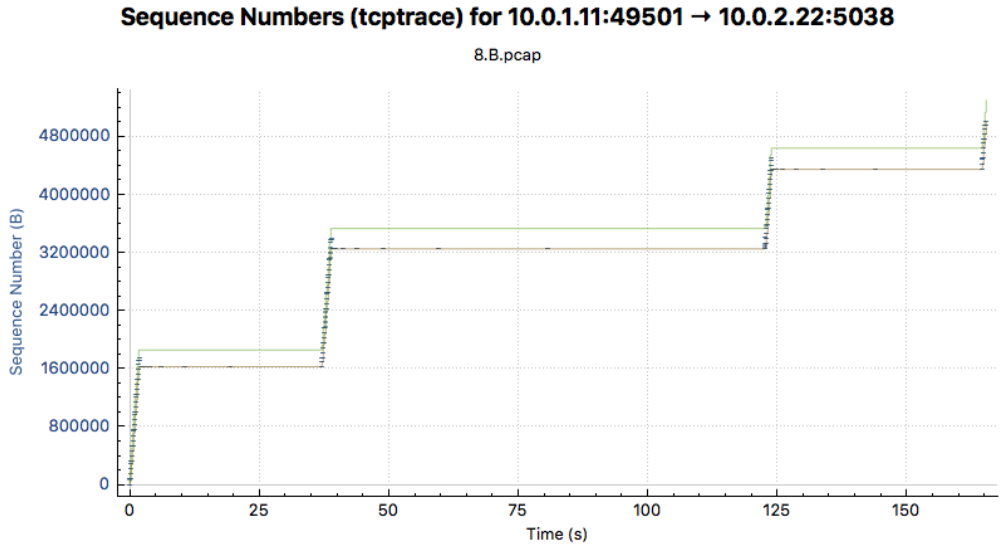
5. Use the instructions from Exercise 6-c to build a Time-Sequence Graph (tcptrace) in Wireshark for the TCP connection. Study the graph at the time instants when the cable is reconnected and TCP sender resumes transmission. Use the navigating features to zoom in to parts of the graph that are of interest.

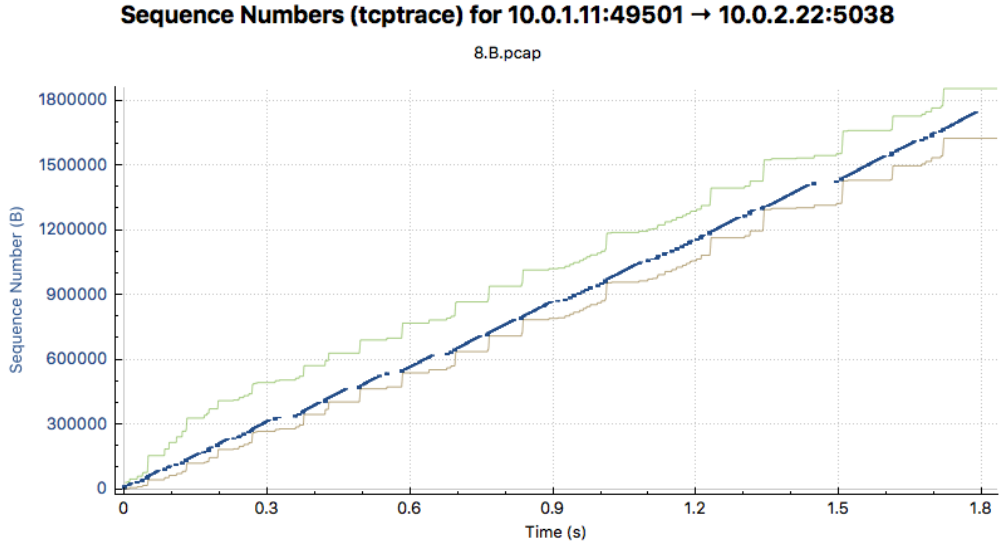
The outcome of this experiment is dependent on the data rate of the link between Router1 and Router2, and Router3 and Router4, respectively. The outcome of the experiment is different when the Ethernet link between Router1 and Router2 is running at 10 Mbps or at 100 Mbps. The above settings are optimized for a 10 Mbps Ethernet link between Router1 and Router2.

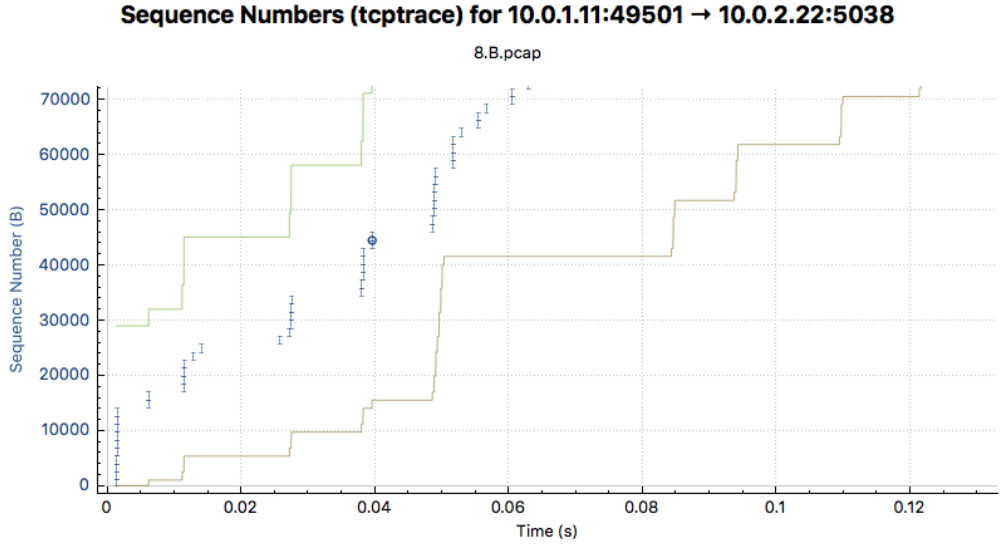
6. Follow the instructions from Exercise 6-c to generate image files for the Time- Sequence Graph (tcptrace). Save enough images so that you can use the graphs to answer the above questions in your lab report. Make sure you include an image that shows a portion of the graph that illustrates the slow start phase. Compress the files and copy the files to a floppy disk.

Question 8.B)

Include the answers to the questions from Step 5. Use the saved image files to support your answers. Annotate the events in this graph, and explain the events that you observe, e.g. segments dropped, retransmission, congestion window, slow start, congestion avoidance, fast recovery, etc.


Sequence Numbers (tcptrace) for 10.0.1.11:49501 → 10.0.2.22:5038
8.B.pcap
Sequence Number (B)
4800000
4000000
3200000
2400000
1600000
800000
0
0
25
50
75
100
125
150
Time (s)


Sequence Numbers (tcptrace) for 10.0.1.11:49501 → 10.0.2.22:5038
8.B.pcap
Sequence Number (B)
1800000
1500000
1200000
900000
600000
300000
0
0
0.3
0.6
0.9
1.2
1.5
1.8
Time (s)


Sequence Numbers (tcptrace) for 10.0.1.11:49501 → 10.0.2.22:5038
8.B.pcap
Sequence Number (B)
70000
60000
50000
40000
30000
20000
10000
0
0
0.02
0.04
0.06
0.08
0.1
0.12
Time (s)

Question 8.B.5.a)

Try to observe periods when the TCP sender is in a slow start phase and when the sender switches to congestion avoidance. Verify that the congestion window follows the rules of the slow start phase.

At first, the sender sends 10 packets, so the cwnd is set to 10 initially. When a first ACK is received, two packets are sent because the cwnd grew by 1. Then, 3 ACKs are received more or less at the same time, and 6 packets are sent out: the cwnd just grew by 3. This behavior is characterized by slow start.

Question 8.B.5.b)

Can you deduct the size of the ssthresh parameter during the times when the congestion window is small?

Question 8.B.5.c)

Can you find occurrences of fast recovery?