

Computer Networks Lab

based on

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

and on

**Computer Networking - A Top-Down Approach
by James Kurose and Keith Ross**

*Adapted for
'Computernetwerken'
by Johan Bergs and Stephen Pauwels*

Tim Verhaegen
Pim Van den Bosch
Group 12

March 30, 2023

Lab 2

Transport Layer Protocols: User Datagram Protocol (UDP) and Transmission Control Protocol (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 analyse measurements of a TCP connection
- How TCP performs retransmissions
- How TCP congestion control works
- How (un)fair bandwidth is divided between multiple TCP and UDP streams

2.1 Lab 2

This lab explores the operation of the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP), two of the major 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 (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, a TCP receiver informs the TCP sender how much data it may 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. Part 1 compares the performance of data transmissions in TCP and UDP. Part 2 explores how TCP and UDP deal with IP fragmentation. The remaining parts address important aspects of TCP. Part 3 explores connection management, Parts 4 and 5 look at flow control and acknowledgements, Part 6 explores retransmissions, and Part 7 is devoted to congestion control. Finally, Part 8 looks into fairness between different UDP and TCP streams.

Part 1. Learning how to use iperf

The `iperf3` command is a tool used to generate UDP and TCP traffic loads. Together with `ping6` and `traceroute6`, `iperf3` is an essential utility program for debugging problems in IP networks. Running the `iperf3` tool consists of setting up a `iperf` server (receiver) on one host and then a `iperf` client (sender) on another host. Once the `iperf` client is started, it sends the specified amount of data as fast as possible to the `iperf` server.

Note that both the `iperf` and the `iperf3` commands exist. For the remainder of the lab exercises **do not use** `iperf`, **but always use** `iperf3`, unless specifically stated otherwise!

Some useful `iperf3` commands:

- `iperf3 -s`
Starts an `iperf` server that listens for traffic on the default `iperf` port.
- `iperf3 -c fc00::1`
Starts an `iperf` client that sends TCP traffic to `fc00::1` on the default port for 10 seconds.

Refer to the manual page of `iperf3` for more options.

By default, `iperf` transmits data over a TCP connection. The `iperf` client opens a TCP connection to an `iperf` server, transmits data and then closes the connection. The `iperf` server must be running when the `iperf` client is started. UDP data transfer is specified with the `-u` option.

Using the `-l [bytes]` option tells `iperf` to send bursts with a set amount of bytes. UDP sends these bursts as soon as they are presented to the Linux kernel. For TCP packets, the Linux kernel can wait a small period of time to group more data together.

The network setup in Figure 2.1 and Table 2.1 is used in part 1 of this lab.

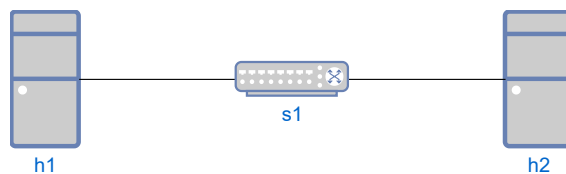


Figure 2.1: Network Topology for Part 1.

Linux PC	Ethernet Interface eth0
h1	fc00:0:0:1::1/64
h2	fc00:0:0:1::2/64

Table 2.1: IP Addresses of the hosts.

Exercise 1: Network setup

1. Write a mininet python script to create the topology as shown in Figure 2.1, and save it as [L2-1-1.py](#). Configure the IP addresses of the interfaces as given in Table 2.1.

2. Run the mininet script.
3. Verify that the setup is correct by issuing a `pingall` command in mininet.

Exercise 2: Transmitting data with UDP

This exercise consists of setting up a UDP data transfer between two hosts, h1 and h2, and observe the UDP traffic.

1. Open xterm terminals on both h1 and h2.
2. On h1, start Wireshark and start capturing packets on the `h1-eth0` interface.
3. On h2, start an iperf server.
4. On h1, start a iperf client that transmits UDP traffic to h2. Limit the length of the buffer to send to 500 bytes, and limit the total amount of bytes to send to 10000.
5. Stop the Wireshark capture on h1 and save the captured traffic to `L2-2-1.pcapng`.
6. For the following questions, filter out UDP traffic with a display filter and ignore the first small 4 byte packet or packets; this is a sort of "handshake" that iperf3 is performing and has nothing to do with the actual data transfer.

Use the data captured with Wireshark in [L2-2-1.pcapng](#) to answer the questions. Support your answers with the saved Wireshark data.

L2-2-1 How many packets are exchanged in the data transfer? How many Internet Protocol (IP) packets are transmitted for each UDP datagram? What is the size of the UDP payload of these packets?

/1

20 packets are exchanged in the data transfer. A single IP packet is sent per UDP datagram. Each of these packets have an UDP payload size of 500 bytes.

L2-2-2 Compare the total number of bytes transmitted, in both directions, including Ethernet, IP, and UDP headers, to the amount of application data transmitted.

/1

A packet contains a total of 562 bytes including headers. Since the UDP payload has a size of 500 bytes, we have an overhead of 11.24

L2-2-3 Explain the fields in the UDP headers.

/1

We have the following fields:

1. source and destination port: used for multiplexing data to/from application layer

- 2. a checksum: used to check for bit errors
- 3. length: length of the UDP headers + UDP payload

L2-2-4 Observe the port numbers in the UDP header. How did the iperf client select the source port number?

/1

The port number is: 55407. It was randomly chosen from a list of free ports, by the operating system.

Part 2. IP Fragmentation of UDP traffic

In this part of the lab, you observe the effect of IP fragmentation on UDP 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 typically is 1500 bytes. If an IP datagram exceeds the MTU size, an error message is reported and the original packet has to be split into smaller packets.

When a UDP datagram is fragmented, its payload is split into multiple IP packets, each satisfying the limit imposed by the MTU. Each fragment is an independent IP packet, and is routed in the network independently from the other fragments. Fragmentation only happens at the sending host. Intermediate hops are not allowed to fragment IPv6 packets (in IPv4 this is different - this will be shown in another lab). 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.

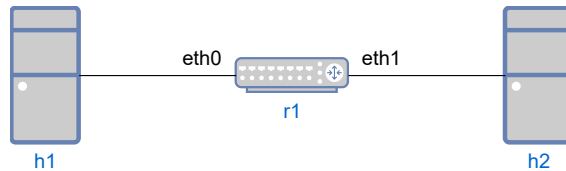


Figure 2.2: Network Topology for Part 2.

Linux PC	Ethernet Interface eth0	Ethernet Interface eth1
h1	fc00:0:0:1::1/64	N/A
h2	fc00:0:0:2::2/64	N/A
r1	fc00:0:0:1::10/64	fc00:0:0:2::10/64

Table 2.2: IP Addresses of the hosts and router.


You explore the issues with IP fragmentation of UDP transmissions in the network configuration shown in Figure 2.2, with h1 as sending host, h2 as receiving host, and r1 as intermediate IP router.

Exercise 3: UDP and Fragmentation

In this exercise you observe IP fragmentation of UDP traffic. In the following exercise, use `iperf3` to generate UDP traffic between h1 and h2, across IP router r1.

1. Write a python script that sets up the mininet topology shown in 2.2. Make sure you configure the IP addresses as shown in table 2.2. Save the script to [L2-3-1.py](#).
2. Start mininet and verify that the network is configured correctly by issuing a `pingall`.
3. Start Wireshark on the `h1-eth0` interface of h1 and start to capture traffic. Do not set any filters.
4. Use `iperf3` to generate UDP traffic between h1 and h2. The connection parameters are selected so that IP Fragmentation does not occur initially:
 - Start an `iperf3` server on h2.
 - On h1, start an `iperf3` client that transmits UDP traffic to h2. Limit the length of the buffer to send to 1000 bytes, and limit the total amount of bytes to send to 10000.

5. Next, try sending larger datagrams by issuing the previous `iperf` command again, but this time, instead of 1000, choose 2000 as buffer length.
6. Stop the traffic capture on h1 and save the Wireshark output to `L2-3-1.pcapng`.

 Wireshark reassembles fragmented IP packets by default. In order to actually see what is going on without Wireshark reassembling, verify the following settings in Wireshark (you should keep the same settings for all future labs as well):

- Preferences -> Advanced -> `ipv6.defragment`: **FALSE**.
- Preferences -> Advanced -> `ip.defragment`: **FALSE**.
- Preferences -> Protocols -> IPv6 -> Reassemble IPv6 datagrams: **OFF**.
- Preferences -> Protocols -> IPv4 -> Reassemble IPv4 datagrams: **OFF**.

Use the data captured with Wireshark in `L2-3-1.pcapng` to answer the questions. Support your answers with the saved Wireshark data.

L2-3-1 From the saved Wireshark data, select one UDP datagram that is fragmented. For each fragment of this datagram, determine the values of the fields in the IP fragmentation header.

/1

An example of a fragmented UDP packet is packet number 74. Together with packet 74 itself, packet number 75 forms the entire UDP packet.

For packet 74, the values of the headers are:

1. Next Header: UDP
2. Reserved octet: 0x00
3. Offset: 0
4. Reserved bits: 0
5. More Fragments: Yes
6. Identification: 0x3cd4c2b1

For the fragment in packet 75, the values of the headers are:

1. Next Header: UDP
2. Reserved octet: 0x00
3. Offset: 181
4. Reserved bits: 0
5. More Fragments: No
6. Identification: 0x3cd4c2b1

L2-3-2 What is the maximum size of an UDP datagram that does not require fragmentation? What is the amount of actual data that is transmitted in such a datagram?

/1

The maximum size of an UDP datagram without fragmentation is equal to the MTU of the underlying link layer minus the size of IP headers. If we have Ethernet as our link layer (MTU of 1500), and use IPv6 (40 bytes header), we have 1460 bytes left for the UDP datagram. Since UDP has 8 bytes of headers, 1452 bytes are used for the payload.

L2-3-3 For the iperf session that resulted in fragmented datagrams: How large is the first datagram? How large is the second? How much application data is in both fragments?

/1

The first datagram has a size of 1496 bytes (UDP headers + UDP payload + IP header), the second one has a size of 600 bytes (IP header + fragment data section). Combined there is 2000 bytes of data in these packets.

L2-3-4 Why don't intermediate routers reassemble fragmented IP packets?

/1

IP fragments/packets do not necessarily go through the same router. It would also result in a sizeable and unnecessary overhead.

Exercise 4: The effect of fragmentation on performance.

For this exercise, we start from the same topology as we used in the previous exercise. However, in order to avoid dealing with very large trace files, we will modify the topology a bit so the links between the router and the hosts is capped to 1Mbps. You can do this by supplying an extra argument `bw=X` to the `addLink` methods in your python script, where X is the bandwidth of the link expressed in Mbps:

```
| addLink(h1, r1, bw=1)
```

1. Modify your python script so the links are set to 1Mbps and save it as [L2-4-1.py](#).
2. Once again, start mininet and verify the connectivity with the `pingall` command.
3. Start an iperf server on h2.
4. Start capturing packets with wireshark on h1.
5. On h1, start an iperf session to h2. The parameters for the iperf session this time are:
 - Make iperf run for 10 seconds and then stop.
 - Set the target bandwidth to 1Mbps. This ensures we will completely fill the link to its maximum capacity.

- Use UDP
 - Limit the buffer length to 1452.
6. Save the output of the iperf server to L2-4-1.txt and the wireshark trace to L2-4-1.pcapng.
 7. Start a new wireshark session on h1.
 8. Run the same iperf command again on h1, but this time use a buffer length of 1453.
 9. Save the output of the iperf server to L2-4-2.txt and the wireshark trace to L2-4-2.pcapng.
 10. Again start a new wireshark session on h1, and run the same iperf command, but now use 2778 as buffer length.
 11. Save the output of the iperf server to L2-4-3.txt and the wireshark trace to L2-4-3.pcapng.
 12. Finally, repeat the same steps again to run an iperf session with a buffer length of 2115. Save the outputs again to L2-4-4.txt and the wireshark trace to L2-4-4.pcapng.

Use the data captured in [L2-4-1.txt](#), [L2-4-2.txt](#), [L2-4-3.txt](#), [L2-4-4.txt](#), [L2-4-1.pcapng](#), [L2-4-2.pcapng](#), [L2-4-3.pcapng](#) and [L2-4-4.pcapng](#) to answer the questions. Support your answers with the saved Wireshark data and iperf output.

L2-4-1 Look at the summary line of iperf3 that displays the throughput throughout the entire 10 second session. Look at the **receiver** report! Compare the throughput of the four iperf sessions. How much difference is there between the biggest and lowest reported throughput?

/1

In iperf3 sessions 1, and 3, the throughput is around 950 on average. The throughput of sessions 2 and 4 is slightly lower with an average of around 900.

L2-4-2 Explain why not every session results in the same throughput. Use your wireshark traces to support your answer.

/1

There is no fragmentation in the first session due to the size of the UDP datagram that we send. We can see this in packet 22 in our trace: [L2-4-2.pcapng](#). On the other hand, in the second session, very small UDP payloads are sent, causing the headers to be more of a significant overhead.

L2-4-3 One of the sessions will have the lowest reported throughput. Why that specific session?

/1


Session two has the lowest throughput as the UDP datagrams are split up into two due to fragmentation, and the final fragment contains very little data compared to headers.

Part 3. TCP and iperf

Up until now, we have been using `iperf3` over UDP. The next parts of the lab will focus on the behaviour of TCP. As you will see, TCP is a *connection-oriented* protocol, in contrast to UDP, which is *connectionless*. Moreover, TCP provides reliable, ordered and error-checked delivery of data. In the next exercises, all these concepts will be touched upon. We will start by running a simple `iperf` session, as we did with UDP.


Exercise 5: Transmitting data with TCP

Use the same topology as shown in figure 2.2 and table 2.2.

 **In order to interpret the results in this exercise correctly, it is necessary to make another slight modification to the python script!** Right after the `net.start()` line, insert the following line: `net["h1"].cmd("ethtool -K h1-eth0 tso off")`. Save your modified script to `L2-5-1.py`.

The reason for this change is because by default *TCP segmentation offload* or *tso* is enabled on Linux systems. This is a technology where (parts of) the TCP/IP processing is offloaded to the network controller. This technique reduces the processing overhead of the network stack on the CPU. However, when it is enabled, your operating system typically “sees” (much) larger packets than are actually sent on the network hardware (less, but larger packets = less processing overhead). The network card then does the *actual* processing. This can be misleading when you look at your Wireshark traces because you don’t see what is actually happening on the network. You see what the CPU of your machine *thinks* is happening...

Therefore, it is recommended to turn this technique off for exercises where this matters. For the remainder of the course, when it is mentioned to turn off *tso* or TCP segmentation offload, you are expected to insert a similar line in your python scripts. When nothing is mentioned, turning off *tso* is not required (but it is OK if you always turn it off by default in your scripts).

 This is one of the few exercises where you **must use iperf instead of iperf3**. Read up on the man page of `iperf` as the syntax is slightly different from `iperf3`!

1. On h1, start Wireshark and start capturing packets on the network interface of h1.
2. On h2, start an `iperf` server.
3. Start a `iperf` client on h1 that transmits TCP packets to h2. Limit the total amount of bytes to send to 10000.
4. Stop the Wireshark capture on h1, and save the captured traffic to `L2-5-1.pcapng`.

Use the data captured with Wireshark in **L2-5-1.pcapng** to answer the questions.

L2-5-1 How many packets contain application data? How many bytes of application data do those packets contain?

/1

If we filter the traces by applying `"tcp.srcport == 47492"` we can see the relevant packets that contain the application data.

If we go to `statistic>conversations>tcp` and we limit according to the filter we can see that there were 925 packets sent.

L2-5-2 Compare the total number of bytes transmitted, in both directions, including Ethernet, IP, and TCP headers, to the amount of application data transmitted.

/1

The total amount of data sent both ways is quite a bit larger than just the data application being transmitted. The overhead is large.

L2-5-3 Inspect the TCP headers. Which packets contain flags in the TCP header? Which types of flags do you observe and what is their purpose?

/1

At the start we see SYN, to initialize the TCP connection, afterwards it's mostly ACK's, which indicate that the process is running as expected.

L2-5-4 Compare the total amount of bytes transmitted in this TCP data transfer, and the UDP data transfer in exercise 1.

/1

As expected the total amount of bytes transmitted by TCP is larger than UDP.

Part 4. TCP and Fragmentation

Exercise 6: TCP and Fragmentation


TCP avoids 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 that is known as Path MTU Discovery, and that works as follows: when a router can't forward a packet because it is larger than the MTU of the link, it discards the packet and generates an Internet Control Message Protocol (ICMP) error message of type "Packet Too Big" (in IPv6) or "Fragmentation needed; DF bit set" (in IPv4). 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.

Linux PC	MTU size of eth0	MTU size of eth1
h1	1500	N/A
h2	1280	N/A
r1	1500	1500

Table 2.3: MTU Sizes

For this exercise, continue to use the same python script as in the previous exercise. Once you have started up the script, perform the steps mentioned below.

 You should switch back to using `iperf3` for this exercise!

1. Modify the MTU of the interfaces with the values as shown in Table 2.3. In Linux, you can view the MTU values of all interfaces in the output of the `ifconfig` command. For example, on h2, you type:

```
| h2% ifconfig
```

The same command is used to modify the MTU value. For example, to set the MTU value of interface h2-eth0 on h2 to 1280 bytes, use the `ifconfig` command as follows:

```
| h2% ifconfig h2-eth0 mtu 1280
```

2. Start Wireshark on h1 and start to capture traffic with no filters set.
3. Start an `iperf` server on h2, and an `iperf` client on h1 that sends TCP packets to h2. Limit the duration of the `iperf` session to 1 second.
4. Save the Wireshark output to `L2-6-1.pcapng`.
5. Now change the MTU size on interface eth1 of r1 to 1280 bytes. Change the MTU size of interface eth0 on h2 to 1500 bytes.

6. Start Wireshark on h1 again.
7. Repeat the iperf transmission in step 3.
8. Save the Wireshark output to L2-6-2.pcapng.

! When answering the following questions, take into account that `iperf3` actually sets up **two** TCP connections: the first one is always a control connection over which client and server exchange statistics etc. This TCP connection is also present when you perform a UDP iperf. In the case of a TCP `iperf3`, the *second* TCP connection is the one that is doing the actual data transmission!

Use the data captured with Wireshark in [L2-6-1.pcapng](#) to answer the following questions.

L2-6-1 Do you observe fragmentation before changing the MTU on r1? If so, where does it occur? If not, why not? Explain your observation. /2

No fragmentation is observed. Since Path MTU Discovery is used, it is possible to find the smallest MTU size along the network path before sending packets. In this case, h1 would discover that the path to h2 has an MTU size of 1280 bytes and would send packets with a size equal to or less than 1280 bytes, avoiding fragmentation.

Use the data captured with Wireshark in [L2-6-2.pcapng](#) to answer the following questions.

L2-6-2 Do you observe fragmentation after changing the MTU on r1? If so, where does it occur? Explain your observation. /1

Unlike IPv4, IPv6 does not allow routers to fragment packets. Instead, when a router receives a packet that is too large to be forwarded over the next link with a smaller MTU, it drops the packet and sends an ICMPv6 "Packet too big" message back to the sender. We can find multiple such "Packet too big" messages in the trace.

L2-6-3 Describe how Path MTU Discovery is performed. /1

It's performed as follows:

- h1 sends a big packet to h2.
 - r1 sees the packet is too big for the link to h2.
 - r1 drops the packet and tells h1 the maximum allowed size (1280 bytes).
 - h1 adjusts packet size to 1280 bytes or less.
 - h1 sends smaller packets to h2, avoiding the need for fragmentation.
-

Part 5. 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 ssh to set up a TCP connection and observe the control packets that establish and terminate a TCP connection. The experiments involve h1 and h2 in the same network as shown in Figure 2.2. You can continue to use the python script from the previous exercise.

Exercise 7: 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.

1. This part of the lab only uses h1 and h2 in the network configuration in Figure 2.2. Verify that the MTU values of all interfaces of h1, and h2, and r1 are set to 1500 bytes, which is the default MTU for Ethernet networks.
2. Start Wireshark on h1 to capture traffic of the ssh connection. Do not set any filters.
3. Start up the ssh daemon on h2 as follows:

```
| h2% /sbin/sshd
```
4. Establishing a TCP connection: Establish a ssh session from h1 to h2. Use the username "computernetwerken" and the password for the VM to log in on h2.
5. Closing a TCP connection (initiated by client): On h1, type `exit` in the ssh session to terminate the connection.
6. Save the Wireshark output as `L2-7-1.pcapng`.

Use the data captured with Wireshark in [L2-7-1.pcapng](#) to answer the following questions.

L2-7-1 Identify the packets of the three-way handshake (refer to the frame numbers). Which flags are set in the TCP headers? Explain how these flags are interpreted by the receiving TCP server or TCP client.

/1

The packets relevant to the three way handshake in our trace are packets 2,3 and 4. The flags set in these packet's TCP headers are SYN, SYN-ACK and ACK respectively.

- SYN flag: Indicates that the sender wants to establish a connection. The receiving TCP server (h2) interprets this as a connection request and responds with a SYN-ACK packet.
- SYN-ACK flag combination: Indicates that the server (h2) acknowledges the connection request and sends its own synchronization information. The receiving TCP client (h1) interprets this as an acceptance of the connection request and responds with an ACK packet.
- ACK flag: Indicates that the client (h1) acknowledges the server's (h2) synchronization information. The receiving TCP server (h2) interprets this as the completion of the three-way handshake, and the TCP connection is considered established.

L2-7-2 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 chosen by h1 and h2? How is this initial sequence number chosen?

/1

The initial sequence number chosen by h1 is 2997518125 and for h2 it is 3158495452. Initial sequence numbers can be random or based on a system algorithm to enhance security. Modern OS use various factors to generate unpredictable sequence numbers.

L2-7-3 Identify the first packet that contains data. What is the sequence number used in the first byte of data sent from the TCP client to the TCP server?

/1

The first packet containing data is packet no 5. The sequence number is 2997518125. The sequence number for the first byte of data sent from the TCP client (h1) to the TCP server (h2) will be the initial sequence number from the SYN packet plus 1. This is because the SYN flag consumes one byte in the sequence number space, even though it doesn't carry actual data.

L2-7-4 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.

/1

The last 4 packets in the trace are the packets involved in closing the connection. The flags in these packets are FIN, ACK, FIN and ACK respectively. The process of closing a TCP connection is called the four-way handshake, which consists of the following steps:

- FIN: The client (h1) sends a packet with the FIN flag set to indicate that it has finished sending data and wants to close the connection.
- ACK: The server (h2) sends a packet with the ACK flag set to acknowledge the receipt of the FIN packet from the client.
- FIN: The server (h2) sends a packet with the FIN flag set to indicate that it has also finished sending data and wants to close the connection.

- ACK: The client (h1) sends a packet with the ACK flag set to acknowledge the receipt of the FIN packet from the server.
-

Part 6. Retransmissions in TCP

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

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

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

2. Multiple ACKs have been received for the same segment. A duplicate 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 still the same network as shown in Figure 2.2.

Exercise 8: TCP Retransmissions

The purpose of this exercise is to observe when TCP retransmissions occur. As before, you transmit data from h1 to h2. Here, data is sent over a link that is set to 1Mbps and will create a 1% packet loss. When a packet gets lost, you will see that it gets retransmitted.

1. Modify the python script you used in the previous exercise by adding the `loss=1` to every link, similar to setting the bandwidth cap earlier. Also for this exercise, double-check that the bandwidth is still capped to 1 Mbps. Save your script to `L2-8-1.py`.

2. Start Wireshark on h2 and capture traffic on interface h2-eth0. Set a display filter to TCP traffic.
3. Make sure an iperf server is running on h2.
4. On h1, start an iperf client that sends 10 seconds worth of data to h2.
5. When the data transfer finishes, stop the Wireshark capture and save it to L2-8-1.pcapng

Refer to examples in the saved Wireshark data (L2-8-1.pcapng). Include images of graphs where possible.

When you scroll through the wireshark trace, you will find (probably several) places where re-transmits are occurring. You can easily identify these places because wireshark will mark these lines in black and red. Refer to one such an occurrence in your wireshark trace to answer the next questions.

L2-8-1 Which packet(s) were lost?

/1

Packet 169 has the "TCP Previous segment not captured" label, this indicates that a packet with the sequence number expected by the receiver was not present in the capture.

L2-8-2 Which packet(s) were retransmitted?

/1

We can see that the packet 169 is retransmitted a bit later in in packet 179, the label of this packet is aptly called "TCP Retransmission"

L2-8-3 How does the sender know the receiver missed one (or more) packets?

/1

Before the package is retransmitted there are a number of Duplicate ACKS (packets 172, 174, 176, 178) coming from the receiver letting the sender know that there is something wrong. Eventually when the sender receives three duplicate ACKs, it assumes that a packet was lost and triggers a TCP Retransmit

L2-8-4 Are only lost packets retransmitted, or are there also other packets that are retransmitted? Why or why not?

/3

There are 3 cases where packets may be retransmitted:

- RTO: Sender resends if no acknowledgment within timeout, even if not actually lost.
- Out-of-order: Receiver sends duplicate ACKs for last in-order packet. 3 duplicate ACKs trigger Fast Retransmit, but may result in unnecessary retransmissions.

- Spurious Retransmissions: Resending may occur due to network issues, even if packet correctly received. Caused by factors like latency spikes, congestion, or sender misinterpretation.
-

Part 7. 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). 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 and to see the slow start and congestion avoidance phases of a TCP session.

Exercise 9: Observing TCP congestion control

The network configuration used is still that in Figure 2.2. To observe the slow start features, we will once again apply slight modifications to the python script of the previous exercise to make things easier to follow. We will keep the 1 Mbps bandwidth limit, but we remove the 1% packet loss that was introduced in the previous exercise. We will now also add a large delay on the links, so that the slow start phase will be lasting longer and is easier to observe. We are going to add 240 ms of delay to the link between r1 and h2, which is more or less equivalent to what would be expected if traffic is routed over a geostationary satellite. You could think of h1 and h2 as ground stations, and r1 as the satellite in this scenario. In order to add delay to a link, similar to capping the bandwidth, we need to add another parameter to that link: `delay="240ms"`.

When h1 sends a data segment to h2, the ACK for that segment will take about 0.5 seconds to arrive to h1 (240ms delay for the data segment, and the same for the ACK). This spaces out the packets nicely so slow start is easier to follow.

1. Modify the python script for the topology so the link between r1 and h2 has a delay of 240 ms, and neither of the links have the 1% packet loss. Save it to `L2-9-1.py`.
2. Start a wireshark capture on h1.
3. Start an iperf server on h2.
4. Start an iperf client on h1 that transmits 10 seconds worth of data to h2.
5. Save your wireshark trace to `L2-9-1.pcapng`.

Refer to examples in the saved Wireshark data ([L2-9-1.pcapng](#)). Include images of graphs where possible.

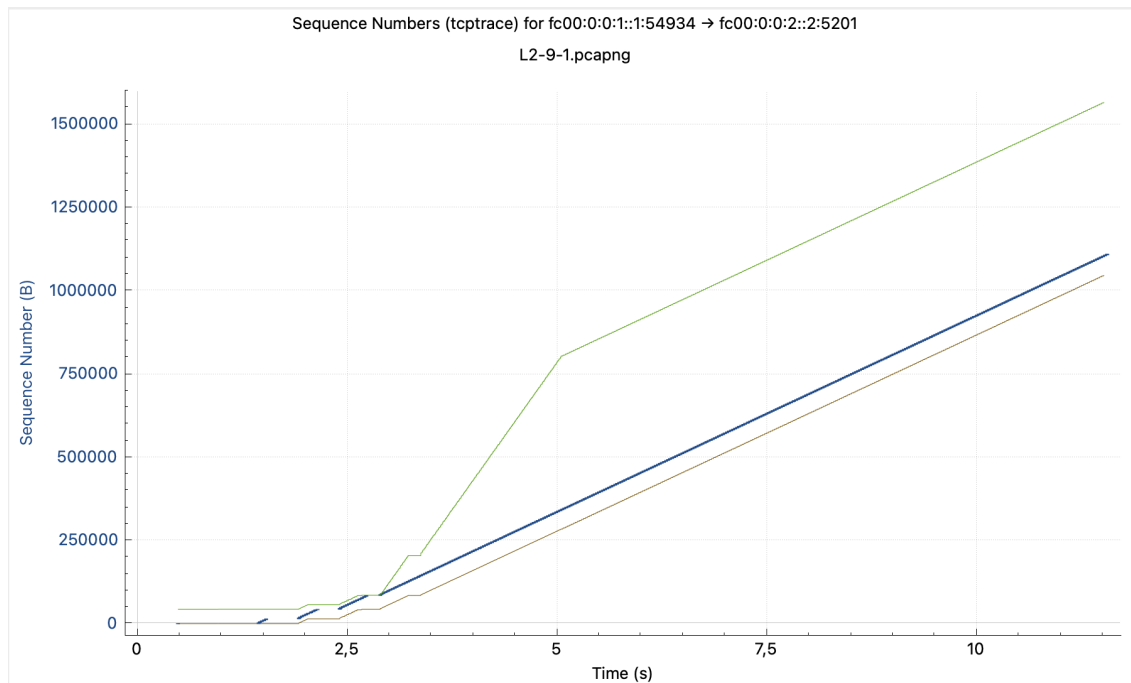


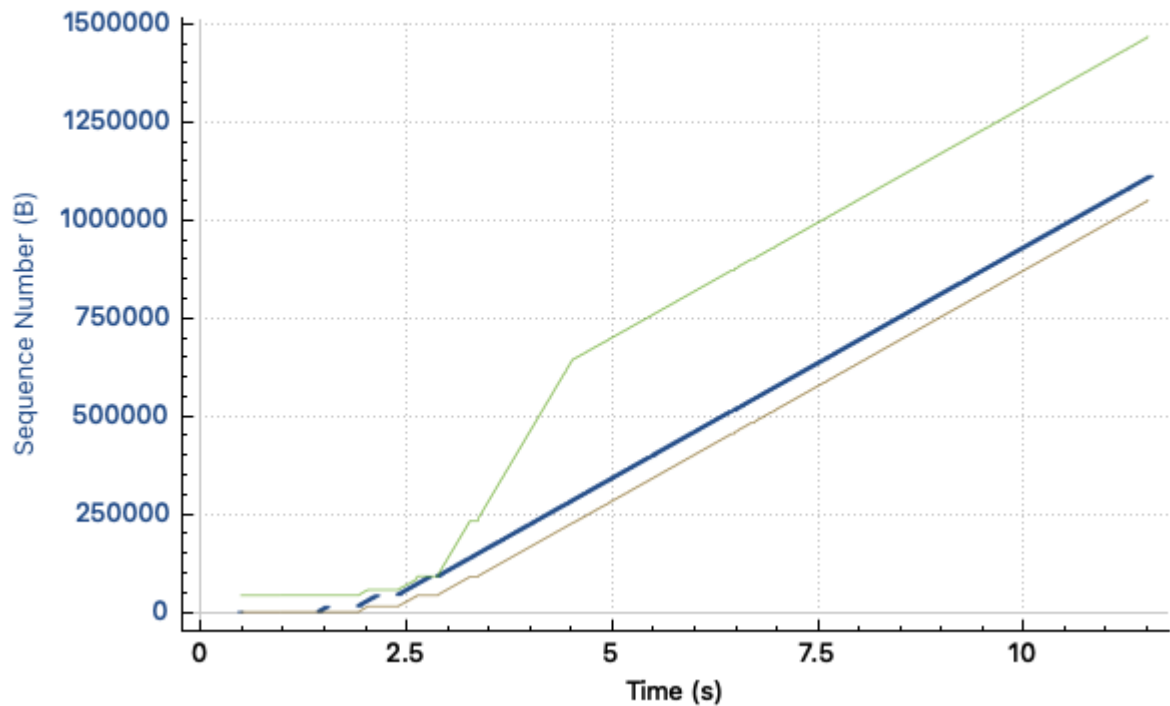
Figure 2.3: tcptrace graph.

L2-9-1 In your wireshark trace, select a packet that belongs to the TCP iperf session. Select a packet that is sent from h1 towards h2. Next, select *Statistics -> TCP Stream Graphs -> Time Sequence (tcptrace)*. You should see a graph like in Figure 2.3. If you don't see a similar graph, check if clicking the "Switch Direction" button helps. You can zoom in on the start of the graph to have a closer look at the slow start phase of the TCP session. Export the graph as `tcptrace.png`. and include it in your report. You can also include extra images of zoomed-in parts of the graph to highlight any details. What is the meaning of the curves on the graph? What do they represent? Using the graph, can you explain how slow start works?

/2

Sequence Numbers (tcptrace) for fc00:0:0:1::1:59508 → fc00:0:0:2::2:5201

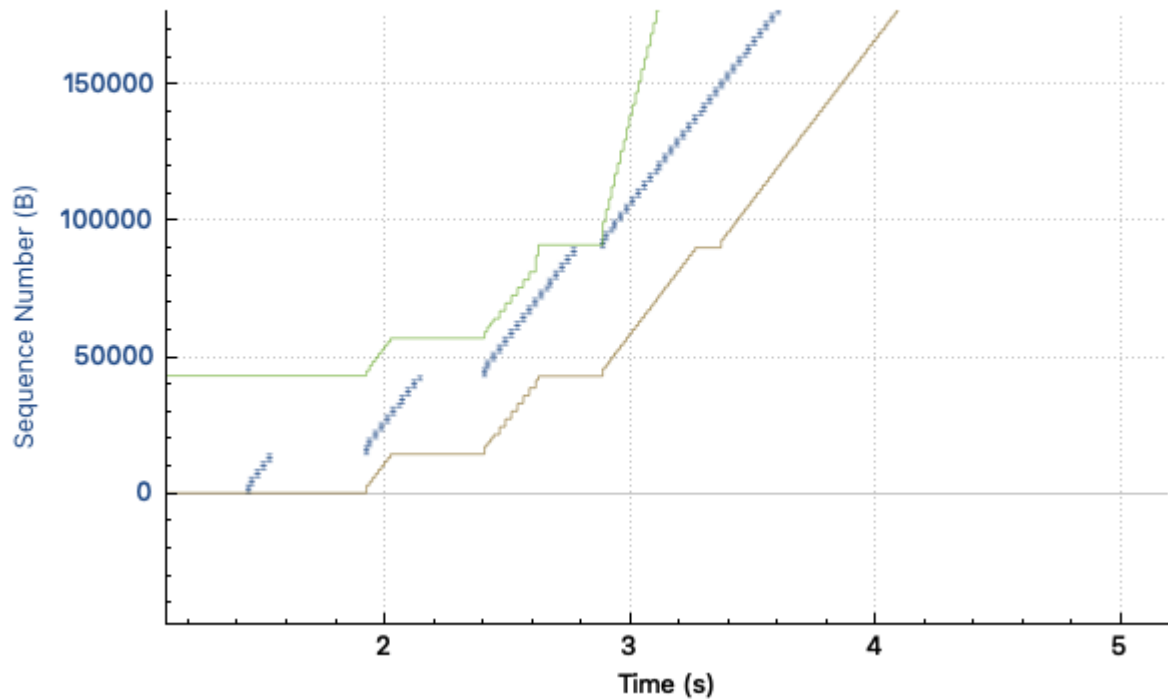
L2-9-1.pcapng



In Wireshark's TCP graph, blue indicates data sent, green shows data acknowledged, and orange represents duplicate acknowledgments due to out-of-order packets. TCP's slow start phase gradually increases the sender's rate until congestion occurs. The graph displays a sawtooth pattern, with the blue line representing the sending rate increasing and decreasing as congestion is detected and resolved.

Sequence Numbers (tcptrace) for fc00:0:0:1::1:59508 → fc00:0:0:2::2:5201

L2-9-1.pcapng



L2-9-2 Using the graph, indicate where the behaviour of the slow start phase changes. How can you tell? What is happening differently after the change?

/2

To identify slow start phase changes on the graph, observe the blue line's slope change. Initially, it increases gradually, but then it becomes less steep or even decreases, indicating the sender is experiencing congestion and adjusting its rate accordingly. This slower rate helps maintain network stability and avoid packet loss.

Part 8. Fairness

Exercise 10: Create and perform your own experiments

For this exercise you are going to create your own experiments that showcase the workings of fairness with TCP and UDP.

! When you create this experiment, it is very important that, on **any link** in your network where you apply **bandwidth limitations**, you also need to add another, extra parameter to the link: `enable_red=True`.

Without that extra parameter, you will encounter a phenomenon known as “Bufferbloat”, which is a well-known cause of high latency and jitter in packet-switched networks caused by excess buffering of packets. TCP uses the occurrence of packet drops to determine the available bandwidth between two hosts. The algorithms speed up the data transfer until packets start to drop, then slow down the transmission rate. Ideally, they keep adjusting the transmission rate until it reaches an equilibrium speed of the link. With a large buffer that has been filled, the packets will arrive at their destination, but with a higher latency. The packets were not dropped, so TCP does not slow down once the uplink has been saturated, further filling the buffer. Newly arriving packets are dropped only when the buffer is fully saturated. To TCP, a congested link can appear to be operating normally as the buffer fills. The TCP algorithm is unaware the link is congested and does not start to take corrective action until the buffer finally overflows and packets are dropped.

A nice article, written by Jim Gettys and Kathleen Nichols, that digs deeper into this problem has been published in the “Communications of the ACM” magazine in January 2012, and can be found on <https://cacm.acm.org/magazines/2012/1/144810-bufferbloat/fulltext>

How the extra `enable_red=True` parameter solves this problem, is out of scope for this exercise.

Your experiment must consist of the following aspects:

1. A network topology to run the experiments on.
2. A description of the steps to perform in order to set up the experiment.
3. The experiment steps itself.
4. A conclusion about what you noticed and a callback to the theory about fairness.

As more than one experiment might be needed, steps 2 and 3 can be repeated as much as required.

Make sure you showcase the following situations within your experiments:

1. What happens when multiple TCP connections are present?
2. What happens when multiple connections and at least one UDP stream are present?
3. How does fairness work with two TCP connections, where every connection has a different round trip time (RTT)? Can you illustrate this?

When you answer this question, make sure that you write it down so that others can easily reproduce your experiments. People who want to perform your experiments should not make assumptions or guess what they have to do.

Tips:

1. To avoid having huge traces in Wireshark, make sure you limit the bandwidth of the links (limiting the bandwidth of one link on the entire path is sufficient).
2. Some questions may require a different topology. You can make multiple topologies for all experiments.
3. Always test your network topology before starting to perform the actual experiments. Make sure the topology behaves as you would expect.
4. Don't forget about the `tsso off` feature!
5. Make sure you include all relevant python scripts, output, wireshark traces, ... in your report.

L2-10-1 Describe your complete experiment and conclusions.

/10

The first experiment consists of 4 hosts connected by a single switch. The script for this can be found in [L2-10-1.1.py](#). Host 4 will serve as the receiving end of 3 TCP connections. All 4 links have been limited to a bandwidth of 1 mbps. Hosts 1, 2 and 3 will attempt to send 1 data at a rate of 1 mbps. To run the experiment, we first launch Mininet and open open 3 terminals on h4, and 1 terminal for h1, h2 and h3. We start an iperf3 server on the 3 terminals running on h4, assigning a different port number to each. Then, we start 3 iperf3 sessions from terminals running on hosts 1, 2 and 3. The output for the sessions can be found in:

1. [L2-10-1.experiment-1-session-1-client.txt](#)
2. [L2-10-1.experiment-1-session-1.txt](#)
3. [L2-10-1.experiment-1-session-2-client.txt](#)
4. [L2-10-1.experiment-1-session-2.txt](#)
5. [L2-10-1.experiment-1-session-3-client.txt](#)
6. [L2-10-1.experiment-1-session-3.txt](#)

We exclude the first couple of packets since the clients did not start at the same time. Looking at the server logs for all 3 sessions, we notice how the bandwidth was split up equally for all 3 sessions/hosts.

Acronyms

ACK Acknowledgement

ICMP Internet Control Message Protocol

IP Internet Protocol

IPv6 IP version 6

IPv4 IP version 4

RTT round trip time

RTO retransmission timeout

TCP Transmission Control Protocol

UDP User Datagram Protocol

MTU Maximum Transmission Unit

MSS Maximum Segment Size