Lab

6

**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 TCP performs retransmissions

Updated: February 2021

**Table of Content**

[Study Material for Lab 6 3](#_Toc65426237)

[Prelab 6 4](#_Toc65426238)

[Lab 6 – Transport Layer Protocols: UDP & TCP 5](#_Toc65426239)

[Part 1. Setting up the Network Topology and Using the “nc” ommand for Data Transmission 7](#_Toc65426240)

[Exercise 1.a- Network setup 7](#_Toc65426241)

[Exercise 1-b. The “nc” command 7](#_Toc65426242)

[Part 2. Data Transmission using TCP and UDP with “nc” 10](#_Toc65426243)

[Exercise 2-a. TCP and Connection Management 10](#_Toc65426244)

[Exercise 2-b. Exploring MSS negotiation in a TCP connection 12](#_Toc65426245)

[Exercise 2-c. TCP and Transmitting to a non-existing host 12](#_Toc65426246)

[Exercise 2-d. TCP and Transmitting to a host with a non-existing port 13](#_Toc65426247)

[Exercise 2-e. Transmitting data with UDP 14](#_Toc65426248)

[Part 3. IP Fragmentation of UDP and TCP traffic 15](#_Toc65426249)

[Exercise 3-a. UDP and Fragmentation 16](#_Toc65426250)

[Exercise 3-b. TCP and IPv4 Fragmentation 18](#_Toc65426251)

[Exercise 3-c. UDP and IPv6 Fragmentation 21](#_Toc65426252)

[Exercise 3-d. TCP and IPv6 Fragmentation 23](#_Toc65426253)

[Part 5. Retransmissions in TCP 25](#_Toc65426254)

[Exercise 5-a. TCP Retransmissions using GBN 25](#_Toc65426255)

[Exercise 5-b. TCP Retransmissions using SACK 27](#_Toc65426256)

# Study Material for Lab 6

1. **TCP and UDP**: Read the overview of TCP and UDP available at   
   <https://www.garykessler.net/library/tcpip.html>
2. **IP Fragmentation**: Refer to the Wikipedia entry for “Path MTU Discovery” and the references therein for information on IP Fragmentation and Path MTU Discovery.  
   <https://en.wikipedia.org/wiki/Path_MTU_Discovery>
3. **TCP Retransmissions**: Refer to RFC 2988, which is available at  
   <https://en.wikipedia.org/wiki/Karn%27s_algorithm>
4. **TCP Congestion Control**: Refer RFC 2001, which is available at  
   <http://www.faqs.org/rfcs/rfc2001.html>  
   and read about TCP congestion control.

## Prelab 6

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

1. Explain the role of port numbers in TCP and UDP.
2. Answer the following questions on Path MTU Discovery:
3. How does TCP decide the maximum size of a TCP segment?
4. How does UDP decide the maximum size of a UDP datagram?
5. 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?
6. Explain why a TCP connection over an Ethernet segment never runs into problems with fragmentation.
7. 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?
8. Answer the following questions about TCP acknowledgements:
9. What is a *delayed acknowledgement*?
10. What is a *piggybacked acknowledgement*?
11. Describe, in your own words, how the retransmission timeout (RTO) value is determined in TCP.
12. Answer the following questions on TCP flow control and congestion control:
13. Describe the sliding window flow control mechanism used in TCP.
14. Describe the concepts of *slow start* and *congestion avoidance* in TCP.
15. Explain the concept of *fast retransmit* and *fast recovery* in TCP.

# Lab 6 – Transport Layer Protocols: UDP & TCP

This lab explores the operation of the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP), the two 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 segment*. When a UDP segment is transmitted, the datagram is encapsulated in an IP header and delivered to its destination. There is usually one UDP segment for each application message. UDP does not create smaller segments from the Application data, leaving fragmentation to the IP layer so(type to) accommodate the link layer MTU size. However, with the move to now allow IP fragmentation, UDP, like TCP, now creates smaller segments from the Application data to fit the link layer MTU when path probing is enabled.

The operation of TCP is more complex. First, TCP is a connection-oriented protocol, in which 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 that 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, acknowledgments, 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 (cumulative ACK). When a TCP sender does not receive an ACK, the data is assumed lost and is retransmitted. With cumulative ACKs, TCP does not allow for the reception of out of sequence segments. Any segment that is received out of order, due to a loss, or a discard due to an error, of a previous segment, is also dropped. With cumulative acknowledgments, a TCP receiver cannot request the retransmission of specific segments. For example, if the receiver has obtained segments 1, 2, 3, 5, 6, 7 with cumulative acknowledgments the receiver can send ACKs only for segments 1, 2, 3 but not for 5, 6, 7. Segments 5, 6, 7 are discarded as they are now considered out of order.

The problem can be remedied with an optional feature of TCP, which is called ***selective acknowledgments* (SACKs)**. Here, in addition to acknowledging the highest sequence number of contiguous data that has been received correctly (cumulative ACK), a receiver can acknowledge additional blocks of sequence numbers. The range of these blocks is included in the TCP header as an option. The use of SACK for a connection is negotiated in the TCP header options field during the setup phase of the TCP connection. Recently it was uncovered that SACK has some severe security holes. In this lab we disable **SACK** to facilitate the observation of Seq#, ACK# and retransmissions.

TCP has two mechanisms that control the amount of data that a TCP sender can transmit. First, the 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. In Part 1 we setup the router serial interfaces and the network configuration as shown in Figure 6.1 with IP addresses as shown in Table 6.1. This configuration is used for all parts in this lab. Please also **NOTE** that we turn **SACK** off for TCP error recovery. We also turn off **CDP** and **LOOP** packets. In Lab 3 we showed you how to do this when configuring a router. Instructions are given in Part 2 Exercise 2-c. Please do this every time you start a PC in Lab 6. We also introduce the “netcat” or also known as the “nc” command. Part 2 compares the performance of data transmissions in TCP and UDP. We also observe TCP connection management. Part 3 explores how TCP and UDP deal with IP fragmentation, Path Discovery MTU (PDMTU) and we have added an exercise on IPv6 to illustrate PDMTU with ICMPv6. Part 4 explores TCP retransmissions. For that we turn on (enable) SACK to show the difference between GBN and SACK.

Timeline

Description automatically generated

Figure 6.1 Network topology for Lab 6

|  |  |  |  |
| --- | --- | --- | --- |
| PC | IPv4 address of *eth0* | IPv4 address of *eth1* | Default Gateway |
| PC1 | 10.0.1.11/24 | – | 10.0.1.1 |
| PC3 | 10.0.3.33/24 | – | 10.0.3.1 |
| **Cisco Router** | **IPv4 address of** ***FastEthernet0/0*** | **IPv4 address of** ***Serial2/0*** | **Default Gateway** |
| Router1 | 10.0.1.1/24 | 10.0.2.1/24 | 10.0.2.2 |
| Router2 | 10.0.3.1/24 | 10.0.2.2/24 | 10.0.2.1 |

Table 6.1 IP addresses of the PCs and Routers

## Part 1. Setting up the Network Topology and Using the “nc” ommand for Data Transmission

Using the topology shown in Figure 6.1, and the IP addresses in Table 6.1, set up the GNS3 configuration. Note that we are using the **serial** interfaces (Serial2/0) to interconnect the Routers.

### Exercise 1-a. Network setup

1. First set up the ethernet and serial interfaces on the routers. Here shown for *Router1* interface FastEthernet0/0 and Serial2/0. And add the default gateway. Repeat for *Router2* with default gateway 10.0.2.1 and appropriate IP addresses as listed in Table 6.1

Router1# **configure terminal**

Router1(config)# **no ip routing**

Router1(config)# **ip routing**

Router1(config)# **interface FastEthernet0/0**

Router1(config-if)# **ip address 10.0.1.1 255.255.255.0**

Router1(config-if)# **no shutdown**

Router1(config-if)# **interface Serial2/0**

Router1(config-if)# **ip address 10.0.2.1 255.255.255.0**

Router1(config-if)# **no shutdown**

Router1(config-if)# **end**

Router1(config)# **ip route 0.0.0.0 0.0.0.0 *10.0.2.2***

Router1(config)# **exit**

1. After you have configured the 2 PCs with the IP addresses as given in Table 6.1, you need to **disable SACK** (selective repeat error recovery), on both *PC1* and *PC3*. Shown here for *PC1*. Repeat for *PC3.*

PC1$ **sudo sysctl -w net.ipv4.tcp\_sack=0**

Or

PC1$ **sudo echo “0” > /proc/sys/net/ipv4/tcp\_sack**

1. Add default routes to the routing tables of *PC1* and *PC3*. For *PC1* the command is as follows. *PC3* default gateway is 10.0.3.1.

PC1$ sudo ip route add default via 10.0.1.1

1. Verify that the setup is correct by issuing a ping command from *PC1* to *PC3*:

PC1$ ping –c 2 10.0.3.33

### Exercise 1-b. The “nc” command

**netcat**, or **nc,** as it usually is referred to as, is a simple Unix utility which reads and writes data across network connections, using TCP or UDP protocol. It is designed to be a reliable "back-end" tool that can be used directly or easily driven by other programs and scripts. At the same time, it is a feature-rich network debugging and exploration tool, since it can create almost any kind of connection you would need and has several interesting built-in capabilities.

In the simplest usage, "nc **host port**" creates a TCP connection to the given **port** on the given target **host**. Your standard input is then sent to the **host**, and anything that comes back across the connection is sent to your standard output. This continues indefinitely, until the network side of the connection shuts down. Note that this behavior is different from most other applications which shut everything down and exit after an end-of-file on the standard input.

netcat can also function as a server, by listening for inbound connections on arbitrary ports and then doing the same reading and writing. With minor limitations, netcat doesn't really care if it runs in "client" or "server" mode -- it still shovels data back and forth until there isn't any more left. In either mode, shutdown can be forced after a configurable time of inactivity on the network side.

And it can do this via UDP too, so netcat is possibly the "udp telnet-like" application you always wanted for testing your UDP-mode servers. UDP, as the "U" implies, gives less reliable data transmission than TCP connections and some systems may have trouble sending large amounts of data that way, but it's still a useful capability to have.

|  |
| --- |
| **The syntax of the command to start nc**  nc *[-options] hostname port[s] [ports] ...*  nc *-l -p port [-options] [hostname] [port]*  A partial list of options of the command are given below. Option can be specified by a hyphen (-) followed by a single letter.  **Options**  -b  Allow UDP broadcasts.  -c *string*  Specify shell commands to exec after connect (use with caution). The string is passed to /bin/sh -c for execution. See the *-e* option if you don't have a working /bin/sh (Note that POSIX-conformant system must have one).  -e *filename*  Specify filename to exec after connect (use with caution). See the *-c* option for enhanced functionality.  -g *gateway*  Source-routing hop point[s], up to 8.  -h  Display help.  -i *secs*  Delay interval for lines sent, ports scanned.  -l  Listen mode, for inbound connects.  -n  Numeric-only IP addresses, no DNS.  -o *file*  Hex dump of traffic.  -p *port*  Local port number (port numbers can be individual or ranges: lo-hi [inclusive]).  -q *seconds*  After EOF on stdin, wait the specified number of seconds and then quit. If *seconds* is negative, wait forever.  -r  Randomize local and remote ports.  -s *addr*  Local source address.  -t  Enable telnet negotiation.  -u  Enable UDP mode.  **-v**  Verbose (use twice to be more verbose)  **-w** ***secs***  Timeout for connects and final net reads  **-C**  Send ^F as line ending.  -z  Zero-I/O mode [used for scanning]  **-T *type***  Set TOS flag (type may be one of "Minimize-Delay", "Maximize-Throughput", "Maximize-Reliability", or "Minimize-Cost").  Read more at: <https://www.commandlinux.com/man-page/man1/nc.1.html> |

## Part 2. Data Transmission using TCP and UDP with “nc”

In this lab we will use netcat or nc as the actual program is called, to observe and test TCP and UDP transmissions.

### Exercise 2-a. TCP and 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 (ISN), the maximum segment size (MSS) and the size of the windows for the sliding window flow control – receive window. TCP requires three or four packets to close a connection. Each end of the connection can be closed separately, requiring 4 packets. This is called a half-close on each side. If both sides close at the same time, then the FIN packet and the ACK can be combined and transmitted in the same segment, giving rise to only 3 packets for closing.

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.

By default Wireshark will keep track of all TCP sessions and **convert** all Sequence Numbers (SEQ numbers) and Acknowledge Numbers (ACK numbers) into **relative** numbers. This means that instead of displaying the **real/absolute** SEQ and ACK numbers in the capture window, Wireshark will display SEQ and ACK numbers relative to the first seen segment for that conversation.

This means that all SEQ and ACK numbers always start at **0** for the first packet seen in each conversation. This makes the numbers much smaller and easier to read and compare than the real numbers which normally are initialized to randomly selected numbers in the range 0 – (232-1) during the SYN phase.

For the Lab Questions related to TCP connection set-up and teardown below, please disable this feature when viewing the Wireshark captured traffic data. You can do that by going to Wireshark “Preferences” and under the protocol tab, look for TCP. Select TCP and in the window find “Relative sequence number” and uncheck it. When done with the questions below, go back and “check” enable “Relative sequence number” again. Relative numbers are easier to work with in general.

This exercise consists of setting up a TCP data transfer, using the command nc, between two hosts, *PC1* and *PC2*, and observe the TCP connection management traffic.

1. Start Wireshark to capture packets on interface eth0 of *PC1* (connection to the switch). Recall that you need you to deselect “Relative sequence number”.[[1]](#footnote-1)
2. On *PC3*, start the nc listening command on port 10086 (i.e., *PC3* is acting as the server) so that it can receive packets being sent to it from a *PC1* (i.e., *PC1* is acting as a client). The port number above is randomly picked, you are free to pick any free port you like:

PC3$ **nc -l 10086**

1. Use the nc command to establish a TCP connection from client *PC1* to server *PC3*:

PC1:~$ **nc 10.0.3.33 10086**

1. Now send the following message from *PC1* to *PC3*. Type it in *PC1*’s console and press enter. On *PC3*’s console, you should see *PC1*’s message echoed (displayed) on the screen. Watch the traffic capture, you will observe a TCP connection being setup and the transfer of TCP data. Press enter at the end of the sentence to initiate the transmission and conclude it. You can type as many sentences as you want.

PC1$ **Hello Bob! Have not seen you in ages. Hope all is going well**.

1. Stop the nc processes (i.e., terminate the connection) on both *PC1* and *PC3* using the following command on each PC’s console, shown here for *PC1*:

PC1$ **^C**

1. ![A picture containing text, iPod, electronics

   Description automatically generated]()In the traffic capture window you will observe the TCP connection being terminated.
2. Stop the data capture. Save the Wireshark output. **Note**: save this Wireshark output for later use. You will use it to compare it to the Wireshark output of Exercise 2-e.



Lab Questions/Report:

Use the saved Wireshark output to answer questions related to the TCP connection – specifically the TCP Connection Set-up and Teardown.

Analyze the TCP segments of the transmitted packets during connection set up:

* 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 and TCP client.

From : [Three-Way Handshake - an overview | ScienceDirect Topics](https://www.sciencedirect.com/topics/computer-science/three-way-handshake#:~:text=The%20TCP%20handshake,as%20shown%20in%20Figure%203.8.)

1. PC1->PC3 SYN (synchronize: seq: 919494744)
2. PC3->PC1 SYN, ACK (synchronize: seq: 4195270838 , acknowledgement: 919494744 + 1 = 91949474)
3. PC1->PC3 ACK (acknowledgement: 4195270838 + 1 = 4195270839)

* During the connection setup, the TCP client and TCP server exchange their initial sequence number (ISN#) that they will use for data transmission. What are the initial sequence numbers of the TCP client and the TCP server?
* Observe the port numbers in the TCP header. How did nc sender select the source port number?
  1. [protocol theory - Why does source port and destination port differ, in my network communications? - Network Engineering Stack Exchange](https://networkengineering.stackexchange.com/questions/40850/why-does-source-port-and-destination-port-differ-in-my-network-communications)
  2. [networking - How are source ports determined and how can I force it to use a specific port - Super User](https://superuser.com/questions/1118735/how-are-source-ports-determined-and-how-can-i-force-it-to-use-a-specific-port#:~:text=The%20port%20number%20is%20chosen,used%20is%20Operating%20System%20dependent.)
  3. Its from ephemeral port range

Analyze the TCP segments of the transmitted packets during data transfer:

* Identify the first packet that contains application data. What is the sequence number used in the first segment of application data sent from the TCP client to the TCP server? 919494745
* The TCP client and TCP server exchange window sizes to get the maximum amount of data that the other side can send at any time. Determine the values of the receive window sizes for the TCP client and the TCP server. 64256
* What is the MSS (maximum segment size) value that is negotiated between the TCP client and the TCP server? 1460 bytes
* How many packets are exchanged in the data transfer (1 tcp payload)? How many packets are transmitted for each TCP datagram (2 (1 tcp payload+ack))? What is the size of the TCP payload of these packets(data: 62 bytes)?
* Inspect the fields in the TCP headers. Which fields in the headers do not change in different packets?
  1. FOR PC1->PC3 source and destination port does not change and same for PC3->PC1.

Analyze the TCP segments of the transmitted packets during connection tear down:

* Identify the packets that are involved in closing the TCP connection. Which flags are set? FIN & ACK
* Explain how these flags are interpreted by the receiving TCP server and TCP client. How many transmissions were involved in the tear down?
  1. Each tear down have 2 message (fin+ack, ack)
  2. FIN: graceful shutdown, ACK: acknowledgement

### Exercise 2-b. Exploring MSS negotiation in a TCP connection

In this exercise, you will explore how a TCP connection negotiates the MSS size during a TCP three-way handshake to conform to the lowest MTU size on the 2 end devices.

1. Start a Wireshark capture on interface eth0 of *PC1* (connection to the switch).
2. Modify the MTU size of *PC3*’s interface eth0 to 700

PC3$ **ip link set dev eth0 mtu 700**

1. On *PC3*, start the nc listening command on port 10086 (*PC3* is acting as the server) so that it can receive packets being sent to it from a client (i.e., *PC1*, acting as a client). The port number above is randomly picked, you are free to pick any free port you like:

PC3$ **nc -l 10086**

1. Use the nc command to establish a TCP connection from client *PC1* to server *PC3*:

![A picture containing text, iPod, electronics

Description automatically generated]()PC1:~$ **nc 10.0.3.33 10086**

1. Use command ^C on each PC to terminate the nc process.
2. Save the Wireshark capture.



Lab Question/Report:

1. What is the MSS value on each interface? How is it calculated? What is the final agreed upon value?700-40 (header) 1500-40(header)
2. Use your saved Wireshark capture to explain how a TCP connection negotiates its MSS during the setup phase.
   1. In options of tcp they send heir each and pick min.

### Exercise 2-c. TCP and Transmitting to a non-existing host

Here you observe how a TCP client tries to establish a connection to a TCP server (host) that does not exist. It finally gives up after several attempts at reaching the host.

1. Start a Wireshark traffic capture on interface eth0 of *PC1* to the switch.
2. Set a static entry in the neigh cache on *PC1* for the *non-existing* host at IP address 10.0.1.100. Note that neither the IP nor the MAC address exist.

PC1$ sudo ip neigh add 10.0.1.100 lladdr 01:02:03:04:05:06 dev eth0

Display *PC1*’s neigh cache to make sure the entry is registered in the cache..

PC1$ ip neigh show

1. From *PC1*, establish an ncconnection to the non-existing host:

PC1$ nc 10.0.1.100 10086

1. Wait for a while. Watch the Wireshark traffic capture. You will see that TCP makes several attempts and then finally gives up.
2. On *PC1*, remove the static entry in the neigh cache which you added in step 2.

![A picture containing text, iPod, electronics

Description automatically generated]()PC1:~$ sudo ip neigh del 10.0.1.100 dev eth0

1. Stop and save the Wireshark output.



Lab Questions/Report:

Use the saved Wireshark capture to answer the following questions:

1. How often does the TCP client try to establish a connection(a lot more that 6 times)? How much time elapses between repeated attempts to open a connection(it’s exponential delay like 2 sec, 4 sec, 8 sec, 16 seconds, 32 seconds & finally 64 seconds)?
2. Does the TCP client **terminate** (using **FIN** bit flag) or **reset** (using **RST** bit flag) the connection, when it gives up with trying to establish a connection? NO, it gives up after fixed number of attempts = 6.
3. Why does this experiment require you to set a static neigh cache entry? Without that PC1 ARPs for 10.0.1.100 first and doesn’t start nc when ARP fails.

### Exercise 2-d. TCP and Transmitting to a host with a non-existing port

Here you observe how a TCP client tries to establish a connection to a TCP server (host) with a non-existing port. It finally gives up after several attempts to reach the port at the host.

1. Start a Wireshark traffic capture on interface eth0 of *PC1* to the switch.
2. On *PC3*, start the nc listening command on port 10086 (*PC3* is acting as the server) so that it can receive packets being sent to it from a client (i.e., *PC1*, acting as a client). The port number above is randomly picked, you are free to pick any free port you like:

PC3$ **nc -l 10086**

1. Use the nc command to establish a TCP connection from client *PC1* to server *PC3* on a different port than the one it is listening on:

PC1$ **nc 10.0.3.33 10042**

1. ![A picture containing text, iPod, electronics

   Description automatically generated]()Watch the Wireshark traffic capture. The receiver will reject the connection.
2. Stop and save the Wireshark output.

Lab Questions/Report:

Use the saved Wireshark capture to answer the following questions:

1. Examine the TCP segment sent from the receiver. What flags do you see are set?Based on that,what happens to the connection? RST (reset) flag from : [What is a TCP Reset (RST)? | Pico](https://www.pico.net/kb/what-is-a-tcp-reset-rst)
2. After which client terminates: [TCP RST FLAG - IP With Ease](https://ipwithease.com/tcp-rst-flag/#:~:text=The%20TCP%20RST%20flag%20indicates,because%20of%20a%20fatal%20error.&text=RST%20bit%20will%20be%20set,which%20no%20process%20is%20listening.)

### Exercise 2-e. Transmitting data with UDP

Here, you will repeat Exercise 2-a., but use UDP for data transfer.

1. Start Wiresharkand capture packets on interface eth0 between *PC1* and the switch. For this exercise you don’t need to disable the “Relative sequence numbers” feature.
2. On *PC3*, start the nc listening command on port 10086 (*PC3* is acting as the server) so that it can receive packets being sent to it from a client (i.e., *PC1*, acting as a client) via UDP. The port number above is randomly picked, you are free to pick any free port you like:

PC3$ **nc -u -l 10086**

1. Use the nc command to establish a UDP connection from client *PC1* to server *PC3*:

PC1:~$ **nc -u 10.0.3.33 10086**

1. Now send the following message from *PC1* to *PC3*. Type it in *PC1*’s console and press enter when done. On *PC3*’s console, you will see *PC1*’s message echoed (displayed) on the screen. Watch the traffic capture, you will observe the transfer of TCP data. Press enter at the end of the sentence to initiate and conclude the transmission of the data.

![A picture containing text, iPod, electronics

Description automatically generated]()PC1$ **Hello Bob! Have not seen you in ages. Hope all is going well**.

1. Use ^C to stop the nc processes on both *PC1* and *PC3*.
2. Stop the data capture. Save the Wireshark output.

Lab Questions/Report:

Comparing TCP and UDP connections and data transmission.

Use the data captured from the Wireshark captures in Exercises 2-a. and 2-e. to answer the following questions. ***Enable* “**Relative sequence number” for both captures to make traffic analysis easier.

* How many ***data*** packets are transmitted by *PC1*, and how many data packets are transmitted by *PC3* for both TCP and UDP? How many ACKs does TCP send in both directions?
  + PC3 and PC1 only sends 1 data which is also the only tcp packet per message
  + NO ACK are present.
* What are the sizes of the TCP segment? UDP segment?
  + PAYLOAD + 8 Byte header and as payload is different so is the size.
* Compare the ***total*** number of overhead bytes transmitted, in both directions, that is Ethernet, IP, and UDP/TCP headers, to the amount of application data transmitted. How much more overhead data does TCP incur compared to UDP? Please include all packet exchanges for both protocols.
  + UDP: 42 bytes extra apart from data for one message.
  + TCP: 132 extra bytes: which have 66 bytes of ack and 66 bytes for data headers.

## 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 (segment for TCP, and datagram for UDP) 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 *Don’t 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 a *Path Discovery MTU* (PDMTU) scheme that determines a Maximum Segment Size (MSS) which does not result in fragmentation. It does this by probing the path with the **DF** bit set. Every time the datagram encounters a smaller MTU size, and error is returned to the sender and the transport layer is required to adjust the MSS value to it in the received MTU size returned in the ICMP error packet. Once the packet goes through all the way to the destination with no more ICMP errors reported due to packet size, the transport layer recognizes that it has taken care of the smallest MTU on the path. It then proceeds to transport the application data using the new MSS value discovered via probing. UDP has a maximum payload size that is not negotiated, as it does not setup a connection. In the past it used to be set to a very high value (65K). Now it is set in the kernel to a default value to conform to the default MTU size of the host interface. Since UDP does not negotiate a payload size, IP does not set the **DF** bit in the header that causes an ICMP error to occur and is reported back to the sender to react by decreasing the payload size. For UDP, IP handles it by doing fragmentation.

In IPV6 we do not have a DF bit. TCP avoids fragmentation using PDMTU, reacting to the ICMPv6 “too big” packet by adjusting its MSS to fit the indicated MTU size. For UDP, just like in IPv4, IPv6 takes care of fragmentation. It uses the “Fragment Header” to carry all the parameters associated with fragmentation and reassembly. The next header field is used to identify the next header as the “Fragment Header”. This is then followed by the UDP header and the “fragmented” payload.

In this part you explore IP fragmentation using IPv4 and IPv6. We use the same configuration as in Figure 6.1, and use Table 6.1 for the IPv4 exercises (3-a., 3-b.) and Table 6.2 (shown below) for the IPv6 exercises (3-c, 3-d).

|  |  |  |  |
| --- | --- | --- | --- |
| PC | IPv6 address of *eth0* | IPv6address of *eth1* | Default Gateway |
| PC1 | autoconfigured | – | - |
| PC3 | autoconfigured | – | - |
| **Cisco Router** | **IPv6 address of** ***FastEthernet0/0*** | **IPv6 address of**  ***Serial2/0*** | **Default Gateway** |
| Router1 | fd01:2345:6789:1::1/64 | fd01:2345:6789:2::1/64 | fd01:2345:6789:2::2 |
| Router2 | fd01:2345:6789:3::1/64 | fd01:2345:6789:2::2/64 | fd01:2345:6789:2::1 |

Table 6.2 IPv6 Address

In Cisco IOS, you can view the MTU values of all interfaces using the show interfaces command:

Router> enable

Router# show interfaces

Note that the value displayed is the value of the link layer (L2) MTU **PLUS** the link layer (L2) headers. So for an Ethernet link it will be 1514 (that is 1500 (payload) +14 (Ethernet header and Trailer)). The command to modify the MTU value on interface Serial 2/0from *default* to **size** is as follows (shown for *Router*): **NOTE**: “**size**” needs to include the headers and trailers of the link layer (L2).

Router# configure terminal

Router(config)# interface Serial2/0

Router(config-if)# mtu size

### Exercise 3-a. UDP and Fragmentation

In this exercise you observe IP fragmentation of UDP traffic. We use ncto generate UDP traffic between *PC1* and *PC3*, across the two routers *Rourter1* and *Router2* and the serial link. We gradually increase the size of the UDP datagrams, by increasing the carried bytes in the payload, until fragmentation occurs. You will observe that IP headers do not set the *DF* bit for UDP payloads.

1. Verify that the network is configured as shown in Figure 6.1 and Table 6.1.
2. Verify that the routers are configured as shown in Figure 6.1 and Table 6.1. The routers should be configured as described in Exercise 1-a. Remember to disable “cdp” and “loop” traffic.
3. View the MTU values of all interfaces on the router using the “show interfaces” command, shown here for *Router1*:

Router1> enable

Router1# show interface S2/0

Repeat for interface F0/0.![A picture containing text, iPod, electronics

Description automatically generated]() What are the MTU sizes for (F0/0) and (S2/0) interfaces of the router? Fill in the table below.

| Router | MTU size of Interface *FastEthernet0/0* | MTU size of Interface *Serial2/0* |
| --- | --- | --- |
| Router1 | 1500 | 1500 |

Record the values and save.

1. Start Wireshark captures on the eth0 interfaces of both *PC1* and *PC3* and the switches. Do not set any filters.
2. Use ncto generate UDP traffic between *PC1* and *PC3*. The connection parameters are selected so that IP Fragmentation does not occur initially.
3. Start the nc listening command on the server *PC3* with the UDP option -u to receive UDP packets being sent from client.

PC3$ nc -u -l 10086

1. Use the following command on *PC1* to create a file of size **N** bytes, called “a.txt” with the character “a” that will not cause fragmentation. Set **N** = 512.

PC1$ for i in {0..N}; do echo -n “a” >> a.txt; done;

1. Use the following command to transfer file “a.txt” from *PC1* to *PC3* using UDP with the nc command:

PC1$ cat a.txt | nc -p 10086 -u 10.0.3.33 10086

1. Delete the current file “a.txt” using the command

PC1$ rm a.txt

1. ![A picture containing text, iPod, electronics

   Description automatically generated]()Repeat steps 8-9 with a larger file size by changing the parameter “**N**” to 512, 1024, 2048, …. until you observe fragmentation in the Wireshark capture.
2. When done, i.e., you observed fragmentation, stop the Wireshark capture and save the output (remember to delete the last used file “**a.txt**”.
3. Now we will change the MTU **size** on the router serial interfaces to 500. The commands are shown here for *Router1* interface Serial 2/0. Repeat for *Router2*. Use **size** = 500.

Router1# configure terminal

Router1(config)# interface Serial2/0

Router1(config-if)# mtu size

1. Start a Wireshark capture on *PC1* eth0 to the switch and on *Router1* Serial2/0 interface.
2. Now generate a file “**a.txt**” on *PC1* that is larger than the MTU size on *Router1* interface Serial2/0 but less than *PC1*’s default MTU. Use **N**=1024. (Note that *PC1*’s MTU is 1500, standard for Ethernet.)

PC1$ for i in {0..N }; do echo -n “a” >> a.txt; done;

1. Send file “a.txt” from *PC1* to *PC2* using UDP with the following nc command:

PC1$ cat a.txt | nc -p 10086 -u 10.0.3.33 10086

1. Delete file “**a.txt**” on PC1 with the command “rm a.txt”.
2. Now generate a file “**a.txt**” on *PC1* that is larger than the MTU size of *PC1*’s MTU (=1500, standard for Ethernet) and *Router1* interface Serial2/0 (set to 500 above). Use **N**=1624:

PC1$ for i in {0..N }; do echo -n “a” >> a.txt; done;

1. Send file “a.txt” from *PC1* to *PC3* using UDP with the following nc command:

PC1$ cat a.txt | nc -p 10086 -u 10.0.3.33 10086

1. Delete file “**a.txt**” on PC1 with the command “rm a.txt”.
2. ![A picture containing text, iPod, electronics

   Description automatically generated]()Terminate the nc process with ^C on both *PC3* and *PC1*.
3. Stop the traffic capture and save the Wireshark outputs.
4. Do not reset the MTU value to default. We will use MTU = 500 in the next exercise (3-b.).

Lab Questions/Report:

1. For steps 7-9, using the Wireshark data capture saved in Step 11:

* What is the default MTU size on the serial Interface (Serial2/0)? 1500
* Determine the UDP segment size at which fragmentation occurs for the case when using the default MTU value. 2048 ie above 1500
* Determine the maximum size of the UDP segment (whole or fragment) that the system can transport. 1500 + 14 header
* From the **first** saved Wireshark data, select one IP datagram that is fragmented. Look at the complete datagram after fragmentation. For each fragment of this datagram, determine the values of the fields in the IP header that are used for fragmentation (*Identification:3437, Fragment Offset:1480, Don’t Fragment Bit: 0, More Fragments Bit: 1*).

1. For steps 12-14, with *Router1* MTU set to 500 and the Wireshark data saved in Step 21:

* What did you observe that was different from the case with the default MTU (Steps 7-9)?
  + For case 1 with 1624 first PC1 R1 only 2 fragments cos mtu 1500 but thereon btw R1 and R2 and R2 to R3 we have 4 fragments because of 500 s2/0 mtu.
* From the saved Wireshark data on *Router1* Serial2/0 interface, find the IP datagram N=1024) that is fragmented at the router interface only. 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, Don’t Fragment Bit, More Fragments Bit*).
* Repeat above but select the IP datagram (N=1624) that was fragmented at the *PC1* interface (from Wireshark capture at *PC1*) and then again at *Router1* interface (from Wireshark capture at *Router1* (N=1624). Show data of both fragmentation operations (i.e., at *PC1* and then at *Router1*). Observe the value of the MF flag in each fragment and how it is set for the fragments as they flow from one link to the next. What happens with the fragment that had MF=0 set on *PC1* interface when it is transmitted by *Router1* on S2/0 interface?
  + Remains same by following the fragment offset

1. Why does IP not set the DF bit for UDP transmissions? Because the IP can perform fragmentation.

### Exercise 3-b. TCP and IPv4 Fragmentation

TCP tries to avoid fragmentation from occurring in the network (i.e., at the IP layer in the routers) 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 as an “option” in the TCP header of the first transmitted TCP segment. Each side sets the MSS so that no fragmentation occurs at the outgoing network interface on either side, 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 (Don’t Fragment) bit in all IP datagrams. When a router needs to fragment an IP packet with the DF bit set, it discards the packet and generates an ICMP error messageof type “*Destination unreachable; Fragmentation needed”*. Upon receiving such an ICMP error message, the TCP sender reduces the segment size. This continues until a segment size is determined which does not trigger an ICMP error message.

1. If you are **NOT** continuing from Exercise 3-a. you need to repeat Steps 1 and 2 to setup the PCs and the routers as in Figure 6.1 and Table 6.2.
2. Before you start this exercise, you need to turn off all TCP options (as they increase the TCP header size beyond 20 bytes and as such will cause the need to fragment even though the payload fits the MTU-20-20 rule ).Disable **SACK** on the PCs as we did in Exercise 1-a. The **Timestamps** option is often enabled too by default as it helps TCP determine delays in the network. We will disable that too.
3. Disable **SACK** and **Timestamps** on the PCs using the following commands, shown here for *PC1*. Repeat for *PC3*.

PC1$ sudo sysctl -w net.ipv4.tcp\_timestamps=0

PC1$ sudo sysctl -w net.ipv4.tcp\_sack=0

1. If you are continuing from Exercise 3-a, the MTU of the routers serial interfaces (S2/0) should still be set to 500. If not, please modify **size** of MTU to “500” using the commands as shown in Step 12 in Exercise 3-a.
2. In Linux, you can view the MTU values of all interfaces in the output of the ip addr show command. For example, on *PC1*, you type:

PC1$ ip addr show

The command ‘ip link set’ is used to modify the MTU value. For example, to set the MTU value of interface eth0 on *PC1* to 500 bytes, use the ip link command as follows:

PC1$ sudo ip link set dev eth0 mtu 500

Use the following command to view settings on *PC1* interface eth0:

PC1$ ip addr show eth0

1. For now you will **NOT** change those values.
2. Start Wireshark captures on the eth0 interface of *PC1* and on Serial2/0 interface of *Router1*.
3. Make sure that MTU Path Discovery is activated (MTU probing) on *PC1*.

You can check if probing is set by using the syctl command as shown below. If the value returned is “0” it is disabled, if “1” it is disabled by default and enabled when an ICMP blackhole is detected, and if “2” it is always enabled. E.g., on *PC1*, type:

PC1$ sysctl net.ipv4.tcp\_mtu\_probing

1. Enable MTU probing on *PC1* with the following command:

PC1$ sysctl -w net.ipv4.tcp\_mtu\_probing=2

Or

PC1$ echo "2" > '/proc/sys/net/ipv4/tcp\_mtu\_probing'

1. You do **not** need to do this for *PC3*.
2. Start an nc receiver on *PC3* with the following command:

PC3$ **nc -l** 10086

1. Now generate a file “**a.txt**” on *PC1* that is larger than the MTU size on *Router1* interface Serial2/0 but less than *PC1*’s default MTU. Use **N**=1024. (Note that *PC1*’s MTU is 1500, standard for Ethernet.)

PC1$ for i in {0..N }; do echo -n “a” >> a.txt; done;

1. Send file “**a.txt**” from *PC1* to *PC2* using TCP with the following nc command:

PC1$ cat a.txt | nc -p 10086 10.0.3.33 10086

1. ![A picture containing text, iPod, electronics

   Description automatically generated]()Delete file “**a.txt**” on *PC1* with the command “rm a.txt”.
2. Stop the Wireshark captures and save the outputs.
3. Now generate a file “**a.txt**” on *PC1* that is larger than the MTU size of *PC1*’s MTU (=1500, standard for Ethernet) and *Router1* interface Serial2/0 (set to 500 above). Use **N**=1624:

PC1$ for i in {0..N }; do echo -n “a” >> a.txt; done;

1. Start Wireshark captures on the eth0 interface of *PC1* and on Serial2/0 interface of *Router1.*
2. Send file “**a.txt**” from *PC1* to *PC3* using TCP with the following nc command:

PC1$ cat a.txt | nc -p 10086 10.0.3.33 10086

1. Delete file “**a.txt**” on *PC1* with the command “rm a.txt”.
2. ![A picture containing text, iPod, electronics

   Description automatically generated]()Terminate the nc process on both *PC1* and P*C3* with ^C.
3. Stop the traffic captures and save the Wireshark outputs.
4. Reset the value of the router MTU to the default size that you recorded in Exercise 3-a., Step 3. (Note for the Cisco IOS routers in Labs (3600 series), “default” size = 1500 on both FastEthernet Serial interfaces. The Serial link uses the HDLC protocol with 16 bit CRC check).



Lab Questions/Report:

From the Wireshark traffic captures saved in Step 15:

1. Do you observe fragmentation? If so, where does it occur? Bter1 r2

* Should it have occurred with PDMTU? Yeah to detect

1. 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.

* What is the value of the recommended **MTU** in the ICMP message sent to *PC1*? 500
* Based on that, what should the adjusted MSS value be set to for *PC1*? 500 – headers(40)
* What did you observe the MSS value to be? 500

From the Wireshark traffic captures saved in Step 21:

1. Is there any difference in behavior in the transmission of file a.txt with N=1624 from what you observed in the transmission of file a.txt with size N=1024? NO
   1. What is the size of the TCP segment transmitted by *PC1? 500+66* What MTU size does it conform to? 500
   2. Did you observe fragmentation? If so, where does it occur? Yes, btw R1 R2 wireshark

Note: You will notice that even though TCP participates in the MTU probing process, the calculation of MSS value, based on the received MTU value in the ICMP error message, is incorrect. It sets MSS to 500, where it should set it to 500 – 20 - 20 = 460. This seems to be a bug in Ubuntu v18 through v20. This was not observed in older versions of Ubuntu, such as v16.

**RESUME FROM HERE**

### Exercise 3-c. UDP and IPv6 Fragmentation

UDP behaves similarly with IPv6 as it does with IPv4. The fragmentation is left up to the IP layer to handle. In IPv6 we do not have the DF bit. The IP layer automatically fragments any large packets from UDP that do not fit into the MTU at the link layer. IPv6 uses the “Next Fragment” header to assist with the fragmentation process and reassembly at the destination.

We continue with the same topology as shown in Figure 6.1. But add to the configuration of the two routers, *Router1* and R*outer2* the IPv6 addresses given in Table 6.2. The PCs will automatically autoconfigure both a link-local address and a global unicast address.

1. Configure the two routers, *Router1* and R*outer2* using the IPv6 addresses given in Table 6.2. Commands given here are for *Router1*. Repeat for *Router2*. Note that we set the mtu size to 1280. That is the minimum required MTU for IPv6.

Router1# **configure terminal**

Router1(config)# **interface FastEthernet0/0**

Router1(config-if)# **ipv6 address fd01:2345:6789:1::1/64**

Router1(config-if)# **no shutdown**

Router1(config-if)# **interface Serial2/0**

Router1(config-if)# **ipv6 address fd01:2345:6789:2::1/64**

Router1(config-if)# **no shutdown**

Router1(config-if)# **mtu 1280**

Router1(config-if)# **exit**

Router1(config)# **ipv6 unicast-routing**

Router1(config)# **ipv6 route fd01:2345:6789:3::/64 S2/0 fd01:2345:6789:2::2**

Router1(config)# **exit**

1. Before continuing with this exercise, we need to modify the MTU cache expiration time of IPv6 on the Linux hosts. By default, this cache lasts for 600 seconds, that is too long for our experiments. We need to modify this value so we wait for a shorter time to observe certain behaviors. Use the following command to change the expiration time on a Linux PC. Shown here for *PC1*. You do not need to do this for *PC3*.

PC1$ sudo sysctl -w net.ipv6.route.mtu\_expires=60

1. First start a nc receiver with UCP over IPv6 on *PC3* with the following command:

PC3$ nc -6 -l –u 10086

1. Start a Wireshark capture on interface eth0 on *PC1* and on interface S2/0 on *Router1*.
2. On *PC1*, create a file that is smaller than 1280 bytes to guarantee that no fragmentation need occur on the Serial link. To calculate the maximum value of the payload to fit in an MTU of 1280 you need to realize that the **IPv6 header is 40 bytes and UDP is 8 bytes**. For example, **N** = 800 will fit fine. Create the file using the following command:

PC1$ for i in {0..N}; do echo -n “a” >> a.txt; done;

1. Using the autoconfigured global unicast IPv6 address of *PC3*, initiate an nc client on *PC1* to send the file “a.txt” to server *PC3*:

PC1$ cat a.txt | nc -6 -u *IPv6\_address\_of\_PC3* 10086

1. Delete the file “a.txt” from *PC1*.
2. Now, on *PC1*, create a file that is bigger than > 1280 bytes but smaller than 1452 (1500-40-8). A value of 1280 < **N** <1452 bytes will guarantee the need to fragment on the router serial interface, but not on PC1’s interface. Use the following command to create file “a.txt” with **N** = 1380.

PC1$ for i in {0..1380}; do echo -n “a” >> a.txt; done;

1. Using the autoconfigured global unicast IPv6 address of *PC3* to initiate an nc client on *PC1* to send the file “a.txt” to server *PC3*:

PC1$ cat a.txt | nc -6 -u *IPv6\_address\_of\_PC3* 10086

1. Delete the file “a.txt” from *PC1*.
2. Now, on *PC1*, create a file that is bigger than > 1500 bytes that requires fragmentation at *PC1* eth0 interface and at *Router1* S2/0 interface. Use the following command to create file “a.txt” with **N** = 1880.

PC1$ for i in {0..N}; do echo -n “a” >> a.txt; done;

1. Using the autoconfigured global unicast IPv6 address of *PC3* to initiate an nc client on *PC1* to send the file “a.txt” to server *PC3*:

PC1$ cat a.txt | nc -6 -u *IPv6\_address\_of\_PC3* 10086

1. Delete the file “a.txt” from *PC1*.![A picture containing text, iPod, electronics

   Description automatically generated]()
2. Stop the Wireshark captures and save.



Lab Questions/Report:

From the saved Wireshark data:

1. Find the IP datagram that was transmitted in Step 6. Look at the complete datagram and:

* Identify the IPv6 header. What is the value of the Next Header field? UDP(17)
* Identify the UDP header showing the source and destination ports and payload length.
  + 1. Src Port: 10086, Dst Port: 10086, length: 809 (total length: 863)

1. Find the IP datagrams that were transmitted in Step 9.
   * How many fragments were generated 1? On which link? 2 on s2/0
   * Find the first fragment. Identify the IPv6 header. What is the value of the Next Header field? Next Header: Fragment Header for IPv6 (44)
   * Identify all the fields in the Fragment Header.

Fragment Header for IPv6

Next header: UDP (17)

Reserved octet: 0x00

0000 0000 0000 0... = Offset: 0 (0 bytes)

.... .... .... .00. = Reserved bits: 0

.... .... .... ...1 = More Fragments: Yes

Identification: 0x8b4819a2

Data (1232 bytes)

1. Find the IP datagrams that were transmitted in Step 12.
   * How many fragments were generated? On *PC1* interface 2? On *Router1* interface? 2
   * Find the first fragment on each link (*PC1* and *Router1*). Identify the IPv6 header. What is the value of the Next Header field? Fragment Header for IPv6 (44)
   * Identify all the fields in the Fragment Header.

Fragment Header for IPv6

Next header: UDP (17)

Reserved octet: 0x00

0000 0100 1101 0... = Offset: 154 (1232 bytes)

.... .... .... .00. = Reserved bits: 0

.... .... .... ...0 = More Fragments: No

Identification: 0x8b48ced7

[2 IPv6 Fragments (1889 bytes): #130(1232), #131(657)]

User Datagram Protocol, Src Port: 36078, Dst Port: 10086

Data (1881 bytes)

### Exercise 3-d. TCP and IPv6 Fragmentation

TCP also collaborates with IPv6 to prevent fragmentation inside the network at the routers, this section is dedicated to demonstrating how TCP works with IPv6 fragmentation and how IPv6 MTU probing operates.

1. If you are continuing for Exercise 3-c. above skip to Step 4. If not, configure the two routers, *Router1* and R*outer2* using the IPv6 addresses given in Table 6.2. Commands are given in Exercise 3-c. Step 1. for *Router1*. Repeat for *Router2*.
2. The PCs will automatically autoconfigure both a link-local address and a global unicast address.
3. Before continuing with this exercise, we need to modify the MTU cache expiration time of IPv6 on the Linux hosts. By default, this cache lasts for 600 seconds, that is too long for our experiments. We need to modify this value so we wait for a shorter time to observe certain behaviors. Use the following command to change the expiration time on a Linux PC. Shown here for *PC1*. You do not need to do this for *PC3*.

PC1$ sudo sysctl -w net.ipv6.route.mtu\_expires=60

1. If you are continuing from Exercise 3-b., you can skip to Step 5. If not, as before, we need to disable **SACK** and **Timestamps** on the PCs using the following commands, shown here for *PC1*. Repeat for *PC3*.

PC1$ sudo sysctl -w net.ipv4.tcp\_timestamps=0

PC1$ sudo sysctl -w net.ipv4.tcp\_sack=0

1. First start a nc receiver with TCP over IPv6 on *PC3* with the following command:

PC3$ nc -l 10086

1. Start a Wireshark capture on interface eth0 on *PC1* and interface S2/0 on *Router1*.
2. On *PC1*, create a file (segment payload) to prevent fragmentation. To calculate the maximum value of **N** that will allow the payload to fit in an MTU of 1280 you need to realize that the IPv6 header is 40 bytes and TCP without options is 20 bytes, so **N** = 800 is a good choice. Create the file using the following command:

PC1$ for i in {0..N}; do echo -n “a” >> a.txt; done;

1. Using the autoconfigured global unicast IPv6 address of *PC3* to initiate an nc client on *PC1* to send the file “a.txt” to server *PC3*:

PC1$ cat a.txt | nc -6 *IPv6\_address\_of\_PC3* 10086

1. Now, on *PC1*, first delete file “a.txt” and then create a new file that is bigger than > 1280 bytes that will guarantee the need to fragment. A value of **N** = 1360 will guarantee fragmentation at the router interface. Using the following command create file “a.txt”.

PC1$ for i in {0..N}; do echo -n “a” >> a.txt; done;

1. Using the autoconfigured global unicast IPv6 address of *PC3* to initiate an nc client on *PC1* to send the file “a.txt” to server *PC3*:

PC1$ cat a.txt | nc -6 *IPv6\_address\_of\_PC3* 10086

1. ![A picture containing text, iPod, electronics

   Description automatically generated]()Stop the Wireshark captures and save.

Lab Questions/Report:

Use the saved Wireshark output to answer the following:

1. Explain how IPv6 MTU path discovery works using what you observed in the Wireshark outputs. Show the ICMPv6 packets that were used to communicate the between *Router1* and *PC1*.

* Identify what field in the ICMPv6 message forced a change at PC1. What was changed at PC1?
  + THE R1 sends packet too big with MTU 1280 after which PC1 sends packets with 1294 total size and total tcp payload 1208. Which is success.

1. Compare the data transmissions from Step 8 to that in Step 10. Describe what you observed. How many segments were transmitted at the *PC1* interface in each step. 1st fail only one 1447 then after packet too big: two segments of size 1294 + 239 ( barring out of order or other retransmissions).

## Part 5. 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.
2. 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 *Karn’s algorithm*, measurements are not taken for retransmitted segments. Instead, when a retransmission occurs, the current RTO value is simply doubled.

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

A disadvantage of cumulative acknowledgements in TCP is that a TCP receiver cannot request the retransmission of specific segments. For example, if the receiver has obtained segments 1, 2, 3, 5, 6, 7 cumulative acknowledgements only permit to send 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.

### Exercise 5-a. TCP Retransmissions using GBN

The purpose of this exercise is to observe TCP retransmissions using GBN. Set up the GNS3 network configuration using the topology shown in Figure 6.1 and the IP addresses as given in Table 6.1. If you are continuing from Part 4, make sure you delete the IPv6 addresses on the routers before you configure the routers with the IPv4 as given in Table 6.1 and reset all MTU and timer values.

For the PCs, make sure you **disable** “SACK”.

As before, we transmit data from the client *PC1* to the listening server on *PC3*. The data rate of the serial link in this part is set to 1200bps. When you disconnect one of the links (shutdown), ACKs cannot reach the sending host. As a result, a timeout occurs and the sender performs retransmissions.

1. Set the data rate (referred to as clock rate on Cisco routers) of the Serial link to 1200bps. Here we show the commands for *Router1*, repeat for *Router2*:

Router1# configure terminal

Router1(config)# interface Serial2/0

Router1(config-if)# clock rate 1200

1. 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, and then clicking the right arrow, (  ) next to the display filter bar.
2. Test the link by issuing a ping from *PC1* to *PC3* for a count of -c 5. It should be successful.
3. Start an nc listening/receiving server on *PC3*:

PC3$ nc -l 10086

1. Start Wireshark capture on interface eth0 on *PC1* to the switch.
2. Now generate a file “a.txt” on *PC1* that is very large: N=1000000:

PC1$ for i in {0..1000000}; do echo -n “a” >> a.txt; done;

1. Send file “a.txt” from *PC1* to *PC2* using UDP with the following nc command:

PC1$ cat a.txt | nc 10.0.3.33 10086

1. Wait for five seconds after starting the file transfer above, then bring down interface Serial2/0 ion *Router1.*

Router1(config)# interface Serial2/0

Router1(config-if)# shutdown

1. Wait for five seconds, then bring interface Seria2/0 on *Router1* back up again.

Router1(config)# interface Serial2/0

![A picture containing text, iPod, electronics

Description automatically generated]()Router1(config-if)# no shutdown

1. When the nc file transmit is complete (you see no more activity in the Wireshark data capture window), stop the Wireshark traffic capture and save the output data.
2. Leave the clock rate as if for the next exercise on retransmissions with SACK.

Lab Questions/Report: **DO AGAIN**

Analyze the Wireshark output and answer the following questions:

* **When you brought down the serial link, observe the time instants when retransmissions took place. How many packets were retransmitted at one time? one**
* Did you observe the cumulative ACKs? And the discards at the receiver due to out of order delivery? (i.e., the receiver sent a duplicate ACK) yes
* Try to derive the time when a packet is retransmitted. Use data to back-up your answer.
  + Roughly 10 seconds according the sequence of a retransmitted packet.
  + Got another with 50 seconds

### Exercise 5-b. TCP Retransmissions using SACK

Repeat the above experiment, but now you enable “SACK” on the PCs. If you are starting a new, then you need to set the clock rate as shown in Exercise5-a. and follow all the steps as given above **EXCEPT do not disable** SACK on the PCs.

1. If you are continuing form 5-a., before you proceed with the experiment, **enable SACK** (selective repeat error recovery), on the two PCs, *PC1* and *PC3*, as follows: (shown here for *PC1*. Repeat for *PC3)*

PC1$ **sudo sysctl -w net.ipv4.tcp\_sack=1**

Or

PC1$ **sudo echo “1” > /proc/sys/net/ipv4/tcp\_sack**

1. Repeat the same steps as in Exercise 5-a.

Lab Questions/Report:

Analyze the Wireshark output and answer the following questions:

* **When you brought down the serial link, observe the time instants when retransmissions took place. How many packets were retransmitted at one time?**
* Did you observe the selective ACKs? (yes in tcp options SACK) Did you observe any discards at the receiver due to out of order delivery? (i.e., observe any duplicate ACK). (yes dup sack)
* Try to derive the time when a packet is retransmitted. Use data to back-up your answer.
  + Got one for 22 seconds and another with 40 seconds.

1. To **disable relative sequence numbers** and instead display them as the real absolute **numbers**, go to the TCP preferences and untick the box for **relative sequence numbers**.

   **Edit > Preferences > Protocols > TCP**

   Uncheck the option: "*Relative sequence numbers*" [↑](#footnote-ref-1)