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 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 of 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 in Part 5 when we study TCP and reliable delivery.

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 1 below with IP addresses as shown in Table 1. Part 2 introduces the "netcat" command (nc) and compares the performance of data transmissions in TCP and UDP. Part 3 explores how TCP and UDP deal with IP fragmentation. The remaining parts address important components of TCP. Part 4 explores connection management in different settings; Part 5 explores TCP retransmissions.

All parts in this lab use the topology as shown in Figure 1 using IP addresses as given in Table 1. Before setting up the network configuration in Part 1, you will be asked to first change the interface configuration of the Cisco 3640 Router to include a **Serial Interface** in slot 3. A serial connection is a direct, physical-layer connection. We use it here to simplify the configuration of the topology.

In this lab we only use the virtual LINUX devices Alice and Bob. No VPCs.

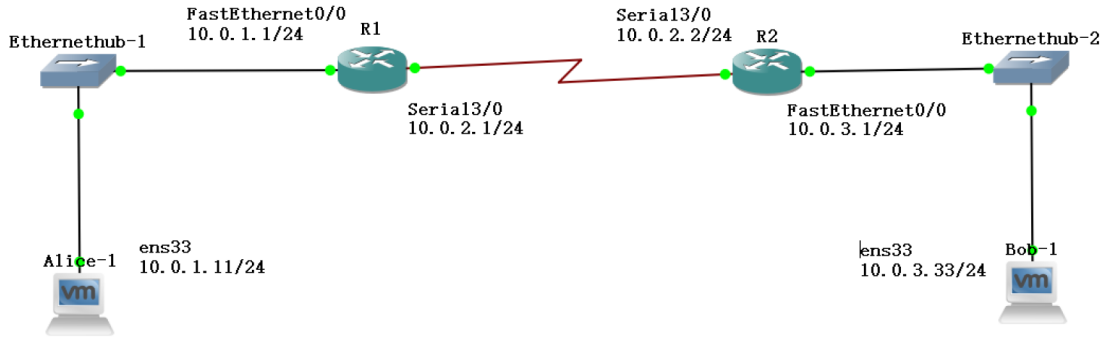

Figure 1 Network topology for the Lab

As discussed in our lectures, each Network Card ("Interface") must be assigned an IP address in order to connect to a network such as the Internet.

- **A personal computer** usually has **one network card**, and in a Linux OS it has the user-friendly identifier of "eth0". To assign an IP to this network card, we use the "ifconfig" system command, as shown in the table below. Additionally, to complete the configuration of a PC we need to define a "gateway" IP. This is the network card IP that is found at the other side of the Ethernet cable. It acts as the "next hop" for the PC-to-network communication, and is defined via the "route" command, as shown below.
- **A router** usually has many network cards, and its role is to dispatch incoming packets from one card to another (i.e., from an input to an output). Each separate network card must be assigned an IP address. Moreover, we need to define "routing rules", i.e., how to map input-to-output cards. This configuration is done via the command shown in the table below.

PART 1. Setting up the network topology using the serial interfaces on the Router

Exercise 1(A) Cisco Router Setup with Serial Interface

1. Do this before you start your project - **do not** drag any routers to the project screen before configuring the serial interface on the router image as shown below.

2. Go to:
   - Edit -> Preferences
   - **(Mac**: GNS3 -> Preferences)

3. In the left-hand pane, click on the arrow next to “Dynamips”, then click on the sub-menu “IOS routers” and click “Edit” as shown in Figure 2 below.

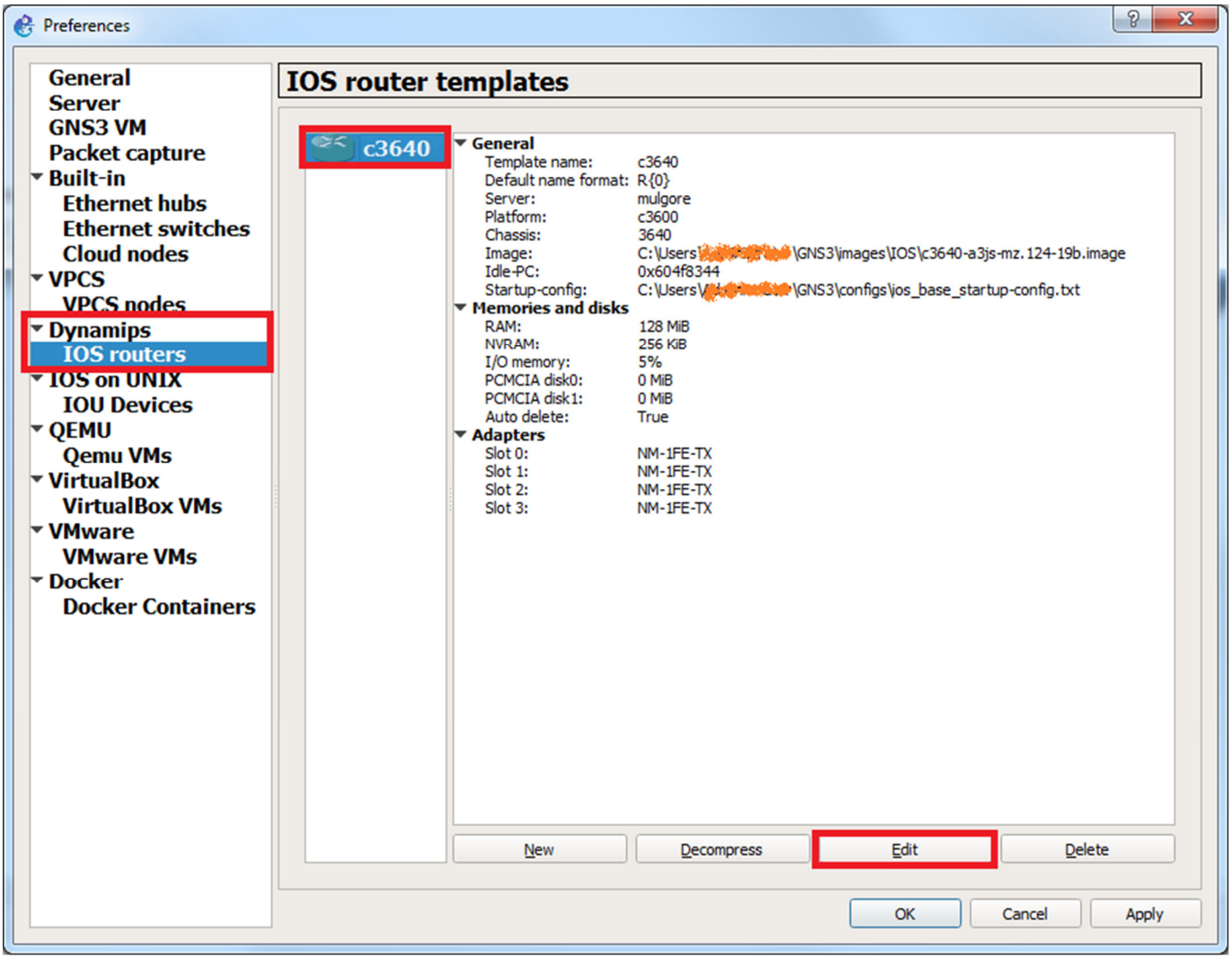

Figure 2 Router Interface Setup

4. Then click on Slots as shown in Figure 3 below. Select slot 3 under Adapters and from the dropdown menu select NM-4T as highlighted in the figure.

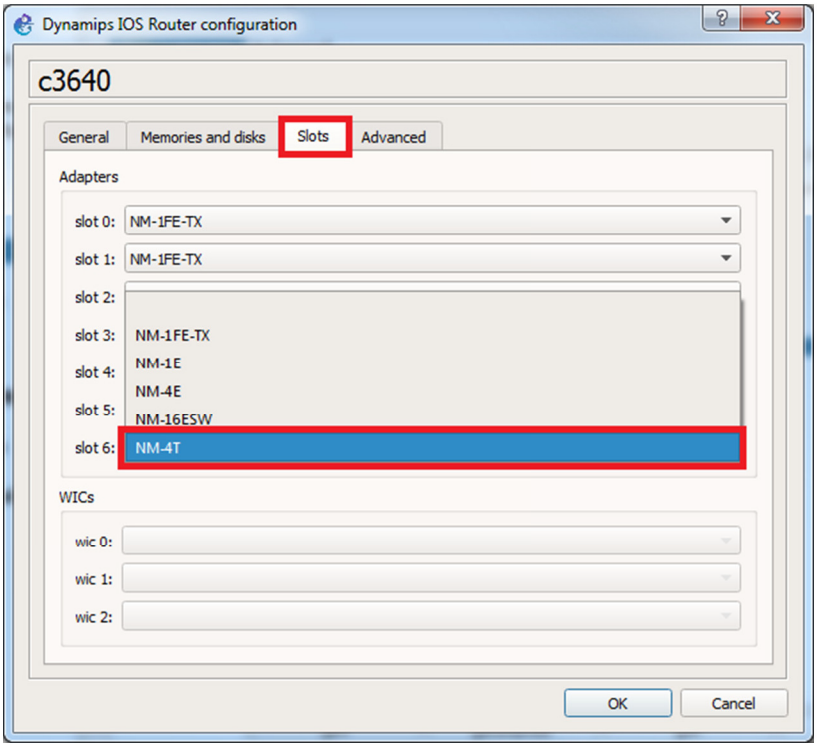

Figure 3 Slot Allocation

5. You should see the following. Click OK.

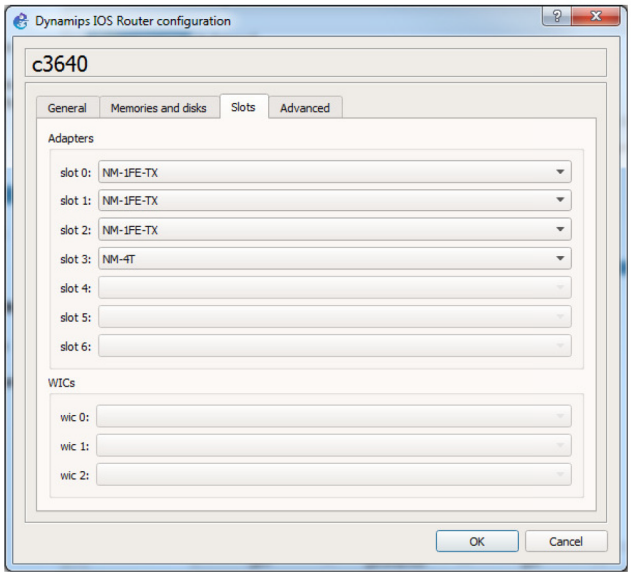

Figure 4

6. Then in the next screen, click Apply then OK. Your router will now have a serial interface with 4 links (Serial3/0-3).

Exercise 1(B). Network setup

1. Connect the network as shown in Fig 1, using a serial link between R1 and R2. Configure the serial interface on R1.

```
R1# conf terminal
R1(config)# interface Serial3/0
R1(config-if)# shutdown
R1(config-if)# ip address 10.0.2.1 255.255.255.0
R1(config-if)# no shutdown
R1(config-if)# exit
```

2. Do the same for router R2 and its serial interface. How should you change the commands above?

3. Configure all other networks components as shown below:

For now, we will use the following table as a cheat-sheet, to get an early feeling of these commands. In the coming labs we will study these commands and their meaning in more detail!

PCs	**eth0**		**Default Gateway**
Alice	10.0.1.11 / 24		10.0.1.1
REMINDER: You can connect to the VM with username: root password: cisco ➢ ifconfig eth0 10.0.1.11/24 ➢ route add default gw 10.0.1.1			
Bob	10.0.3.33 / 24		10.0.3.1
REMINDER: You can connect to the VM with username: root password: cisco ➢ ifconfig eth0 10.0.3.33/24 ➢ route add default gw 10.0.3.1			
Routers	**FastEthernet0/0**	**Ser3/0**	**Default Gateway**
R1	10.0.1.1/24	10.0.2.1/24	10.0.2.2
➢ enable ➢ config t ➢ interface f0/0 ➢ shutdown ➢ ip address 10.0.1.1 255.255.255.0 ➢ no shutdown ➢ exit ➢ exit ➢ config t ➢ ip route 10.0.3.0 255.255.255.0 10.0.2.2 ➢ exit			
R2	10.0.3.1/24	10.0.2.2/24	10.0.2.1

```
➢ enable
➢ config t
➢ interface f0/0
➢ shutdown
➢ ip address 10.0.3.1 255.255.255.0
➢ no shutdown
➢ exit
➢ config t
➢ ip route 10.0.1.0 255.255.255.0 10.0.2.1
➢ exit
```

Table 1 - IP addresses of the VMs and Routers

4. Verify that the setup is correct by issuing a `ping` command from Alice-1 to Bob-1.

   ```
   Alice-1% ping 10.0.3.33 –c 5
   ```

   ```
   Bob-1% ping 10.0.1.11 –c 5
   ```

5. Now we disable **SACK** on both Alice-1 and Bob-1 with the following command:

   ```
   Alice-1% sysctl -w net.ipv4.tcp_sack=0
   ```

   ```
   Bob-1% sysctl -w net.ipv4.tcp_sack=0
   ```

6. The console will display the following confirming the change:

   ```
   net.ipv4.tcp_sack = 0
   ```

7. ALTERNATIVELY you can use:

   ```
   Alice-1% echo 0 > /proc/sys/net/ipv4/tcp_sack
   ```

   ```
   Bob-1% echo 0 > /proc/sys/net/ipv4/tcp_sack
   ```

PART 2 Data Transmission using Netcat - “nc” command

In this lab we will use `netcat` or `nc` as the actual program is called, to test TCP and UDP transmissions. The following tables summarize the main uses of the command `nc`.

NAME

nc - TCP/IP swiss army knife

SYNOPSIS

nc *[-options] hostname port[s] [ports] ...*
nc *-l -p port [-options] [hostname] [port]*

DESCRIPTION

netcat 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. Netcat, or "nc" as the actual program is named, should have been supplied long ago as another one of those cryptic but standard Unix tools.

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.

You may be asking "why not just use telnet to connect to arbitrary ports?" Valid question, and here are some reasons. Telnet has the "standard input EOF" problem, so one must introduce calculated delays in driving scripts to allow network output to finish. This is the main reason netcat stays running

until the *network* side closes. Telnet also will not transfer arbitrary binary data, because certain characters are interpreted as telnet options and are thus removed from the data stream. Telnet also emits some of its diagnostic messages to standard output, where netcat keeps such things religiously separated from its *output* and will never modify any of the real data in transit unless you *really* want it to. And of course, telnet is incapable of listening for inbound connections, or using UDP instead. Netcat doesn't have any of these limitations, is much smaller and faster than telnet, and has many other advantages.

OPTIONS

-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

-G num
source-routing pointer: 4, 8, 12, ...

-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.

-b
allow UDP broadcasts

-r
randomize local and remote ports

-s addr
local source address

-t
enable telnet negotiation

-u
UDP mode

```
-v
      verbose [use twice to be more verbose]
-w secs
      timeout for connects and final net reads
-C
      Send CRLF as line-ending
-z
      zero-I/O mode [used for scanning]
-T type
      set TOS flag (type may be one of "Minimize-Delay", "Maximize-Through-
      put", "Maximize-Reliability", or "Minimize-Cost".)

      Read more at: https://www.commandlinux.com/man-page/man1/nc.1.html
```

Exercise 2(A). Transmitting data with TCP

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 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 display, Wireshark will display a SEQ and ACK number 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 - (2^32)-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 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 untick it. When done with the questions below, go back and “tick” enable “Relative sequence number” again. Relative numbers are easier to work with.

1. Start Wireshark on link connecting Hub1 to Alice-1 to capture the traffic exchange.

2. Start the nc listening command on Bob-1 on port 10086 (Bob-1 is acting as the server) so that it can receive packets being sent to it from Alice-1 (Alice-1 is acting as the client). The port number above is randomly picked, you are free to pick any free port you like.

   ```
   Bob-1% nc -l -p 10086
   ```

3. Use the nc command to establish a TCP connection from Alice-1 to Bob-1.

   ```
   Alice-1% nc 10.0.3.33 10086
   ```

4. Now send the following message from Alice-1 to Bob-1. Type it in Alice-1's console and press enter when done. On Bob-1's console, you should be able to see Alice-1's message echoed on the screen. Watch the traffic capture, you should see a TCP connection.

   ```
   Alice-1% Hello Bob! Have not seen you in ages. Hope all is going well.
   ```

5. Stop the nc processes running on Alice-1 and Bob-1. Terminate the process with:

   ```
   Ctrl+C (^C)
   ```

6. Stop the data capture. Save the Wireshark output.

Self-practice material – interesting topics to study at home

TCP Connection Set-up and Teardown. When done, keep the Wireshark output for next set of questions after UDP transmission.

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.
- During the connection setup, the TCP client and TCP server tell each other the initial sequence number (ISN#) they will use for data transmission. What are the initial sequence numbers of the TCP client and the TCP server? Note: Wireshark displays ISN as starting at Seq=0, that is not the real ISN. To view the real ISN#, you have to deselect "Relative sequence number" as directed in the introduction to Exercise 1(C).
- Identify the first packet that contains application data. What is the sequence number used in the first byte of application data sent from the TCP client to the TCP server?
- 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 window sizes for the TCP client and the TCP server.
- What is the MSS value that is negotiated between the TCP client and the TCP server?
- Describe the closing process of the TCP connection?

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 in these packets? Explain how these flags are interpreted by the receiving TCP server and TCP client. How many transmissions were involved in the tear down?

Exercise 2(B) Transmitting UDP data.

1. Start Wireshark on link connecting Hub1 to Alice-1 to capture the traffic exchange.

2. Start the `nc` listening command on Bob-1 with the UDP option `-u` to receive UDP packets being sent from Alice-1:

   ```
   Bob-1% nc -u -l -p 10086
   ```

3. Establish a connection between Alice and Bob using UDP.

   ```
   Alice-1% nc -u 10.0.3.33 10086
   ```

4. Now send the following message from Alice-1 to Bob-1. Type it in Alice-1's console and press enter when done. On Bob's console, you would be able to see the same message echoed on the screen. Check the Wireshark capture. This time the message is being transmitted using the UDP protocol.

   ```
   Alice-1% Hello Bob! Have not seen you in ages. Hope all is going well.
   ```

5. Terminate the `nc` process on each of Alice-1 and Bob-1 with `^C`.

6. Save the Wireshark output. Stop data capture.

Self-practice material – interesting topics to study at home

Comparing TCP and UDP connections and data transmission.

Use the data captured with Wireshark in Exercises 2(A) and 2(B) to answer the following questions. **Note**, you should have enabled the Relative sequence number feature again to make traffic analysis easier.

- How many data packets are transmitted by Alice, and how many data packets are transmitted by Bob for both TCP and UDP? How many ACKs does TCP send in both directions?
- What are the sizes of the TCP segment? UDP segment?
- Compare the total number of overhead bytes transmitted, in both directions, that is Ethernet, IP, UDP/TCP headers, to the amount of application data transmitted. How much more overhead data does TCP incur compared to UDP?
- Which flags are set in the TCP data packets?
- Take the largest UDP segment and the largest TCP segment that you observed and compare the amount of application data that is transmitted in each. Are they different?

PART 3. IP Fragmentation of UDP and TCP Traffic

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

When an IP datagram is fragmented, its payload is split across 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 the higher layers, it incurs considerable overhead and, therefore, should be avoided. TCP tries to avoid fragmentation with a Path MTU Discovery scheme that determines a maximum segment size (MSS), which does not result in fragmentation. UDP in recent years has adopted it too.

In this part, you explore the issues with IP fragmentation of TCP and UDP transmissions in the network configuration shown in Figure 1 and Table 1, with Alice-1 as sending host, Bob-1 as receiving host.

Exercise 3(A). UDP and Fragmentation

In this exercise you will observe IP fragmentation of UDP traffic. As before, you will use `nc` to generate UDP traffic between Alice-1 and Bob-1, and gradually increase the size of UDP segment until fragmentation occurs. You will observe that the IP header does not have the DF bit set (i.e., DF=1) for UDP payloads.

To observe the UDP segment size at which fragmentation occurs, we gradually increment the size of the UDP segment by increasing the size of the file we are transmitting between the two hosts. The file is called "a.txt" and resides on the client Alice-.

1. Start Wireshark on link connecting Hub1 to Alice-1 to capture the traffic exchange.

2. Start the `nc` listening command on the server Bob-1 with the UDP option `-u` to receive UDP packets being sent from client Alice-1:

   ```
   Bob-1% nc -u -l –p 10086.
   ```

3. Use the following command on Alice-1 to create a file of size **N** bytes, called "a.txt" with the character "a".

   ```
   Alice-1% for i in {0..N}; do echo -n "a" >> a.txt; done;
   ```

Start with N=256. You may want to create many filenames, e.g., a256.txt, a512.txt, a1024.txt and so on, to avoid reusing the same file erroneously. In that case, obviously change 'a.txt' to the new filename in the commands that follow.

4. Use the following command to transfer file "a.txt" using UDP with the `nc` command.

```
Alice-1% cat a.txt | nc -u 10.0.3.33 10086
```

5. Repeat steps 3-4 with a larger file size by changing the parameter "**N**" to 512, 1024, 2048, …. until you observe fragmentation in the Wireshark capture.

6. Terminate the `nc` process with `^C`. Stop the traffic capture and save the Wireshark output.

Self-practice material – interesting topics to study at home

- Determine the UDP segment size at which fragmentation occurs.
- Determine the maximum size of the UDP segment (whole or fragment) that the system can transport.
- From the 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, Fragment Offset, Don't Fragment Bit, More Fragments Bit).

Exercise 3(B). TCP and Fragmentation

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

- When a TCP connection is established, it negotiates the maximum segment size (MSS) to be used. 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, when it transmits segments. The smaller value is adopted as the MSS value for the connection.

- The exchange of the MSS addresses MTU constraints only at the hosts, not at the intermediate routers. To determine the smallest MTU on the path from the sender to the receiver, TCP employs a method known as Path MTU Discovery, which works as follows. The sender always sets the DF bit in all IP datagrams. When a router needs to fragment an IP packet with the DF bit set, it discards the packet and generates an ICMP error message of type "destination unreachable; fragmentation needed". Upon receiving such an ICMP error message, the TCP sender reduces the segment size. This continues until a segment size that does not trigger an ICMP error message is determined.

1. Modify the MTU of the serial interfaces on the routers with the values shown in Table 2.

Routers	**MTU size on Serial3/0**
R1	500
R2	500

Table 2. MTU sizes.

In Cisco IOS, you can view the MTU values of all interfaces using the `show interfaces` command. For example, on **R1**, you type:

```
R1> enable
R1# show interfaces
```

The command to modify the MTU value is as follows:

```
R1# configure terminal
R1(config)# interface Serial3/0
R1(config-if)# mtu 500
```

2. Make sure that MTU Path Discovery is activated (MTU probing) on Alice-1 and Bob-1.

   You can check if probing is set by using the `syctl` command. 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 Alice-1, type:

   ```
   Alice-1% sysctl net.ipv4.tcp_mtu_probing
   ```

   Enable MTU probing on Alice-1 and Bob-1 with the following command:

   ```
   Alice-1% sysctl -w net.ipv4.tcp_mtu_probing=2
   ```

   Or

   ```
   Alice-1% echo "2" > '/proc/sys/net/ipv4/tcp_mtu_probing'
   ```

3. Make sure that probing is set (enabled) on both **Alice-1** and **Bob-1**.

4. Start the `nc` listening command on Bob-1 so that you will be able to receive packets being sent from Alice-1:

   ```
   Bob-1% nc -l –p 10086
   ```

5. Start Wireshark on link connecting Hub1 to Alice-1 to capture the traffic exchange.

6. Transmit from Alice-1 to Bob-1 the file "a.txt" you created in part 2(A) (use the last one you created with largest value for N).

   ```
   Alice-1% cat a.txt | nc 10.0.3.33 10086
   ```

7. Terminate the `nc` process with `^C`. Stop the traffic capture and save the Wireshark output.

8. When done with this exercise, **reset** the MTU value to 1500 on Serial3/0 interface of both routers.

Self-practice material – interesting topics to study at home

- Do you observe fragmentation? If so, where does it occur? Explain your observation.
- Describe how ICMP error messages are used for Path MTU Discovery. Look at the first TCP segment that is sent after Alice-1 has received each ICMP error message. Note the segment size.