LAB FIVE - 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 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 5.1 below with IP addresses as shown in Table 5.1. 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. In part 6 we introduce a TCP security vulnerability.

All parts in this lab use the topology as shown in Figure 5.1 using IP addresses as given in Table 5.1. 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. In this lab we only use the Ubuntu devices Alice and Bob. No PCs. In part 6 you will remove router R2 and add Mallory to the configuration as shown in Figure 5.5 and use Table 5.3 to setup the IP addresses.

Figure 5.1 Network topology for Lab 5

|  |  |  |  |
| --- | --- | --- | --- |
| **PCs** | **eth0** |  | **Default Gateway** |
| Alice | 10.0.1.11 / 24 |  | 10.0.1.1 |
| Bob | 10.0.3.33 / 24 |  | 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 |
| R2 | 10.0.3.1/24 | 10.0.2.2/24 | 10.0.2.1 |

Table 5.1 IP addresses of the PCs and Routers

# 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:
* **Windows**: Edit -> Preferences
* **Mac**: GNS3 -> Preferences
1. 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 5.2 below.



Figure 5.2 Router Interface Setup

1. Then click on Slots as shown in Figure 5.3 below. Select slot 3 under Adapters and from the dropdown menu select NM-4T as highlighted in the figure.



Figure 5.3 Slot Allocation

1. You should see the following. Click OK.



Figure 5.4

1. 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

Please follow these steps for every part and exercise. You need to **disable SACK** (or selective repeat), switch to **super user (su root)** on the VMs, and use the **serial** interfaces.

1. Connect the Ethernet interfaces of Alice and Bob as shown in Figure 5.1. Configure the IP addresses of the interfaces and default gateways as given in Table 5.1. These do not change through-out the lab.
2. Recall that you need to switch to root for the Ubuntu hosts using the following command:

**Alice**% su root

**Bob**% su root

1. Connect the routers as shown, using a serial link between R1 and R2. Configure the IP addresses of the interfaces and default gateways as given in Table 5.1. Below we show how to configure the serial interface on R1.

**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

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

**Alice%** ping 10.0.3.33 –c 5

**Bob%** ping 10.0.1.11 –c 5

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

**Alice%** sysctl -w net.ipv4.tcp\_sack=0

**Bob%** sysctl -w net.ipv4.tcp\_sack=0

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

net.ipv4.tcp\_sack = 0

1. Or you can use:

 **Alice%** echo “0” > /proc/sys/net/ipv4/tcp\_sack

 **Bob%** echo “0” > /proc/sys/net/ipv4/tcp\_sack

# PART 2 Data Transmission using Netcat - “nc” command on Ubuntu

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.

|  |
| --- |
| NAMEnc - 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-Throughput", "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 to capture the traffic exchange.
2. Start the nc listening command on Bob on port 10086 (Bob is acting as the server) so that it can receive packets being sent to it from a client (e.g., Alice, acting as a client). The port number above is randomly picked, you are free to pick any free port you like.

 **Bob%** nc -l 10086

1. Use the nc command to establish a TCP connection from client Alice to server Bob.

 **Alice%** nc 10.0.3.33 10086

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

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

1. Stop the nc processes running on both Alice and Bob using the following command on each terminal:

 Ctrl+C (^C)

1. In the traffic capture window you should see the TCP connection being terminated.
2. Stop the data capture. Save the Wireshark output.

**Lab Questions**

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 to capture the traffic exchange.
2. Start the nc listening command on Bob with the UDP option -u to receive UDP packets:

 **Bob%** nc -u -l 10086

1. Establish a connection between Alice and Bob using UDP with the following command:

 **Alice%** nc -u 10.0.3.33 10086

1. Now send the following message from Alice to Bob. Type it on Alice’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%** Hello Bob! Have not seen you in ages. Hope all is going well.

1. Terminate the nc process on each of Alice and Bob with ^C.
2. Stop the traffic capture and save the Wireshark output.

**Lab Questions:**

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 5.1 and Table 5.1, with Alice as sending host (client), Bob as receiving host (server).

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

**R**> enable

**R#** show interfaces

The command to modify the MTU value on interface **int** from default to **size** is as follows:

**R#** configure terminal

**R(config)#** interface **int**

**R(config-if)#** mtu **size**

### 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 and Bob, and gradually increase the size of UDP segment until fragmentation occurs. Normally you should observe that the IP header does not have the DF bit set (i.e., DF=1) for UDP payloads. However protocol implementations differ when it comes to end systems and it seems that for Linux we have observed that that the IP header has the DF bit set (i.e., DF=1) for UDP payloads less than the MTU size of the local interface, and the IP header will have DF bit unset (i.e., DF=0) if the packet size exceeds the MTU of the local interface. We will start off with the default MTU size and then change that to show you the above implementation issue oddity.

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. First check the default MTU size on the routers using the command shown above.
2. Start Wireshark on link connecting Hub1 to Alice to capture the traffic exchange.
3. Start the nc listening command on the server Bob with the UDP option -u to receive UDP packets being sent from client Alice:

**Bob%** nc -u -l 10086.

1. Use the following command on Alice to create a file of size **N** bytes, called “a.txt” with the character “a”:

**Alice%** for i in {0..N}; do echo -n “a” >> a.txt; done;

Start with N=256.

1. Use the following command to transfer file “a.txt” from Alice to Bob using UDP with the nc command:

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

1. Repeat steps 4-5 with a larger file size by changing the parameter “**N**” to 512, 1024, 2048, …. until you observe fragmentation in the Wireshark capture. **Delete** “a.txt” for every value of N before you do the next one using command:

**Alice%** rm a.txt

1. Delete “a.txt” on Alice. Stop the traffic capture and save the Wireshark output.
2. Now we will change the MTU size on router **R1** interface Serial 3/0 to 500 as follows:

 **R1#** configure terminal

 **R1(config)#** interface Serial3/0

 **R1(config-if)#** mtu 500

1. Start Wireshark capture on link connecting Hub1 to Alice.
2. Now generate file “a.txt” on Alice that is just a bit larger than the MTU of the R1 interface Serial3/0,

**Alice%** for i in {0..512}; do echo -n “a” >> a.txt; done;

1. Send file “a.txt” from Alice to Bob using UDP with the following nc command:

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

1. Delete file “a.txt” on Alice. Terminate the nc process with ^C on both Bob and Alice. Stop the traffic capture and save the Wireshark output.

**Lab Questions:**

* What is the default MTU size on the Serial Interface (ser3/0).
* Determine the UDP segment size at which fragmentation occurs for the case when using the default MTU value.
* 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).
* For steps 8-11, with MTU set to 500, what did you observe that was different from the case with the default MTU?

### 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 5.2.

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

Table 5.2. MTU sizes.

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

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, type:

**Alice%** sysctl net.ipv4.tcp\_mtu\_probing

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

**Alice%** sysctl -w net.ipv4.tcp\_mtu\_probing=2

Or

**Alice%** echo "2" > '/proc/sys/net/ipv4/tcp\_mtu\_probing'

1. Make sure that probing is set (enabled) on both **Alice** and **Bob**.
2. Start the nc listening command on Bob so that you will be able to receive packets being sent from Alice:

**Bob%** nc -l 10086

1. Start Wireshark on link connecting Hub1 to Alice to capture the traffic exchange.
2. Transmit from Alice to Bob the file “a.txt” you created in part 3(A) (use the last one you created with largest value for N).

**Alice%** cat a.txt | nc 10.0.3.33 10086

1. Terminate the nc process on both Bob and Alice with ^C. Stop the traffic capture and save the Wireshark output.
2. When done with this exercise, **reset** the MTU value to 1500 on Serial3/0 interface of both routers.

**Lab Questions:**

* 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 has received each ICMP error message. Note the segment size.

# PART 4. TCP and connections to non-existing entities

### The network configuration for this part is the network as given in Figure 5.1 and Table 5.1.

### Exercise 4(A). Requesting a connection to a non-existing host

In this exercise we observe how a TCP client tries to establish a connection to a host that does not exist.

1. Start Wireshark on link connecting Hub1 to Alice to capture the traffic exchange.
2. Set a static entry in the ARP table of Alice for a non-existing IP address **10.0.1.100**:

**Alice%** arp –s 10.0.1.100 00:01:02:03:04:05

1. From Alice, establish an nc session to port 10086 on a non-existing host:

**Alice%** nc 10.0.1.100 10086

1. Terminate the Wireshark traffic capture and save the output.

**Lab Questions:**

* How often does the TCP client try to establish a connection? How much time elapses between repeated attempts to open a connection?
* Does the TCP client terminate or reset the connection when it gives up trying to establish a connection?
* Why does this experiment require setting a static ARP table entry?

### Exercise 4(B). Requesting a connection to a non-existing port

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

1. Start Wireshark on link connecting Hub1 to Alice to capture the traffic exchange.
2. Establish a TCP connection to port 80 of on Bob. Note that there is no TCP server running on Bob that is listening on this port number:

**Alice%** nc 10.0.3.33 80

1. Terminate the Wireshark traffic capture and save the output.

**Lab Questions:**

* How does TCP at the remote host close this connection?
* How long does the process of ending the connection take?

# 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:

* **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 time 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 (RTT) 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 acknowledgment for that segment. The RTT value is calculated based on these round-trip measurements. Following a heuristic called *Karn’s algorithm*, measurements are not taken for retransmitted segments. Instead, when a retransmission occurs, the current RTT 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 in indication that a packet has been lost. In this case, the TCP sender expedites a fast retransmit by sending an ACK for each packet that is received out of order.

The exercise in this part explores aspects of TCP retransmissions that do not require access to internal timers. Unfortunately, the round-trip time measurements and the RTT values are difficult to observe and are therefore not included in this lab.

The network configuration for this part is the network as given in Figure 5.1 and Table 5.1.

### Exercise 5(A). TCP Retransmissions

The purpose of this exercise is to observe when TCP retransmissions occur. In this part of the lab, you will transmit data from Alice to Bob. When you disconnect the connection to Bob, ACKs cannot reach the sending host Alice. As a result, a timeout occurs and the sender performs retransmissions.

1. Setting the clock rate on the Router serial link.

We will set the clock rate on the serial link to a very low bit rate. We show the commands here for the serial interface of **R1**. Repeat for **R2**.

**R1#** configure terminal

**R1(config)#** interface Serial3/0

**R1(config-if)#** clock rate 1200

This sets the clock rate of the serial link to 1200bps.

1. Test the link by issuing a ping from Alice to Bob. Proceed to step 6, if the ping is successful. If not, check the configuration again.
2. Start Wireshark on link connecting Hub1 to Alice to capture the traffic exchange. Set a display filter for TCP.
3. Start the nc listening command on Bob so that you will be able to receive packets from Alice:

**Bob%** nc -l 10086

1. Use the following command to create a huge (N=1,000,000) text file (a.txt) on Alice:

**Alice%** for i in {1..1000000}; do echo -n “a” >> a.txt; done;

1. Now transmit the huge file you just created to Bob using the nc command on Alice:

**Alice%** cat a.txt | nc 10.0.3.33 10086

1. Wait for five seconds after starting nc above, then bring down **R1** Serial3/0 interface.

**R1(config)#** interface Serial3/0

**R1(config-if)#** shutdown

1. Wait for five seconds after bringing the interface down and bring the Serial interface up again.

**R1(config)#** interface Serial3/0

**R1(config-if)#** no shutdown

1. When nc file transmit is complete is complete (you see no more activity in the Wireshark data capture pane), stop Wireshark traffic capture and save the output data.

**Lab Questions:**

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 cumulative ACKs? And the discards at the receiver due to out of order delivery? (i.e., the receiver sent a duplicate ACK)
* **Try** to derive the algorithm that sets the time when a packet is retransmitted. Use data to back-up your answer. Is there a maximum time interval between retransmissions?
* After how many retransmissions, if at all, does the TCP sender stop to retransmit the segment? Describe your observations.

# PART 6. Exploring security vulnerabilities in TCP

In the previous parts of this lab, we examined how a TCP connection is established and observed the data exchange between a client and a host during a TCP session.

This part of the lab aims to introduce one of the major security vulnerabilities in the design of the original TCP. Specifically, we will examine a security vulnerability that can occur during connection establishment.

For this part of the lab, you need to add to the network topology the Lubuntu VM Mallory as shown in Fig 5.5. Alice will act as a TCP server on the local network and Bob, as the client, will connect to Alice via a TCP connection. The attacker, in this case Mallory, will intervene to prevent a communication channel from being setup using a Denial of Service (DOS) attack.



Figure 5.5

Use the following table to configure the device network interfaces:

|  |  |  |  |
| --- | --- | --- | --- |
| **PCs** | **eth0** |  | **Default Gateway** |
| Alice | 10.0.1.11 / 24 |  | 10.0.1.1 |
| Bob | 10.0.2.12 / 24 |  | 10.0.2.1 |
| Mallory | 10.0.2.22 / 24 |  | 10.0.2.1 |
| **Routers** | **FastEthernet0/0** | **FastEthernet1/0** | **Default Gateway** |
| R1 | 10.0.1.1/24 | 10.0.2.1/24 |  |

Table 5.3

### Exercise 6(A). TCP Denial of Service (DOS) attack using SYN flooding

In this exercise we simulate a Denial of Service (DOS) attack emanating from Mallory to disrupt the establishment of a communication channel to the server Alice by flooding TCP SYN packets with random source IP addresses. Alice will be pre-occupied with sending SYN-ACK packets back to the non-existing IP addresses and in the process have lots of open connections waiting for ACKs to establish them. No responses will be forth coming and Alice will be left hanging with several timers waiting to establish or finally terminate the session setup attempts. This will prevent a legitimate client such as Bob to get access to Alice. It is possible that Bob might get his SYN packet thru to Alice. In this experiment you will be required to hop immediately between Bob’s console and Alice’s to hopefully observe the inability to setup a session (i.e., string of characters do not appear on Alice’s terminal after you press enter on Bob’s terminal). It may finally succeed as Bob will make several attempts, but if all works out as provisioned with Mallory, Bob will fail and stop trying.

To perform the TCP SYN flooding attack, you need to manually turn off the SYN flooding attack defense mechanism, ***SYN cookie***. We will first run the experiment with SYN cookie active and then show what happens when we deactivate it.

1. Start a TCP server on Alice with the following command:

**Alice%** nc -l 1234

1. Start a Wireshark capture on the link between Mallory and Ethernethub-1 and another capture on the link between Alice and R1.
2. Now on Mallory run the following command to flood SYN packets to Alice:

**Mallory%** hping3 –-flood -S -p 1234 --rand-source 10.0.1.11

1. On Bob, run the following command to setup a TCP connection to server Alice.

**Bob%** nc 10.0.1.11 1234

1. On Bob’s terminal, type a string of characters (e.g., Hello, I am BOB and want to chat with you). Press Enter to send the information to Alice. If the connection is successful, i.e., Mallory’s flooding of SYNs to Alice did not impact Alice, you will see the string of characters on Alice’s terminal.
2. Use ^C to terminate the nc program on Bob and Alice, and the hping3 command on Mallory.
3. Stop the Wireshark captures and save the output.
4. Now we will deactivate the SYN flooding defense mechanism on Alice. To do that you execute the following command to turn off the SYN cookie on Alice:

**Alice%** sysctl -w net.ipv4.tcp\_syncookies=0

1. Start the TCP server on Alice with the following command:

**Alice%** nc -l 1234

1. Start a Wireshark capture between Mallory and Ethernethub-1 and another capture between Alice and R1.
2. Now on Mallory run the following command to flood SYN packets to Alice:

**Mallory%** hping3 –-flood -S -p 1234 --rand-source 10.0.1.11

1. Re-initiate a TCP connection from Bob to Alice again using the following command

**Bob%** nc 10.0.1.11 1234

1. On Bob’s terminal, type a short string of characters (e.g., Hello, I am BOB). Press Enter to send the information to Alice and then switch immediately to Alice’s terminal.
2. On Alice, examine the Active Internet connection table using the following command. Take a screenshot of the output.

**Alice%** netstat -natp

1. Stop Wireshark captures and save the outputs.
2. Use ^C to terminate the nc program on Bob and Alice, and the hping3 command on Mallory.
3. Re-enable SYN cookie on Alice:

**Alice%** sysctl -w net.ipv4.tcp\_syncookies=1

**Lab Questions:**

* 1. Was the first connection between the client Bob and server Alice, was it successful, i.e., did Alice receive the string of characters from Bob? What did you observe in the Wireshark outputs to explain the success of the connection?
	2. For the second attempt between Bob and Alice, was the connection successful, i.e., did Alice receive the string of characters from Bob? What did you observe in the Wireshark outputs and Alice’s Active Internet connection table to explain the failure of the connection?