POLITECNICO DI TORINO

Master's Degree in Communications and Computer Networks Engineering

Network Measurment Laboratory

Supervisors	Candidate
Prof. Marco MELLIA	**Razieh HEIDARIAN**

01 2019

Table of Contents

Chapter 1

Echo services

1.1 Analysis of the ECHO service

First of all, we configured our test bed, we assigned IP address 172.16.0.1/26 to Host1, 172.16.0.2/26 to Host2, and 172.16.0.3/26 to Host3. By using this configuration, the network address is 172.16.0.0, while the broadcast address is 172.16.0.63, so we can have at most 62 hosts. Then we enabled time, chargen and echo services on all three hosts by editing the file "/etc/inetd.conf" in root mode, to have write permission, and adding these 3 lines:
time stream tcp nowait root internal
echo stream tcp nowait root internal
chargen stream tcp nowait root internal

then we restarted the inetd service by running the command "/etc/init.d/openbsd-inetd restart" and we checked the status of the running services both for TCP and UDP connections by using the command "netstat -t -u -l". Plus, offloading capabilities have been disabled through the command ethtool, in order to prevent the NIC to take care of control information of other layers and make it as simple as possible.

We run Wireshark and start capturing packets of our interface in the background. We use telnet to open a connection from one host to another host in the LAN (telnet allows us to open a remote connection).

To contact the time service Telnet asks TCP to open a connection, depending if there is service time running or not on the server, client will get the response (SYN/ACK or RST). In our server, time service is ON (each port has an address, referring to specific application or protocol, there are standard ports time service

port is 37), so TCP is able to connect successfully. Client sends SYN to open a connection and does synchronization, server will send a SYN/ACK and then client will answer it with an ACK, it completes three-way-handshake, server will send client some data (TIME response frame), then the client sends ACK back and the connection is closed immediately by server, server will send FIN-ACK, client will reply by sending FIN-ACK, and server finally send ACK to complete tear down phase. In figure 1.1 we see what happens in terminal when we telnet the time service and when this service is ON, plus in figure 1.5, we see in Wireshark three-way-handshake process, transferring data as well as tear down phase.

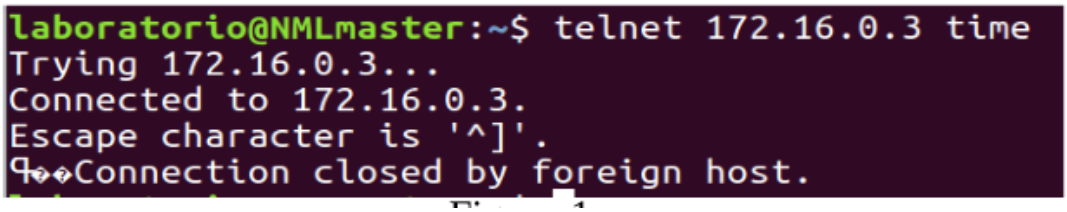

Figure 1.1: Telnet time

To contact echo server which is enable on our hosts, application asks TCP to open a connection to host 172.16.0.1 on port 7 which is port number of echo service. TCP sends a SYN, SYN gets received by server thanks to the lower layers, and then check if there is echo application running on the server, it can open a connection. Server sends a SYN/ACK (an ACK which also carries a SYN because TCP wants to open a bidirectional service, so needs synchronization) client receives a SYN/ACK and then reply with an ACK. So we see SYN then SYN/ACK and after that ACK which means three-way-handshake is successful because echo service on our hosts is ON. Telnet will answer I am able to open a connection, and it is connected to 172.16.0.1 as we can see in Figure 1.2. Now, there is a virtual connection between our telnet and echo server of host with IP address 172.16.0.1.

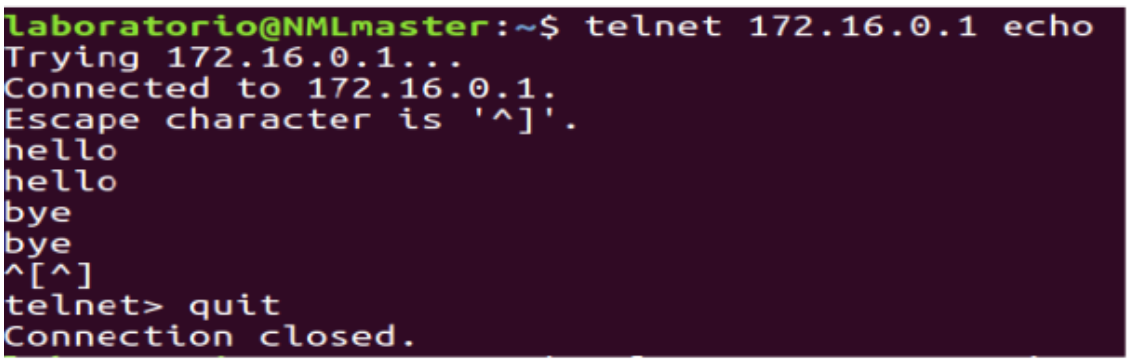

Figure 1.2: Telnet echo

whatever we type on our application, when we press return (Enter) button on the keyboard, will be sent to the server. Whole text will be sent to the server, not character by character. hence, it reduces lots of overhead. As it is depicted in Figure 1.3, the text is encoded by ASCII encoding and at the end of each message we must press Enter to transmit to the server which its ASCII code is “0d” for carriage return and “0a” for line feed. This gets sent to TCP which sends it to TCP of 172.16.0.1 then sent it to application which process it and eventually replies to us. As an example echo server receives a “hello” message, processes it and replies to us, echoing back what we typed which is “hello”. Whatever we type here will be sent as a request to other end and we will get back the response. Finally, when we are done, we want to close the connection. It will not be sent to the other end, then we enter to the command mode. To exit of telnet, we just type "quit". This tells telnet I want to exit, telnet will tell TCP I want to close connection, TCP will close the connection, the other TCP (on 172.16.0.1) receives FIN/ACK message which tells that client (172.16.0.2) wants to close the connection, echo answers FIN/ACK to accept closing, finally client sends an ACK to complete tear down phase. Connection is closed, TCP tells telnet I am done and telnet prints on screen “connection closed”.

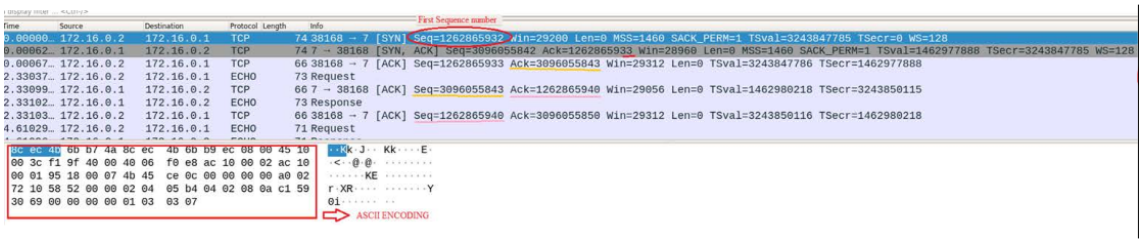

Figure 1.3: Wireshark analyze

After doing telnet, we see in Wireshark connection setup, data exchange phase,

and also connection tear down phase. It is illustrated in figure 1.5. As it is shown in figure 1.3the port number on the server (destination) is 7, which is port number of echo service, but in the source side the port number is 38168. In this Linux implementation, it takes any free ports which are not already in used, with a number higher than 1024. We cannot use for two connections the same source port because it may create a conflict. If I have 2 connections between my client and the same server with the same service and the same destination port the system cannot correctly multiplex and de-multiplex messages sent by first or second application. So we need to have 2 different ports otherwise system cannot distinguish between them. Therefore, we need to have different source ports (client ports) that system can correctly multiplex/de-multiplex messages sent by different TCP connection. Connection will be identified by five fields, IP address of client, IP address of server, port number of the client (source), destination port on server and protocol (ex. TCP). As we know TCP connection is bidirectional, and a reliable connection, so it uses ARQ mechanism. To implement any ARQ mechanism we need ACK, and sequence numbers. It is depicted in Figure-3, that actual sequence number does not start numbering from zero, it has been chosen randomly among 2 32 number, because of security reasons. Imagine an attacker, it could be easy to get server port number (among 2 16 port numbers), IP address of client and server, but by using random sequence number (guessing among 2 32 numbers) can make attacking for him/her harder. The reason why it selects a random number as initial sequence number is to avoid others inject and corrupt the payload exchange by source and destination. Client and server must perform a synchronization of sequence number. Because server will discard any frame that does not have the exact sequence number it expects (either a duplicate frame it already acknowledged, or an out-of-sequence frame it expects to receive later.)

The client on either side of a TCP session maintains a 32-bit sequence number it uses to keep track of how much data has been sent. This sequence number is included on each transmitted packet, and acknowledged by the opposite host as an acknowledgement number to inform the sender host that the transmitted data was received successfully and it waits for next bytes. Sequence number evolves by adding payload length to previous value (itis Byte wise). ACK numbers are sequence number that has gotten plus payload length plus one, hence ACK will give us the next Byte needed. In data exchange phase the receiver keeps track of the sequence number of the next frame it expects to receive, and sends that number with every ACK that it sends. For example, the initial relative sequence number shown in packet number 1 in figure 1.3 is 0 (naturally), while the actual sequence number is 1262865932 Decimal. At the beginning notice that the acknowledgement number has been increased by 1 although no payload data has yet been sent by the client. This is because the presence of the SYN or FIN flag in a received packet

triggers an increase of 1 in the sequence (This does not interfere with the accounting of payload data, because packets with the SYN or FIN flag set do not carry a payload). For the next packets as it is discussed sequence numbers will increase by adding payload length, and the ACK numbers of a host are in correspondence to the sequence numbers of the other host in a TCP connection .

TCP is an ARQ protocol, Go-Back-N. It allows to have a better performance by decreasing the total number of messages exchanged. By looking at Wireshark, we found out there is no an ACK for every packet, sometimes it is written in data packet which it is called piggybacking. When window in transmitter is greater than one at the receiver we can use delayed ACK and doing piggybacking. Each ACK consumes 66 bytes, so we saved this amount by doing piggybacking. In fact, it will delay ACKs deliberately to save bandwidth. To implement it we need an agreement between receiver and transmitter to use the same option. We also telnet for host with IP address 172.16.0.1 and trying to connect to port number 1234. what we expect is that application telnet starts and asks support from TCP to open a connection, TCP will send a SYN message, try to open a connection and sends it to other end, the receiver receives a SYN message to port 1234, checks if there is an application listening and able to process data on port 1234, there is no such an application then fails. TCP has a problem, there is no such application that data can be delivered, in TCP protocol format it sends a RST-ACK. therefore, three-way-handshake is not successful, and then replies the below message that shows handshaking was not successful and connection was refused. Totally, server refuses an incoming connection or TCP reacts to undesired connections by sending a RST-ACK.

```
laboratorio@NMLmaster:~$ telnet 172.16.0.1 1234
Trying 172.16.0.1...
telnet: Unable to connect to remote host: Connection refused
laboratorio@NMLmaster:~$
```

Figure 1.4: Telnet port 1234

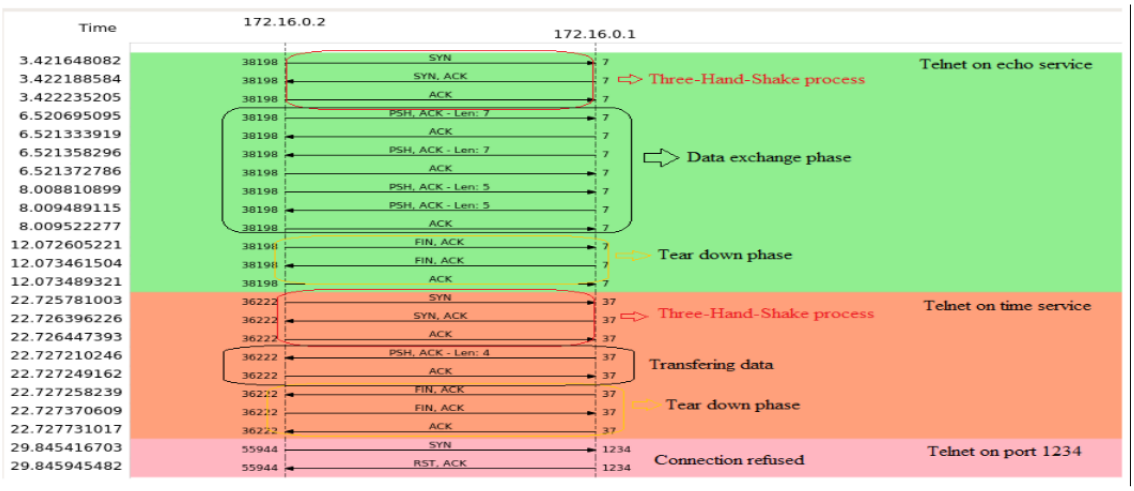

Figure 1.5: Wireshrk analyze

In this case, the number of TCP connection requests that we see on conversation window of Wireshark depends on how many telnets we are running, while Wireshark is capturing in background. It is given in figure 1.6 that we have 3 TCP connections and one UDP connection.

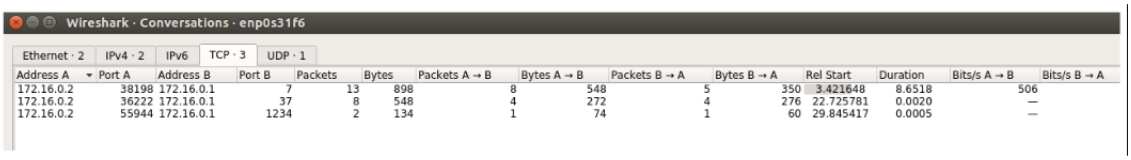

Figure 1.6: TCP analyze

First line belongs to connection to echo service with source IP address 172.16.0.2, port number in client side is 38198 and the number of packets that has been sent is 13 and amount of them is 898 bytes. Also we can see destination(server) IP number is 172.16.0.1, and also port number of echo service is 7, the number of packets that server has sent to the client is 5 which is 350 bytes. The second line is related to telnet time service which is run on port 37. Also in third line, it is shown the number of total packet for telnet connection on port number 1234 is 2 which one is for SYN in client side and another one is for RST-ACK in receiver side. As it is given to exchange application messages, the network carried more bytes than what we thought, which is because of protocol overhead (TCP overhead, IP overhead, and etc.) Finally, we disconnected the server which was Host3 with IP address 172.16.0.3 by unplugging the cable, in Host3 the routing table will not change because layers are independent and what happens in physical layer is nothing to do with other layers (routing table is working in layer 3), also we see that line

card is UP but not RUNNING any more. Because UP means card is working at Network layer, but Running means the card is working at physical layer. Then after unplugging the cable, we tried to echo it (Host3) but we got this answer that has been shown in figure 1.7, because we are not connected with Host3 anymore, and there is no way to reach it.

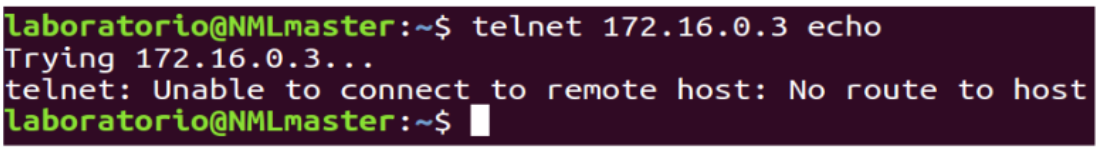

Figure 1.7: Telnet echo

In other case, we disconnected server after connection establishment, the client tries to send data but, since it cannot receive the acknowledgment, it repeats the transmission till timeout expires. This happens because, for some time after unplugging the cable neither the routing table nor the ARP table are modified, thus for all the layers above the physical one everything seems to work and re-transmissions occur.

Chapter 2

Nmap services

2.1 Analysis of the NMAP command

Network mapper (NMAP) is a tool used for port scanning. Port scanning is used to check which ports are open. The goal of this lab is to run a scan by NMAP utility as a normal user and also privileged user which by scanning of ports would be able to distinguish which services are running in first 100 ports using either TCP or UDP protocols. In our experiment, we configured our LAN by connecting two hosts (H1 and H2) via Ethernet through a switch. IP address of host1 is 172.16.0.1/26 and IP address of host2 is 172.16.0.2/26.

2.2 TCP scanning:

TCP protocol requires an initial three-way-handshake, which can be exploited to evaluate the status of a port. Essentially, nmap probes the target host with a SYN message on the desired port, and it is able to determine whether a service is listening on that port or not, by just looking at the answer. If the reply is a SYN-ACK, the target host has accepted the connection and nmap can immediately conclude that the port is open, whereas in the case the reply is a RST-ACK, the port is closed. In both cases the nmap application receives a feedback from the remote host, hence one attempt per port is generally enough to take a decision without ambiguity. In this case our network is LAN which devices are connected with a cable around one meter so the probability of message loss is almost zero. Although it rarely happens in our simple scenario, a bunch of reasons make it possible in a real network. Nmap faces this issue by making recursively additional attempts, re-transmitting the SYN and hoping that the SYN-ACK comes back. Finally, if still no reply comes, nmap declares the port status as filtered, which may be the case if the target is behind a firewall.

2.2.1 TCP port scan (without SUDO):

First we run the wireshark for capturing the packets. then host H1 runs the command "nmap H2 -p 1-100" as a normal user and the port scan is performed through the transmission of 102 TCP SYN packets. since we are running as a normal user so first, it will check that whether the victim is alive or not to avoid wasting time with the whole port scanning by the port 80 and the port 443.

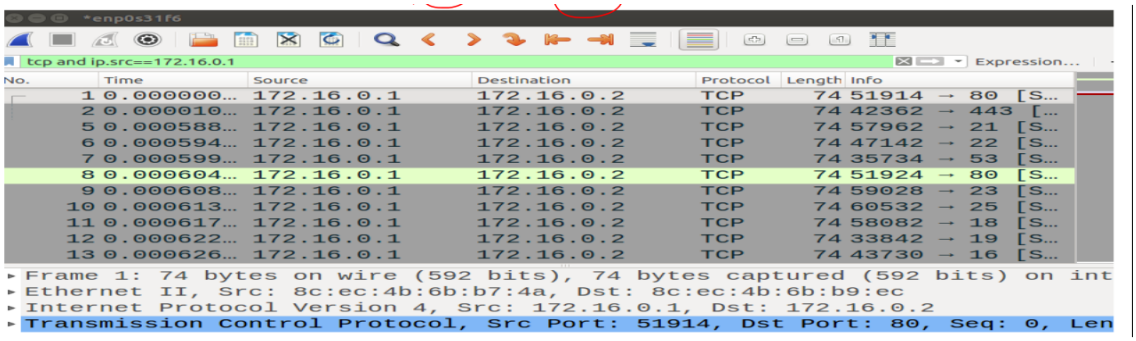

Figure 2.1: Wireshark analyze tcp protocol

Port 80 is the port number assigned to commonly used internet communication protocol, Hypertext Transfer Protocol (HTTP). TCP port 443 is the standard TCP port that is used for website which use SSL. HTTPS used this port. After checking these two ports NMAP start sending the packets and check the first 100 ports in the random order (Otherwise the network administrator will understand that we are doing the port scan). Destination ports are chosen randomly in the range [1,100]. The scanned host H2 immediately answers by sending RST-ACK packets if no service is running on the scanned port or SYN-ACK packets otherwise. As a result, we receive 100 responses in terminal. As it is depicted in figure 2.1 just one port is open which is "ssh" with port number 22, and all other 99 ports are closed, and we receive SYN-ACK for port number 22, and for other 99 ports we get RST-ACK since the ports are closed.

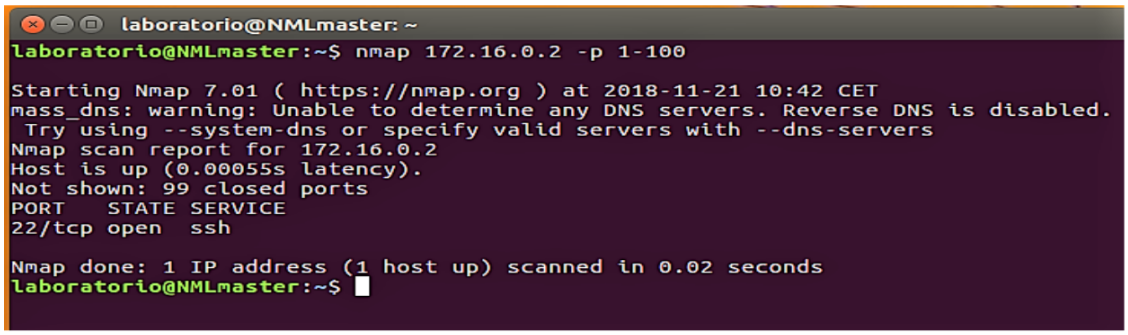

Figure 2.2: NMAP analyze

For the ports which are closed two packets have been exchanged (SYN: from transmitter to receiver, RST-ACK: response of SYS that has been sent by receiver which shows port is closed) and for the ports which are open, there are four packets (SYN, SYN-ACK, ACK and finally RST-ACK). Then we plotted the TCP port scanning versus time. NMAP sends 102 requests and got responses and checked all the ports just once.

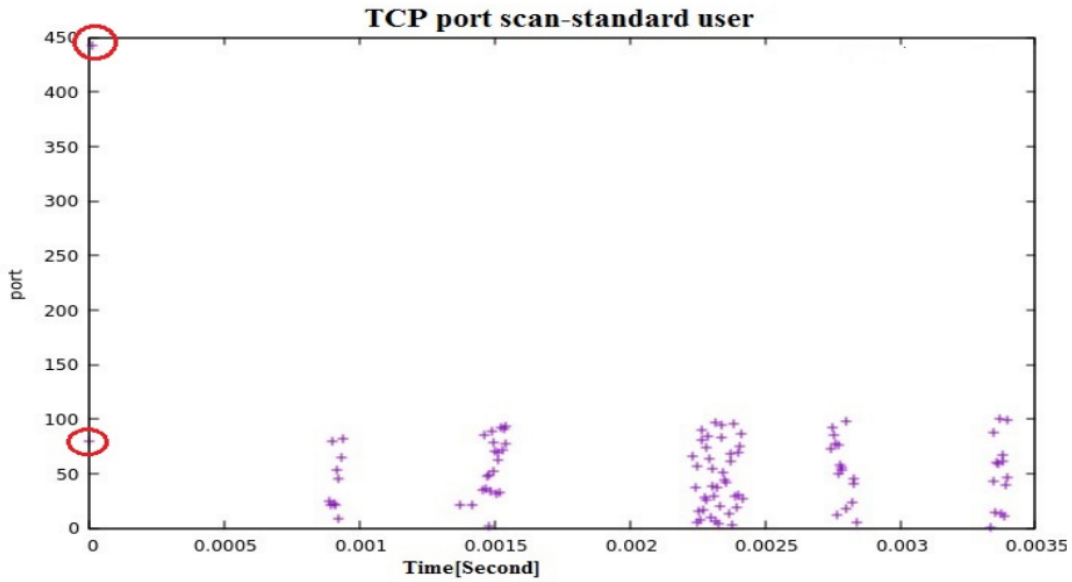

Figure 2.3: TCP port scan standard user

2.2.2 TCP port scan (with SUDO):

When we run NMAP as a root (privilege user), it abuses the ARP protocol to find out if the victim is alive. ARP behaves better than TCP. As an example when firewall blocks ports 80, and 443, TCP will send a request (SYN) and we will get no answer and we believe that victim is not alive, so by using TCP there is less chance that we get answer. Whereas, we don't have a scenario that a host does not answer to ARP request because otherwise host cannot exchange data. Responding to ARP requests is a prerequisite to IP communication on such a network, so it's nearly impossible to block or hide from this type of scan, otherwise it cannot exchange message with anyone because IP will not be able to find the MAC address of destination. It is like having a disconnected host, so the host must always respond to ARP request. It is much reliable to use ARP request rather than TCP request. Normal users are in application layer and can just have connection with layer 4, and are not able to talk to ARP to send a request. When nmap run as root has right to do whatever, nmap forges an ARP request, then asks Ethernet to send a SYN message by forging TCP (when nmap is being run as privileged user it can abuses protocol formats to extract information from remote host), then receiver receives this SYN at TCP, then replies with SYN-ACK, at the end source will get this SYN-ACK up to TCP layer, but at Ethernet nmap will get this SYN-ACK and will find out that the port is open but when TCP gets this message will reply with an error message RST, because TCP had not sent a SYN. It is given in figure 2.4 the needed time for scanning ports as a root is more than required time when user is normal.

As it is illustrated in figure 2.5 , at the beginning it is not needed to check ports 80 and 443 anymore, first well-known ports are checked (e.g. HTTP, SSH, FTP, SMTP, Telnet etc.) and then after some time (around 1 s) all the others randomly. We can notice that the TCP scan duration is very short by comparison with UDP scanning because H1 immediately receives an answer (positive or negative) from H2 for every port checked thanks to TCP being connection oriented. This implies that in this case nmap checks only once a given port.

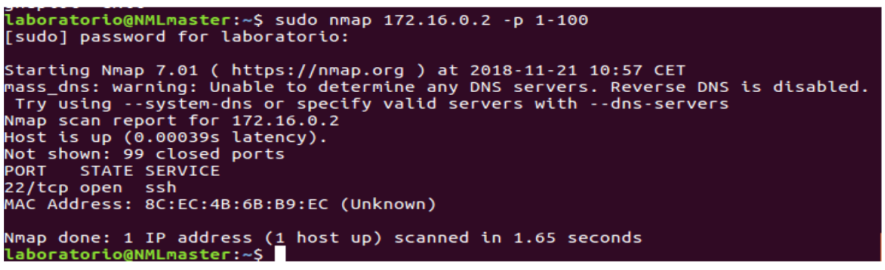

Figure 2.4: Analyze the NMAP with command sudo

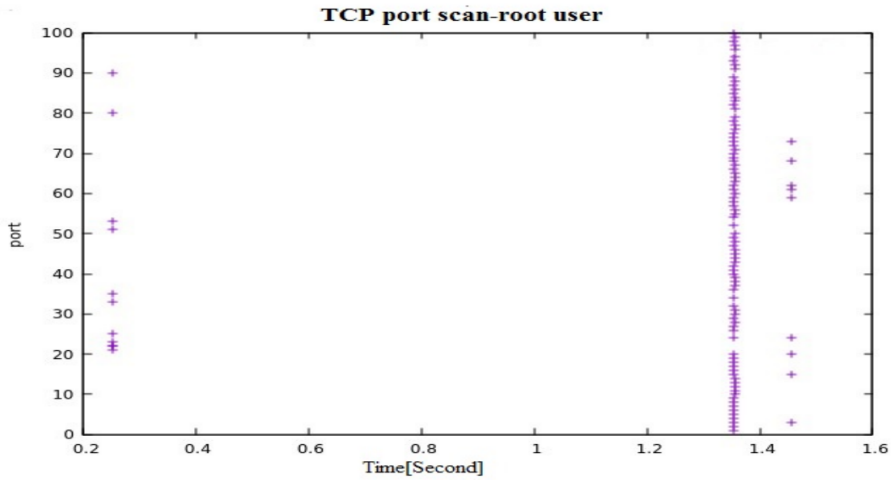

Figure 2.5: TCP port scan root user

2.3 UDP scanning:

UDP is a connection less protocol .NMAP UDP scans are going to take some time to scan because For scanning by UDP message, NMAP would use ICMP messages to discover port status, and if a UDP packet is sent to a port that is not open, the system will respond with an ICMP port unreachable message, but Neither UDP packets, nor the ICMP errors are guaranteed to arrive, so UDP scanners of this sort must also implement re-transmission of packets that appear to be lost and filtered. NMAP retry transmissions just in case the probe or scan was lost. NMAP would detect the limitation rate and it would be slowed down the scanning due to avoid flooding the network with useless packets which target machine will drop. So the scanning host H1 must wait for the ICMP error messages related to each one of the 100 scanned ports and the rate at which those messages are generated is about 1 per second. All things considered, the UDP scanning keeps doing multiple attempts, until something which proves the status of the port is received, either an ICMP Destination unreachable telling the port is closed, or an application message demonstrating that the port is open. As a consequence, the UDP scanning is in general slower and less reliable, because we don't know a prior if and when a response will come back.

2.3.1 UDP PORT SCAN (without SUDO):

UDP port scan cannot be done without the privileged mode because UDP is the best effort protocol and it will not inform the sender if any packets get dropped so

Error message can be shown using ICMP protocol and ICMP requires root access.

```
laboratorio@NMLmaster:~$ nmap -sU 172.16.0.3 -p 1-100
You requested a scan type which requires root privileges.
QUITTING!
```

Figure 2.6: NMAP without sudo

2.3.2 UDP PORT SCAN (with SUDO)

UDP takes larger time as compared to TCP, UDP is connection-less and scanning UDP ports in this case has taken 115 seconds. Almost 1 second for each port. UDP is checking ports randomly, and each port is checked many times. It will check the port until it will get an ICMP error-message and if it will not receive any ICMP error-message it will again check that port. usually this happens because the generation rate of ICMP error messages is much slower than the generation rate of UDP packets and therefore nmap may keep sending UDP packets to a given port until it receives the related ICMP error message. Note that this behavior may be due to not only the "laziness" of ICMP protocol, but also because UDP is unreliable, therefore some UDP packets with a given destination port sent by the scanning host may be lost and nmap may need to send new UDP packets with that destination port. The number of checking each port depends on the operating system at the receiver side (victim). For example, the Linux 2.4.20 kernel limits destination unreachable messages to one per second.

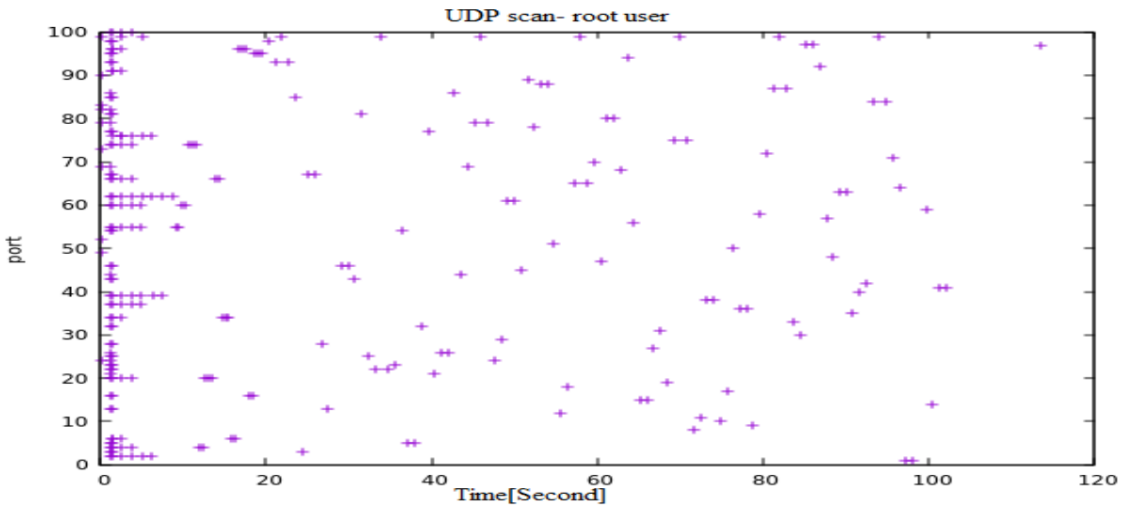

Figure 2.7: UDP scan root user

2.4 Why the TCP scan is much faster than the UDP scan?

TCP is a connection oriented protocol. This means that systems must establish and confirm a stable connection for TCP, the process is commonly known as the "three-way handshake" - before actually sending the data. In case of TCP, it is utilizing half open scanning technique which it sends SYN packet and it would be waiting for response. If it receives SYN-ACK it means the port is open but if it retrieves RST-ACK, it means the port is closed. As the process must happen quickly for a good connection, it can be reasonably expected that an open TCP port will respond in short order when it is probed. This means that the timeout before assuming a port is closed may also be very short. It could also be seen that the TCP scanning duration is too shorter rather than UDP port scanning duration, because of the sender machine would be received port status from target machine rapidly. And scanning each port has been done just once. On the other hand, UDP is a connection less protocol. NMAP UDP scans are going to take some time to scan because For scanning by UDP message, NMAP would use ICMP messages to discover port status, and if a UDP packet is sent to a port that is not open, the system will respond with an ICMP port unreachable message, but Neither UDP packets, nor the ICMP errors are guaranteed to arrive, so UDP scanners of this sort must also implement re-transmission of packets that appear to be lost or filtered. ports rarely send any kind of response, this leaves NMAP to time-out and then retry transmissions just in case the probe or scan was lost. NMAP would detect the limitation rate and it would be slowed down the scanning due to avoid flooding the network with useless packets which target machine will drop. So the scanning host H1 must wait for the ICMP error messages related to each one of the 100 scanned ports and the rate at which those messages are generated is about 1 per second. Every ports may check several times.

Chapter 3

Analyze the Ethernet links

3.1 Performance test on Ethernet Links

The goal of this laboratory is to analyze good-put on a test-bed illustrated below through Wireshark and the speed-test application "nttcp". we consider three hosts: H1, H2 and H3 connected to a switch with below configuration. Then we want to consider two different scenarios, and for each scenario consider 4 possible combination of connections by transport protocols to compute performance at application layer at receiver side. To reach this goal we use "nttcp" utility to measure goodput which is defined as useful data that successfully has received at application layer.

Good-put = Useful data at application layer / time to complete transfer We compute maximum efficiency of channel in different cases

$$\eta TCPfullduplex = \frac{MSS}{MSS + 20_{\mathrm{TCP}} + 20_{\mathrm{IP}} + 38_{\mathrm{ETH}}} = 0.949 \tag{3.1}$$

$$\eta TCPhalfduplex = \frac{MSS}{(MSS + 20_{\mathrm{TCP}} + 20_{\mathrm{IP}} + 38_{\mathrm{ETH}}) + (20_{\mathrm{TCP}} + 20_{\mathrm{IP}} + 38_{\mathrm{ETH}})} = 0.91 \tag{3.2}$$

$$\eta TCPfullduplex = \frac{MSS}{MSS + 8_{\mathrm{UDP}} + 20_{\mathrm{IP}} + 38_{\mathrm{ETH}}} = 0.957 \tag{3.3}$$

Good-put = Capacity of channel * η

In all cases, we considered packet length for TCP connection equal to 1460 Byte, when we have UDP connection we consider the packet length equal to 1472 Byte, this is done to avoid fragmentation, also we chose number of packet that we

transmit equal to 5000. We will firstly predict the good-put by discussing the impact of protocols overhead, probability of collisions, packet dropping probability, and etc. Then, we will eventually validate the expectations through the experimental results. First of all, we assigned IP address to our hosts, H1: 172.16.0.1/26, H2: 172.16.0.2/26, H3: 172.16.0.3/26 We disabled offloading capabilities such as GSO, GRO, TSO, TX, and RX. flow control at layer 2 is off as well. we do not expect to have transmission errors since we used just few meters Ethernet links.

3.1.1 A1: H1 and H2 sending to H3, both through TCP connection

In this case since from H1 to switch link is full duplex and 10Mbps and also from switch to H3 we have 100Mbps and full duplex, we don't expect any losses or collisions for this connection, so we expect maximum good put around 9.5 Mbps. While for another connection which is from H2 to switch with 10 half duplex and from switch to H3 is 100 Mbps full duplex we don't expect losses in application layer because TCP is reliable but we expect some collisions via half duplex link, because on the same link there are both data and ACKs. It is needed to do re-transmission for those packets that are lost due to collision so transmission in connection H2->H3 will take more time. As a consequence, we expect good-put reduction for this connection, less than 9.1 Mbps. There is also no congestion on the switch since the total input rate is around 20 Mbit/s while the output link capacity is 100 Mbit/s. Results of experiment: we have good put reduction in second connection and as the I/O graph shows in figure 3.1 we have randomness in second connection, we observed 20% collision in (H2->H3) connection and good-put equal to 8.25 Mbps which is less than 9Mbps and it is in line with our anticipation.

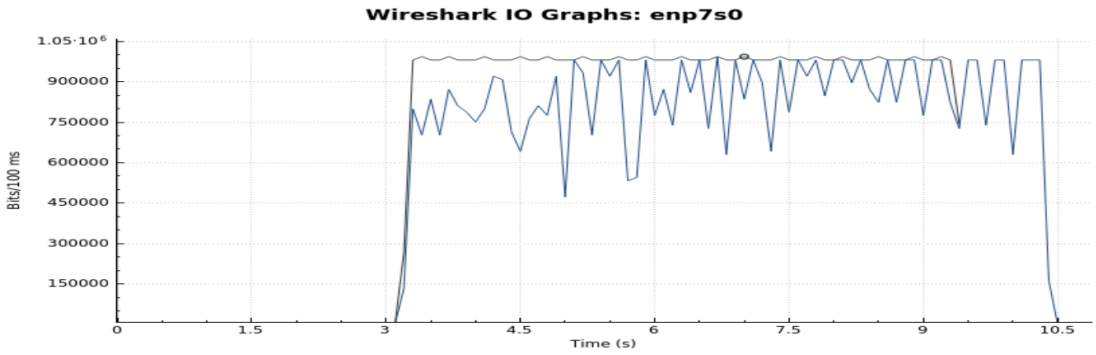

Figure 3.1: Wireshark IO graph

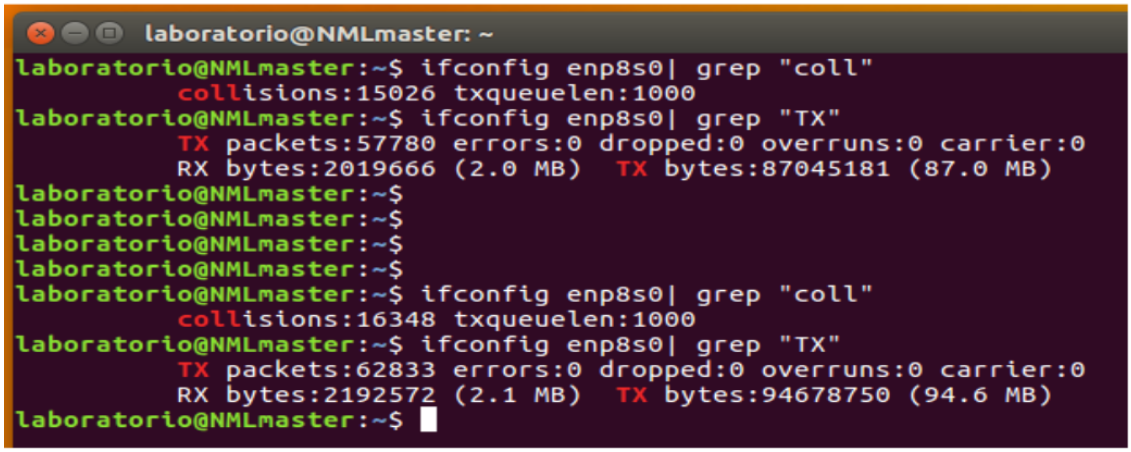

Figure 3.2: Ifconfig

3.1.2 A2: H1 and H2 sending to H3, both through UDP connection

In UDP connection there is no ACK, so we expect no collision in both connections. As the rate of 2 transmitters is equal to 20Mbps and much less than receiver rate which is 100Mbps, we also do not expect to see any congestion in switch and packet losses in application layer. In both connections we expect maximum goodput around 9.6Mbps.

UDP connection	Goodput		Loss Probability (application layer)		Collision Probability (Physical layer)	
	Expected	observed	Expected	observed	Expected	observed
H1→H3	9.6 Mbps	9.57 Mbps	0%	0%	0%	0%
H2→H3	9.6 Mbps	9.55 Mbps	0%	0%	0%	0%

Figure 3.3: Table 1

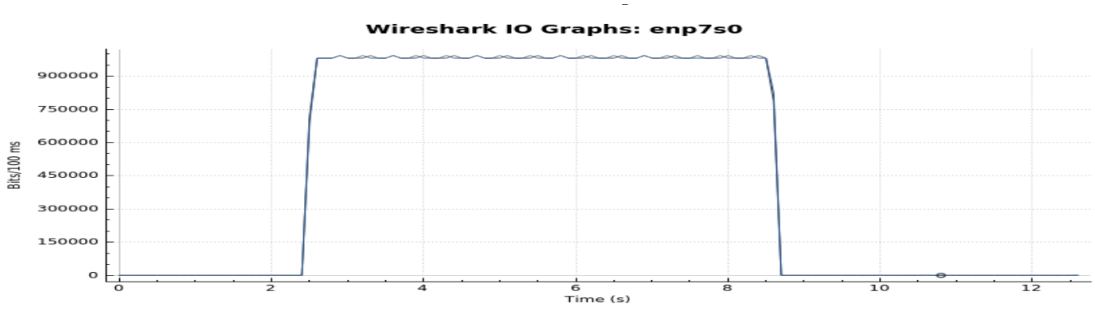

Figure 3.4: Wireshark IO graph

3.1.3 A3: H1 and H2 sending to H3, the first using UDP, the second TCP

H1→H3 through UDP connection, there is no ACK sending to sender, and of course whole this connection is full duplex, so we don't expect to see any collision and the link speed of transmitters is less than receiver. Therefore, we predict not to see any packet dropping due to congestion in switch. The expected goodput is 9.6 Mbps. While in another connection H2→H3, it is based on TCP connection and the link from H2 to switch is half duplex so we expect to see collision in this link because it needs to transfer data and also receives ACK from another side, the connection is TCP and reliable so when we have collision we need to re-transmit those packets, so it takes much time to be finished, as a result we expect goodput less than 9 Mbps or to be more precise around 8, like case A.1 and also we predict to see randomness while H2 is transmitting to H3 due to collision around 25% like case A.1.

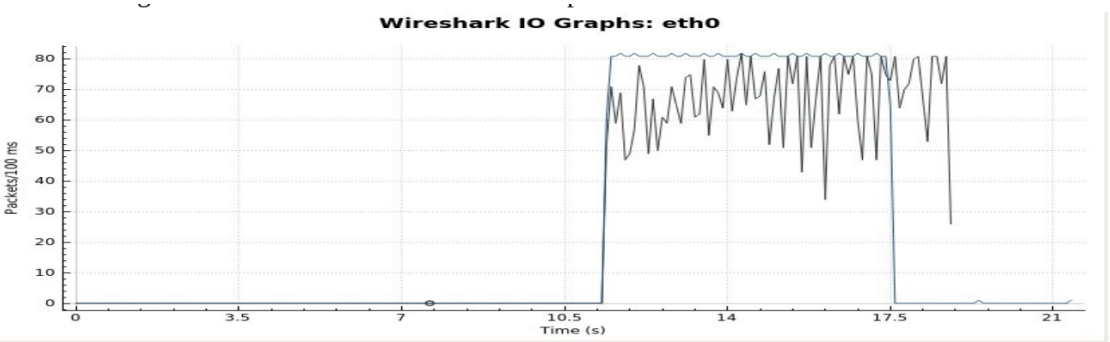

Figure 3.5: Wireshark IO graph

3.1.4 A4: H1 and H2 sending to H3, the first using TCP, the second UDP

H1→H3 through TCP connection, TCP is reliable and it is needed to send ACK, whole this connection link is full duplex, so we expect not to see any collision and the speed of transmitters(H1+H2) is less than receiver. Therefore, we predict not to see any packet dropping due to congestion in switch. The expected goodput is 9.5 Mbps. In another connection H2→H3, it is based on UDP connection and the link from H2 to switch is half duplex, but we will not face with any problem because there is no ACK in case of UDP, so we expect to see no collision in this link, and no congestion, we predict that 2 transmissions finish more or less at the same time. The expected goodput is 9.6Mbps.(results would be like case A.2)

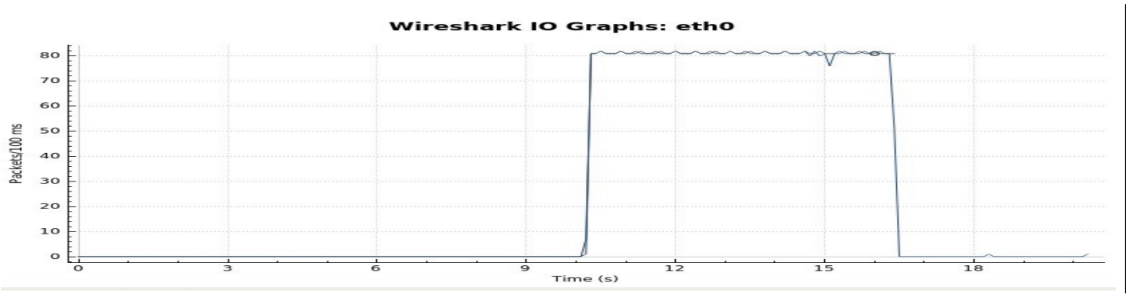

Figure 3.6: Wireshark IO graph

3.1.5 B1: H3 sending to H1 and H2, both through TCP connection

The link speed of sender H3 is much more than receivers (100 Mbps » 20 Mbps) so buffer of switch will be filled very soon and we have dropping packets, as it is TCP and there is congestion control, network will tell transmitter to decrease its speed to avoid dropping packet, so the speed of transmitter must be matched with the speed of receiver. In the first case H3→H1, H3 continues transmitting at rate 10Mbps and H1 receives all packet and sends ACK without having any problem because it is full duplex, so we expect no collision anymore and expected goodput is around 9.5 Mbps. While in other case H3→H2, the link from switch to H2 is half duplex and we have collision because of receiving data and sending ACK simultaneously on the same half duplex link, so H3 will decrease its speed more. As a result, this connection will take more time and in fact we deal with decreasing in goodput, the expected good put is less than 9Mbps. we expect to see a little packet loss due to congestion in switch only at the very beginning. Losses experienced before the initial phase of congestion control algorithm. But as it is TCP connection and reliable, re-transmission will be done and we expect no losses

in application layer. We expected to get almost the same values such that we got for part A.1.

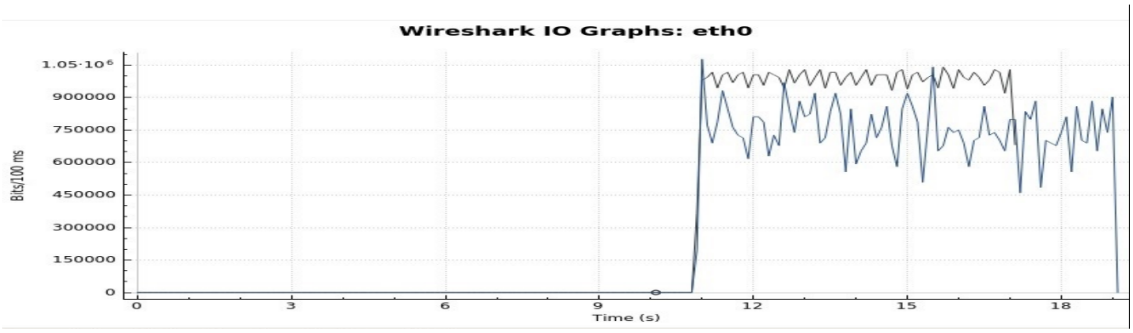

Figure 3.7: Wireshark IO graph

3.1.6 B2: H3 sending to H1 and H2, both through UDP connection

In UDP connection we do not have congestion control. So H3 starts transmitting at rate 100Mbps, as there are 2 flows its speed will be divided by two, fairly, for each connection. So at each connection the speed of sender is 50Mbps, while the speed of both receivers is 10Mbps. Evolution of two connections is really fast, both of them finish receiving data around 2 seconds. We expect lots of packet dropping almost 80% of packets due to the congestion in switch. As we know goodput is useful data at application layer over time to complete transfer, which in this case time to complete transmission is really fast we expect high goodput close to maximum. We do not expect collision because both connections are UDP and on each link we just transmit data and no ACK.

UDP connection	Goodput		Loss Probability (application layer)		Collision Probability (Physical layer)	
	Expected	observed	Expected	observed	Expected	observed
H3→H1	9.6 Mbps	9.55 Mbps	80%	80%	0%	0%
H3→H2	9.6 Mbps	9.56 Mbps	80%	80%	0%	0%

Figure 3.8: Table 2

3.1.7 B3: H3 sending to H1 and H2, the first using UDP, the second TCP

As we know the UDP connection is more aggressive than TCP, so H3 starts to transmit at maximum speed around 100Mbps and the buffer of switch will be filled

very fast but there is no congestion algorithm to decrease the speed of sender.H3 will continue transmission at maximum rate until it finishes the transmission to H1, we expect lots of packet dropping almost 90 percent, and also maximum goodput because transmission is very fast. While in second connection at the beginning H2 receives almost nothing because UDP connection is using whole bandwidth, then after finishing first transmission, second connection will achieve whole bandwidth, but it is TCP connection and the speed of transmitter must be in line with the speed of receiver to avoid congestion in switch. link from switch to H2 is half duplex and we need to receive data and send ACK through this link, so we have collision and transmitter decreases its speed even more, hence duration of this connection is more, we will have decreasing in goodput. The expected goodput is much less than 9.1Mbps.

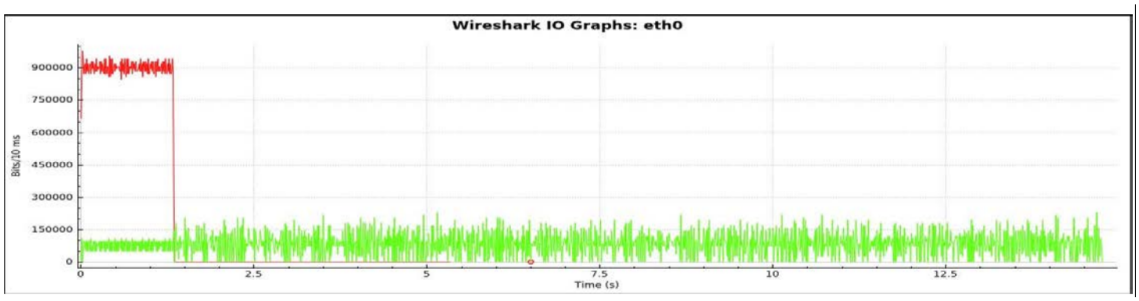

Figure 3.9: Wireshark IO graph

3.1.8 B4: H3 sending to H1 and H2, the first using TCP, the second UDP

Like previous case, UDP connection behaves greedy, so H3 starts to transmit at maximum speed around 100Mbps and the buffer of switch will be filled very fast but there is no congestion algorithm to decrease the speed of sender.H3 will continue transmission at maximum rate until it finishes the transmission to host2, we expect lots of packet dropping almost 90 percent . As this transmission is very fast we expect maximum goodput around 9.6 Mbps . While in second connection at the beginning H1 receives almost nothing because UDP connection is using whole bandwidth, then after finishing UDP transmission, TCP connection will achieve whole bandwidth, but it is TCP connection and the speed of transmitter must be in line with the speed of receiver to avoid congestion in switch. Transmission time will increase and we expect goodput lower than 9 Mbps. Link from switch to H1 is full duplex, we need to receive data and send ACK through this link, as it is full duplex we have no collision between ACKs and data. So by comparison with case B1 we expect to see less goodput from H3 to H1.

Mixed connection	Goodput		Loss Probability (application layer)		Collision Probability (Physical layer)	
	Expected	observed	Expected	observed	Expected	observed
H3→H1(TCP)	< 9 Mbps	8.4 Mbps	0%	0%	0%	0%
H3→H2(UDP)	9.6 Mbps	9.57 Mbps	90%	90%	0%	0%

Figure 3.10: Table 3

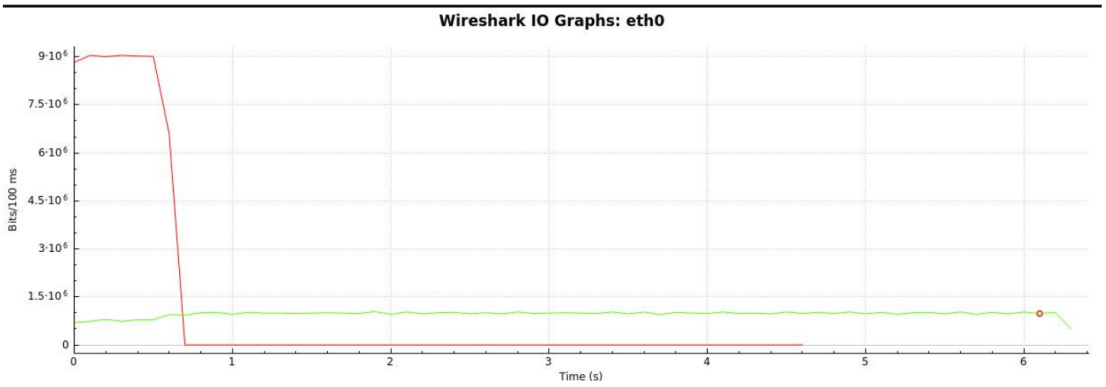

Figure 3.11: Wireshark IO graph

Above photo belongs to transmitter side(H3).

Chapter 4

Analyze the Ethernet an Wi-Fi links

4.1 Performance test over Wi-Fi

We configured the testbed, so that H1 and H2 are connected via Wi-Fi to the AP, and disconnect the cables from the Ethernet ports. H3 is connected using the wired Ethernet. We assigned IP address198.16.0.11 to host1, 192.68.0.12 to host2, and 192.168.0.3 to host3. Subnet mask was 255.255.255.0.

Figure 4.1: Host configuration

As usual all offloading capabilities (such as gso, gro, tso, tx, rx,..) are disabled, also flow control on layer two is OFF. Max speed for our Wi-Fi interface at physical layer is 54 Mbps, but because of huge amount of overhead of 802.11, overhead of IP and etc. the maximum speed cannot be further than half of it (around 27 Mbps), also we know that Wi-Fi is half duplex. Ethernet interface speed is 10 Mbps and it is Full duplex. We are not in an isolated channel, may be other users are using the same channel that we are using, and we are sharing the medium, the speed may decrease. So, our prediction may won't be precise. Thus, we repeated our

experiment several times to get an average goodput and also getting minimum and maximum goodput. We considered the following scenarios when the size of packet for TCP connection is 1460 Bytes and when the connection is UDP it is 1472 Bytes, also number of packet that we transmit is equal to 5000.

4.1.1 Case A.1: Single flow - E3 sends data to W1 through a TCP connection[Eth => Wi-Fi]

first of all, the speed of Ethernet is 10Mbps and the speed of Wi-Fi is 27Mbps, there is no problem by having bottleneck, so there is no congestion. All packets that E3 is sending will be received by W1, as it is TCP connection W1 must send ACK back. Wi-Fi is half duplex, and the method that is used to avoid collision is CSMA/CA. In this situation, data link layer in our network attempt to avoid collisions by transmitting only when the channel is sensed to be "idle", hence the time of transmission will increase. Consequently, it will lead to a goodput a bit less than maximum goodput approximately 9.5 Mbps. The average goodput that we got was 9 Mbps and minimum goodput was 8.86, this randomness could be because other students at the same time were transmitting through channel 1 which we were using, but totally our network was reliable, and the results were matched with our prediction. TCP is reliable, so we do not expect any losses in application layer.

Expected goodput	Average goodput	Max goodput	Min goodput
Less than 9.5 Mbps	9 Mbps	9.2 Mbps	8.86 Mbps

Figure 4.2: Table 3

4.1.2 Case A.2: Single flow - E3 sends data to W1 through a UDP connection [Eth => Wi-Fi]

In this scenario we are using UDP connection that does not use ACKs. Therefore, W1 does not need to sense the channel to be idle and sends ACKs back to E3. We predict to have maximum goodput at around 9.6 Mbps in ideal situation, also we don't expect to have any losses due to congestion in switch as the speed of transmitter is lower than the speed of receiver. Only when many users are sharing the channel, the speed of Wi-Fi link will decrease and may be become a bottleneck, therefore we may have losses because of congestion, but we did not experiment this. Differently from the scenario with TCP, if a packet is lost because of interference, that packet is lost forever. UDP does not try to recover from losses and this surely has its impact on the final goodput. Moreover, UDP does not perform congestion control, so if the channel is very noisy, the sender won't slow down and will keep

losing more and more packets, which will affect the goodput.

Expected goodput	Average goodput	Maximum goodput	Minimum goodput	Average losses
9.6 Mbps	9.5 Mbps	9.53 Mbps	6.97 Mbps	0 %

Figure 4.3: Table 4

4.1.3 Case B.1: Single flow - W1 sends data to E3 through a TCP connection [Wi-Fi => Eth]

The speed of wireless link (27 Mbps) is greater than the speed of Ethernet link (10 Mbps). Therefore, bottleneck is Ethernet channel. So the buffer of access point will be full very soon, we have congestion, and consequently, we expect to have packet dropping. It is evident, the congestion control algorithms that has been done by network will react to this packet loss and transmission rate in source (W1) will decrease immediately, and W1 starts to send data at rate 10 Mbps. Hence, there is no packet dropping anymore. We expect goodput around 9.5 Mbps. The average goodput is 9.4Mbps which satisfied our prediction. also it is sending data through TCP connection, so if any packet drop, it will be re-transmitted. As a result, the number of packet that W1 has sent is equal to number of packet that E3 has received. There are no losses in application layer. (The results are like case A1)

	Bytes	Real s	CPU s	Real-MBit/s	CPU-MBit/s	Calls	Real-C/s	CPU-C/s
l	7300000	5.79	0.01	10.0866	4594.8072	5000	863.58	393391.0
1	7300000	6.20	0.08	9.4133	720.0099	5043	812.87	62174.8

Figure 4.4: Analyze packets

4.1.4 Case B.2: Single flow - W1 sends data to E3 through a UDP connection [Wi-Fi => Eth]

Like previous case Ethernet link is bottleneck. But in UDP there is no need to send ACKS, no congestion control algorithm to reduce the transmission rate. Consequently, we should experience a lot of losses. The goodput is the useful data at application layer over transmission time, because transmission time is very fast we anticipate to have maximum goodput around 9.6 Mbps. what we obtained was on average 9.5 which was in line with what we expected.

4.1.5 Case C: Single flow - W1 sends data to W2

In both scenario two mediums are Wi-Fi and they are sharing the same channel. Each transmission of one packet goes twice on the same physical medium. Because for each packet that has been sent it goes twice from W1 to AP and from AP to W2. Also we have to take it into account that in the case of TCP connection we need to send ACKs. If we had just one Wi-Fi flow through TCP connection, goodput would be around 22Mbps and for the case of UDP connection it would be 25Mbps. But here because of sharing Wi-Fi link by different flows we expect to have poor goodput. In the case of UDP because there is no contention to achieve Wi-Fi link for sending back acknowledgments we expect higher goodput. We won't have losses in any case, as both hosts are pushing packets to the AP at a largely sustainable rate. As Wi-Fi is using CSMA/CA to avoid collision, we expect no collision. Also the speed of transmitter and receiver is the same so we don't expect to have bottle neck and seeing any congestion in layer 2.

connection	Expected goodput	Average goodput	Max goodput
W1 -> W2 (TCP)	5-6 Mbps	4.3 Mbps	4.96 Mbps
W1 -> W2 (UDP)	6-7 Mbps	6.5 Mbps	7.15 Mbps

Figure 4.5: Table 5

4.1.6 Case D.1: Bidirectional flow-E3 and W1 exchange data-both through a TCP connection[Wi-Fi<=> Eth]

In this case we have two TCP connections coexisting on the Wi-Fi link, so this medium will be shared among them. In ideal case, we expect both transmissions finish more or less at the same time and expected goodput in both cases is around 9.5 Mbps. Both transmitter and receiver at each side have the same speed so we do not expect to see congestion and packet lose.

	Bytes	Real s	CPU s	Real-MBit/s	CPU-MBit/s	Calls	Real-C/s	CPU-C/s
l	7300000	5.74	0.01	10.1753	4976.5658	5000	871.17	426075.8
1	7300000	6.21	0.08	9.4075	722.8796	5043	812.37	62422.6
l	7300000	6.31	0.14	9.2591	405.7021	5043	799.55	35033.5
1	7300000	6.09	0.01	9.5917	6610.8218	5000	821.21	565995.0
	Bytes	Real s	CPU s	Real-MBit/s	CPU-MBit/s	Calls	Real-C/s	CPU-C/s
l	7300000	6.19	0.01	9.4345	4571.4286	5000	807.75	391389.4
1	7300000	6.60	0.08	8.8540	690.4869	5044	764.72	59637.3
l	7300000	6.35	0.15	9.1902	395.8034	5044	793.76	34185.5
1	7300000	5.98	0.01	9.7689	6368.5932	5000	836.38	545256.3

Figure 4.6: Analyze packets

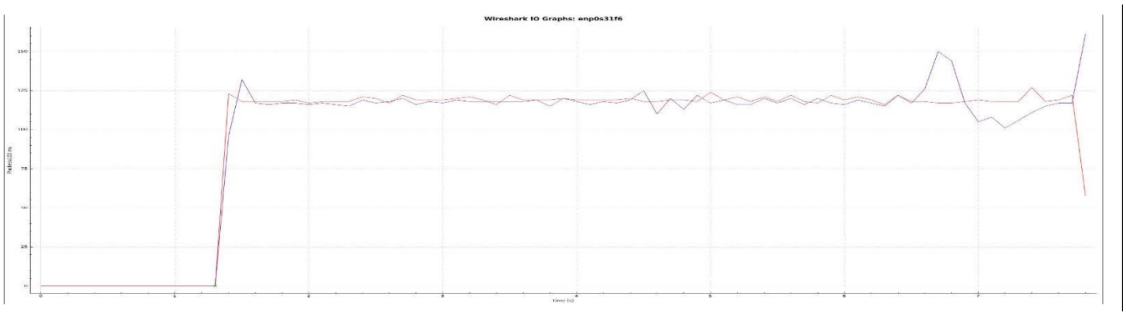

Figure 4.7: Wireshark IO graph

Connection	Expected goodput	Average goodput	Min goodput	Max goodput
E3 → W1 (TCP)	Around 9.5 Mbps	9.25 Mbps	9.19 Mbps	9.30 Mbps
W1 → E3 (TCP)	Around 9.5 Mbps	9.20 Mbps	8.7 Mbps	9.47 Mbps

Figure 4.8: Table 6

4.1.7 Case D.2: Bidirectional flow-E3 and W1 exchange data-both through a UDP connection[Wi-Fi<=> Eth]

Wi-Fi source starts to send packets, and it could fill up the switch buffer too fast. But in this case, we do not have congestion control algorithm to slow down the transmission, therefore the Wi-Fi source transmits at maximum rate, and we expect to deal with lots of dropping. As we told in case B, goodput is the amount of useful data over transmission time, as transmission time is fast we do not face with decreasing goodput due to dropping packets. So we expect goodput around 9.6 Mbps from W1 to E3. Transmitting from W1 to E3 will be finished sooner, then another transmission can use whole bandwidth and will compensate less speed at the beginning. We expect goodput for connection E3 to W1 almost 9.6 Mbps and also we anticipate not to see dropping because there is no bottleneck in this case, and speed of transmitter is less than the speed of receiver. As it is shown the W1 first finishes transmission, thanks to higher speed of Wi-Fi here. And it is what we expected. Also we got the average goodput for both connections almost 9.5 Mbps which is in line with our expectation.

Connection	Expected goodput(at RX)	Observed goodput(at TX)	Observed goodput(at RX)
W1 → E3 (UDP)	Almost 9.6 Mbps	23 Mbps	9.56 Mbps
E3 → W2 (UDP)	A bit less than 9.6 Mbps	9.65 Mbps	9.53 Mbps

Figure 4.9: Table 7

4.1.8 Case D.3: Bidirectional flow-E3 to W1 use TCP and W1 to E3 use UDP

W1 is sending to E3 through UDP at rate almost 27 Mbps. Sender has higher speed than receiver so we have congestion in switch, but there is no congestion algorithm so we expect many losses around 50%. UDP is more aggressive than TCP, it uses all the bandwidth, another connection through TCP decreases its speed, so UDP connection finishes transmission first. we expect average goodput around maximum which is 9.6 Mbps for UDP connection. In another case there is no problem of congestion, we expect no packet dropping and as a result no re-transmitting. Just it has to wait for another connection to be finished, so in this period transmission rate is almost zero, then it starts to transmit at maximum speed. We expect average goodput much less than 9.(half of the time transmission rate is almost zero, and then it is 9.5 Mbps)

Connection	Expected goodput	Observed goodput	Average losses
W1 → E3 (UDP)	Almost 9.6 Mbps	9.48 Mbps	50%
E3 → W1 (TCP)	<< 9	6.5 Mbps	0%

Figure 4.10: Table 8

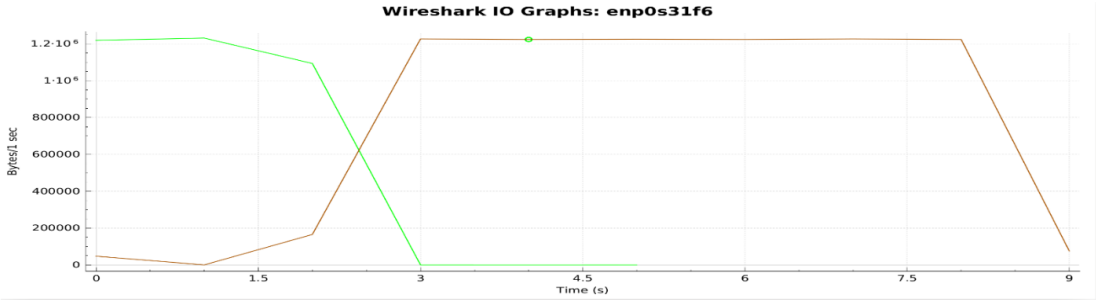

Figure 4.11: Wireshark IO graph

4.1.9 Case D.4: Bidirectional flow-E3 to W1 use UDP and W1 to E3 use TCP [Wi-Fi=>Eth and Eth=>Wi-Fi]

In the case that W1 is transmitter we are using TCP congestion the same as what we told in case D1, we have congestion at layer two, we have congestion control, it will reduce its speed with what is coordinated to other link. Again the Good-put was limited by the speed of Ethernet. So our expectation on average is around 9.5Mbps. maybe we see some packet drops just at the beginning but as it is TCP, and reliable. We have re-transmission. Also in another case that E3 is sender as

its speed is less than receiver there is no congestion problem. We expect no packet loss and goodput around 9.6 Mbps.

Connection	Expected goodput(at RX)	Observed goodput(at TX)	Observed goodput(at RX)
W1 → E3 (TCP)	Almost 9.5 Mbps	9.8 Mbps	9.48 Mbps
E3 → W1 (UDP)	Almost 9.6 Mbps	9.4 Mbps	9.4 Mbps

Figure 4.12: Table 9

4.1.10 Case E: E3 sends to W1, and W2 sends data to E3

In this scenario we consider a bidirectional data flow Ethernet and Wi-Fi. It is composed by three hosts. Two of them are connected through Wi-Fi to the AP, so they share the same link, while the third host is connected to the switch with a wire, in which E3 sends data to W1, and W2 sends data to E3 at the same time. In this case we expect a very similar behavior to what we observed in case D. The only difference is that this time the Wi-Fi channel is accessed also by W2 instead of only W1, but because the number of flows that are sharing the link are the same it will not affect the final result. We do experiment with new transmitter to verify that the result is more or less the same with the case D. We may expect some higher delay due to the fact that now we have two CSMA instances competing for the channel access instead of only one, so we may have collisions and re-transmissions at layer 2.

E1:TCP-TCP

In this scenario W2 sends data to E3 and E3 sends data to W1.

Connection	Expected goodput	Average goodput	Min goodput	Max goodput
E3 → W1 (TCP)	Around 9.5 Mbps	9.25 Mbps	9.19 Mbps	9.30 Mbps
W2 → E3 (TCP)	Around 9.5 Mbps	9.20 Mbps	8.8 Mbps	9.47 Mbps

Figure 4.13: Table 10

E2:UDP-UDP

In this scenario we assume that two connections start almost at the same time. There is no congestion control algorithm as well as ACKs. W2 will get access to the wireless channel first, because W2 has access to channel sooner rather than W1. Since UDP is aggressive, we expect W2 connection to finish first. W2 will send packets at a higher speed than that at which the switch manages to forward them on the Ethernet link, since the bottleneck is represented by the Ethernet

link, so we expect lots of losses since a congestion control algorithm is not present. Expected value for W2 goodput: 9.6 Mbps. Finally, we expect the other connection to finish much later since it's limited by the capacity of 10 Mbps of the Ethernet link. Therefore, the expected value for E3 goodput should be lower than 9.6 Mbps.

Connection	Expected goodput(at RX)	Observed goodput(at TX)	Observed goodput(at RX)
W1 → E3 (UDP)	Almost 9.6 Mbps	23 Mbps	9.56 Mbps
E3 → W2 (UDP)	A bit less than 9.6 Mbps	9.65 Mbps	9.53 Mbps

Figure 4.14: Table 11

E.3: Mixed Bidirectional flow E3 sends to W1 through TCP, W2 sends data to E3 through UDP [W2 =>Eth, W1 <=Eth]

Because of the same reason that we explained before that the speed of W2 is higher, so fill up the buffer of the switch very soon and we have dropping packet because of congestion and there is no congestion control. But as transmission finishes fast we expect Maximum throughput. On the other side TCP, finding all the wireless channel busy, had to wait the end of the UDP transmission in order to get some bandwidth, so it postponed the beginning. So we expect much less goodput than 9.5 Mbps. Like case D3 we expect average goodput around 6 Mbps.

Connection	Expected goodput(at RX)	Observed goodput(at TX)	Observed goodput(at RX)
W2 → E3 (UDP)	Almost 9.6 Mbps	24.5 Mbps	9.48 Mbps
E3 → W1 (TCP)	Almost 6 Mbps	6 Mbps	6 Mbps

Figure 4.15: Table 12

E.4: Mixed Bidirectional flow E3 sends to W1 through UDP, W2 sends data to E3 through TCP [W2 =>Eth, W1 <=Eth]

Like previous cases, the bottleneck imposed by Ethernet, so the speed of media on long run will be 10Mbps. Wireless channel is shared between W1 and W2, because of congestion control W2 transmit at rate 10Mbps on average so we expect average goodput on E3 around 9.5, and in another case when E3 is transmitter through UDP connection we expect goodput about 9.6 Mbps.

Connection	Expected goodput(at RX)	Observed goodput(at TX)	Observed goodput(at RX)
W2 → E3 (TCP)	Almost 9.5 Mbps	9.8 Mbps	9.48 Mbps
E3 → W1 (UDP)	Almost 9.6 Mbps	9.4 Mbps	9.4 Mbps

Figure 4.16: Table 13

4.1.11 Case F: Bidirectional flow W2 sends data to W1, and W1 sends data to W2

As before, we will analyze measurements for all the 4 case, that is TCP-TCP, TCP-UDP, UDP-TCP and UDP- UDP. We have two transmitter accessing to the shared medium. The sharing of the wireless link does not depend on transport protocol, it is layer two sharing mechanism and it is fair based on the packets. So transmitters access to the channel whatever is the transport protocol (TCP or UDP), they have the same chance to transmit. These flows will go to the Access point, Access Point has to re-transmit the same data, so there is another attempt to access the channel by the AP which again is contending for the wireless link as well as two other wireless transmitters, so we have 3 transmitters that tend to transmit. In the case of UDP, it will finish transmission first. TCP has to send back some Acknowledgments, so again there is another access to the channel, so we get again some traffic on the same shared channel. We would expect no losses because the layer two offer a reliable channel, there might be some losses in case the AP get congested but if it has enough buffer that should not be a big problem. We would expect to get maximum 6-7 Mbps for both UDP and TCP provided that they transmit at the same time, if there is shift over time it is not true. We don't expect big difference between UDP and TCP they get roughly the same share. UDP is more aggressive if there are losses, but if there are no losses which is apparently this case we do not see much difference. TCP is more bloated. It has to send more header, must send acknowledgment, but it does not have big impact. Eventually, there might be a bit of bias that UDP is a little bit faster but this can be randomly.

Connection	Average goodput	Max goodput	Min goodput
W2 →W1 (TCP) W1 → W2 (TCP)	4.6 Mbps 4.6 Mbps	5.5 Mbps 5.1 Mbps	4.1 Mbps 4 Mbps
W2 →W1 (UDP) W1 → W2 (UDP)	4.8 Mbps 4.8 Mbps	5.7 Mbps 5.6 Mbps	3.8 Mbps 4 Mbps
W2 →W1 (UDP) W1 → W2 (TCP)	5.3 Mbps 4 Mbps	6.3 Mbps 4.7 Mbps	4.8 Mbps 3.8 Mbps
W2 →W1 (TCP) W1 → W2 (UDP)	4.2 Mbps 5.5 Mbps	5 Mbps 6.1 Mbps	3.7 Mbps 5 Mbps

Figure 4.17: Table 14