TABLE OF CONTENTS

24/30

Lot of errors

Some comments are missing

INSTRUMENTS USED

- **Switch**

Brand: OfficeConnect
Type: Dual Speed Switch 5

- **3 Computers**

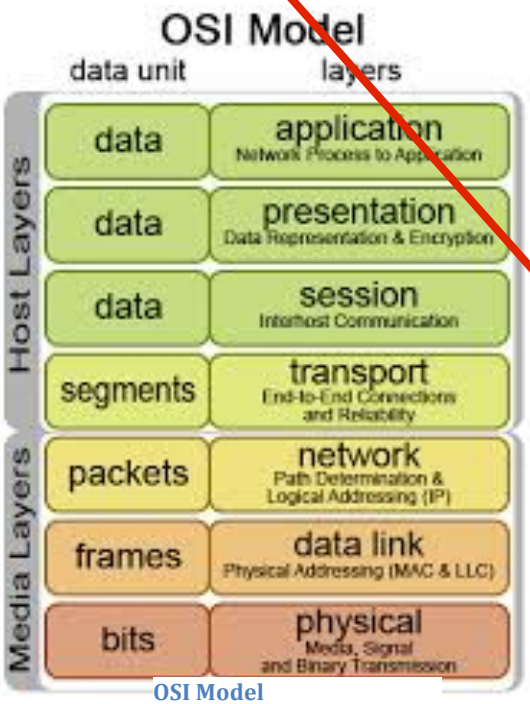
Brand: HP

- **3 Cables**

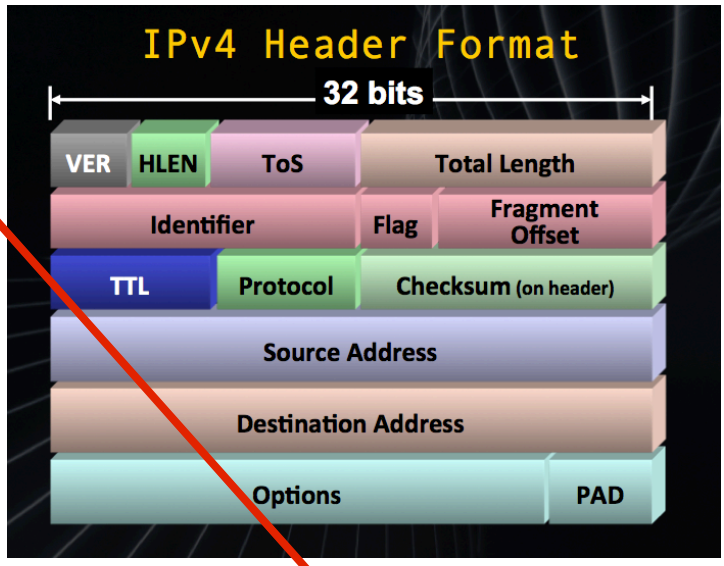
INTRODUCTION

(Some brief notions on IP, Netmask, MAC address, and ARP)

Probably the easiest way to describe the Internet is by defining it a "**network of networks**". It is a global system of interconnected computer networks which can be private, public, academic, company or government owned. The way all these devices are linked together is by using standard protocols which serve specific purposes. In the internet protocol suite, the principal communication protocol is the **IP protocol** (also known as IPv4, or IP Version 4). In the OSI model (Open Systems Interconnection), we can find it on layer 3 – the network layer, and its header format is the following:



OSI Model



Header Format

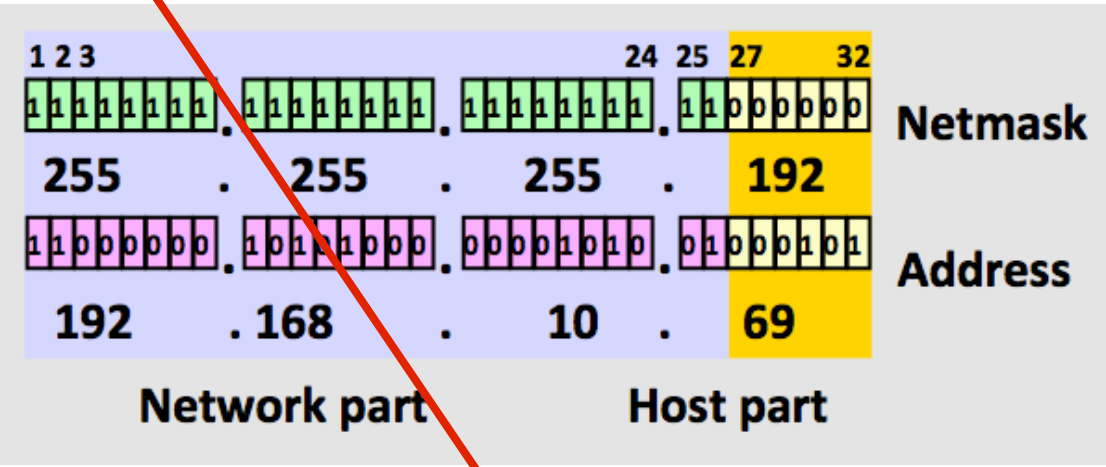
Among the header sections we recall the:

- *Identifier* – An identification field used to identify the group of fragments of a single IP datagram.
- *Flags* – A 3-bit field used to control or identify fragments (reserved, don't fragment, more fragments).
- *Fragment offset* - specifies the position of an IP fragment relative to the beginning of the unfragmented IP datagram.
- *TTL (Time to live)* – prevents datagrams from persisting in networks. At each router hop the TTL is decreased by one. When it reaches 0 the packet is discarded.
- *Source Address* – IP source address

- *Destination Address* – IP destination address

IP is a connectionless protocol for use on packet-switched networks. IPv4 uses 4 bytes long addresses written in decimal digits separated by dots, such as 95.244.86.131, 192.168.1.10, and so on…

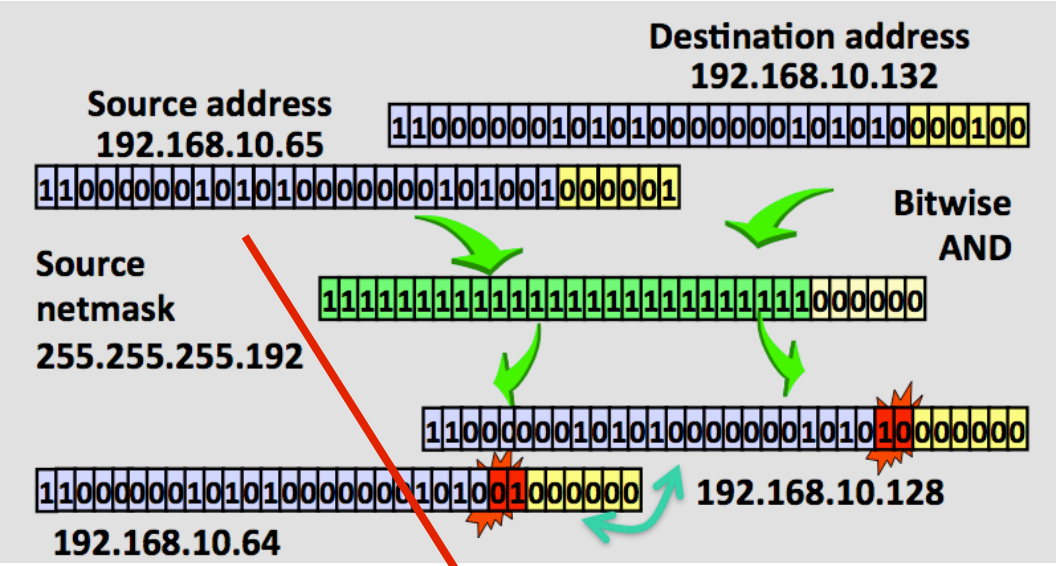
In order to better understand the concept of "network of networks", let's introduce what is known as the **Netmask**. The netmask is a further 8-byte value that always goes "hand-in-hand" with an IP address. An IP address always has some bits which are used to define a network, and some bits to define particular hosts present in the network. The netmask is what sets the boundary between the "network part" of an IP address and the "host part", and it is always composed by a succession of ones followed by a succession of zeroes, the ones indicating the bits of the IP address useful to identify the network and the zeroes the bits useful to identify the host.



Moreover, a netmask is called in such a way because it literally "masks" the network part of an IP address. In fact, by computing the bitwise AND between a given IP address and its netmask it is possible to identify its network part (i.e. the network address of that particular logical IP subnet which that IP address belongs to). Therefore the netmask is of fundamental importance, as it can be used to determine whether two hosts belong to the same network (IP logical subnet) or not, by simply computing the AND between their IP addresses and the netmask in question. For example, consider the following images:

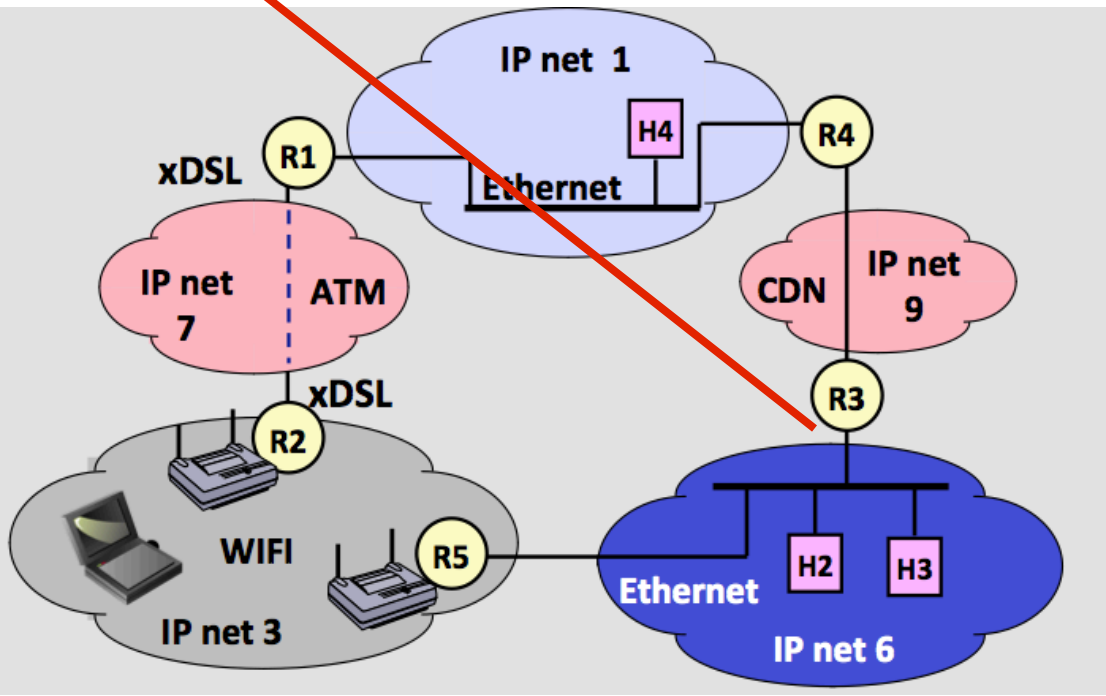
CASE 1

CASE 2



In CASE 1, the two hosts with IP addresses 192.168.10.101 and 192.168.10.65, given the netmask 255.255.255.192, belong to the same logical IP subnet which is the result of the bitwise AND between the addresses and the netmask: 192.168.10.64.
This is not the situation for CASE 2. In fact, once the AND between the hosts' IP addressed and the netmask is performed, the two results are different, meaning that the two hosts belong to two different IP subnets (specifically with addresses 192.168.10.128 and 192.168.10.64).

Having this in mind, the concept of the internet being a "network of networks" can be clarified with the aid of the following image:

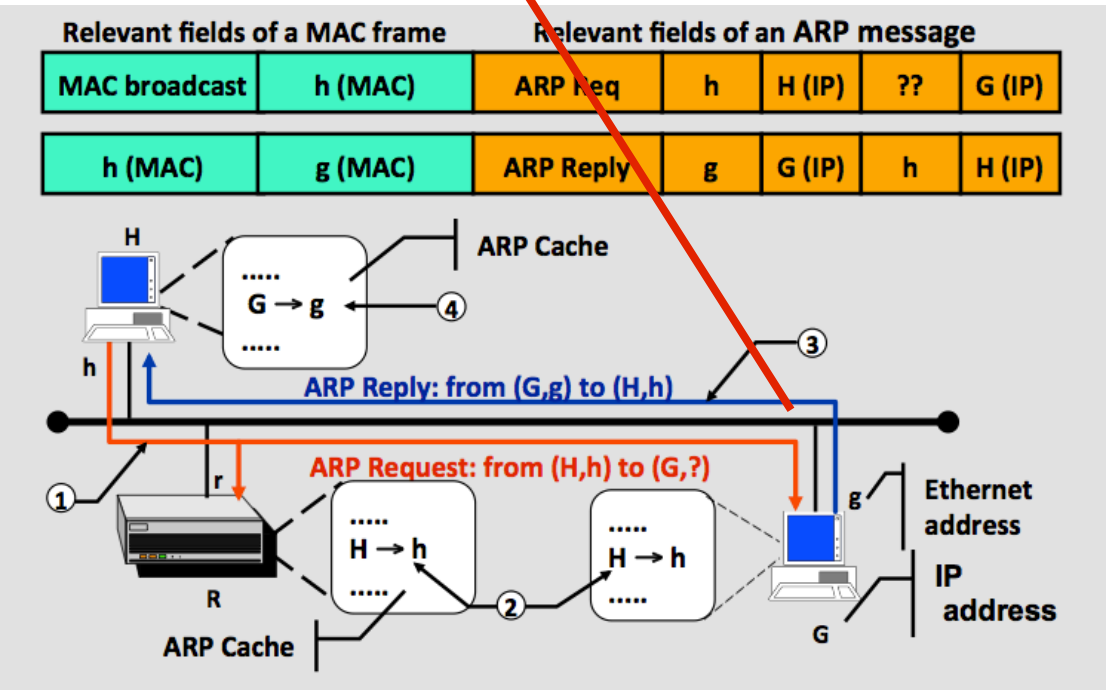


Each "cloud" corresponds to a logical IP subnet, which is identified by the network part of the IP addresses. The netmask has the purpose of "hiding" the network address

within the IP address of each host belonging to a particular subnet (also called "physical network"). Whenever a host needs to send a packet to a destination, first the check with the netmask and the AND is performed. If the two hosts happen to be on different physical networks then routers are involved (via routing tables, etc…). Otherwise, if the source and destination host lie on the same physical network (CASE 2), they are said to be "directly connected" and the packet is sent directly to the destination's host MAC address.

The **MAC address** is a unique identifier attached to network interfaces for communication on the data link (layer 2) network. It is expressed in a 6-byte hexadecimal format (such as 01:23:45:67:89:ab). They are used as a network address for most IEEE 802 technologies, including Ethernet. Any IP logical subnet may have different hosts, but, in order for the network to work properly, the MAC addresses in such a subnet must be unique. When a host needs to send packets to another host which is directly connected, or when a packet reaches, through a router, the LIS (logical IP subnet) which the destination host belongs to, the destination's MAC address needs to be known by either the source host in the first scenario, or by the router in the second scenario. The way the MAC address is resolved is via the ARP protocol (Address Resolution Protocol).

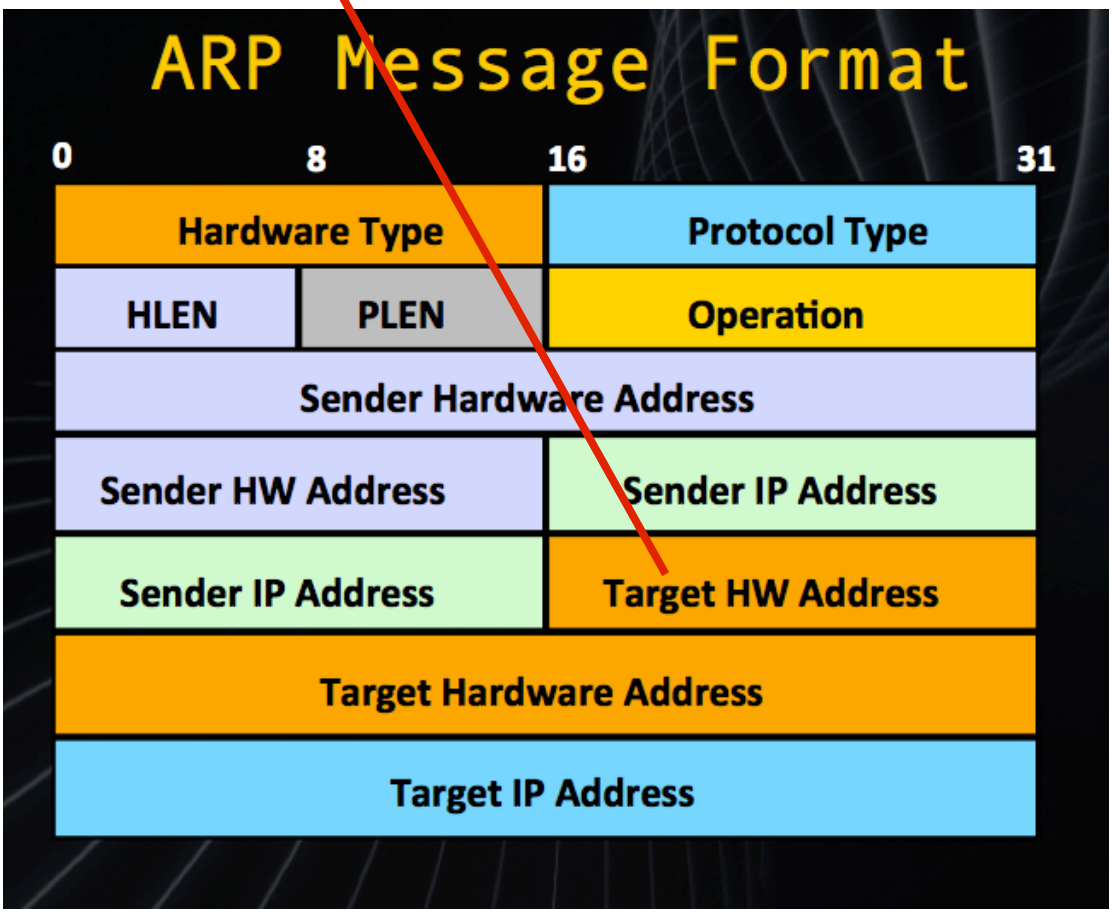
The **ARP** is a solicitation protocol based on broadcast, the purpose of which is to perform a mapping between layer 3 (IP) and layer 2 (MAC) addresses. When a host needs to send frames to another host which is directly connected, the first thing the source host does is to check its **ARP cache**, a particular cache containing mappings between MAC and IP addresses of the hosts lying on the same physical network, to check if it already has the destination host's MAC address. If it does, the packets are sent. Otherwise an ARP request is executed.
An ARP request is a message sent on broadcast throughout the physical network in question. Let's observe the following diagram to understand how it works:



ARP requests are encapsulated in Ethernet frames, and the ARP cycle can be described as follows:

1. The source host checks its ARP cache and realizes that the destination's host MAC address is not present. Therefore it generates an ARP request sending it on broadcast, and attaching to it its source MAC address. In the figure above it is asking, in plain words, "which MAC address possesses the IP address <G>?"
2. Since the ARP request is sent on broadcast, whoever listens to it can map H's MAC address to its IP address (this is shown being done by the router ad the destination host up above), although this is not strictly necessary for the correct functioning of the protocol.
3. The destination host G replies to the original ARP request by inserting its MAC address and sending the frame back to host H, changing the format of the message from "request" to "reply". Now, the ARP message is no longer on broadcast. →ARP requests are broadcasted, replies are unicasted.
4. The original source host H maps G's IP address to its MAC address and updates its ARP table.

The ARP message format is the following:

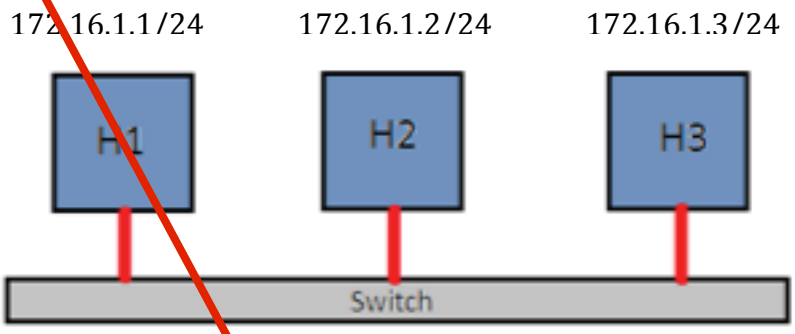


The HW (hardware addresses) refer to the MAC addresses. The only particular thing worth noticing is that this message format is the same for both ARP requests and replies, in the sense that practically the same ARP message is sent as a reply. The only fields that need to be updated are the operation field which will be changed from request to reply, and, obviously, the target HW address which is no longer the

broadcast but the host's which issued the request in the first place. Sender and target addresses are inverted and all the rest remains untouched.

IN THE LAB

The purpose of this investigation is to familiarize with the above functions embedded in networks in general. In order to do this we set up our own LAN (Local Area Network) composed of three hosts and a switch. No routers were involved. A diagram with the hosts' addresses and netmask is provided below:



All the hosts were configured from the terminal using the command "ifconfig". This investigation is mainly based on the use of the **ping command** in order to check connectivity.

Ping is a computer network program used to test the reachability of a particular host on an IP network and to measure the RTT (Round Trip Time) for messages sent from a source to a destination host (i.e. how much time it takes for the packet to go back and forth).

```
root@laboratorio:/home/laboratorio# ping 172.16.1.1 -c3
PING 172.16.1.1 (172.16.1.1) 56(84) bytes of data.
64 bytes from 172.16.1.1: icmp_req=1 ttl=64 time=1.22 ms
64 bytes from 172.16.1.1: icmp_req=2 ttl=64 time=0.578 ms
64 bytes from 172.16.1.1: icmp_req=3 ttl=64 time=0.610 ms

--- 172.16.1.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.578/0.803/1.222/0.297 ms
```

Ping operates by sending **ICMP** (Internet Control Message Protocol) messages and waiting for ICMP replies. There are many varieties of ICMP messages, but here we will focus only on the ones that ping uses: ICMP echo request and ICMP echo reply. The format of the ping command is the following:

```
ping -s(SIZE) -c(NUM OF PING MESSAGES) <IP Address>
```

In the ping command it is in fact possible not only to decide how many messages to send, but there is also the possibility of defining the size of the packet being sent. Although this is a very important functionality, it is not particularly relevant in this investigation.
The purpose of this experience is instead to analyse what happens in case the ping command is used under certain peculiar conditions. These include pinging the broadcast address, executing ping when there are duplicate IP addresses on the same LAN, pinging hosts when the netmask being used is wrong (the concept of "wrong" netmask will become clear in short), and pinging when the netmask is wrong and the broadcast address has been assigned to a host.

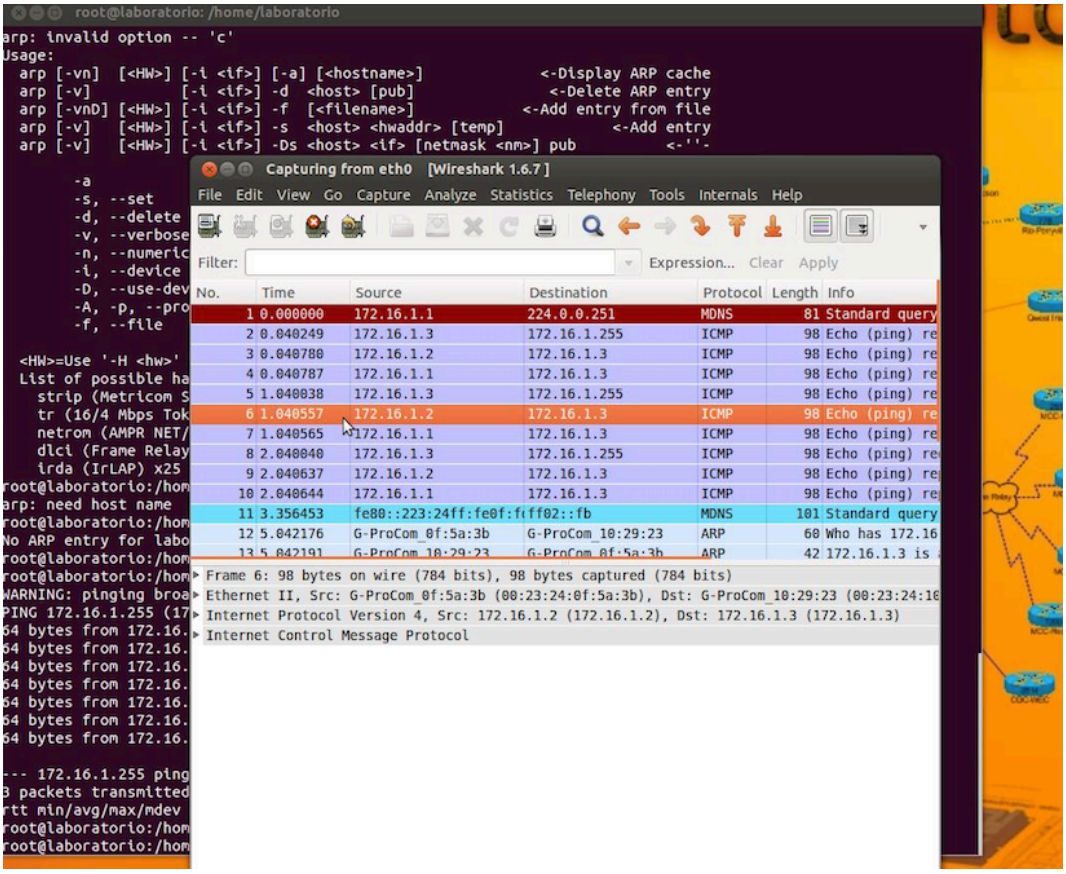
ADVANCED PING

Pinging the broadcast address:

/25

1. Suppose now that a host, say H3, pings the broadcast address. Being the netmask /24, the broadcast address is given by 172.16.1.255.
   Since the ping is sent on broadcast, both H1 and H2 will receive it and send ICMP replies to the host who executed the ping command, as it is possible to see from the picture below. H1 and H2 will also update their arp caches with H3's IP address. However, H3 does not update its cache with H1's and H2's address because it did not ping them directly but pinged the broadcast address instead, and no MAC address is associated to the broadcast address, therefore H3's arp table remains unvaried.

NO



2. Now let's analyse what happens when a host pings the network address. Let's recall that the netmask is 255.255.255.0, therefore by computing the bitwise AND with any IP address it is clear that the network address will result to be X.X.X.0.
   When we tried to ping the network address from H1, we obtained the following result:

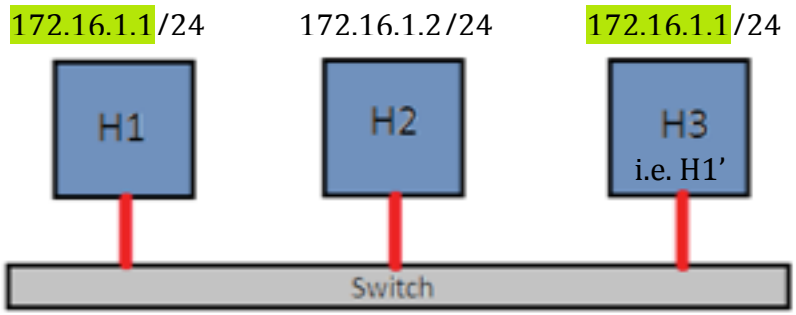
```
root@laboratorio:/home/laboratorio# ping 172.16.1.0
Do you want to ping broadcast? Then -b
root@laboratorio:/home/laboratorio# ping 172.16.1.0
Do you want to ping broadcast? Then -b
```

   The shell is asking if we want to ping the broadcast address, i.e. the network address is just the network's "identifier", it is not related to any machines connected to it. There is no purpose in trying to ping a network which a host already belongs to, since of course it will be reachable.

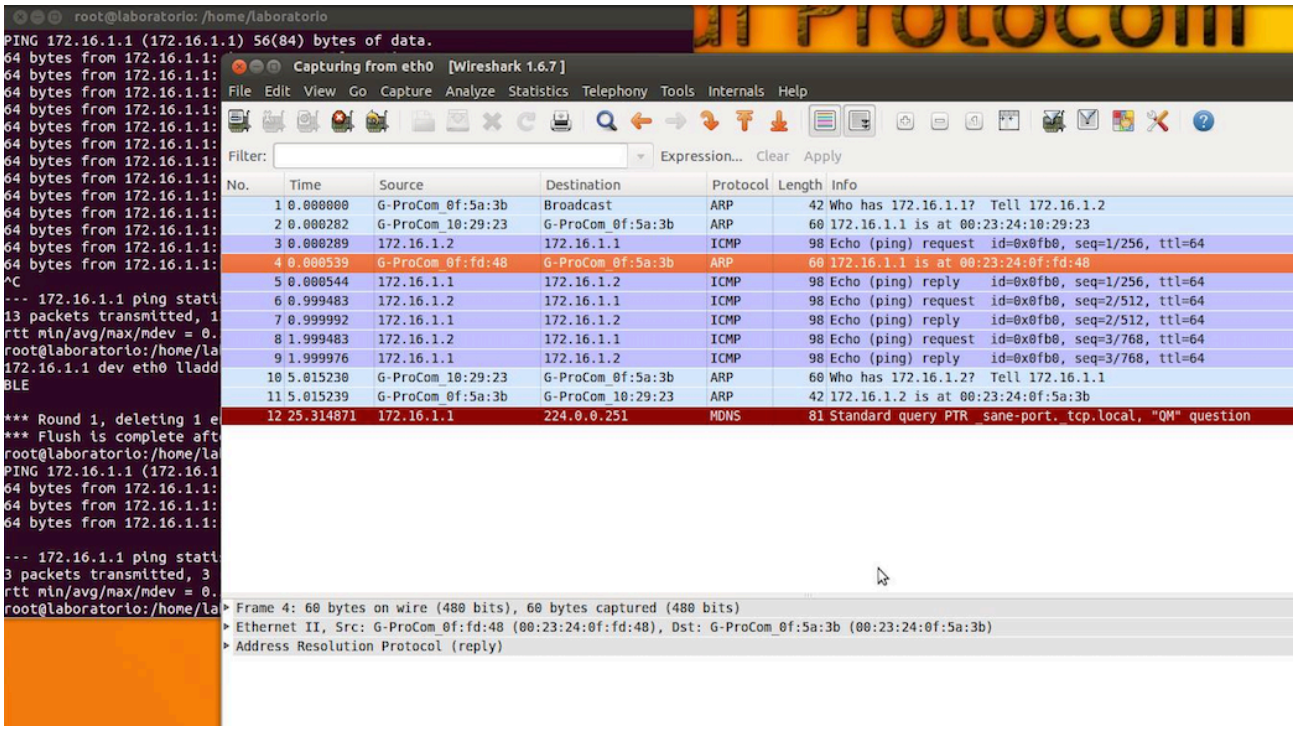
NO

Duplicate Addresses

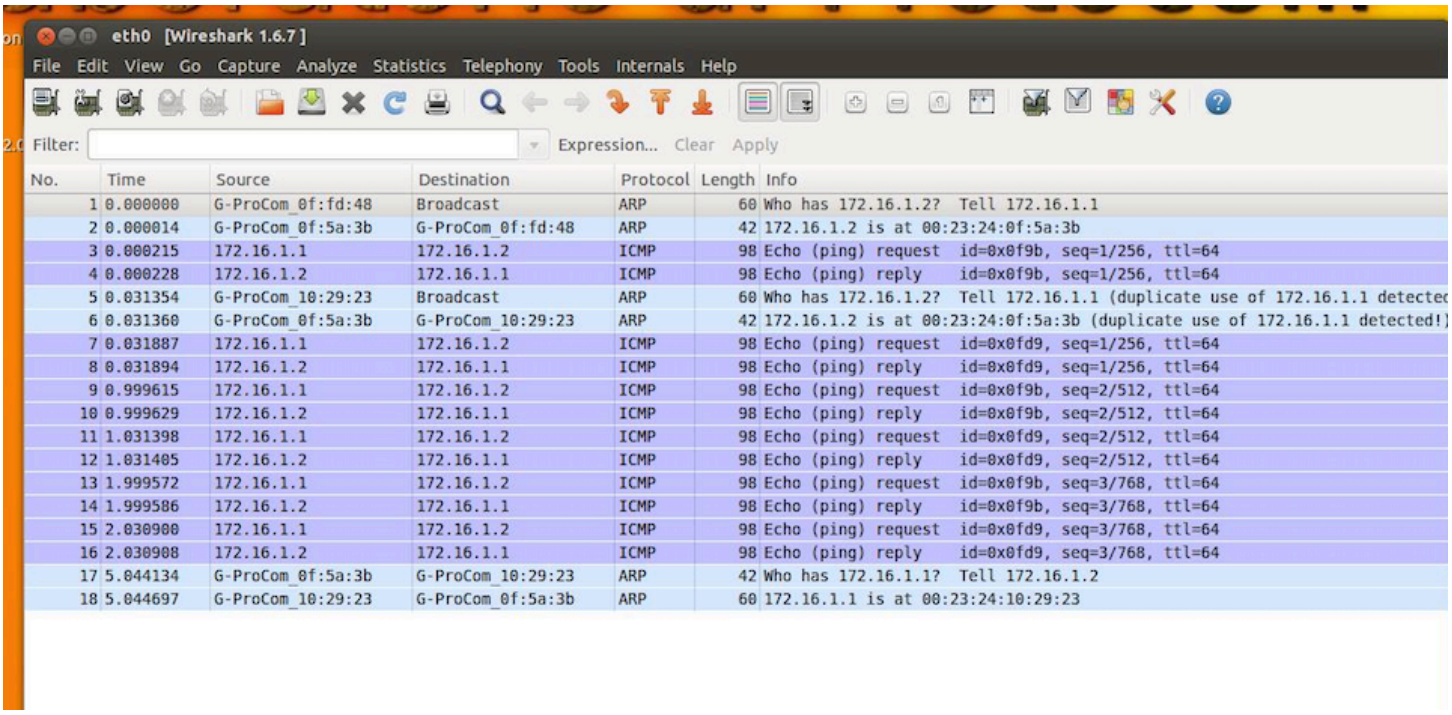
Now suppose that by mistake we configured the LAN in such a way that two hosts have the same IP address (H1 and H1'). Our LAN then looks something like this:



1. Now we want to see what happens when H2 pings H1. Before executing the ping we empty the ARP table of H2. As it is possible to notice from the snapshot below, the first thing H2 does is send a broadcast ARP request. One of the H1 answers first, and H2 sends him the ICMP packets.
   Once all the ping commands have been correctly executed H2 will update its cash with the H1 who answered first, which is also the one that was receiving the ping replies from H2. Although in the end the final ARP request (i.e. the ARP request to H2) is sent by the H1 that was receiving the ICMP packets, both H1 and H1' will have their ARP caches updated with the mapping between H2's MAC and IP address.



2. Suppose now that both H1 and H1' ping H2 at the same time. Let's analyze the screenshot below:



Who says that?

The second H1 who sends the ARP request is detected by H2 as "duplicated", as it is possible to notice from line 5. In fact at this point H2 has already received an ARP request, but is now detecting a second one from the same source IP address. Therefore this is notified in the ARP message: "duplicate use of 172.16.1.1 detected!".
H2 replies to both H1s. In fact the ping command executed on H1 and H1' had this format:

NO!!!!

```
ping -c3 172.16.1.2
```

but, although H1 and H1' issued only 3 requests each, H2 sent 6 replies. Afterwards, H2 sends the final ARP request to H1 (line 17) in order to update its cache. The H1 to whom this ARP request is sent is the last of the two who sent the ARP request to H2 (i.e. the H1 who was classified as "duplicated").
In both H1s the ARP table is updated with H2's address as a result of the ARP requests they sent to it in the beginning (line 1 and line 5).

Who receives those responses???

Wrong Netmask

By wrong netmask we mean configuring the hosts in such a way that they appear as belonging on different subnets. As a consequence of this they might have difficulties in detecting each other even though they are physically connected on the same LAN. Moreover, it may also happen that one host sees another one as belonging to the same subnet, while the second one does not detect the network which the first one belongs to.
In this particular case we configured our LAN in such a way that H1 sees H2 as belonging to its subnet, but H2 sees H1 as NOT belonging to his subnet, i.e. we used different netmasks: 172.16.1.130/24 for H1, and 172.16.1.2/25 for H2. We also changed the IP address for H1 because in this way when H2 computes the bitwise and between its IP address and its netmask and between H1's address and its netmask, H2 sees H1 as belonging to a different network. To clarify this:

From H1's perspective:

172.16.1.130 & 255.255.255.0 = 172.16.1.0 → network H1 lies on
172.16.1.2 & 255.255.255.0 = 172.16.1.0 → network H2 lies on

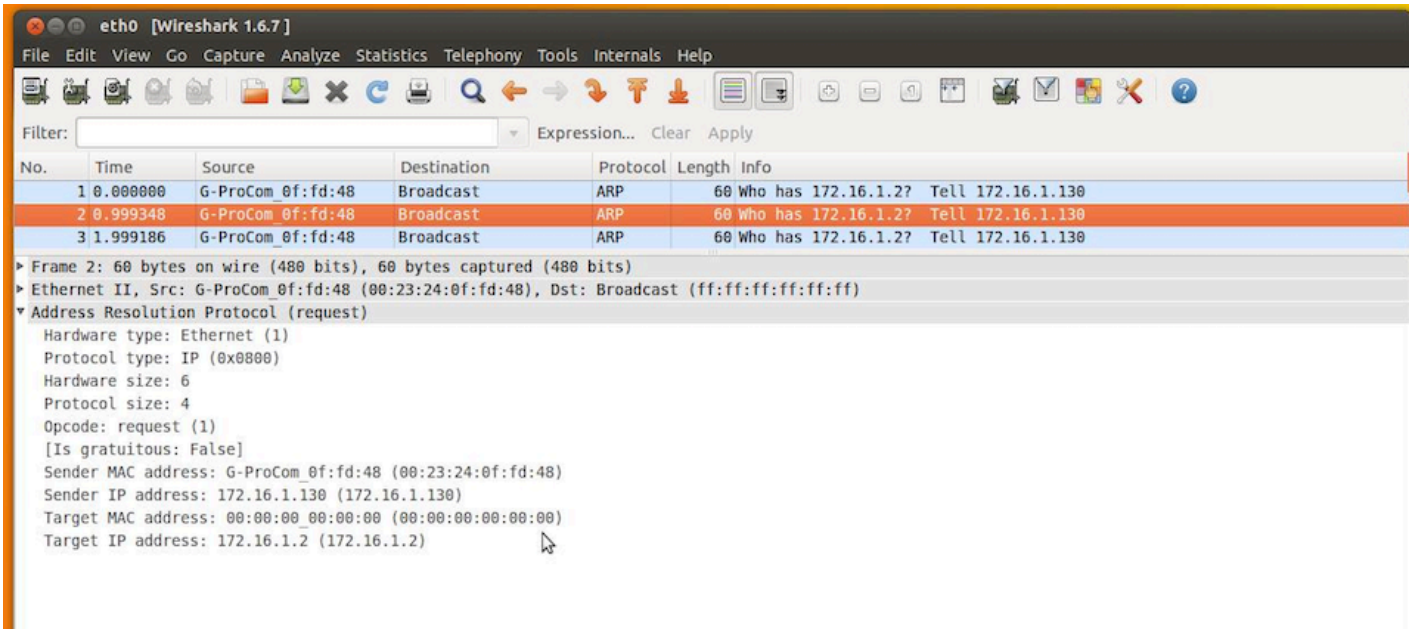
→ H1 and H2 belong to the same LIS.

From H2's perspective:

172.16.1.2 & 255.255.255.128 = 172.16.1.0 → network H2 lies on
172.16.1.130 & 255.255.255.128 = 172.16.1.128 → network H1 lies on

→H1 and H2 belong to two different LISs.

1. Now suppose H1 pings H2. H1 sends a broadcast ARP because it sees H2 as belonging to his subnet, but H2 never replies because from its perspective H1 is on a different LAN. Therefore there are no updates in ARP tables from either side and no packets are sent. This is shown in the picture reported below:



2. Suppose now that H2 now pings H1. H2 does not see H1, so we have a network unreachable error. Therefore no packets are sent and the ARP tables are not updated in any way.

```
root@laboratorio:/home/laboratorio# ping 172.16.1.130 -c3
connect: Network is unreachable
```

Wrong Netmask and conflict with broadcast address

As a last point of interest in this investigation we consider the following scenario. Suppose we configure the LAN in such a way that H1 has address 172.16.0.127/24, and H2 has address 172.16.0.1/25. Therefore we have:

From H1's perspective:

172.16.0.127 & 255.255.255.0 = 172.16.0.0 → network H1 lies on
172.16.0.1 & 255.255.255.0 = 172.16.0.0 → network H2 lies on

→ H1 and H2 belong to the same LIS.

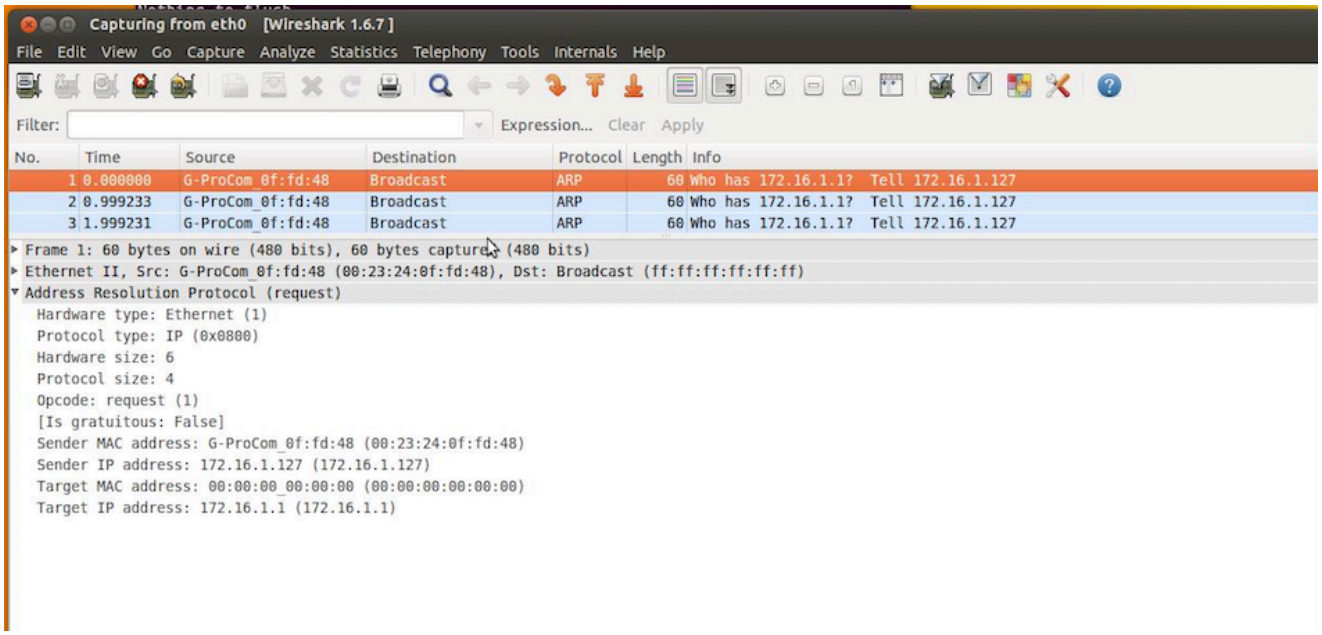
From H2's perspective:

172.16.0.1 & 255.255.255.128 = 172.16.0.0 → network H2 lies on
172.16.0.127 & 255.255.255.128 = 172.16.0.0 → network H1 lies on

→H1 and H2 belong to the same LIS.

However, H1 has the broadcast address of the network which H2 lies on. This means that H2 should not be able to send packets to H1, since from H2's perspective it would be as sending packets to the broadcast address, which, by definition, does not belong to a specific machine.
Consider the following two scenarios:

1. Suppose H1 pings H2. H1 sends ARP requests because it sees H2 as belonging to the same subnet. However H2 never sends replies because from its perspective it would be the same as sending an ARP reply in broadcast, which, as we have seen before, never happens simply because ARP replies are designed to be unicasted. No ARP tables are updated.



2. Observe the picture below. As a last scenario now suppose that H2 is pinging H1. But H2 sees H1's IP address as its own network broadcast address. Therefore H2 starts sending ICMP requests on broadcast (line 1). H1 sees the messages and sends an ARP request with H2's IP address as destination (line 2), in order to resolve its MAC address.
   However, for the same reasons explained on the previous point (i.e. ARP replies are unicasted by definition), H2 sends no reply. Neither H1's ARP table nor H2's are updated.

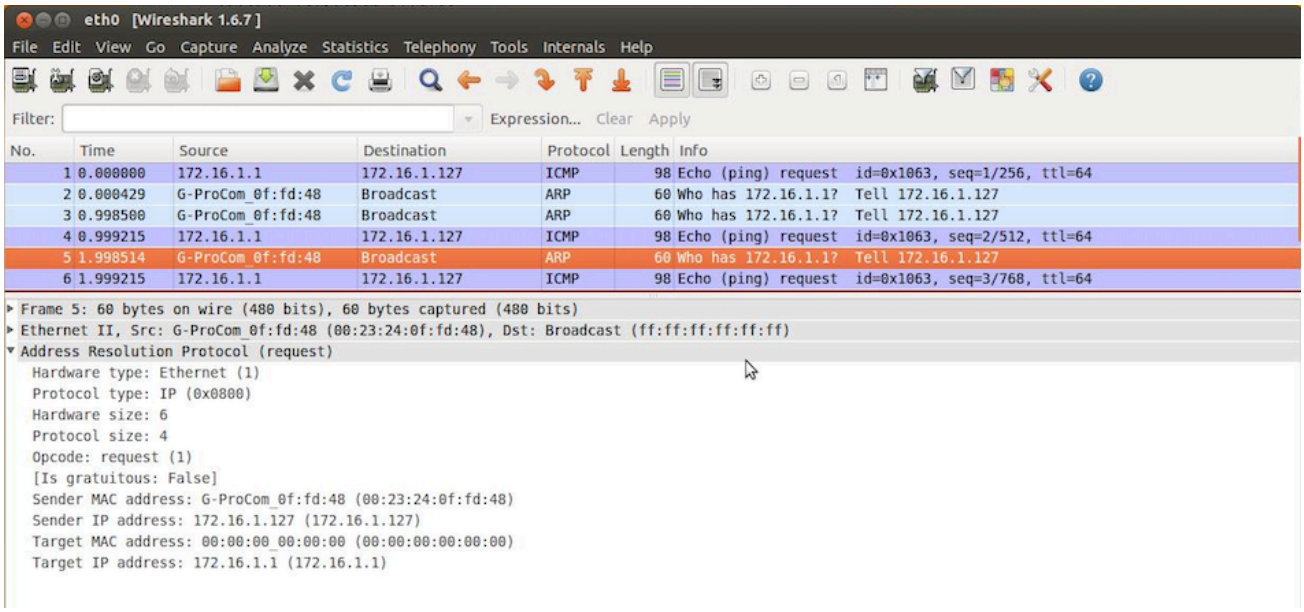
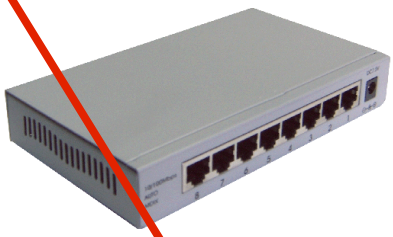


TABLE OF CONTENTS

INSTRUMENTS USED

- **Switch**



Brand: Office Connect
Type: Dual Speed Switch 5

- **3 Computers**

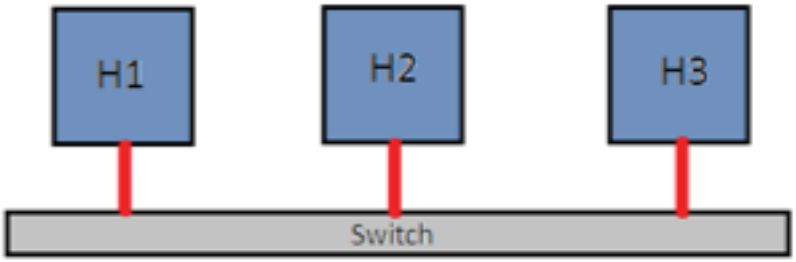
Brand: HP

- **3 Cables**

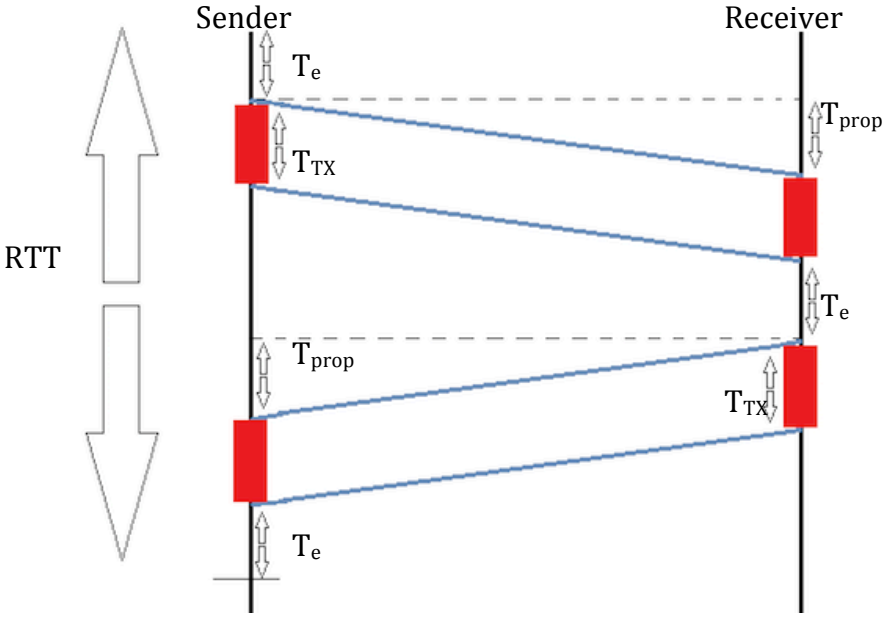
INTRODUCTION

We have already seen in the previous investigation how the command "ping" can be used for connectivity checks and what the outcomes are in different scenarios (e.g. wrong Netmasks, pinging the Broadcast address, having two hosts configured with the same IP address and so on…). In this investigation we will instead use ping for a different purpose: estimating the capacity offered by the physical layer.

The **capacity** of a physical layer is defined as "the rate at which data can be transmitted over a given channel under given conditions".[1] Before proceeding, let us recall how the equipment is setup and describe how the investigation will be carried out.



As usual the three hosts are connected to each other via a switch. The goal of the experience is to estimate the capacity, C, of the links by using ping only. First of all, in the ping statistics we find the Round Trip Time, RTT, which is related to the speed with which data is sent. The following diagram can help us provide a clear and visual definition of RTT:



[1] http://www.google.it/url?sa=t&rct=j&q=&esrc=s&source=web&cd=4&ved=0CFcQFjAD&url=http%3A%2F%2Fweb.cs.wpi.edu%2F~rek%2FUndergrad_Nets%2FC04%2FPhysicalLayer.ppt&ei=a4d3U-CLLs6w7AaIIw&usg=AFQjCNHM9r-d8J8u2uG84qmv_FQOUkpsbg&sig2=gCYO-uDi_FFhCFSu3h2UBw&bvm=bv.66917471,d.ZGU

In the above diagram let us define:

- $\mathbf{T_e}$ – Elaboration time. That is the delay it takes for a node in the network to process incoming data and deliver a new packet.
- $\mathbf{T_{TX}}$ – Transmission time. The time it takes for a node in the network to fully send a packet.
- $\mathbf{T_{prop}}$ – Propagation time. The time it takes for data to travel along the physical channel in between two nodes.

Having defined the above parameters, it is clear that the RTT, the time it takes for a packet to go back and forth two nodes at the extremes of the network, can be expressed as:

$$RTT = 2T_{TX} + 3T_e + 2T_{prop}$$

From the above we can derive an expression for the capacity.

DESCRIBING THE METHODOLOGY

By definition, the capacity C can be described as:

$$C = \frac{D}{T_{TX}}$$

where D is the amount of total data contained in the packets.

Substituting in the above equation we get:

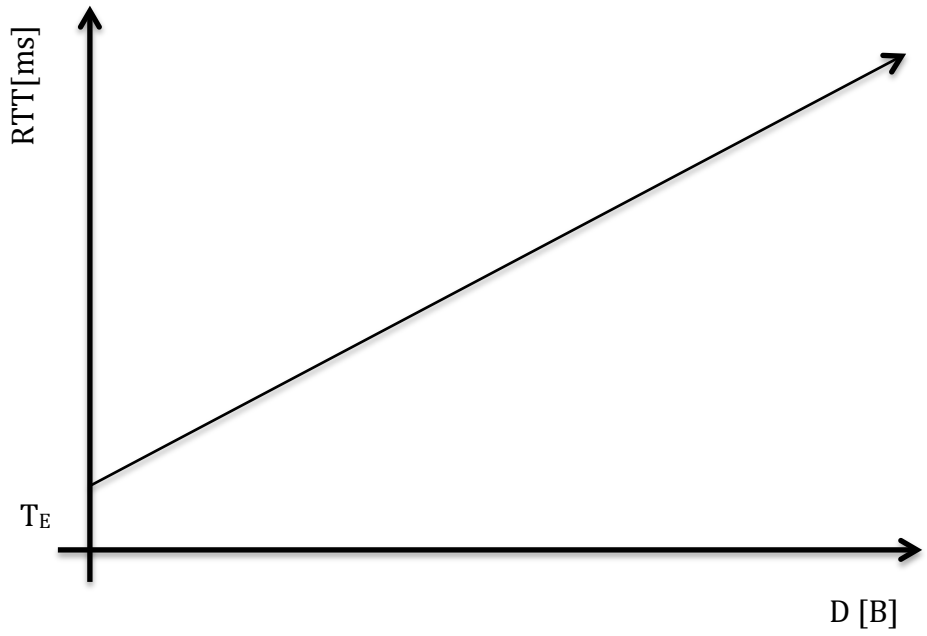
$$RTT = 2\frac{D}{C} + 3T_e + 2T_{prop}$$

If we neglect the propagation time T_{prop} and the elaboration time T_e we get the much more simplified expression

$$RTT = 2\frac{D}{C}$$

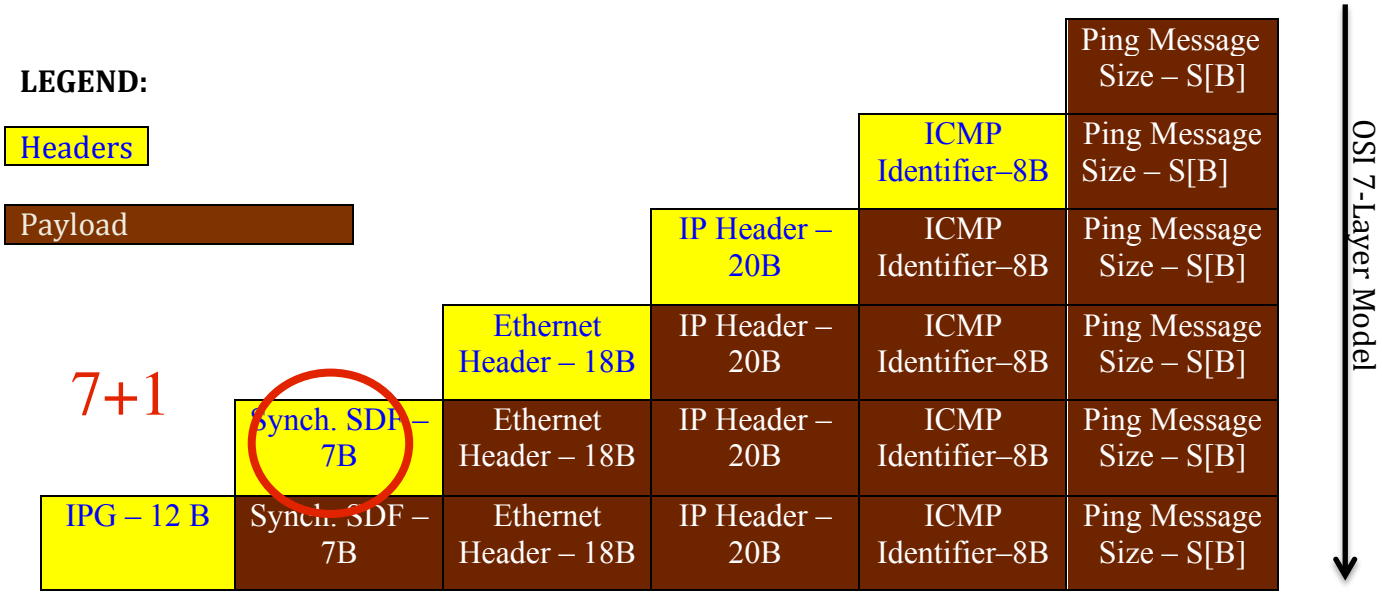
$$\Rightarrow \boldsymbol{C = \frac{2D}{RTT}}$$

Before going into further detail on how to compute the capacity using ping, let us first consider some points on the size of the PDU being transmitted, D. We would expect the two quantities to be linearly correlated. That is, if we make a plot of the RTT vs. S we expect the result to be something similar to the following graph:



Where $T_E=3T_e$. In fact notice that if D=0, then $T_{TX}=0$, and as a consequence RTT=T_E. Also, keep in mind that before T_e was assumed to be 0 in order to derive a very simple expression for the RTT, but in reality it cannot be completely neglected. This instead is not true for T_{prop}: being the propagation time dependent on the speed of light, and being the channel used extremely short (less than 10m), we can safely assume this quantity to be 0.

It is really important to stress that D is NOT the size of the ping message, which we shall address as S, but the size of the PDU being sent over the channel. That is, D is the size of the ping message plus all the additional headers. The following diagram can better clarify this concept:



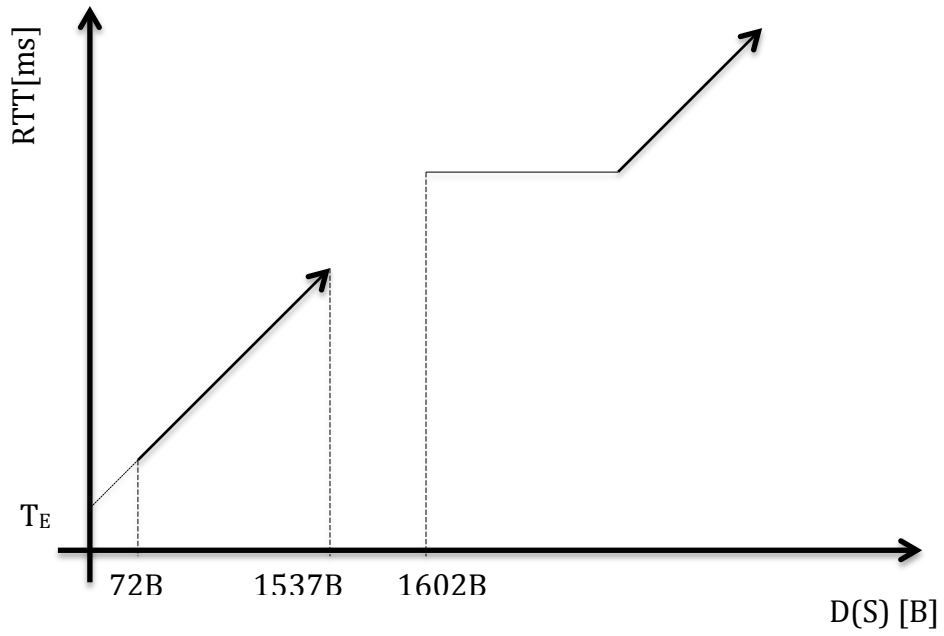
We see that whenever we go down a level a new header is added to the current PDU total size. Therefore, by the time we reach the physical layer, the overall PDU size will be equal to:

$$D(S) = S + 8_{ICMP\ identifier} + 20_{IP\ Header} + 18_{Ethernet\ Header} + 7_{Synch.SDF} + 12_{IPG}$$

$$\Rightarrow \boldsymbol{D(S) = S + 65B}$$

Taking into consideration the effects that all the various headers imply (the analysis of which is made right below the following graph), we expect the graph RTT vs. PDUsize to look something like this:

GRAPH 1



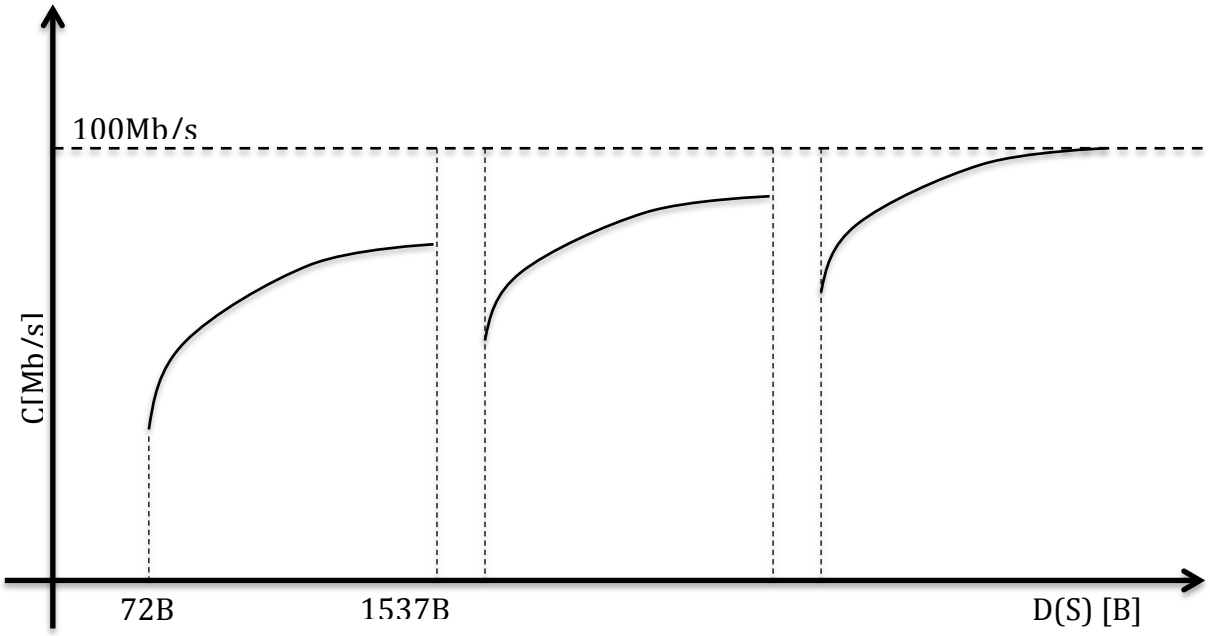
The graph starts at 72B because the header of the PDU is 72B, and, since we always need an initial header, it is not possible to transmit a PDU which has D<72B.
Then we need to keep in mind that the maximum Ethernet payload is 1500B, and if an Ethernet packet exceeds this size the packet is fragmented. If we look at the PDU size at the physical layer, this corresponds to fragmentation occurring when D=1537B (1500+18+7+12). When a packet is fragmented, the 65B header for the new packet is immediately created. This means that the total message size jumps from its current value, say x, to x+65B, and there can be no messages which assume a size in between x and x+65B.
The region of the graph in which the RTT remains constant can is due to padding on Ethernet frames. We know that the minimum Ethernet packet size is 64B, however, it can happen that the amount of useful data that needs to be transferred is less than this minimum required value. To solve this problem "padding" is executed. That is, the Ethernet frame if filled with bytes which carry no useful information, but that can be overwritten if needed. This means that the ping message size can still increase without actually modifying the size of the packets, therefore the RTT remains constant until all the padding bytes have been overwritten. When this happens, the RTT starts to increase again as before until the next critical byte at which fragmentation occurs.

Now let us consider the capacity vs. pdu size graph. Let us recall the formula:

$$\boldsymbol{C = \frac{2D}{RTT}}$$

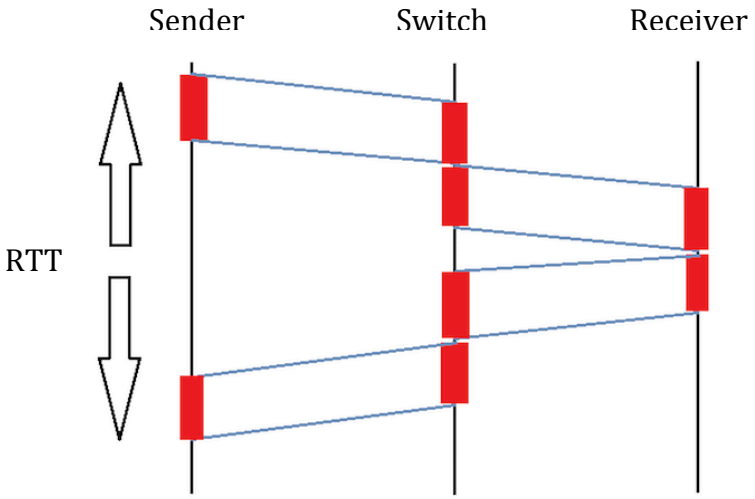
we know that as D increases, so does RTT. Therefore, assuming that the capacity has a maximum possible attainable value, say 100Mb/s, we can expect the graph to look something like:

GRAPH 2



The points at which the capacity suddenly decreases correspond to the points in which the RTT suddenly increases (see GRAPH 1), i.e. when fragmentation occurs.

As a further remark, let us recall that during the experience a switch was provided to connect the three different hosts, hence adding an extra delay:



As before, let us assume $T_{prop}=T_E=0$. Therefore we have that:

NO

$$RTT = 4T_{Tx} = \frac{D}{C}$$

$$\Rightarrow C = \frac{\boldsymbol{4D}}{\boldsymbol{RTT}}$$

And this is the formula for C that will be used when plotting the graphs during the investigation.

PLOTTING THE GRAPHS

The graphs that need to be plotted are the RTT vs PDUsize and Capacity vs PDUsize. The basic idea behind both graphs is using the ping command in the following format:

```
ping -s(SIZE) -c(NUM OF PING MESSAGES) <IP Address>
```

For both graphs we require the PDU size at the physical layer (which can be retrieved from the `(SIZE)` defined in the ping command by adding all the additional headers), and the RTT, which can be retrieved by the ping statistics once the command has been executed. Note that we want the minimum possible RTT, since we want to calculate the maximum possible capacity of the link.
Therefore the idea is to run a large amount of pings and store the required data at the end of each command. The way this can be achieved is by writing a *bash script*, which we can call bash.sh, for instance, and save it on the desktop with a "for" cycle inside:

```
for ((i=0; i<10000; i+=10))

do

echo $i>>size.dat
ping 172.16.1.1 -s $i -c3 -i 0.01|grep min | cut -d '/' -
f4|cut -d "=" -f2>>result.dat

done
```

What this small program does is increase the size of the ping message from 0 to 10000B at intervals of 10B, and send each packet with intervals of 0.01s in between. At each iteration the line `echo $i>>size.dat` appends the current size of the message in a file called "size.dat" which is created on the desktop. Instead the line `|grep min | cut -d '/' -f4|cut -d "=" -f2>>result.dat` in plain words selects the line containing the word "min" inside, crops it according to the delimiter "/" and chooses the 4th field, then crops it again using the delimiter "=" and chooses the 2nd field. This was written in such a way to isolate the min RTT[ms], which is the desired data. This can be better understood by observing the following screenshot:

```
4 packets transmitted, 4 received, 0% packet loss, time 2998ms
rtt min/avg/max/mdev = 0.253/0.364/0.581/0.131 ms
parallels@ubuntu:~$
```

We first select the line containing the field "min". Afterwards we select the field "mdev = 0.253" by using the delimiter "/" and finally we select "0.253", the min RTT, by using the delimiter "=". Through the command `>>result.dat` this value is then appended at each iteration on a file created on the desktop named "result.dat".

Therefore, the first thing we do is change directory to the Desktop and run the bash script through the terminal, by typing in:

```
root@ubuntu:/home/parallels# cd Desktop
root@ubuntu:/home/parallels/Desktop# bash bash.sh
```

At the end of this command the above program is ran and the files “size.dat” and “result.dat” are created on the desktop, containing the size and the min RTT for each ping command executed in the loop. It is convenient to group the two kinds of data on one single file, in order to make the plotting of the graphs easier.

```
root@ubuntu:/home/parallels/Desktop# paste size.dat result.dat>>graphdata.dat
```

The above command creates a new file, “graphdata.dat”, in the same directory as the other two (the desktop). In this file the contents of the other two files are saved in two columns: the first column will contain the data from “size.dat”, i.e. the ping message size, and the second column the data from “result.dat”, i.e. the min RTT for each ping. This file “graphdata.dat” will be the basis for plotting our results.

Plotting RTT vs PDU Size

In order to make our graphs we will use Gnuplot. It can be opened from the terminal:

```
root@ubuntu:/home/parallels/Desktop# gnuplot

        G N U P L O T
        Version 4.6 patchlevel 1    last modified 2012-09-26
        Build System: Linux x86_64

        Copyright (C) 1986-1993, 1998, 2004, 2007-2012
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type set to 'unknown'
gnuplot>
```

Now, since we are plotting the RTT vs PDU Size graph, let us label the axis accordingly (x-axis→PDUsize, y-axis→RTT). This can be done in Gnuplot:

```
gnuplot> set xlabel "PDUsize[B]"
gnuplot> set ylabel "RTT[ms]"
```

Afterwards we can plot the graph using the file "graphdata.dat". Keep in mind that we need the PDU size at the physical layer and not at the application layer. This is dealt with in the following command:

```
gnuplot> plot "graphdata.dat" using ($1+65*($1/1472+1)):2 title "RTT vs PacketSi
ze" with linespoint
```

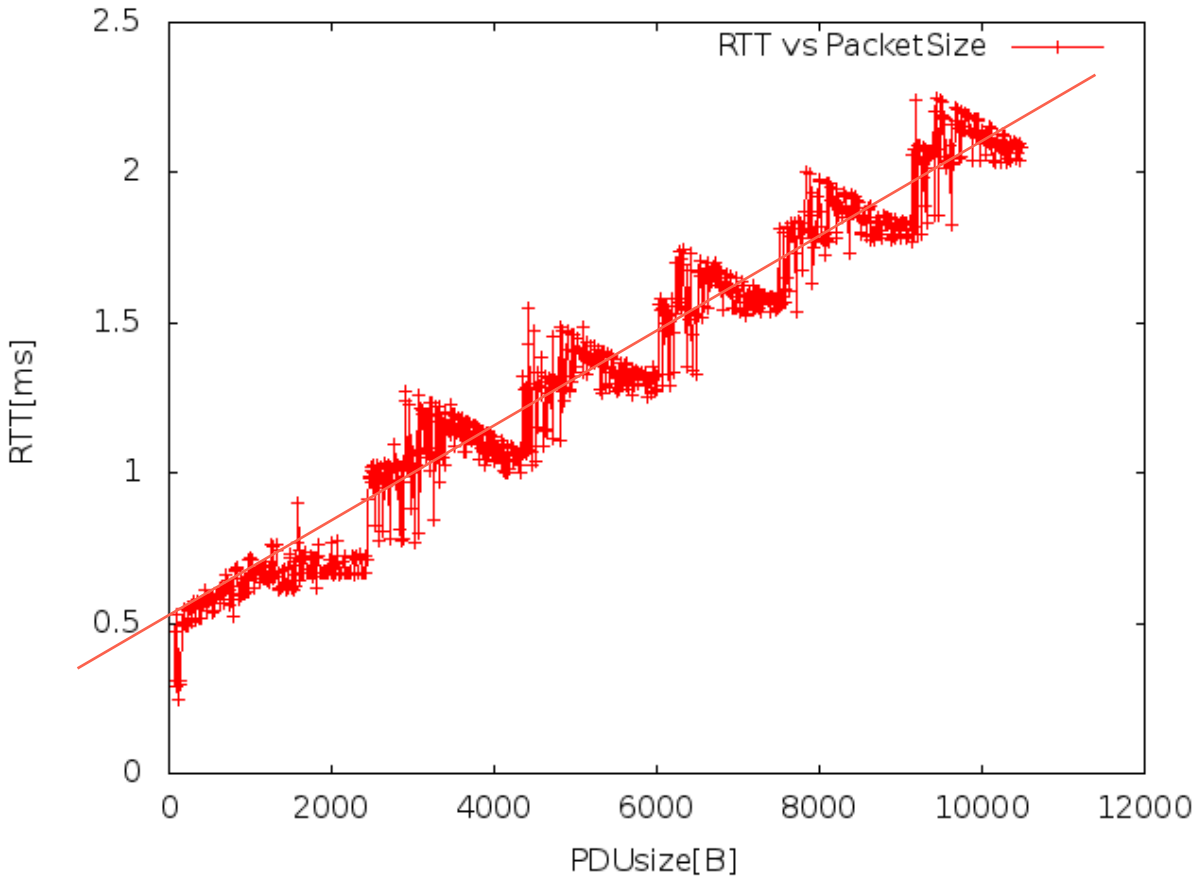
`plot "graphdata.dat" using ...` Means to plot the information contained in the required file under the following specifications: `($1+65*($1/1472+1)):2`. That is, plot on the x-axis these values: `($1+65*($1/1472+1))` and on the y-axis these values: `2`. On the x-axis we have the PDUsize at the physical layer, while on the y-axis the RTT.

This expression `($1+65*($1/1472+1))` means to take each value contained in the first column of the file "graphdata.dat", i.e. the PDU size at the application layer and add the 65B of headers that are required. The operation `$1/1472` returns the integer value of the division (e.g. 1470/1472=0, 1475/1472=1, 3000/1472=2… and so on). Therefore at each critical byte

(1473) a fragmentation occurs and 65 more bytes are added to the PDUsize at the physical layer.
Instead `:2` simply means to make the graph with the values contained in column 2 (min RTT) along the y-axis.

`title "RTT vs PacketSize" with linespoint` sets the title of the graph ("RTT vs PacketSize") and specifies that the points need to be linearly interpolated. Finally, this is the end result for the RTT vs PDUsize plot:

GRAPH 3



After making the plot we needed to export it as a .png file in order to be able to save it. This was done in the terminal by entering the following two commands:

```
gnuplot> set terminal png
Terminal type set to 'png'
Options are 'nocrop font "/usr/share/fonts/truetype/liberation/LiberationSans-Re
gular.ttf,12" fontscale 1.0 size 640,480 '
gnuplot> set output "rttvssizeexpression.png"
```

Afterwards the graph appeared on the desktop as a .png file and it was possible to save it.

Comments

From the above graph it is clear that the RTT increases with respect to the PDU size somewhat linearly, as predicted in the first part of this report. However, it is important to highlight some concepts.

First of all it is possible to notice that the graph presents certain regions in which the RTT actually seems to decrease as the PDU size increases. This does not happen in reality, but the graph has these features due to a quantization error in the ping command.

?!?!

A second error which is present in the graph is the position of the jumps. According to the analysis previously made these should occur around multiples of 1500B, but instead the graph shows them happening at about 2200B, 4100B, 6000B, 7900B, and so on… This inaccuracy is due to the clock in the computer's CPU, which was not fast enough to generate one PDU every with increasing 0.01s as specified in the ping command.

NO!

Finally, GRAPH 3 does not show any region in which the RTT appears to be constant. This is simply due to the scale. In fact, recall that the RTT is constant only when a new packet is created because of fragmentation and padding is executed on the new PDU. However padding is done only on 64B, which on scale of thousands does not really appear.

Plotting Capacity vs PDU size

The procedure with which this graph was made is the same for the previous one. The differences simply lie in labelling the axis and using the data available in "graphdata.dat" to make the graph:

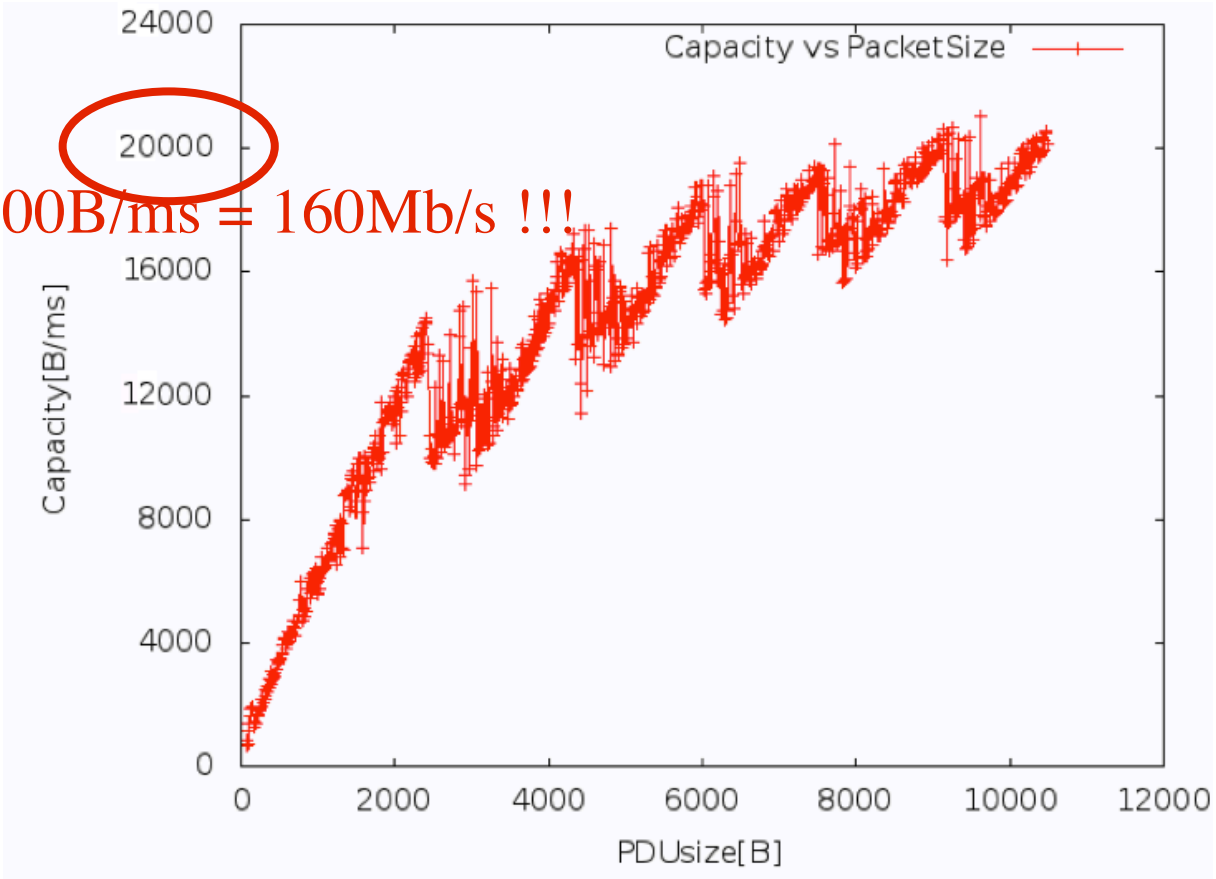
```
gnuplot> set xlabel "PDUsize[B]"
gnuplot> set ylabel "Capacity[B/ms]"
gnuplot> plot "graphdata.dat" using ($1+65*($1/1472+1)):(4*($1+65*($1/1472+1)))/
($2) title "Capacity vs PacketSize" with linespoint
```

Here we are plotting Capacity vs. Size. Therefore `($1+65*($1/1472+1))` refers to the PDUsize at the physical layer while `(4*($1+65*($1/1472+1)))/($2)` refers to the Capacity[B/s] (4D/RTT).

?!?!

The obtained graph is the following:

GRAPH 4



Just as before, this graph was exported and saved by entering these two commands in the terminal:

```
gnuplot> set terminal png
Terminal type set to 'png'
Options are 'nocrop font "/usr/share/fonts/truetype/liberation/LiberationSans-Re
gular.ttf,12" fontscale 1.0 size 640,480 '
gnuplot> set output "Capacity vs PDUsize.png"
```

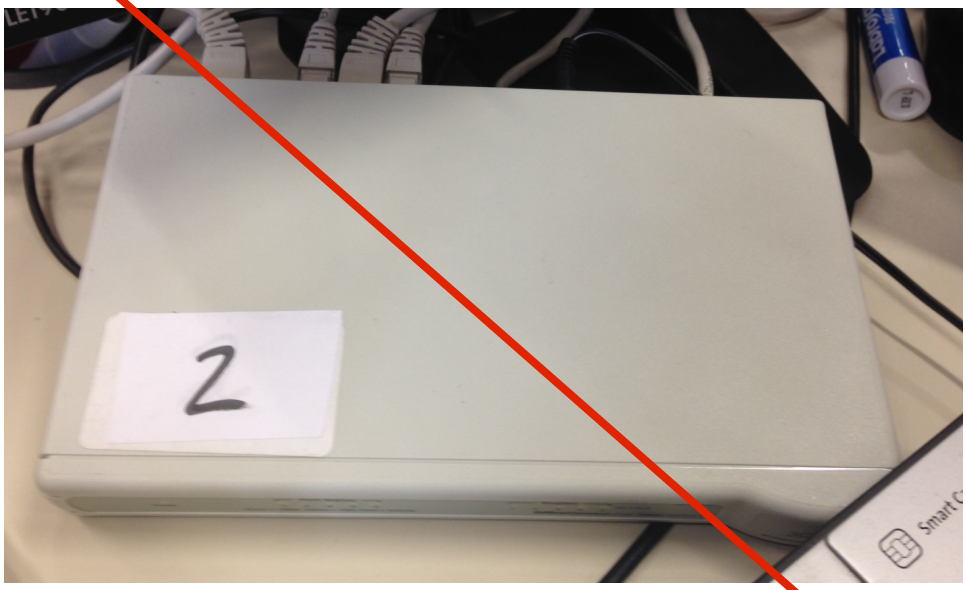
Comments

GRAPH 4 is very consistent with the predictions made. In fact the overall capacity is seen to increase at a decreasing rate as the PDU size at the physical layer increases. Moreover, whenever a packet is fragmented the capacity drops suddenly and then rises again. Although fragmentation occurs at 1500B (max Ethernet payload size), so we would expect the discontinuities to be at multiples of 1500B, the jumps in the above graph are present at about 2200B, 4100B, 6000B, 7900B… The cause for this error is the lack of necessary power from the CPU's clock. Moreover, these values correspond to the PDU sizes at which the discontinuities occur in the RTT vs PDUsize (i.e. GRAPH 3), suggesting that the data is consistent.

TABLE OF CONTENTS

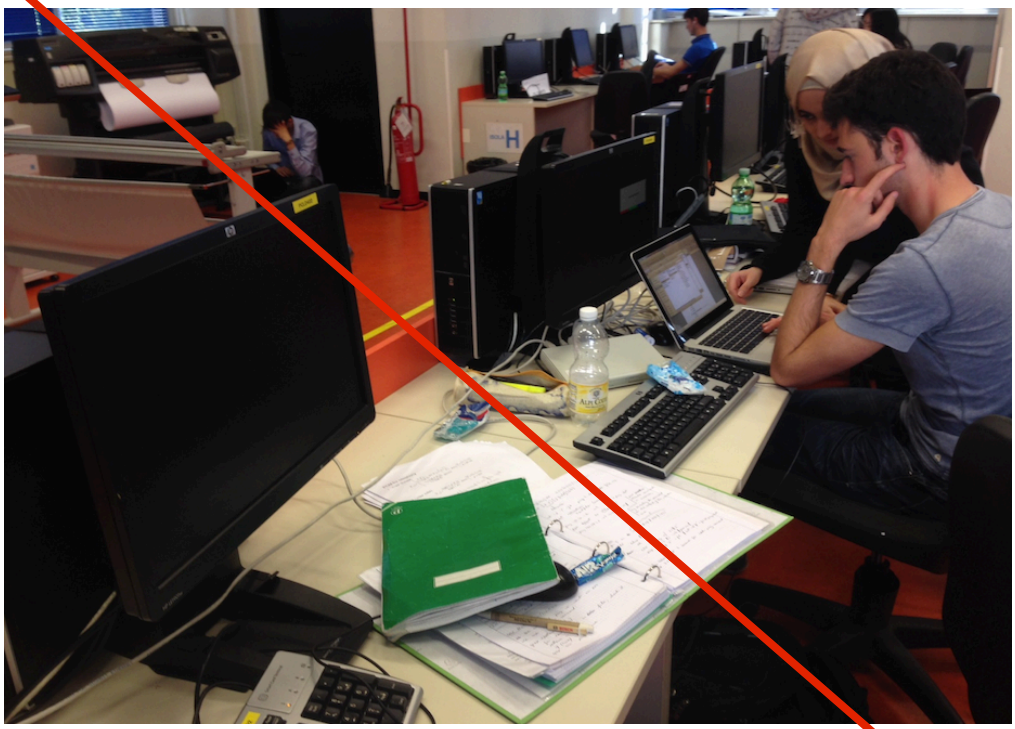
INSTRUMENTS USED

Switch



Brand: Office Connect
Type: Dual Speed Switch 5

3 Computers



Brand: HP

Type: HP LE1901w

3 Cables **Power Supply**

GETTING STARTED

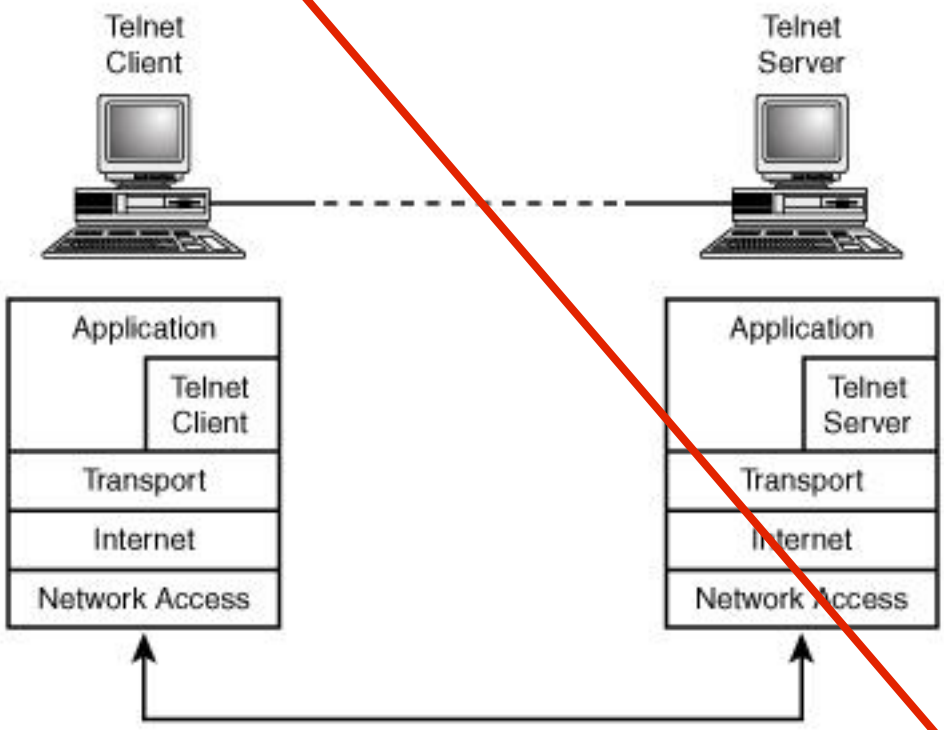
- **A LITTLE BIT OF THEORY**

TCP stands for Transmission Control Protocol. It serves the application layer, and it is being served by the underlying IP layer (layer 3). Due to some unpredictable behaviour in the network layer, IP packets can get lost, duplicated, or arrive out of sequence. TCP is a reliable service in the sense that it detects these errors, it corrects them, it asks for retransmission of lost packets, and guarantees in sequence delivery, being TCP connection oriented. Moreover, TCP provides multiplexing and demultiplexing over the ports in the same host, in order to send or receive on the same stream of bytes data corresponding to different applications. TCP also does flow and congestion control.

TCP	UDP
Reliable	Unreliable
Connection-oriented	Connectionless
Segment retransmission and flow control through windowing	No windowing or retransmission
Segment sequencing	No sequencing
Acknowledge segments	No acknowledgement

UDP stands for User Datagram Protocol. Differently from TCP it provides a connectionless, unreliable service. It sends and receives datagrams (packets) over the network, and does not have neither congestion nor flow control. Similarly as TCP, UDP provides multiplexing and demultiplexing.

The **TELNET** program is the user interface to the TELNET protocol.
Telnet is a network protocol used on the Internet or local area networks to provide a bidirectional interactive text-oriented communication facility using a virtual terminal connection. User data is interspersed in-band with Telnet control information in an 8-bit byte oriented data connection over the Transmission Control Protocol(TCP).
It enters command mode, indicated by its prompt (telnet>). In this mode, it accepts and executes its commands . If it is invoked with arguments, it performs an open command with those arguments. To force telnet application running locally not to send character to the server but to interpret them as commands is enough to enter into the telnet command mode and press '^]?+RETURN. Finally to exit the application it's enough to use quit.



The **Echo Protocol** is a service in the Internet Protocol Suite defined in RFC 862. It was originally proposed for testing and measurement of round-trip times in IP networks. A host may connect to a server that supports the Echo Protocol using the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) on the port number 7. The server sends back an identical copy of the data it received.

The **Character Generator Protocol** (CHARGEN) is a service of the Internet Protocol Suite defined in RFC 864 in 1983 by Jon Postel. It is intended for testing, debugging, and measurement purposes. The protocol is rarely used, as its design flaws allow ready misuse. A host may connect to a server that supports the Character Generator Protocol on either Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) port number 19. Upon opening a TCP connection, the server starts sending arbitrary characters to the connecting host and continues until the host closes the connection. In the UDP implementation of the protocol, the server sends a UDP datagram containing a random

number (between 0 and 512) of characters every time it receives a datagram from the connecting host. Any data received by the server is discarded.

GNUPLOT is a command-line program that can generate two- and three-dimensional plots of functions, data, and data fits. It is frequently used for publication-quality graphics as well as education. The program runs on all major computers and operating systems (GNU/Linux, Unix, Microsoft Windows, Mac OS X, and others). The easiest way to obtain a graph is to save a file trough Wireshark from File → export to select the proper subset of packets you need. After that it's possible to use the command grep to select which line you are interested to and then cut to obtain a column of data you want to save them in a final file.

- **INTRODUCTION**

The goal of this laboratory is to plot the evolution versus time of

- The client sequence number;
- The client acknowledge number;
- The server sequence number;
- The server acknowledge number;
- The size of the packets sent by the client;
- The size of the packets sent by the server;
- The receiver window size advertised by the client;
- The receiver window size advertised by the server;

LABORATORY DESCRIPTION

- **Netstat, gedit / etc/inetd.conf, ethtool –K eth0**

We first use the netstat command which is used to display very detailed information about how the computer is communicating with other computers or network devices. Specifically, the netstat command can show details about individual network connections, overall and protocol-specific networking statistics, and much more, all of which could help troubleshoot certain kinds of networking issues.

netstat –t -l -n →numbers

netstat –t -l →names

Our aim is to have applications at the application layer able to listen for possible requests from client.

- **Configuration of the server and the client**

We have to write on the terminal:

gedit /etc/inetd.conf

This way a file opens and we enable the services echo and chargen by deleting in front of them the #. The computer on which we do this is going to be the client.

Moreover is better to disable some advanced functionalities. To check which one to disable we write *ethtool –K eth0*

And to disable any offloading capability we write:

ethtool –k eth0 rx off
ethtool –k eth0 tx off
ethtool –k eth0 gso off
ethtool –k eth0 gro off

- **telnet**

Now we use telnet to contact the application we are interested in to connect to one of the applications running on another host.
First we need to open Wireshark maintaining the terminal available so we write:

Wireshark & → and we start capturing packets

And after that:

telnet 172.16.1.2 chargen

The user connects to the host using a telnet client. The user receives a stream of bytes. Although the specific format of the output is not prescribed by RFC 864, the recommended pattern (and a de facto standard) is shifted lines of 72 ASCII characters repeating.

A typical CHARGEN service session looks like this:

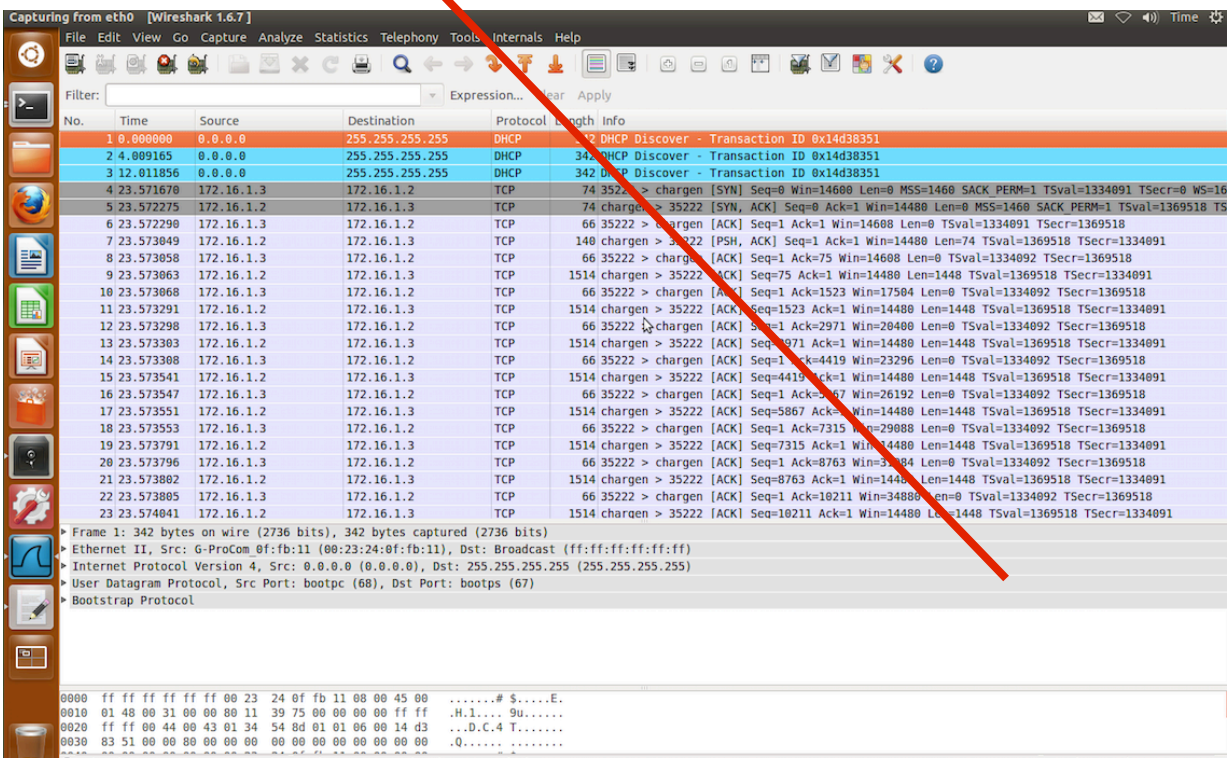
We maximize the window then we minimize it for 10 seconds and finally we maximize it again. After other 10 seconds we enter the telnet command mode by pressing:

'^]'+return

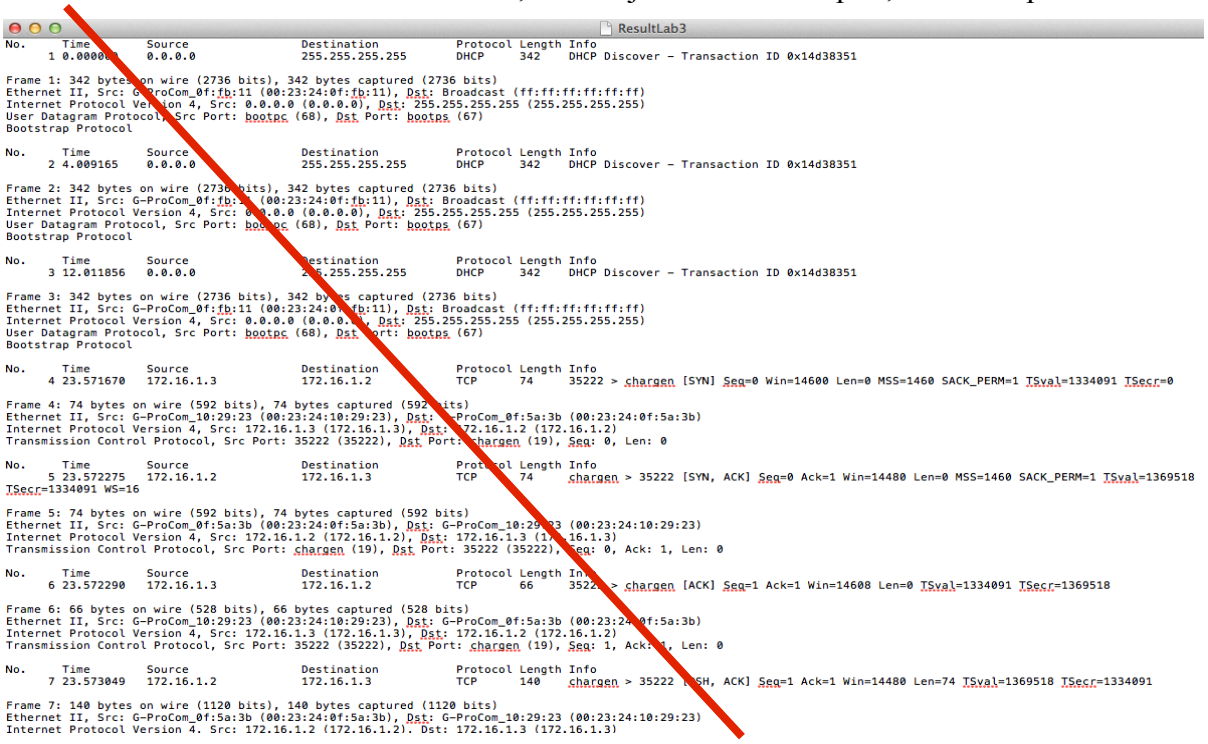
Then we write on the terminal:

quit → to exit

What we get in Wireshark is a list of TCP packets where is possible to see in grey the connection phase ad in violet colour the transmission data phase. From this picture is not possible to see the connection tear down phase.

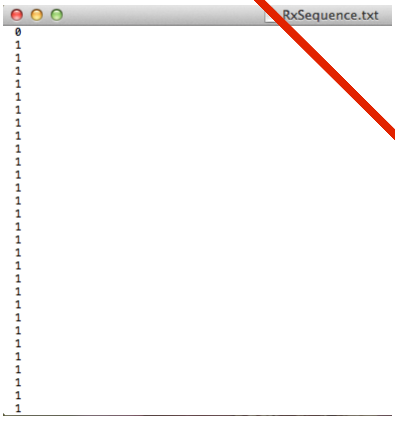


From Wireshark, using the menu, we click on File → export and this way we can save a file that we call ResultLab3 that can be seen, even if just for the first part, from this picture:



- **make the files**

At this point we should create a file that we decided to call “Lab3bashProgs” where we try to find the right way to obtain data we need and save them on other files.



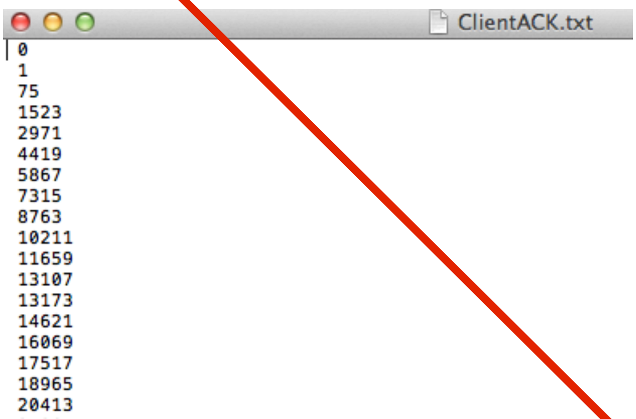
```
RxSequence.txt
0
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
```

1 grep 'Dst Port: chargen (19)' ResultLab3 | cut –d ',' –f 4 | cut –d ':' –f 2 > RxSequence.txt

This is the line written on the terminal to get the column of values which represent the client sequence number. We notice that they are all 1 except the first one which is the one representing the hand-shake opening connection and is just one zero because this is only the client column. Furthermore we should remember that the sequence numbers on wireshark are relative. This is just because it's easier for the user to read. To check which one is the absolute sequence number it' s possible to see it into the section TCP in Wireshark.

2 grep 'Dst Port: chargen (19)' ResultLab3 | cut –d ',' –f 5 | cut –d ':' –f 2 > CientAck.txt

As expected we get a column of increasing acknowledgments and how much is increasing it depends on the length of the packet.

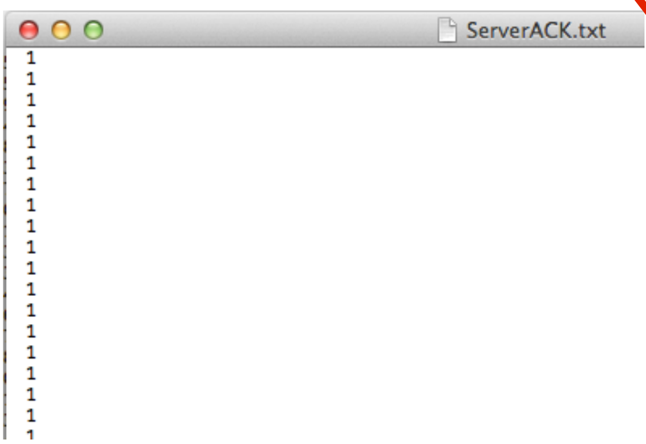


```
ClientACK.txt
0
1
75
1523
2971
4419
5867
7315
8763
10211
11659
13107
13173
14621
16069
17517
18965
20413
```

```
ServerSequence.txt
0
1
75
1523
2971
4419
5867
7315
8763
10211
11659
13107
13173
14621
16069
17517
18965
20413
```

3 grep 'Src Port: chargen (19)' ResultLab3 | cut –d ',' –f 4 | cut –d ':' –f 2 > ServerSequence.txt

It gives a column equal to the one into the file ClientAck.txt. We expected it in fact on wireshark for example we can see that the client first send an ack saying that he received until 74 so he is ready to receive from 75 and the server answer by sending the packet with sequence number = 75.

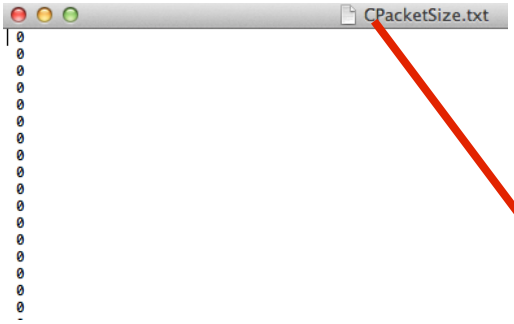


```
ServerACK.txt
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
```

4 grep 'Src Port: chargen (19)' ResultLab3 | cut –d ',' –f 5 | cut –d ':' –f 2 > ServerAck.txt

The resulting column here is exactly the same as the sequence numbers of the client. The first zero is part of the hand-shaking again and all the others prove that client is not sending any packet to the

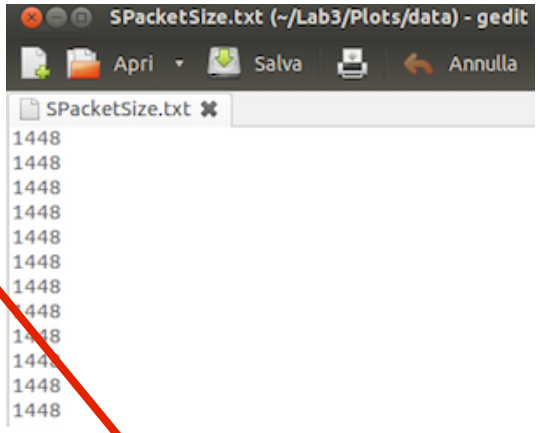
server.



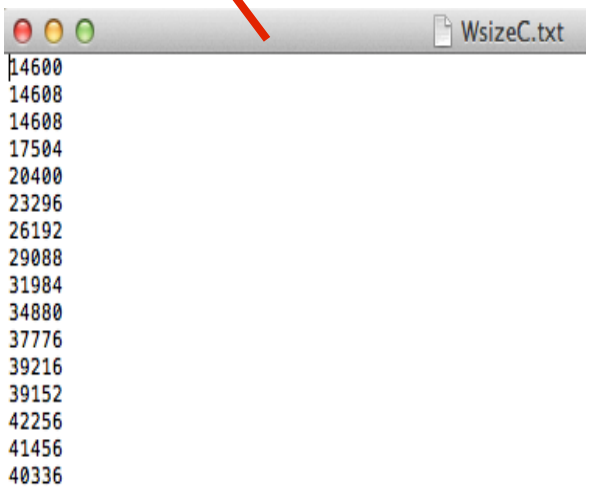
5 grep 'Dst Port: chargen (19)' ResultLab3 | cut –d ',' –f 6 | cut –d ':' –f 2 > CPacketSize.txt

As expected the client is not sending packets so the column is a list of zeros.

6 grep 'Src Port: chargen (19)' ResultLab3 | cut –d ',' –f 6 | cut –d ':' –f 2 > SPacketSize.txt



The Server Packet size is the Default Maximum Segment Size MSS =1448 bytes.



The next two lines of code written in the terminal are made to obtain the window size of the client and the server. The grap is studied to get first all the packets with source 172.16.1.3 and as destination 172.16.1.2 and viceversa in line number 8.

7 grep '172.16.1.3 172.16.1.2' ResultLab3 | tr –s ' ' | cut –d 'W' –f 2 | cut –d '=' –f 2 | cut –d ' ' –f 1 > WsizeC.txt

In this line we add a new command to collect all the spaces in one. The client window size starts with higher values and then oscillates around a certain value lower than 1448 so lower than the default maximum segment size.

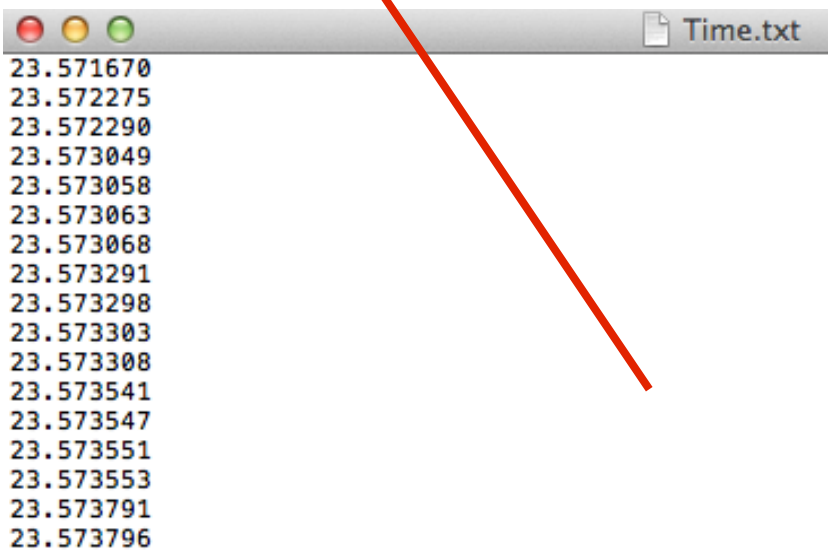
8 grep '172.16.1.2 172.16.1.3' ResultLab3 | tr –s ' ' | cut –d 'W' –f 2 | cut –d '=' –f 2 | cut –d ' ' –f 1 > WsizeS.txt

The result here is a server window size which never change because it doesn't need to since client is not sending any packet to the server.

We can see that the server advertises a default window size of 14480 bytes - which is 10xmss with mss of 1460. To 14600 is added the same tcp amount of overhead.

The last step is needed to obtain the time which is going to represent the x axis of each graph.

9 grep 'Seq=' ResultLab3 | tr –s ' '| cut –d ' ' –f 3 > Time.txt



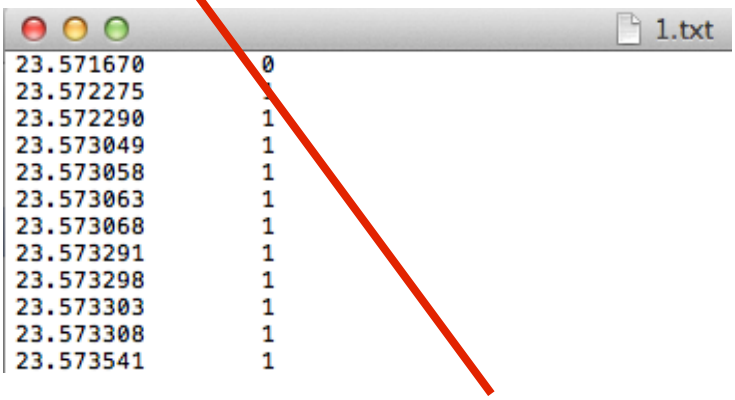
Using the command *past <file1> >file2* we save on different files each of the columns obtained before in parallel with the one of the time so that we can easily make plots using Gnuplot.

- **Gnuplot**

The following image shows the commands needed to make a plot.

```
set xlabel "time"
set ylabel "client sequence number"
set title "Group 2"
plot 2 "1.txt" using 1:2 title "Client Sequence Number" with linespoints
```

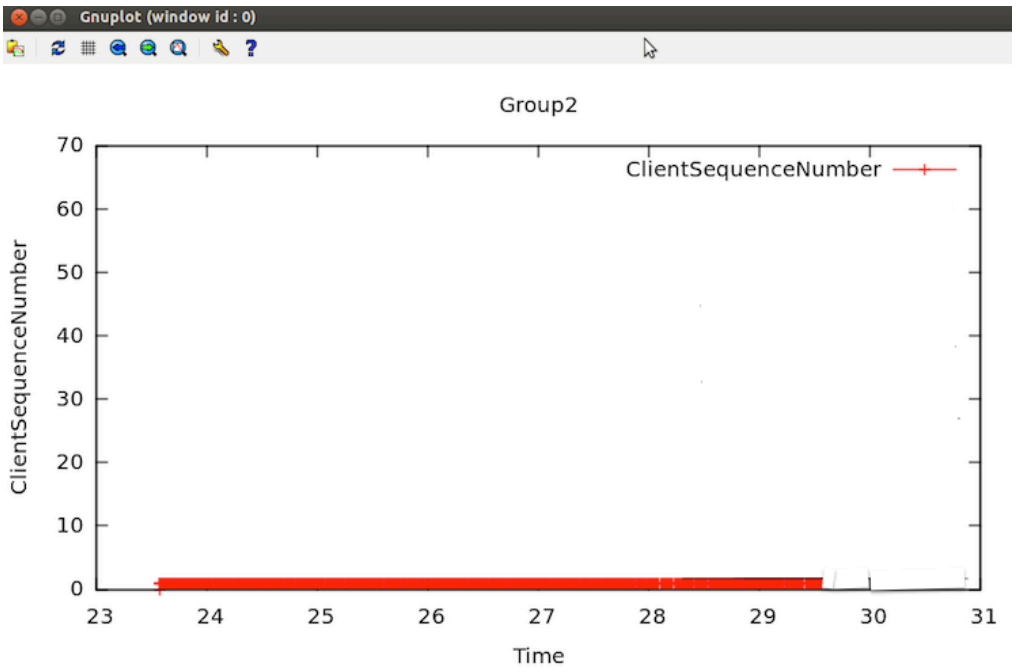
Where 1.txt is:



All the other scripts are similar to this one. We just change the y label, the file we get the data from and the title.

- **Graphs**

The graph of the **client sequence number** is flat as explained before:

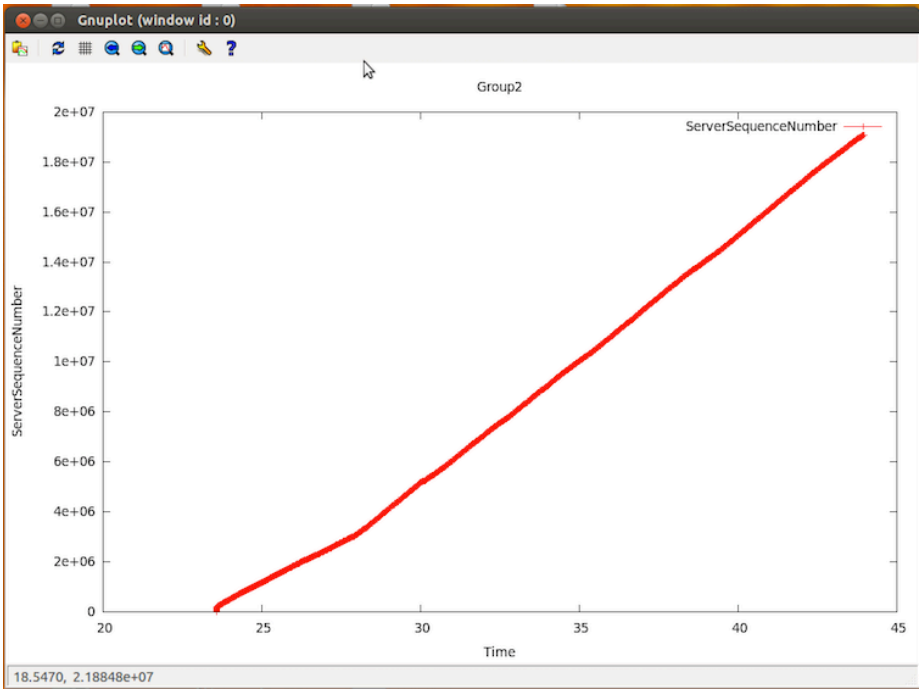


The graph of the **server sequence number** is strictly increasing. The small fluctuation at around 28 seconds is due to the maximization of the window (i.e. more data are sent). The

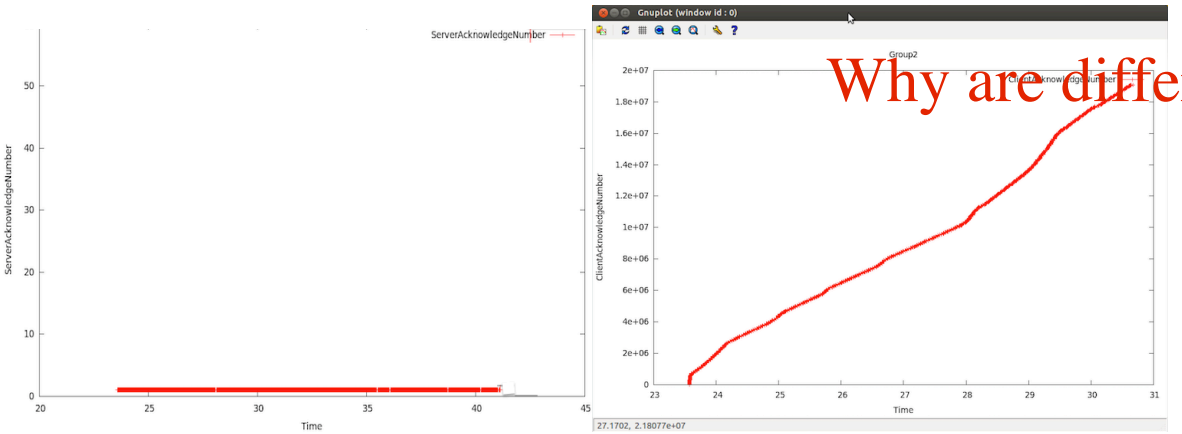
Which window???

graph could also be a little bit different if we enter in the command mode by pressing the escape sequence '^]'+RETURN and after some seconds we return to the previous state again. In that case no data are sent so the graph would have been flat until we reactivated the Chargen service. Using Wireshark it is possible to see some packets where the server is asking to the client to talk (keep-alive) . During the flat period the server would impose zero window, in order to prevent the client from sending more data. After a while the client is ready to talk again (and you can read window updated). An interesting thing is to notice that the client sends spontaneously the message to say that he is ready to receive. So why the server continue to send questions? To keep the server alive in case the message "window update" get lost.

NO



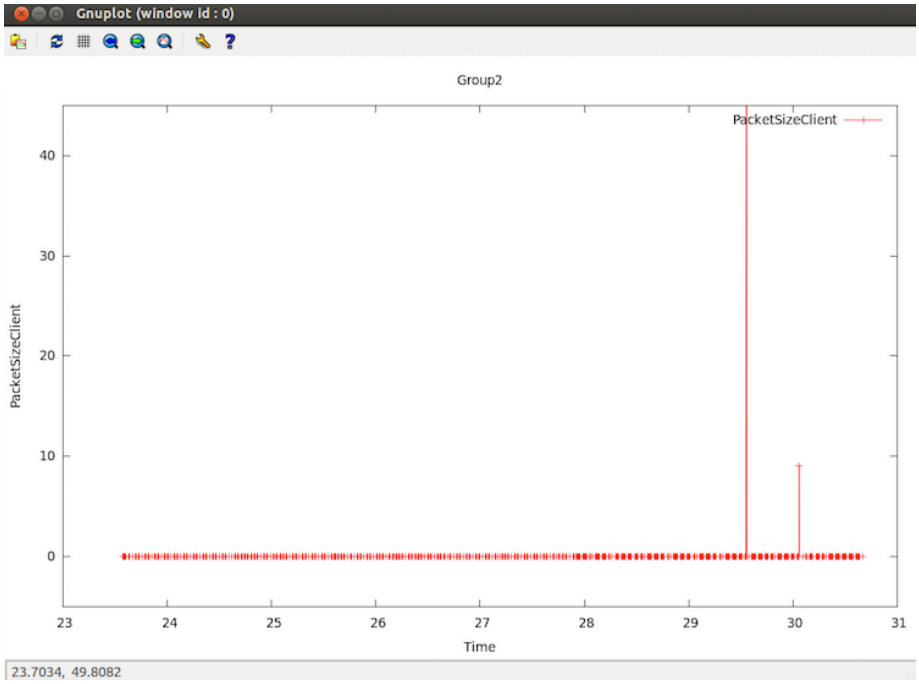
For the **acknowledgments** the two graphics will be equal to the first two but inverted, that is the acknowledgments of the client are equal to the sequence numbers of the server and viceversa.



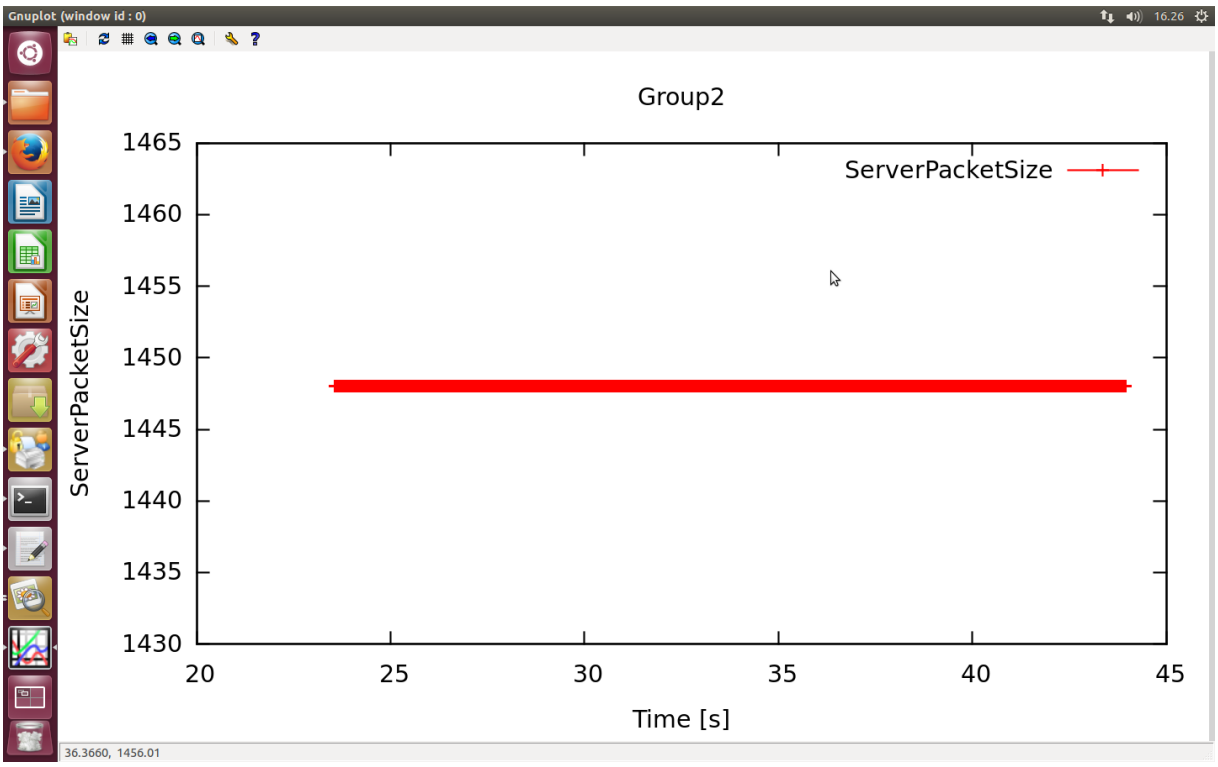
Why are different??

The **packet size of the client** is zero because it doesn't send any data to the server so, as a result, the graph is flat. The two vertical lines represent the fact that in those two moments we pressed the keyboard, that is we communicate with Telnet pressing the keys. For example we

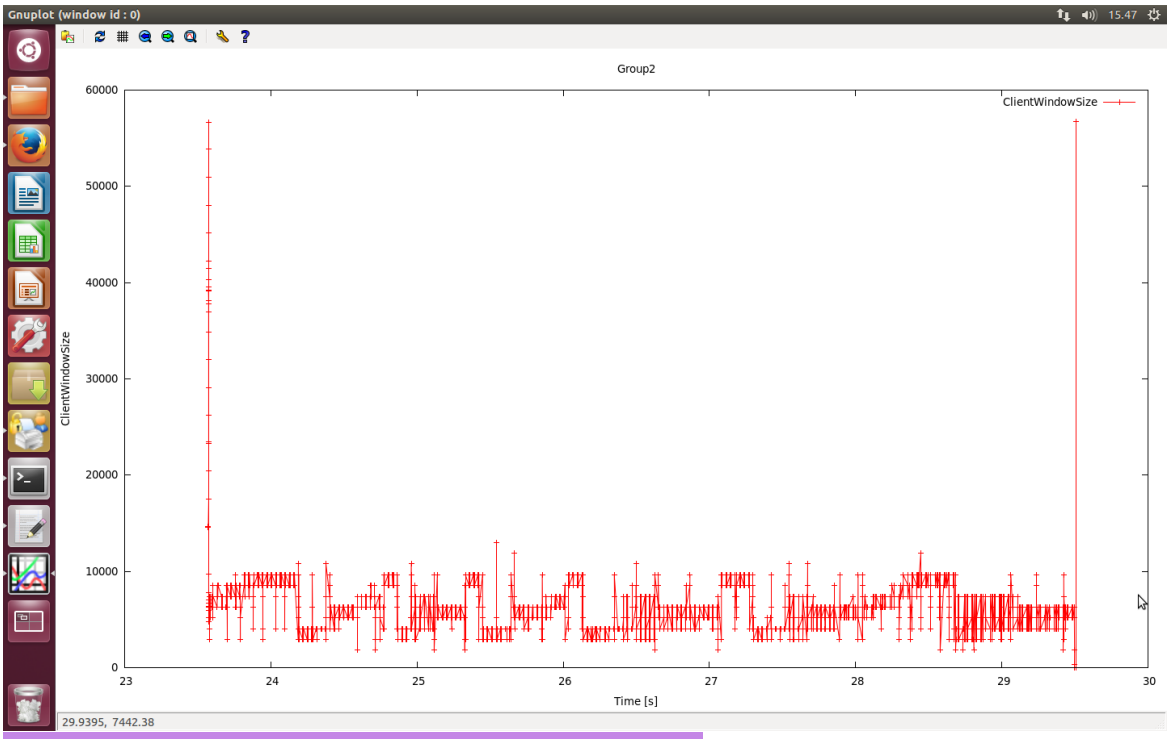
enter in the command mode by pressing on the keyboard '^]'+RETURN. So if we press something on the keyboard we communicate with Telnet as shown in the following graph:



The **packet size of the server** is flat and is the maximum size 1448 bytes. In fact each TCP packet can carry 1448 bytes of useful data, which is 1500-20(IP header)-32(TCP header with timestamp option).



The **receiver window size of the client**, as expected, starts with a value really similar to the server's one and then changes according to the flow and to the risk



of congestions. During a short period of time it increases a lot then it tends to oscillate around values a little bit slower than the one of the server. If I pressed '^]'+RETURN and after a while RETURN again to restart the flux of data, I would have seen on the plot a section with all zero values. The window size changes according to the amount of flow and the risk of congestion.

NO!!!

NO

The **receiver window size of the server** is always the maximum segment size (MSS) value because there is no risk of congestion since the server is not receiving any packet from the client:

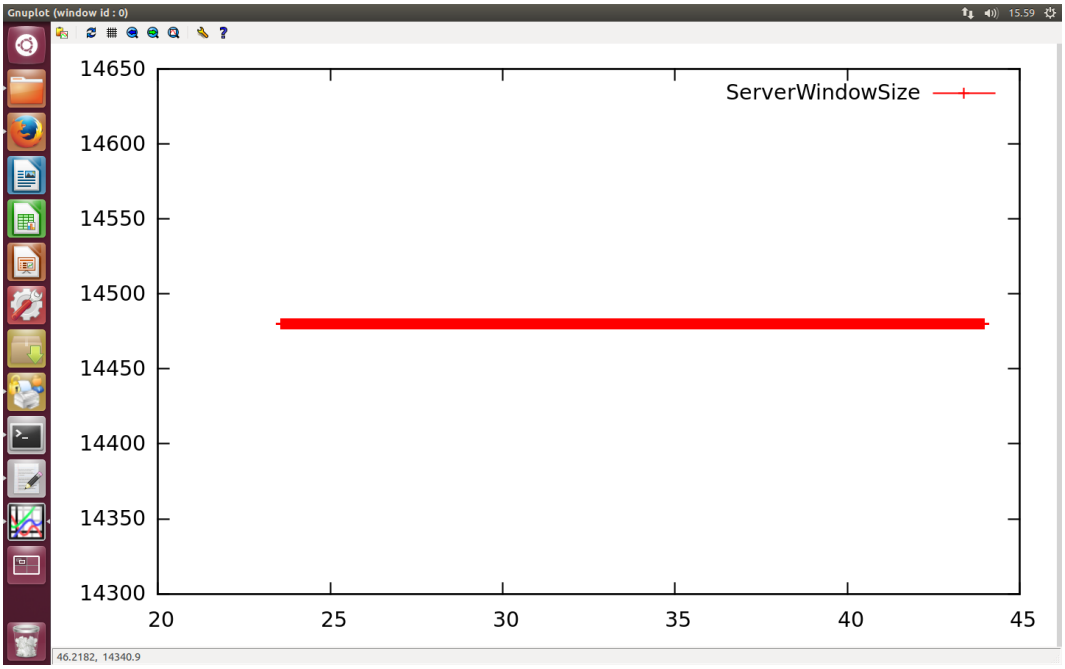


TABLE OF CONTENTS

INSTRUMENTS USED

- **Switch**

Brand:
Type:

- **3 Computers**

Brand:
Type:

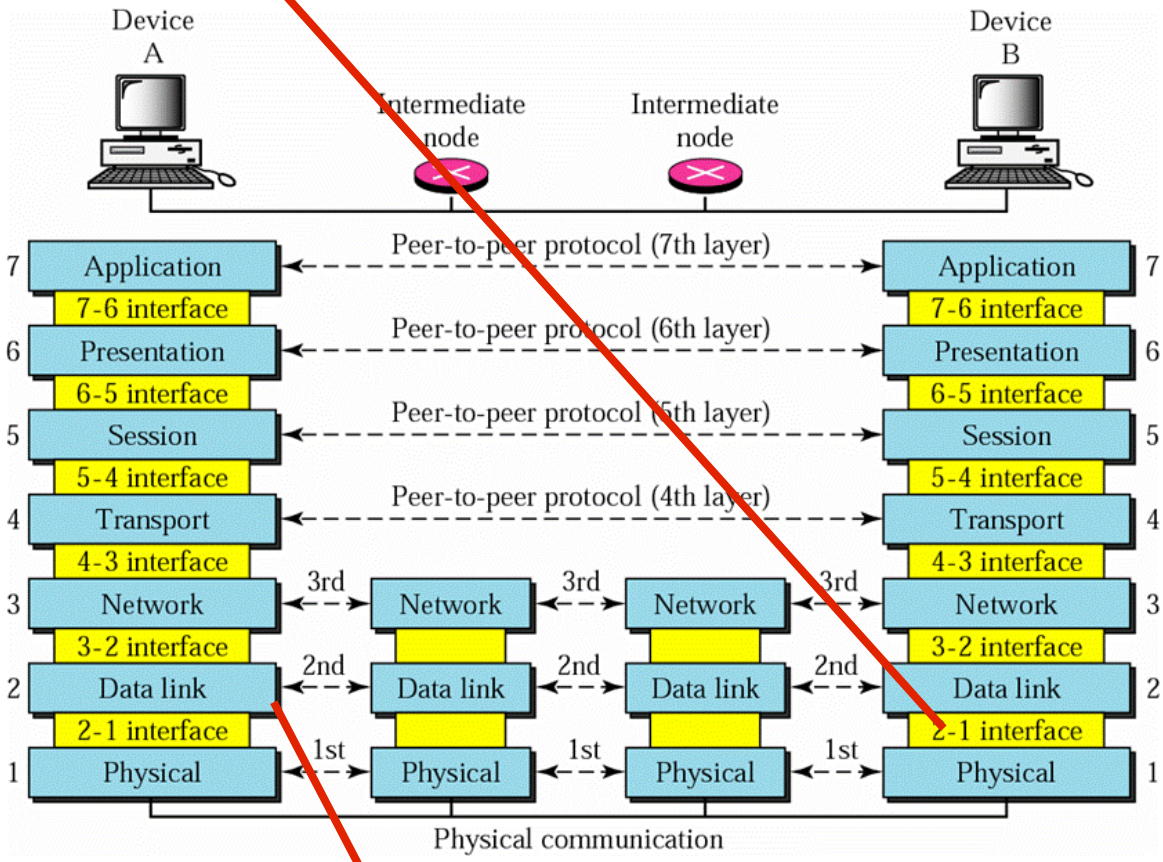
- **3 Cables**

Brand:
Type:

GETTING STARTED

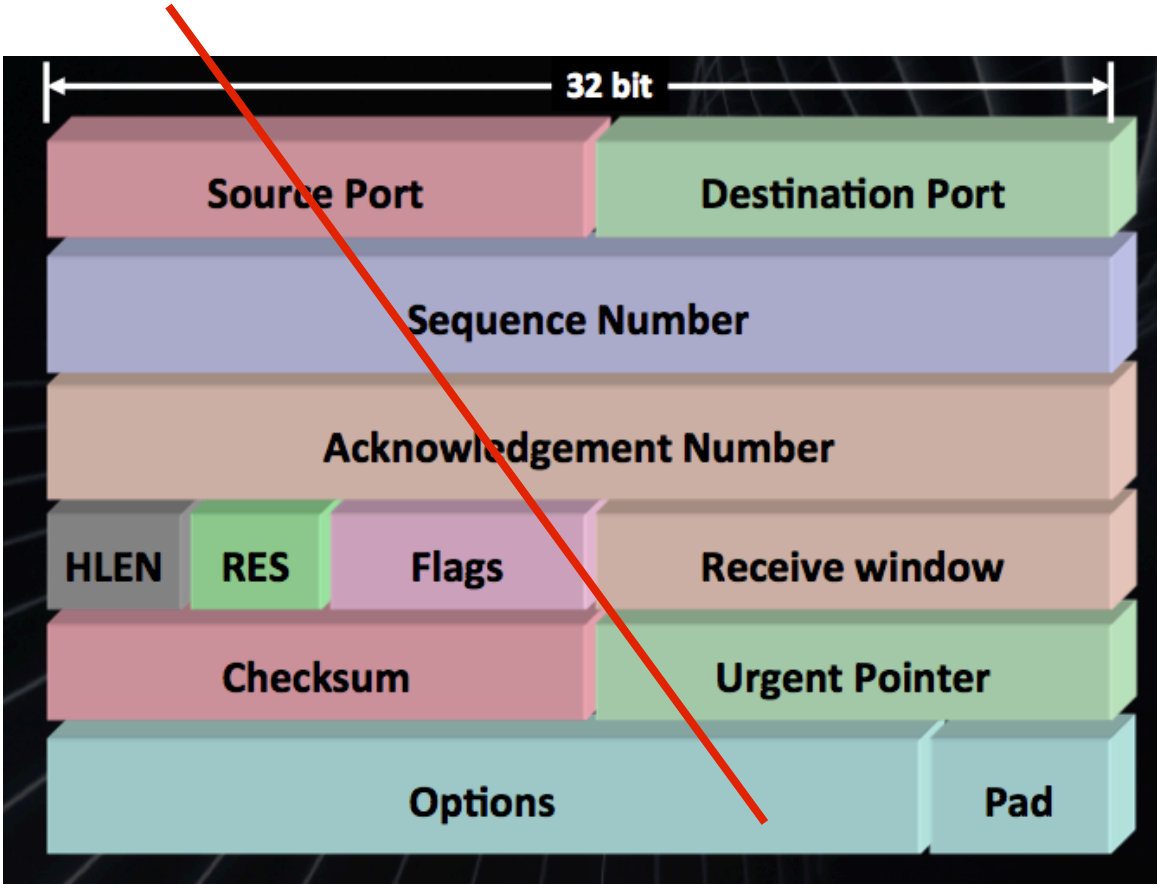
A LITTLE BIT OF THEORY

The OSI model (Open Systems Interconnection) has the purpose of characterizing the internal functions of a communication system, by subdividing it in seven different "abstraction layers". Each of these layers serves the layer above and is served by the layer below. The functionalities of each layer are implemented by one or more entities within the layer itself. Protocols enable one entity on a host to communicate with another entity on a different host on the same layer. The following image helps clarify this concept.



This investigation deals only with layer 4 communication, and in particular with the UDP and the TCP protocols.

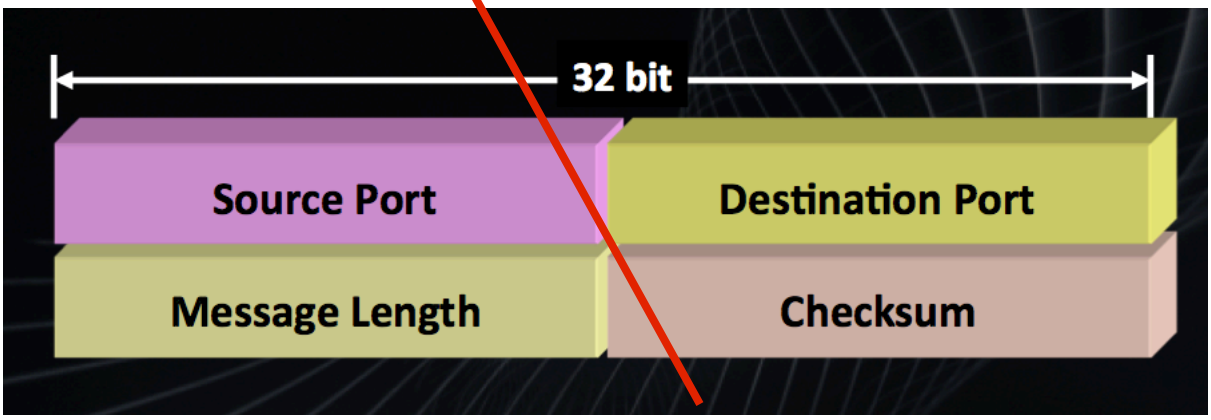
TCP stands for Transmission Control Protocol. It serves the application layer, and it is being served by the underlying IP layer (layer 3). Due to some unpredictable behaviour in the network layer, IP packets can get lost, duplicated, or arrive out of sequence. TCP is a reliable service in the sense that it detects these errors, it corrects them, it asks for retransmission of lost packets, and guarantees in sequence delivery, being TCP connection oriented. Moreover, TCP provides multiplexing and demultiplexing over the ports in the same host, in order to send or receive on the same stream of bytes data corresponding to different applications. TCP also does flow and congestion control. The TCP header format is the following:



Some clarification on the fields:

- Source Port and Destination Port are used for multiplexing/demultiplexing.
- Sequence Number- Gives the number of the sequence of data being sent. Although it can start from any number (safety reasons), we usually consider their relative value for practical reasons.
- Acknowledgement Number-Gives the number of the ACK being received. It is usually cumulative (eg. ACK=26 means all sequences up to 26 have been correctly received).
- HLEN- Gives the length of the header.
- Flags:
    - U- Urgent: This segment contains urgent data
    - SYN- Used for synchronization (opening connection)
    - FIN- Used for ending the connection
    - RST- Used for resetting (abruptly ending) the connection
    - ACK- This sequence contains an acknowledgement
- Receive Window- A number representing the available number of bytes that the receiver still has in its buffer to correctly receive incoming information.
- Checksum- Does error control
- Urgent Pointer- Tells the TCP receiver the position in the segment of the urgent data.

UDP stands for User Datagram Protocol. Differently from TCP it provides a connectionless, unreliable service. It sends and receives datagrams (packets) over the network, and does not have neither congestion nor flow control. Similarly as TCP, UDP provides multiplexing and demultiplexing. Its packet format is the following:



The Source Port, Destination Port, and Checksum fields serve the same purpose as in the TCP header. The only difference is the presence of the Message Length, which can be considered pretty much useless since the IP layer correctly delimits packets.

INTRODUCTION

The goal of this laboratory is to analyze the performance of a file transfer over the network obtained using TCP or UDP at the transport layer. Before proceeding, all the offloading capabilities of the linecard should be deactivated. This is done by entering the following pattern of commands in the terminal:

```
ethtool -K ethX [rx on|off] [tx on|off] [sg on|off] [tso on|off] [ufo on|off] [gso on|off] [gro on|off] [lro on|off]
```

Afterwards, the Ethernet interface is configured such that the speed is 10Mb/s Full Duplex, that is a link that allows communication in both directions:

```
H1: ethtool -s eth1 speed 10 duplex full autoneg on
```

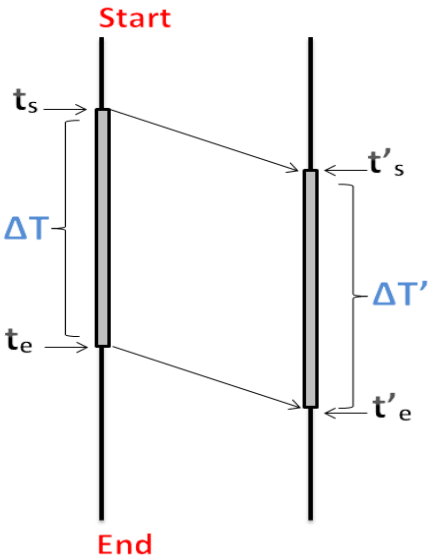
To measure the performance, we will use the nttcp command, which measures the transfer rate on a TCP/UDP multicast connection. This command measures what is known as the "Goodput"- the speed at which useful data is received by the application layer.

Goodput=(Useful data at application layer)/(Time to complete the transfer)

Consider the following image:

$$G_1 = \frac{DATA}{\Delta T}$$

$$G_2 = \frac{DATA}{\Delta T'}$$



Which is the correct goodput? To answer this we must keep in mind that the sender's clock is stopped before the last byte has been transmitted, while the receiver's clock is stopped after the last byte has been received. Therefore G_2 is the goodput that should be taken into account.

Throughout the investigation we will use clients and servers. To run nttcp on the remote host (the server), we simply write the nttcp with –i:

```
H1: nttcp -i &
```

4. Which port is the nttcp server listening to?

As usual, the & is to run the application in background without the need to open a new terminal window.
For example, suppose that H1 is configured as server and H2 as client. On H1 the nttcp application is waiting for connections from client nttcp applications. If we want to contact H1 from H2 we simply write on H2:

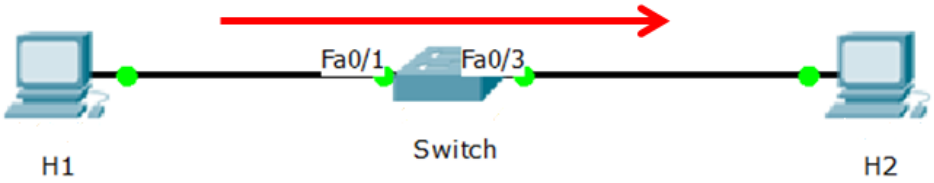
```
H2: nttcp H1
```

We will keep in mind the following useful commands:

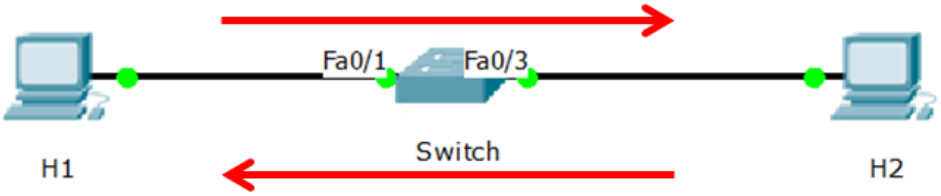
- `-r` defines the receive transfer direction
- `-t` defines the transmit transfer direction
- `-T` print a line
- `-u` Use the UDP protocol instead of the TCP (default)
- `-n` N where N is the number of buffers sent
- `-l` L where L is the length of the buffers

For the performance and measurement of the goodput the investigation consists in considering the following four different cases:

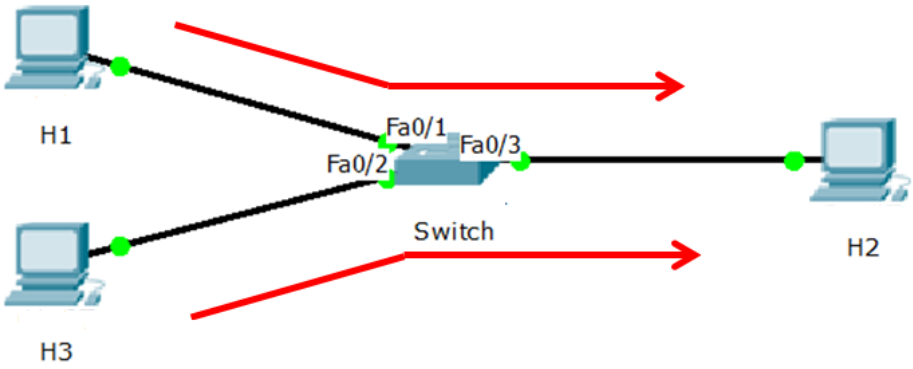
- A - Single flow – FULL DUPLEX: H1 is sending data to H2.



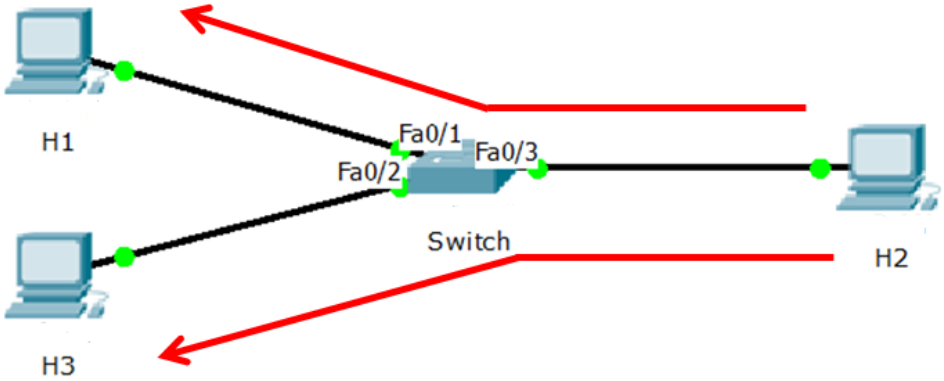
- C - Bidirectional flows – FULL DUPLEX: H1 is sending data to H2 AND H2 is sending data to H1.



- E - Two flows to the same receiver – FULL DUPLEX: H1 sends data to H2 AND H3 sends data to H2.



- G - Two flows from the same sender – FULL DUPLEX: H1 sends data to H2 AND to H3.



For each of the above cases we will consider the following three different situations:

- All flows use TCP at the transport layer
- All flows use UDP at the transport layer
- One flow uses TCP, the second uses UDP at the transport layer

What is the maximum goodput you expect when using UDP at the transport layer?
Being UDP connectionless and not reliable we expect, in both FULL and HALF DUPLEX, goodput to be the same accordingly to which part of the report we are considering.
The reason why we suppose this is going to be the result for the HALF DUPLEX too is because there are no acknowledgement packets going back to the TX.

What is the maximum goodput you expect when using TCP at the transport layer?
In this case the FULL DUPLEX is different from the HALF one because here there are acknowledgement packets going back to the TX..

After these quick predictions we will report in the next pages the results, along with additional comments.

(3)
Tables

What we see on the terminal when we generate the output on the Command Line is:

```
root@laboratorio:/home/laboratorio# nttcp -T -u -r 172.16.1.3
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608    7.01    0.02      9.5746   3355.2754    2049    292.34  102444.9
1  8388608    6.93    0.01      9.6845   5592.4053    2051    295.98  170916.7
root@laboratorio:/home/laboratorio# nttcp -T -u -r 172.16.1.3
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608   14.08    0.02      4.7668   2796.0862    2049    145.54   85371.4
1  8388608   14.02    0.01      4.7853   5592.4053    2051    146.25  170916.7
root@laboratorio:/home/laboratorio#
```

This is the example of what we see on the terminal for the case A or C of the UDP protocol. WHY UDP? Because it is the only one packet oriented. Why case A or C? Because there is just one transmitting flow and one receiving flow.

We observe first the cases where ALL FLOWS USE TCP AT THE TRANSPORT LAYER

Test	Average TCP Goodput per flow			Collision Probability		Loss at the application layer		Comment
	Prediction [Mb/s]	Observed [Mb/s]		Prediction [Mb/s]	Observed [Mb/s]	Prediction [Mb/s]	Observed [Mb/s]	
A	< 10	9.4		0	0	0	0	Comments are immediately below the table.
C	<10	Local Host	Remote Host	0	0	0	0	
		9.15	9.14					

E	<5	4.75	4.74	0	0	0	0	
G	< 5	6.17	4.76	0	0	0	0	

a. Comments on the TCP type of flow

Test A

```
H1: nttcp -T H2
```

```
(In this way we have the default length: 4Kbytes,
and number of buffers sent: 2048)
```

The situation in which there are only two hosts involved and just one of the two is sending packets is the simplest. The goodput we obtained using TCP is 9.4 Mb/s, which is slightly less than 10Mb/s, the cable capacity. We set on the terminal that the connection is full duplex (both parties can talk at once) with this command:

```
H1: ethtool -s eth1 speed 10 duplex full autoneg on
```

Using a full duplex connection, the acknowledgements sent by the receiver are not influencing the goodput speed. Also no packets are lost because of the reliability of the message TCP and there is no bit error rate since we safely assume that a cable connecting two hosts is a reliable connection. We conclude that the goodput is slightly lower than we expected is due to the fact that the headers (37B of Ethernet+20B of IP+20B of TCP) are about 5% of the total data sent and they are not considered as useful data.

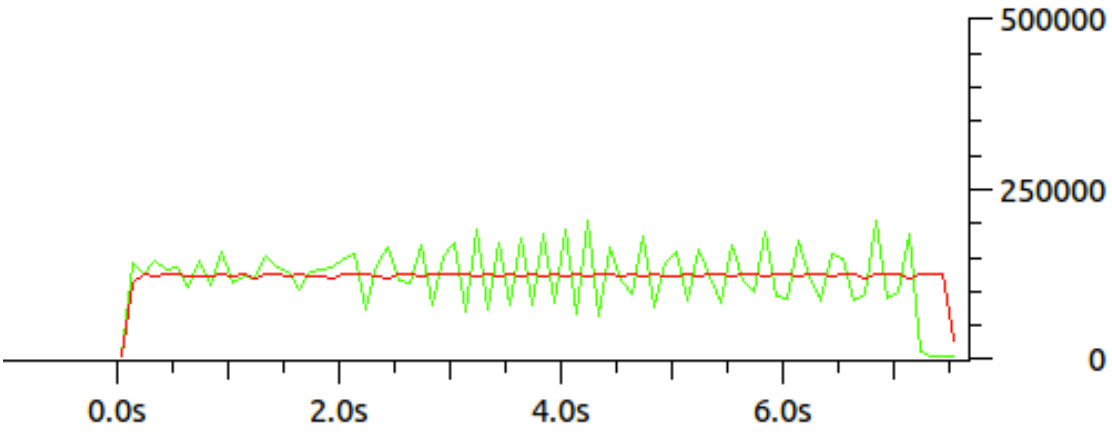
If it was HALF duplex probably the goodput would have been a little bit less than the one we got here because of the acks going back on the same cable and there would have been collisions.

Test C

```
H1: nttcp -T H2 & nttcp -T -r H2
```

Here the test involves two hosts, this time both are transmitting. We still expect to get a goodput similar to 10 Mb/s always for the same reason as before. The results we get at the receivers are respectively 9.15 Mb/s and 9.14 Mb/s. No collisions are expected because of the full duplex and no packet loss because it is a TCP flow. The goodput is slightly lower than the previous case, because now the acknowledgements use a bit of the capacity.

The HALF DUPLEX situation would have taken the double of time.



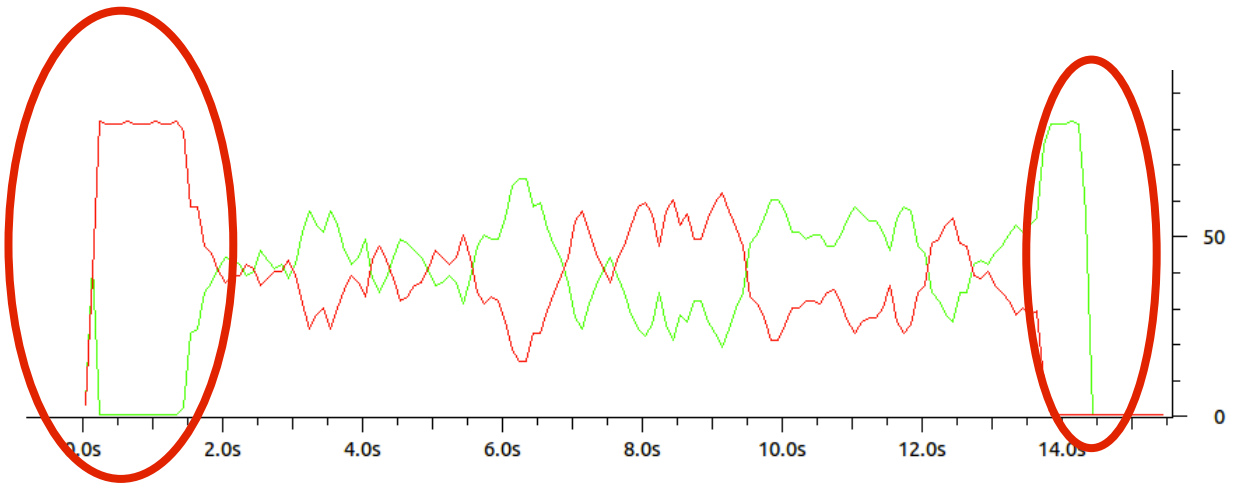
On the y-axis is reported the number of bytes. (Green-TCP, Red-TCP).

Test E

```
H1: nttcp -T H2
H3: nttcp -T H2
```

In this test with two flows (from H1 and H3) to the same receiver (H2) the situation is different from the Goodput point of view while it remains the same for the other two aspects. We expect the Goodput to be smaller than 5 Mb/s for the same reasons we pointed out in Test A. We get in fact 4.75 Mb/s and 4.74 Mb/s.

In the case of HALF DUPLEX the only difference would be the time taken by acknowledgements. In fact H1 and H2 should send data in the same direction so from this point of view is exactly the same having a full duplex or an half duplex. The only difference is that having a full duplex the acknowledgements can flow in the opposite direction without disturbing the data flow.



????

Test G

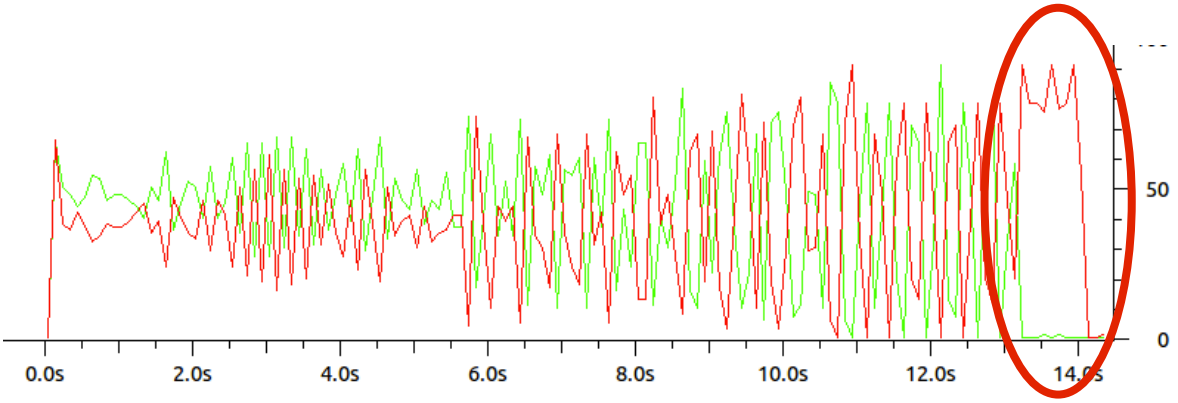
`H1: nttcp -T H2 & nttcp -T H3`

NO

The last test regarding TCP flow is about H1 sending messages to H2 and H3. As for Test E the only data which is different is the Goodput which is expected again smaller than 5 Mb/s. We get 6.17 Mb/s and 4.76 Mb/s. This is probably due to the fact that our host "preferred" one TCP flow over the other one.

With HALF DUPLEX would take a longer time just because of acknowledgements opposite flow as in case E.

This time on the y-axis we see reported the number of packets.



Then we observe the same cases with ALL FLOWS USING UDP AT THE TRANSPORT LAYER

Test	Average UDP Goodput per flow			Collision Probability		Loss at the application layer		Comment
	Prediction [Mb/s]	Observed [Mb/s]		Prediction [Mb/s]	Observed [Mb/s]	Prediction [Mb/s]	Observed [Mb/s]	
A	< 10	9.57		0	0	0	0	Comments are immediately below the table. Please check there.
C	< 10	9.5	9.54	0	0	0	0	
E	< 5	Local Host	Remote Host	0	0	Some packets lost	From 8388608 to 233472	
		0.26	0.20					
G	< 5	4.8	4.7	0	0	0	0	

b. Comments on the UDP type of flow

During all the cases as before there are no collisions because of FULL DUPLEX.

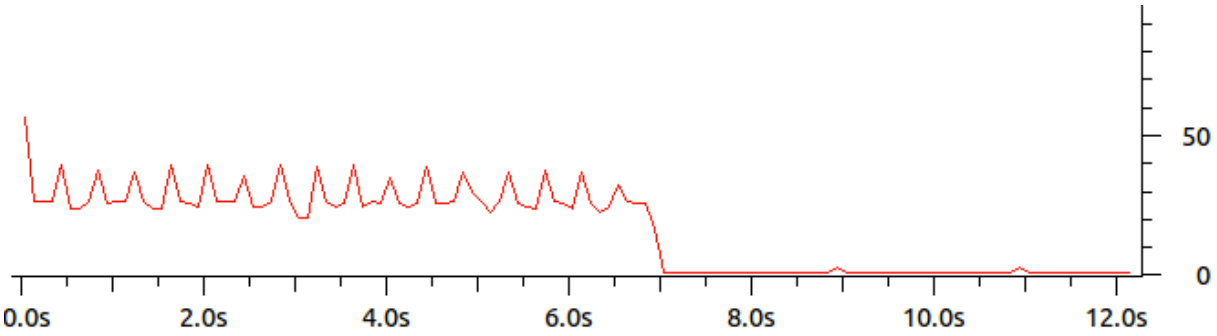
Furthermore if we use HALF DUPLEX there are no differences because in this section of the report we are treating UDP which does not require acknowledgements.

Test A

```
H1: nttcp -T -u H2
```

Now that we are sending packets using UDP in this first case where we have only flow in one direction the other direction is totally unused because there are no acknowledgements going back. In this case the goodput is slightly higher (9.5 Mb/s) than the TCP test: this is due to the fact that UDP header is only 8 Bytes.

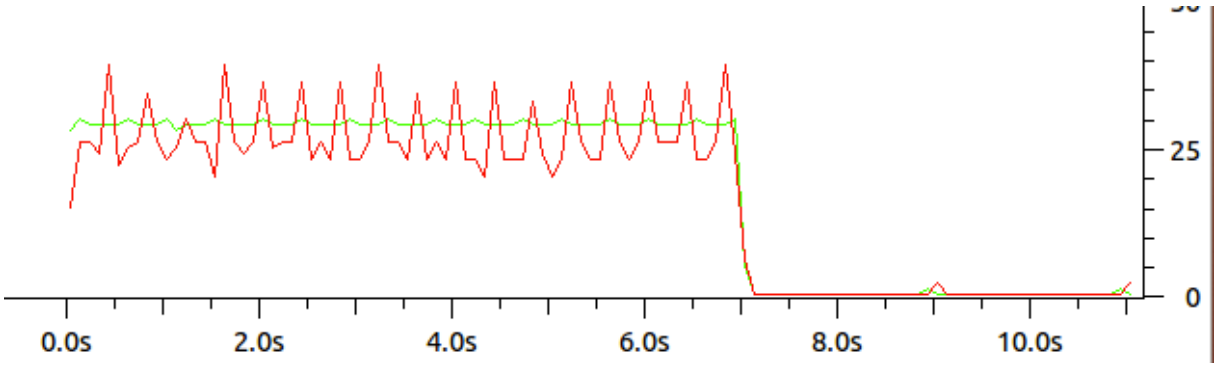
(Number of packets on y-axis).



Test C

```
H1: nttcp -T -u H2 & nttcp -T -r -u H2
```

Here we have a double flow but no acknowledgements, so with respect to the same situation in TCP the goodput is a little bit higher: 9.5/9.54 with respect to 9.15/9.14.



Test E

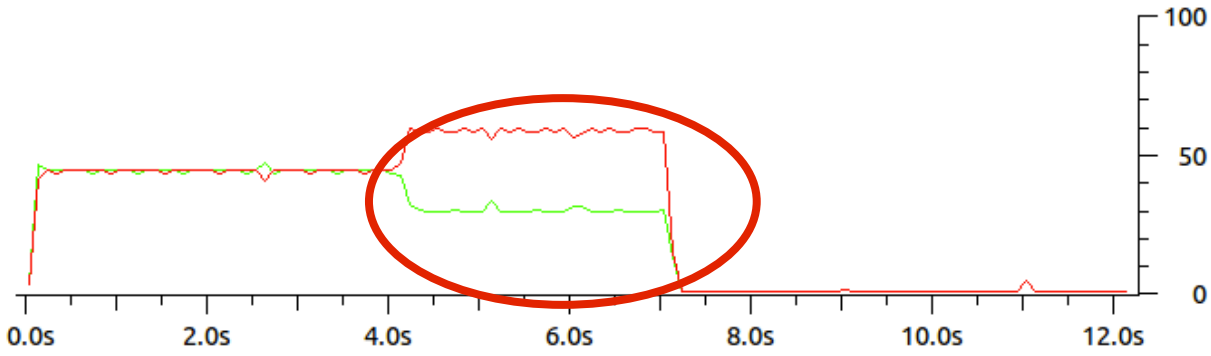
```
H1: nttcp -T -u H2
```

```
H3: nttcp -T -u H2
```

The result we got from this test is the worst one because of queuing at the switch. Being the protocol used UDP it is not guaranteed that all packets reach their destination. In this case most of the packets are lost.

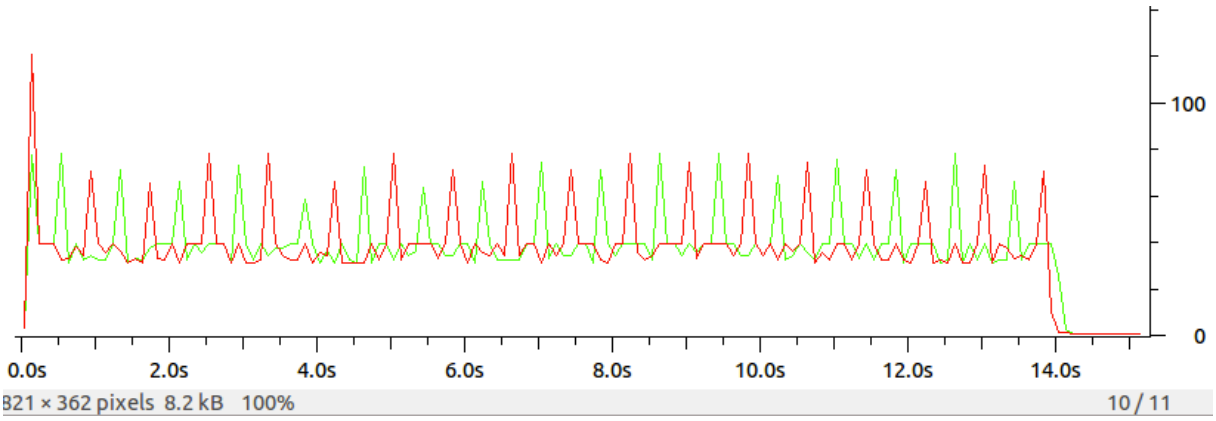
Bytes that should have reached the destination: 8388608

Bytes arrived: 2423472



Test G

```
H1: nttcp -T -u H2 & nttcp -T -u H3
```



This time there is no queuing at the switch, thus no packet get lost. This means that we were right expecting the goodput to be just a little bit less then 5Mb/s, which is the half of the available capacity. We got 4.7 and 4.8 Mb/s.

Finally we observe the same cases with MIXED FLOWS USING BOTH UDP AND TCP AT THE TRANSPORT LAYER

Test	Average TCP Goodput per flow		Average UDP Goodput		Comment
	Prediction	Observed	Prediction	Observed	
C	<10	9.08	<10	9.32	Comments are immediately below the table.
E	<5	5.0	<10	6.5	
G		7.07		4.74	

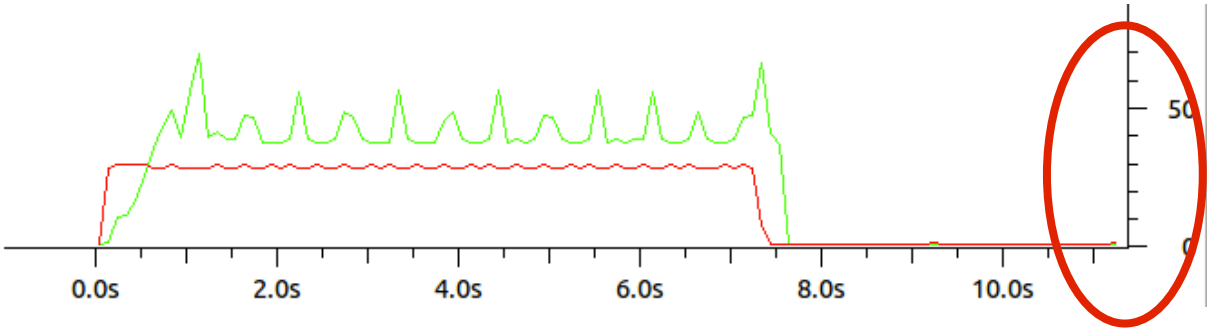
c. Comments on the hybrid TCP/UDP type of flow

Test C

```
H1: nttcp -T H2 & nttcp -T -r -u H2
```

In a FULL DUPLEX situation in the UDP direction the goodput is a little bit higher because the UDP requires a shorted header with respect to TCP. Anyway the UDP goodput is a bit lower than the UDP-UDP case for the sole reason that we have TCP acknowledgements flowing in the same direction. In the opposite direction the goodput is a little bit less because TCP header is 20 Bytes. Please keep in mind that what is shown below are the number of packets. Therefore data is consistent.

(tcp-green, udp-red).

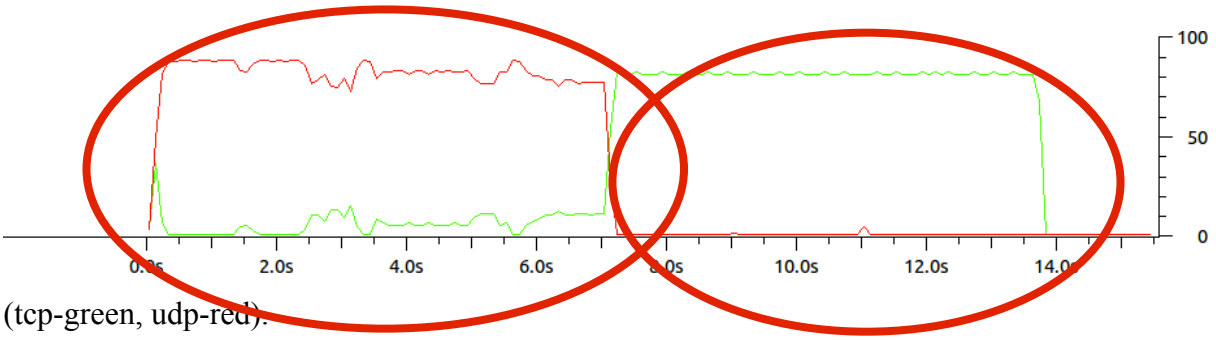


Test E

```
H1: nttcp -T H2
```

```
H3: nttcp -T -u H2
```

H3 is sending a UDP flow while H1 is sending a TCP one and they are both sending them to H2. We expect the UDP and TCP goodput around 5 Mb/s, since they have to share the channel to get to H2. Our test reaveals quite different result though. Frist to be transmitted will be the UDP packets. This is due to the fact that TCP implements flow control, that is it will slow down as soon as it will sense that the channel is full of UDP packets. For this reason the TCP goodput is around 5Mb/s, i.e. twice as much as what we had in case A. About UDP we get 6.5 Mb/s, because over 8388608 bytes just 5767168 arrived. This is probably due o queuing at the switch.



(tcp-green, udp-red).

Test G

```
H1: nttcp -T H2 & nttcp -T -u H3
```

This case turns out to be quite similar to the previous one. This last case doesn't have bytes losses in UDP flow though, reason being that there is no packet loss due to queuing, since both flows are originated by the same sender.

(tcp-green, udp-red).