Very good
Well done and with some interesting insights

30/30

NETWORKING LABORATORY I
CONNECTIVITY CHECK - ADVANCED PING

GIULIO FRANZESE 182969 - MARCO RECENTI 180644 - TOMMASO CALIFANO 180808

ACADEMIC YEAR 2013 - 2014
PROF. MELLIA MARCO

1. PING IN BROADCAST

The first part of our laboratory concern pinging broadcast. Our network address is 172.16.0.0 with netmask 255.255.255.128 so our broadcast address will be 172.16.0.127. In the rfc 1122 section 3.2.2.6

> 3.2.2.6 Echo Request/Reply: RFC-792
>
> Every host MUST implement an ICMP Echo server function that receives Echo Requests and sends corresponding Echo Replies. A host SHOULD also implement an application-layer interface for sending an Echo Request and receiving an Echo Reply, for diagnostic purposes.
>
> An ICMP Echo Request destined to an IP broadcast or IP multicast address MAY be silently discarded.

DISCUSSION:
This neutral provision results from a passionate debate between those who feel that ICMP Echo to a broadcast address provides a valuable diagnostic capability and those who feel that misuse of this feature can too easily create packet storms.

The IP source address in an ICMP Echo Reply MUST be the same as the specific-destination address (defined in Section 3.2.1.3) of the corresponding ICMP Echo Request message.

Data received in an ICMP Echo Request MUST be entirely included in the resulting Echo Reply. However, if sending the Echo Reply requires intentional fragmentation that is not implemented, the datagram MUST be truncated to maximum transmission size (see Section 3.3.3) and sent.

Echo Reply messages MUST be passed to the ICMP user interface, unless the corresponding Echo Request originated in the IP layer.

If a Record Route and/or Time Stamp option is received in an ICMP Echo Request, this option (these options) SHOULD be updated to include the current host and included in the IP header of the Echo Reply message, without "truncation". Thus, the recorded route will be for the entire round trip.

If a Source Route option is received in an ICMP Echo Request, the return route MUST be reversed and used as a Source Route option for the Echo Reply message.

It is discussed about the possibility or not to reply to ping in broadcast.This discussion is not the aim of our report and we are simply going to say that on some machine,depending on settings it may be necessary to force the reply to ping in broadcast
$/bin/echo"0"/proc/sys/net/ipv4/icmp\_echo\_ignore\_broadcast$
$gedit/etc/NetworkManager/Netowork$

$ping172.16.0.127 - c7 - b > ping\_bc.dat$

The device configured as H2 ping the broadcast, hereafter the schematic explanation of what appens
1)we generate at the applicative layer a ping of the desired lenght which is by default 56 byte
2) the icmp layer add the header(8 byte) and pass it to the IP layer
3)the IP layer read the IP destination address and ask himself if it is necessary to generate an arp request The answer is no because everyone already known the mac address for EVERYONE

4)The packet is passed to the ethernet layer and to the physical layer where the final headers are added
We have a packet that is composed by

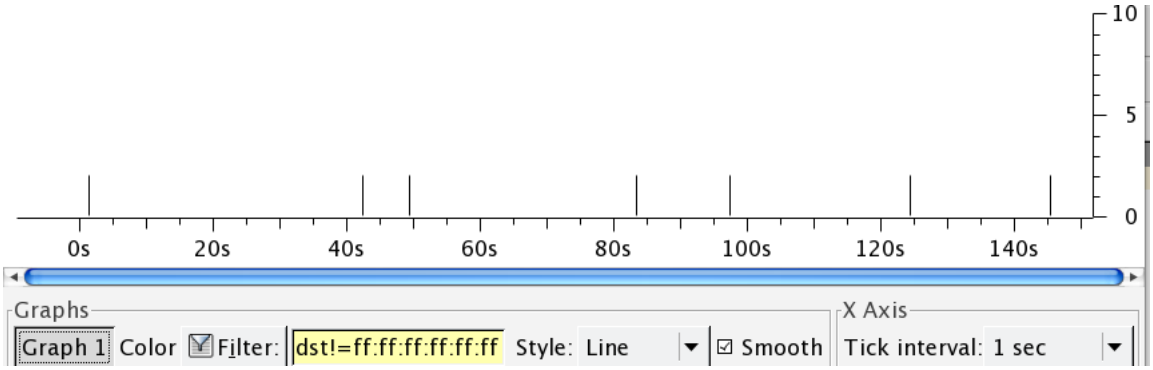
[BYTES] 56 (icmp data)+8 (icmp header)+20 (ip header)+12 (2*6 (mac address)+4 CRC(ethernet)+ 2 (ethertype)= 102

Here there is the first captured packet

```
1.727111      172.16.0.1                172.16.0.127              ICMP
    98 Echo (ping) request  id=0x113e, seq=1/256, ttl=64
```

TOTAL LENGHT 98! What appened is that it is missing the CRC that is added after the "wireshark layer" 102-4=98 Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) Something else we can notice is that the group address bit is setted to $1 \rightarrow BROADCAST$ ....1 .... = IG bit: Group address (multicast/broadcast)
What appens next is that all the hosts in the network receives an icmp echo request.
H2 and H3 have to do an arp request to reply to H1: icmp echo request was in broadcast so no arp were needed, reply are in unicast!
If we check the arp tables of the 3 hosts we see that at first they are empty.
After the arp requests H2(the generator of broadcast ping) will have knowledge about H1 and H3 mac address, H1 will add H2 to is arp table and also H3 will do the same.
When we capture the arp request from H1 and H3 we see that the total lenght of ethernet frame is 42 bytes!
The answer to this strange situation is analogue to the previous one: the capture we've done with wireshark was done after the crc was calculated and the padding were removed.
Something else we can notice is that no icmp echo reply from local loopback are captured from wireshark

After some time we see other arp request but not in broadcast: in unicast



This is not an age refresh but a security check: it is disquised on rfc 1122 sec 2.3.2.1 Unicast Poll:

```
Unicast Poll -- Actively poll the remote host by
periodically sending a point-to-point ARP Request
to it, and delete the entry if no ARP Reply is
received from N successive polls.  Again, the
timeout should be on the order of a minute, and
typically N is 2.
```

On our capture we see that there are several other arp in unicast that we can see applying a filter in wireshark.

Hereafter we have the output on the terminal H1.
One of the most important thing we can see is that the applicative layer recognize duplicates:
it is due to the fact that for every icmp req we send we will have 3 replies with the same seq.number
If we generate N icmp echo request to broadcast, 3*N will be generated by hosts but only 3N-2 will appear on the terminal because when the first reply to the last icmp req will arrive the application layer will stop to receive replies since it think that replies are finished.
Obviously the first one to reply is the local loopback with an average time of 0.08006 ms, this will influence a lot statistics on RTT as average or STD dev

```
oot@laboratorio:/home/laboratorio# arp
oot@laboratorio:/home/laboratorio#
```

The ARP table of the host that has to ping the broadcast is empty

```
root@laboratorio:/home/laboratorio# ping -c 150 172.16.0.127 -b
WARNING: pinging broadcast address
PING 172.16.0.127 (172.16.0.127) 56(84) bytes of data.
64 bytes from 172.16.0.1: icmp_req=1 ttl=64 time=0.039 ms
64 bytes from 172.16.0.2: icmp_req=1 ttl=64 time=0.593 ms (DUP!)
64 bytes from 172.16.0.3: icmp_req=1 ttl=64 time=0.837 ms (DUP!)
64 bytes from 172.16.0.1: icmp_req=2 ttl=64 time=0.034 ms
64 bytes from 172.16.0.2: icmp_req=2 ttl=64 time=0.355 ms (DUP!)
64 bytes from 172.16.0.3: icmp_req=2 ttl=64 time=0.547 ms (DUP!)
64 bytes from 172.16.0.1: icmp_req=3 ttl=64 time=0.035 ms
64 bytes from 172.16.0.2: icmp_req=3 ttl=64 time=0.388 ms (DUP!)
64 bytes from 172.16.0.3: icmp_req=3 ttl=64 time=0.482 ms (DUP!)
64 bytes from 172.16.0.1: icmp_req=4 ttl=64 time=0.033 ms
64 bytes from 172.16.0.2: icmp_req=4 ttl=64 time=0.328 ms (DUP!)
64 bytes from 172.16.0.3: icmp_req=4 ttl=64 time=0.565 ms (DUP!)
64 bytes from 172.16.0.1: icmp_req=5 ttl=64 time=0.024 ms
64 bytes from 172.16.0.2: icmp_req=5 ttl=64 time=0.372 ms (DUP!)
64 bytes from 172.16.0.3: icmp_req=5 ttl=64 time=0.530 ms (DUP!)
64 bytes from 172.16.0.1: icmp_req=6 ttl=64 time=0.032 ms
64 bytes from 172.16.0.2: icmp_req=6 ttl=64 time=0.359 ms (DUP!)
64 bytes from 172.16.0.3: icmp_req=6 ttl=64 time=0.512 ms (DUP!)
64 bytes from 172.16.0.1: icmp_req=7 ttl=64 time=0.034 ms
64 bytes from 172.16.0.2: icmp_req=7 ttl=64 time=0.357 ms (DUP!)
64 bytes from 172.16.0.3: icmp_req=7 ttl=64 time=0.509 ms (DUP!)
64 bytes from 172.16.0.1: icmp_req=8 ttl=64 time=0.033 ms
64 bytes from 172.16.0.2: icmp_req=8 ttl=64 time=0.359 ms (DUP!)
64 bytes from 172.16.0.3: icmp_req=8 ttl=64 time=0.504 ms (DUP!)
64 bytes from 172.16.0.1: icmp_req=9 ttl=64 time=0.033 ms
64 bytes from 172.16.0.2: icmp_req=9 ttl=64 time=0.358 ms (DUP!)
64 bytes from 172.16.0.3: icmp_req=9 ttl=64 time=0.417 ms (DUP!)
```

This FIGURE is the terminal of the host that pings in broadcast and we can see the Duplicate PING (DUP!)

```
root@laboratorio:/home/laboratorio# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
172.16.0.1               ether   c4:2c:03:33:ec:6b   C                     eth0
```

the ARP table of the host 2 after the ping in broadcast of the host 1. It is the same for the host 3.

2. PING NETMASK

The next part of the experiment ask us to ping the netmask (172.16.0.0). The network behavior is exactly the same as the first part of our experiment.

We have to be sure before starting that the arp tables of the 3 hosts are empty. If they are not we can manually delete entries. The Arp table is like the ping in broadcast written before.

3. DUPLICATE ADDRESS

3.1. **H2 pings 172.16.0.1**

. Next step is to sniff a connection in a situation in which two host are configured as 172.16.0.1 and the third one in configured as 172.16.0.2

We can clearly see the first arp request for 172.16.0.1 and two arp replies corresponding to the same arp request Checking the arp tables of H2 we see that only the first one is considered so from the moment it recives the first arp reply (which in this case is from H1') it will associate H1 ip address with H1' mac address.

But this is something strange: according to rfc 826:

> When an address resolution packet is received, the receiving Ethernet module gives the packet to the Address Resolution module which goes through an algorithm similar to the following. Negative conditionals indicate an end of processing and a discarding of the packet.
>
> Notice that the <protocol type, sender protocol address, sender hardware address> triplet is merged into the table before the opcode is looked at. This is on the assumption that communcation is bidirectional; if A has some reason to talk to B, then B will probably have some reason to talk to A. Notice also that if an entry already exists for the <protocol type, sender protocol address> pair, then the new hardware address supersedes the old one. Related Issues gives some motivation for this.

So from the capture we can expect that the arp entry related to ip 172.16.0.1 would be the second one which is supposed tu supersed the old one. Instead this doesn't appen. It is probably due to a mechanism act to prevent Arp Poisoning attack.

According to the same rfc(826) we also see that H1 doesn't need to make an arp request to reply with an echo reply to H1: he already saved the data when the first arp req arrived. This is an obvious functionality act to reduce the amount of traffic in a network where is generally supposed to take place bidirectional communication. We can also notice (as in the previous part) that after some time there is a second arp request in UNICAST. Exploring the packet we can see that the address bit is setted to 0 which is the corresponding for unicast.

1	0.000000	Apple_2f:07:bb	Broadcast	ARP
2	0.000225	Asiarock_60:7f:4a	Apple_2f:07:bb	ARP
3	0.000249	172.16.0.3	172.16.0.1	ICMP
4	0.000366	Apple_33:ec:6b	Apple_2f:07:bb	ARP
5	0.000408	172.16.0.1	172.16.0.3	ICMP
6	0.999359	172.16.0.3	172.16.0.1	ICMP
7	0.999581	172.16.0.1	172.16.0.3	ICMP
8	1.999374	172.16.0.3	172.16.0.1	ICMP
9	1.999597	172.16.0.1	172.16.0.3	ICMP
10	2.999380	172.16.0.3	172.16.0.1	ICMP
11	2.999604	172.16.0.1	172.16.0.3	ICMP
12	3.999383	172.16.0.3	172.16.0.1	ICMP
13	3.999607	172.16.0.1	172.16.0.3	ICMP
14	4.999382	172.16.0.3	172.16.0.1	ICMP
15	4.999611	172.16.0.1	172.16.0.3	ICMP
16	5.008437	Asiarock_60:7f:4a	Apple_2f:07:bb	ARP
17	5.008476	Apple_2f:07:bb	Asiarock_60:7f:4a	ARP
18	5.999328	172.16.0.3	172.16.0.1	ICMP
19	5.999552	172.16.0.1	172.16.0.3	ICMP
20	7.000997	172.16.0.3	172.16.0.1	ICMP
21	7.001214	172.16.0.1	172.16.0.3	ICMP
22	7.999992	172.16.0.3	172.16.0.1	ICMP
23	8.000219	172.16.0.1	172.16.0.3	ICMP
24	8.999362	172.16.0.3	172.16.0.1	ICMP

```
42 Who has 172.16.0.1?  Tell 172.16.0.3
60 172.16.0.1 is at 00:19:66:60:7f:4a
98 Echo (ping) request  id=0x16ea, seq=1/256, ttl=64 (reply in 5)
60 172.16.0.1 is at c4:2c:03:33:ec:6b
98 Echo (ping) reply    id=0x16ea, seq=1/256, ttl=64 (request in 3)
98 Echo (ping) request  id=0x16ea, seq=2/512, ttl=64 (reply in 7)
98 Echo (ping) reply    id=0x16ea, seq=2/512, ttl=64 (request in 6)
98 Echo (ping) request  id=0x16ea, seq=3/768, ttl=64 (reply in 9)
98 Echo (ping) reply    id=0x16ea, seq=3/768, ttl=64 (request in 8)
98 Echo (ping) request  id=0x16ea, seq=4/1024, ttl=64 (reply in 11)
98 Echo (ping) reply    id=0x16ea, seq=4/1024, ttl=64 (request in 10)
98 Echo (ping) request  id=0x16ea, seq=5/1280, ttl=64 (reply in 13)
98 Echo (ping) reply    id=0x16ea, seq=5/1280, ttl=64 (request in 12)
98 Echo (ping) request  id=0x16ea, seq=6/1536, ttl=64 (reply in 15)
98 Echo (ping) reply    id=0x16ea, seq=6/1536, ttl=64 (request in 14)
60 Who has 172.16.0.3?  Tell 172.16.0.1
42 172.16.0.3 is at c8:2a:14:2f:07:bb
98 Echo (ping) request  id=0x16ea, seq=7/1792, ttl=64 (reply in 19)
98 Echo (ping) reply    id=0x16ea, seq=7/1792, ttl=64 (request in 18)
98 Echo (ping) request  id=0x16ea, seq=8/2048, ttl=64 (reply in 21)
98 Echo (ping) reply    id=0x16ea, seq=8/2048, ttl=64 (request in 20)
98 Echo (ping) request  id=0x16ea, seq=9/2304, ttl=64 (reply in 23)
98 Echo (ping) reply    id=0x16ea, seq=9/2304, ttl=64 (request in 22)
98 Echo (ping) request  id=0x16ea, seq=10/2560, ttl=64 (reply in 25)
```

Here it is the capture done on the dev H2

1	0.000000	Apple_2f:07:bb	Broadcast	ARP
2	0.000027	Asiarock_60:7f:4a	Apple_2f:07:bb	ARP
3	0.000232	172.16.0.3	172.16.0.1	ICMP
4	0.000252	172.16.0.1	172.16.0.3	ICMP
5	0.999299	172.16.0.3	172.16.0.1	ICMP
6	0.999321	172.16.0.1	172.16.0.3	ICMP
7	1.999249	172.16.0.3	172.16.0.1	ICMP
8	1.999269	172.16.0.1	172.16.0.3	ICMP
9	2.999184	172.16.0.3	172.16.0.1	ICMP
10	2.999205	172.16.0.1	172.16.0.3	ICMP
11	3.999119	172.16.0.3	172.16.0.1	ICMP
12	3.999139	172.16.0.1	172.16.0.3	ICMP
13	4.999049	172.16.0.3	172.16.0.1	ICMP
14	4.999070	172.16.0.1	172.16.0.3	ICMP
15	5.007848	Asiarock_60:7f:4a	Apple_2f:07:bb	ARP
16	5.008113	Apple_2f:07:bb	Asiarock_60:7f:4a	ARP
17	5.998929	172.16.0.3	172.16.0.1	ICMP
18	5.998951	172.16.0.1	172.16.0.3	ICMP
19	7.000522	172.16.0.3	172.16.0.1	ICMP
20	7.000541	172.16.0.1	172.16.0.3	ICMP
21	7.999452	172.16.0.3	172.16.0.1	ICMP
22	7.999472	172.16.0.1	172.16.0.3	ICMP
23	8.998756	172.16.0.3	172.16.0.1	ICMP
24	8.998779	172.16.0.1	172.16.0.3	ICMP

```
60 Who has 172.16.0.1?  Tell 172.16.0.3
42 172.16.0.1 is at 00:19:66:60:7f:4a
98 Echo (ping) request  id=0x16ea, seq=1/256, ttl=64 (reply in 4)
98 Echo (ping) reply    id=0x16ea, seq=1/256, ttl=64 (request in 3)
98 Echo (ping) request  id=0x16ea, seq=2/512, ttl=64 (reply in 6)
98 Echo (ping) reply    id=0x16ea, seq=2/512, ttl=64 (request in 5)
98 Echo (ping) request  id=0x16ea, seq=3/768, ttl=64 (reply in 8)
98 Echo (ping) reply    id=0x16ea, seq=3/768, ttl=64 (request in 7)
98 Echo (ping) request  id=0x16ea, seq=4/1024, ttl=64 (reply in 10)
98 Echo (ping) reply    id=0x16ea, seq=4/1024, ttl=64 (request in 9)
98 Echo (ping) request  id=0x16ea, seq=5/1280, ttl=64 (reply in 12)
98 Echo (ping) reply    id=0x16ea, seq=5/1280, ttl=64 (request in 11)
98 Echo (ping) request  id=0x16ea, seq=6/1536, ttl=64 (reply in 14)
98 Echo (ping) reply    id=0x16ea, seq=6/1536, ttl=64 (request in 13)
42 Who has 172.16.0.3?  Tell 172.16.0.1
60 172.16.0.3 is at c8:2a:14:2f:07:bb
98 Echo (ping) request  id=0x16ea, seq=7/1792, ttl=64 (reply in 18)
98 Echo (ping) reply    id=0x16ea, seq=7/1792, ttl=64 (request in 17)
98 Echo (ping) request  id=0x16ea, seq=8/2048, ttl=64 (reply in 20)
98 Echo (ping) reply    id=0x16ea, seq=8/2048, ttl=64 (request in 19)
98 Echo (ping) request  id=0x16ea, seq=9/2304, ttl=64 (reply in 22)
98 Echo (ping) reply    id=0x16ea, seq=9/2304, ttl=64 (request in 21)
98 Echo (ping) request  id=0x16ea, seq=10/2560, ttl=64 (reply in 24)
98 Echo (ping) reply    id=0x16ea, seq=10/2560, ttl=64 (request in 23)
```

Here it is the trace of the capture done on the host H1'

So we can see a perfectly matchable capture between this list and the previous one

On the other host (H1) we can only see an arp request and an arp reply, then nothing because all the other messages will be on a two-point communication between H1' and H2 so in our networking topology (3 hosts linked by a switch) on H1 we wont see anything after the two arp messages.

```
 3 19.770333    Apple_2f:07:bb    Broadcast    ARP
 4 19.770364    Apple_33:ec:6b    Apple_2f:07:bb    ARP
```

```
60 Who has 172.16.0.1?  Tell 172.16.0.3
42 172.16.0.1 is at c4:2c:03:33:ec:6b
```

An implementation detail we noticed is that if an arp request is received no security antipoisoning check is done (or at least with our version of operating system, we cannot say it is a general result).

```
--- 172.16.0.1 ping statistics ---
100 packets transmitted, 100 received, 0% packet loss, time 98999ms
rtt min/avg/max/mdev = 0.217/0.258/0.425/0.023 ms
root@laboratorio:/home/laboratorio# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
172.16.0.1               ether   00:19:66:60:7f:4a   C                     eth0
```

This is the ARP table of the Host 2 after that it pings the other two with the same IP adress

```
root@laboratorio:/home/laboratorio# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
172.16.0.3               ether   c8:2a:14:2f:07:bb   C                     eth0
```

This is the ARP table of the Host 1 and 1' after received ping from Host 2

3.2. **H2 is pinged by H1 and H1'**

. In this case H1 and H1' ping H2 and they have the same IP adress.
What happens, at the contrary of the previous situation, is that, when an ARP request is received the ARP table of H2 is refreshed. Due to the implementation of ARP request in unicast after the first one in broadcast, H2 will reply to H1 and H1' alternativly due ARP refreshing. But at the first, due to obvious reason, the first ICMPechoreply will arrive to both host. We can also notice that Wireshark recomnize duplicate use of the IP adress.

```
root@laboratorio:/home/laboratorio# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
172.16.0.1               ether   c4:2c:03:33:ec:6b   C                     eth0
```

This is the ARP table of the Host 2 that is pinged by the other two host with the same ip adress he can see only one MAC adress

```
root@laboratorio:/home/laboratorio# ping -c 150 172.16.0.3
PING 172.16.0.3 (172.16.0.3) 56(84) bytes of data.
64 bytes from 172.16.0.3: icmp_req=1 ttl=64 time=0.821 ms
64 bytes from 172.16.0.3: icmp_req=30 ttl=64 time=0.519 ms
64 bytes from 172.16.0.3: icmp_req=31 ttl=64 time=0.519 ms
64 bytes from 172.16.0.3: icmp_req=32 ttl=64 time=0.502 ms
64 bytes from 172.16.0.3: icmp_req=33 ttl=64 time=0.514 ms
64 bytes from 172.16.0.3: icmp_req=34 ttl=64 time=0.541 ms
```

This is the terminal on the host 1. We can see that he start to ping only after $icmp\_rec = 1$

```
root@laboratorio:/home/laboratorio# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
172.16.0.3               ether   c8:2a:14:2f:07:bb   C                     eth0
```

This is the ARP table of the Host 1: we can see the mac adress of host 2

```
root@laboratorio:/home/laboratorio# ping -c 150 172.16.0.3
PING 172.16.0.3 (172.16.0.3) 56(84) bytes of data.
64 bytes from 172.16.0.3: icmp_req=1 ttl=64 time=0.516 ms
64 bytes from 172.16.0.3: icmp_req=2 ttl=64 time=0.313 ms
64 bytes from 172.16.0.3: icmp_req=3 ttl=64 time=0.303 ms
64 bytes from 172.16.0.3: icmp_req=4 ttl=64 time=0.314 ms
64 bytes from 172.16.0.3: icmp_req=5 ttl=64 time=0.285 ms
64 bytes from 172.16.0.3: icmp_req=6 ttl=64 time=0.311 ms
64 bytes from 172.16.0.3: icmp_req=7 ttl=64 time=0.308 ms
64 bytes from 172.16.0.3: icmp_req=8 ttl=64 time=0.288 ms
64 bytes from 172.16.0.3: icmp_req=9 ttl=64 time=0.314 ms
64 bytes from 172.16.0.3: icmp_req=10 ttl=64 time=0.282 ms
64 bytes from 172.16.0.3: icmp_req=11 ttl=64 time=0.306 ms
64 bytes from 172.16.0.3: icmp_req=12 ttl=64 time=0.291 ms
64 bytes from 172.16.0.3: icmp_req=13 ttl=64 time=0.309 ms
64 bytes from 172.16.0.3: icmp_req=14 ttl=64 time=0.307 ms
64 bytes from 172.16.0.3: icmp_req=15 ttl=64 time=0.308 ms
```

This is the terminal on the host 1'. We can see that he started to ping while host 1 is not pinging (see the $icmp\_req = 1$)

```
root@laboratorio:/home/laboratorio# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
172.16.0.3               ether   c8:2a:14:2f:07:bb   C                     eth0
```

This is the ARP table of the Host 1': we can see the mac adress of host 2

```
11 16.808521      Asiarock_60:7f:4a          Broadcast                 ARP
```

```
60 Who has 172.16.0.3?  Tell 172.16.0.1 (duplicate use of 172.16.0.1 detected!)
```

duplicate use

4. WRONG NETMASK

We configure the hosts:

H1: 172.16.0.1/25
H2: 172.16.0.70/29

We have configured 2 host as described above, and our test will be trying to ping H2 from H1 and viceversa.

H1 believes that H2 belong to his subnet so he will send an arp req but as H1 will receive this packet he will discard it because it is from an IP which is not part of his subnet.
From this H2 behavior we can learn that arp mechanism check also subnet when deciding if an arp request is valid or not.
On the other way when is H2 that tries to ping H1 the arp request does not even leave the device: the ip layer recognize that 172.16.0.1 is unreachable and so we will see the error line Network Unreachable.
So the ARP tables of both the hosts are empty.

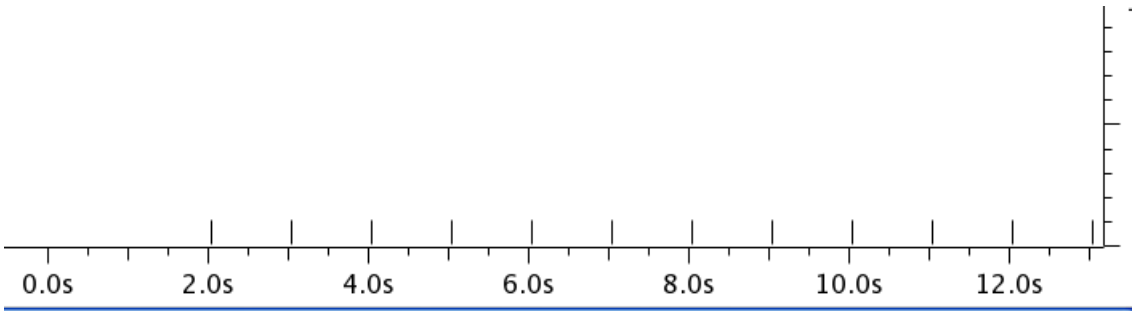
```
root@laboratorio:/home/laboratorio# ping -c 10  172.16.0.70
PING 172.16.0.70 (172.16.0.70) 56(84) bytes of data.
From 172.16.0.1 icmp_seq=1 Destination Host Unreachable
From 172.16.0.1 icmp_seq=2 Destination Host Unreachable
From 172.16.0.1 icmp_seq=3 Destination Host Unreachable
From 172.16.0.1 icmp_seq=4 Destination Host Unreachable
From 172.16.0.1 icmp_seq=5 Destination Host Unreachable
From 172.16.0.1 icmp_seq=6 Destination Host Unreachable
From 172.16.0.1 icmp_seq=7 Destination Host Unreachable
From 172.16.0.1 icmp_seq=8 Destination Host Unreachable
From 172.16.0.1 icmp_seq=9 Destination Host Unreachable
From 172.16.0.1 icmp_seq=10 Destination Host Unreachable

--- 172.16.0.70 ping statistics ---
10 packets transmitted, 0 received, +10 errors, 100% packet loss, time 8999ms
pipe 4
```

H1 is traying to ping H2 but as we can see the destination host is unreachable

```
root@laboratorio:/home/laboratorio# ping  172.16.0.1
connect: Network is unreachable
```

H2 is trying to ping H1 but the network is unreachable as we can see in this figure



every second it send an ARP request

5. WRONG NETMASK AND BROADCAST ADDRESS

We configure the hosts:

H1: 172.16.0.127/24
H2: 172.16.0.1/25

1) H1 want to ping H2

Here appens that H1,recognizing H2 as belonging to his subnet, generate an arp request for H2. But H2 receive an ARP request from an address that for his configuration is the broadcast address (.127), so there's no way that H2 is going to reply to an arp req for the broadcast address.
Only packet we can see are H1 arp requests
So in this case the ARP table are empty.

```
root@laboratorio:/home/laboratorio# ping -c 30  172.16.0.127 -b
WARNING: pinging broadcast address
PING 172.16.0.127 (172.16.0.127) 56(84) bytes of data.
64 bytes from 172.16.0.1: icmp_req=1 ttl=64 time=0.041 ms
64 bytes from 172.16.0.1: icmp_req=2 ttl=64 time=0.031 ms
64 bytes from 172.16.0.1: icmp_req=3 ttl=64 time=0.030 ms
64 bytes from 172.16.0.1: icmp_req=4 ttl=64 time=0.033 ms
64 bytes from 172.16.0.1: icmp_req=5 ttl=64 time=0.029 ms
64 bytes from 172.16.0.1: icmp_req=6 ttl=64 time=0.032 ms
64 bytes from 172.16.0.1: icmp_req=7 ttl=64 time=0.034 ms
64 bytes from 172.16.0.1: icmp_req=8 ttl=64 time=0.031 ms
64 bytes from 172.16.0.1: icmp_req=9 ttl=64 time=0.034 ms
64 bytes from 172.16.0.1: icmp_req=10 ttl=64 time=0.033 ms
64 bytes from 172.16.0.1: icmp_req=11 ttl=64 time=0.033 ms
64 bytes from 172.16.0.1: icmp_req=12 ttl=64 time=0.031 ms
64 bytes from 172.16.0.1: icmp_req=13 ttl=64 time=0.033 ms
64 bytes from 172.16.0.1: icmp_req=14 ttl=64 time=0.030 ms
64 bytes from 172.16.0.1: icmp_req=15 ttl=64 time=0.030 ms
64 bytes from 172.16.0.1: icmp_req=16 ttl=64 time=0.032 ms
64 bytes from 172.16.0.1: icmp_req=17 ttl=64 time=0.030 ms
64 bytes from 172.16.0.1: icmp_req=18 ttl=64 time=0.032 ms
64 bytes from 172.16.0.1: icmp_req=19 ttl=64 time=0.035 ms
64 bytes from 172.16.0.1: icmp_req=20 ttl=64 time=0.032 ms
64 bytes from 172.16.0.1: icmp_req=21 ttl=64 time=0.030 ms
64 bytes from 172.16.0.1: icmp_req=22 ttl=64 time=0.032 ms
64 bytes from 172.16.0.1: icmp_req=23 ttl=64 time=0.030 ms
64 bytes from 172.16.0.1: icmp_req=24 ttl=64 time=0.030 ms
64 bytes from 172.16.0.1: icmp_req=25 ttl=64 time=0.033 ms
64 bytes from 172.16.0.1: icmp_req=26 ttl=64 time=0.033 ms
64 bytes from 172.16.0.1: icmp_req=27 ttl=64 time=0.031 ms
64 bytes from 172.16.0.1: icmp_req=28 ttl=64 time=0.033 ms
64 bytes from 172.16.0.1: icmp_req=29 ttl=64 time=0.032 ms
64 bytes from 172.16.0.1: icmp_req=30 ttl=64 time=0.030 ms
```

In this case we can see that H1 is pinging H2 and consider it like a broadcast

1	0.000000	172.16.0.1	172.16.0.127	ICMP
2	0.000338	Asiarock_60:7f:4a	Broadcast	ARP
3	0.997252	Asiarock_60:7f:4a	Broadcast	ARP
4	0.999969	172.16.0.1	172.16.0.127	ICMP
6	1.997259	Asiarock_60:7f:4a	Broadcast	ARP
7	1.999962	172.16.0.1	172.16.0.127	ICMP
8	2.999972	172.16.0.1	172.16.0.127	ICMP
9	3.000275	Asiarock_60:7f:4a	Broadcast	ARP
0	3.997278	Asiarock_60:7f:4a	Broadcast	ARP
1	3.999967	172.16.0.1	172.16.0.127	ICMP
2	4.997293	Asiarock_60:7f:4a	Broadcast	ARP
3	4.999972	172.16.0.1	172.16.0.127	ICMP
4	6.000090	172.16.0.1	172.16.0.127	ICMP
5	6.000403	Asiarock_60:7f:4a	Broadcast	ARP
6	6.997340	Asiarock_60:7f:4a	Broadcast	ARP
7	7.000002	172.16.0.1	172.16.0.127	ICMP
8	7.997296	Asiarock_60:7f:4a	Broadcast	ARP
9	7.999892	172.16.0.1	172.16.0.127	ICMP
0	8.999974	172.16.0.1	172.16.0.127	ICMP
1	9.000274	Asiarock_60:7f:4a	Broadcast	ARP
2	9.997389	Asiarock_60:7f:4a	Broadcast	ARP
3	9.999965	172.16.0.1	172.16.0.127	ICMP
4	10.997404	Asiarock_60:7f:4a	Broadcast	ARP
5	10.999971	172.16.0.1	172.16.0.127	ICMP
6	11.999975	172.16.0.1	172.16.0.127	ICMP
7	12.000274	Asiarock_60:7f:4a	Broadcast	ARP
8	12.997417	Asiarock_60:7f:4a	Broadcast	ARP
9	12.999967	172.16.0.1	172.16.0.127	ICMP
0	13.997451	Asiarock_60:7f:4a	Broadcast	ARP
1	13.999972	172.16.0.1	172.16.0.127	ICMP

98	Echo (ping) request	id=0x11fa, seq=1/256, ttl=64
60	Who has 172.16.0.1?	Tell 172.16.0.127
60	Who has 172.16.0.1?	Tell 172.16.0.127
98	Echo (ping) request	id=0x11fa, seq=2/512, ttl=64
60	Who has 172.16.0.1?	Tell 172.16.0.127
98	Echo (ping) request	id=0x11fa, seq=3/768, ttl=64
98	Echo (ping) request	id=0x11fa, seq=4/1024, ttl=64
60	Who has 172.16.0.1?	Tell 172.16.0.127
60	Who has 172.16.0.1?	Tell 172.16.0.127
98	Echo (ping) request	id=0x11fa, seq=5/1280, ttl=64
60	Who has 172.16.0.1?	Tell 172.16.0.127
98	Echo (ping) request	id=0x11fa, seq=6/1536, ttl=64
98	Echo (ping) request	id=0x11fa, seq=7/1792, ttl=64
60	Who has 172.16.0.1?	Tell 172.16.0.127
60	Who has 172.16.0.1?	Tell 172.16.0.127
98	Echo (ping) request	id=0x11fa, seq=8/2048, ttl=64
60	Who has 172.16.0.1?	Tell 172.16.0.127

```
98 Echo (ping) request  id=0x11fa, seq=9/2304, ttl=64
98 Echo (ping) request  id=0x11fa, seq=10/2560, ttl=64
60 Who has 172.16.0.1?  Tell 172.16.0.127
60 Who has 172.16.0.1?  Tell 172.16.0.127
98 Echo (ping) request  id=0x11fa, seq=11/2816, ttl=64
60 Who has 172.16.0.1?  Tell 172.16.0.127
98 Echo (ping) request  id=0x11fa, seq=12/3072, ttl=64
98 Echo (ping) request  id=0x11fa, seq=13/3328, ttl=64
60 Who has 172.16.0.1?  Tell 172.16.0.127
60 Who has 172.16.0.1?  Tell 172.16.0.127
98 Echo (ping) request  id=0x11fa, seq=14/3584, ttl=64
60 Who has 172.16.0.1?  Tell 172.16.0.127
98 Echo (ping) request  id=0x11fa, seq=15/3840, ttl=64
```

this is the capture from H1 that it is the same of H2

2) H2 want to ping H1

.1 recognize .127 as his broadcast address,so it will directly generate an icmp echo request with MAC address ff:ff:ff:ff:ff:ff
.127 receive a message from a .1 and to reply he need his mac address,so it generate an arp request to which .1 wont reply thinking is the broadcast address.
There will be no comunication, below the capture confirm what we expected, so, like we expected, the ARP table of the two hosts remain empty.

Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP
Asiarock_60:7f:4a	Broadcast	ARP

60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127
60	Who has 172.16.0.1? Tell 172.16.0.127

capture seeing from H1

Networking Laboratory II
CAPACITY ESTIMATION USING PING

GIULIO FRANZESE 182969 - MARCO RECENTI 180644 - TOMMASO CALIFANO 180808

ACADEMIC YEAR 2013 - 2014
PROF. MELLIA MARCO

1 Methodology to estimate the capacity offered by the physical layer

In order to evaluate the access link capacity we used " ping " tool. This tool provides us informations about : sequence number, RTT, which is related to the speed of the data sent, and TTL

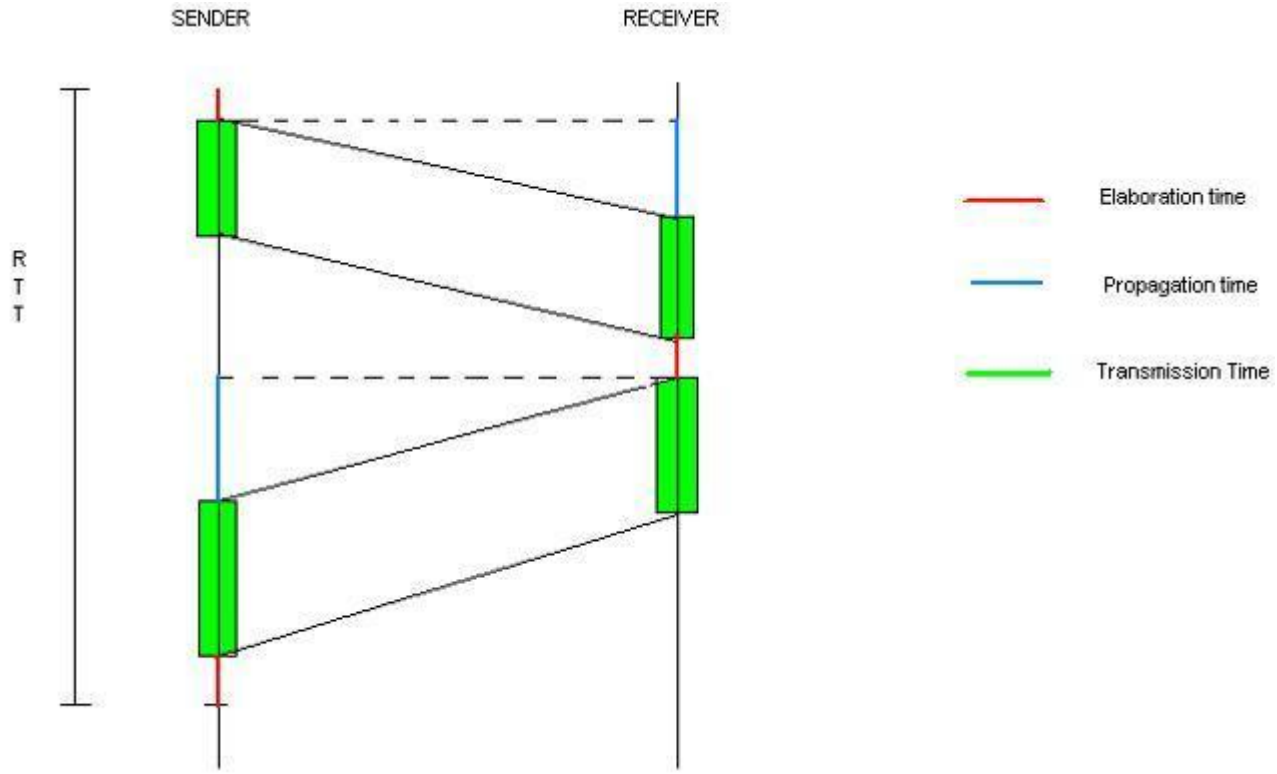


FIGURE 1

We are assuming that :

$$T_{TX} \ll T_\epsilon \tag{1}$$

From the figure we can describe the RTT as the sum of $2T_{TX}$ from sender to receiver and viceversa, $3T_\epsilon$, which is the elaboration time, for the data sent and $2T_p$ (propagation time).

$$RTT = 2T_{TX} + 3T_\epsilon + 2T_p \tag{2}$$

2 Evaluation of capacity and RTT

Assuming a random distribution of packet's elaboration time, we repeated the RTT's measurement many times in order to increase the probability to evaluate the minimum from all the possibles elaboration's time and have a more accurated measurement. D is equal to S, ping PDU size, plus headers' bytes of ICMP containing sequence number identifier (8 B) IP (20 B) Ethernet (18 B) Physical layer (8 B) + IPG (Inter packet gap; 12 B) which leads to a packet of 1538B. We know also that :

$$T_{TX} = \frac{L1data}{C} = \frac{D}{C} \tag{3}$$

So we can approximate RTT as

$$RTT = \frac{2D}{C} \tag{4}$$

We can notice a difference in the measurement of RTT when we change the size of the packet from 1472 B to 1473 B, and so on for their multiples, because we need to transmit another packet since we reached the maximum size permitted. This phenomenon is called fragmentation and it starts when the packet overcome 1538B.

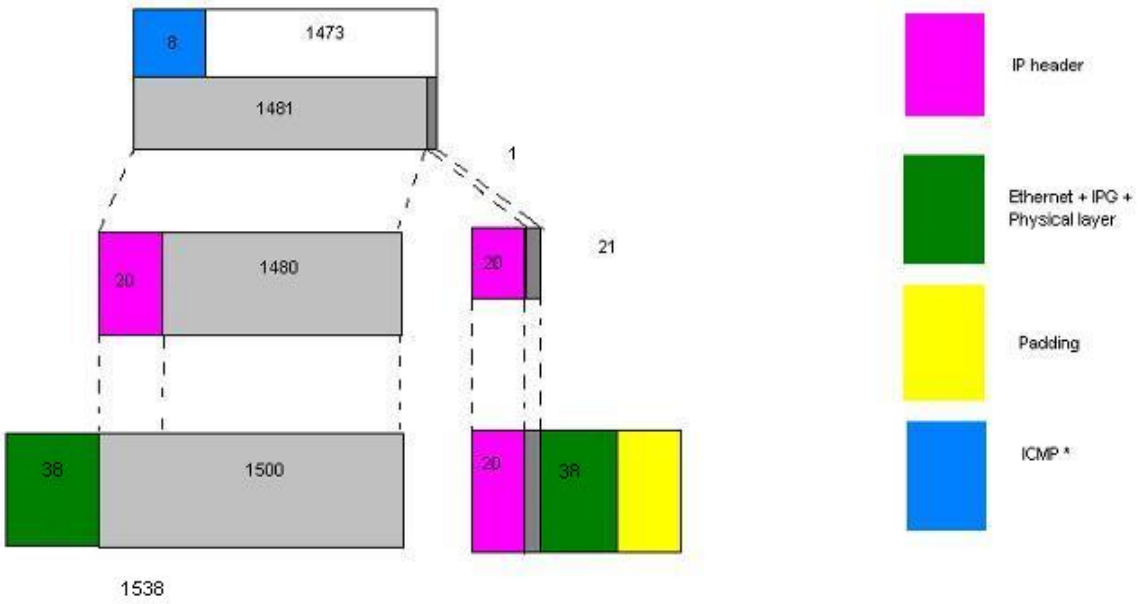
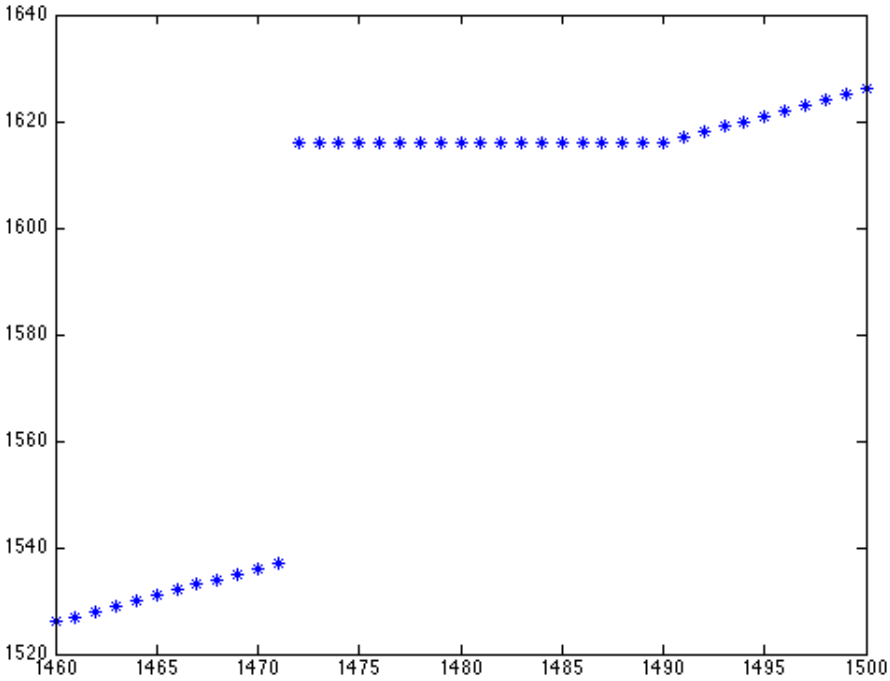


FIGURE 2 : FRAGMENTATION

Since we have a MTU, maximum transfer unit, of 1500 B, if we try to send a packet of 1473 B it will be splitted into two, because we want/need 8 B for ICMP header and 20 B for IP header which sum is 1501B. This behaviour is taken into account when calculation the capacity, infact we did not use S size but D size that is described by the following formula :

$$D[S] = 1538 * floor(\frac{S}{1472}) + c * (64 + 12 + 8) + ramp(mod(S, 1472) - 18) \tag{5}$$

Where c is equal to 0 if s%1472=0 and 1 otherwise



Using a switch to link H1 and H2, the situation is similar to the one in the figure below .We didn't consider in the graph both elaboration and propagation times

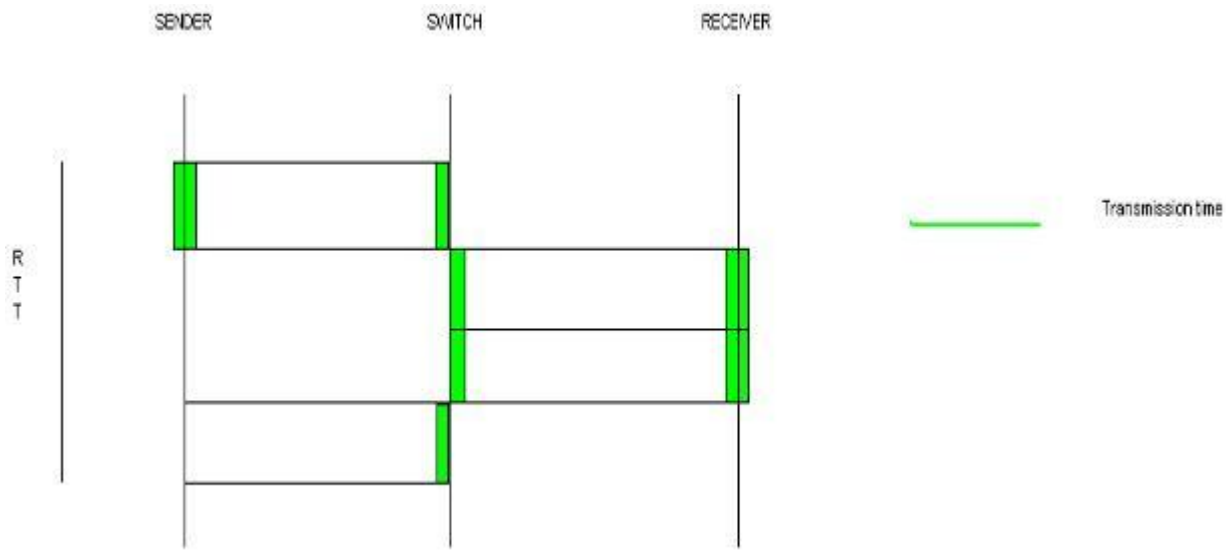


FIGURE 3

In this case we have :

$$RTT = 4T_{TX} + T_\epsilon \tag{6}$$

$$C = \frac{4D}{RTT} \tag{7}$$

If there is fragmentation of packets, (3 in this case) what we have is a situation similar to the figure below (that does not consider propagation and elaboration time)

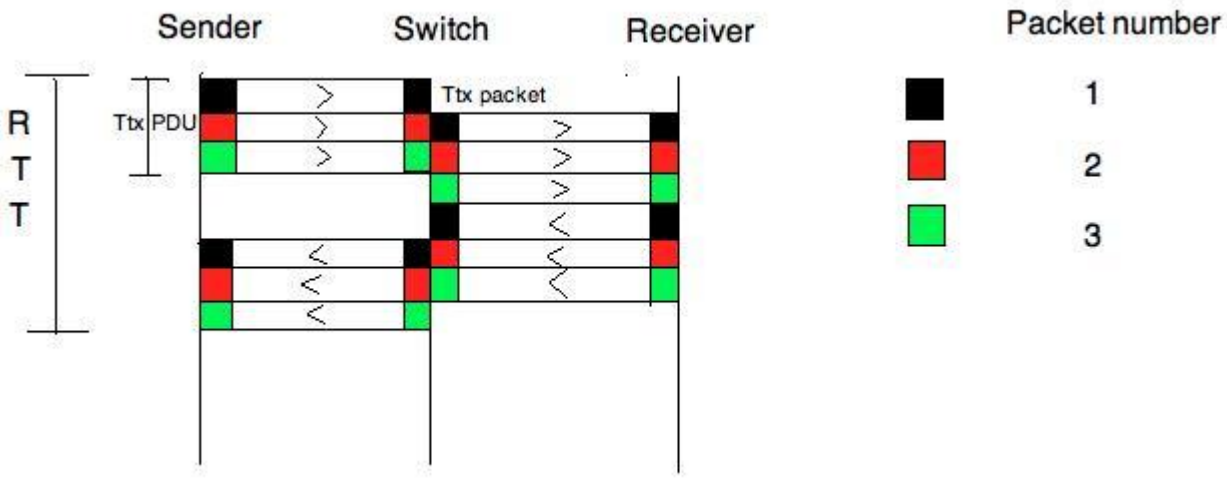


FIGURE 4

So we assume :

$$RTT = 2T_{TX} + 2T_{TX,packets} + T_\epsilon \tag{8}$$

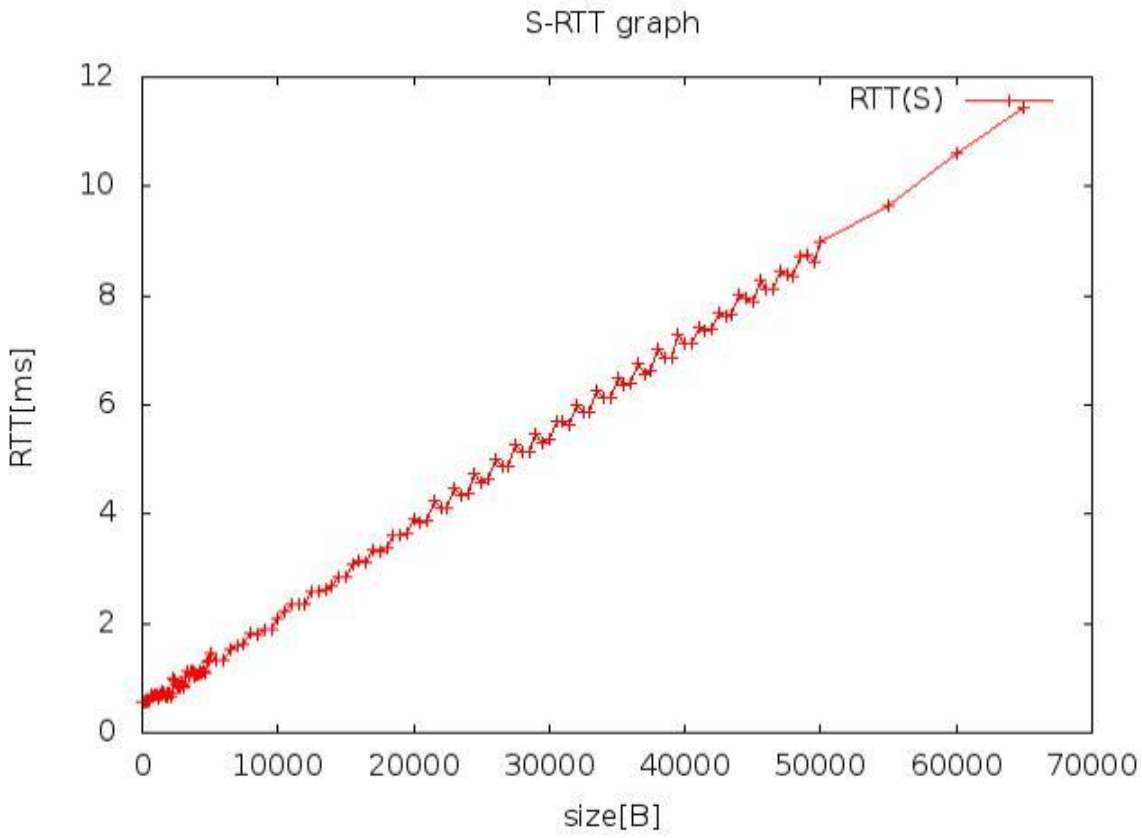


FIGURE 5 : EVOLUTION OF RTT OVER PDU SIZE

Since RTT depends on transmission and elaboration time, we can notice that, as we were expecting, RTT increases linearly, because both T_{TX} and T_{ϵ} depends on PDU size. we can see a difference of the concentration of the points in the graph: it is due to the bash we did.
How we can see in the script first part of the bash is about size from 12 to 5000 at a rate of 50. The second part from 5000 to 25000 at a rate of 100 and the third and last part is form 25000 to 50000 at a rate of 500.
The data are taken from Size1.dat and exported to the file Result.dat that is useful for the gnuplot graph to implement.

```
rm -f size1.dat
rm -f size2.dat
rm -f fin1.dat
rm -f fin2.dat
rm -f fin3.dat
rm -f RTT1.png
rm -f RTT2.png
rm -f appr_CAPACITY.png

#ifconfig eth0 172.16.0.1/24
#ethtool -s eth0 speed 10 duplex full autoneg on


O=66 #8+20+18+7+12
for ((s=12;s<5000;s=s+50))
do
    echo $s>>Size1.dat
    echo  $(( $(( 1532*$(($s/1472)) )) +64+12+8+ $(( $(( $(( $(( $s
        %1472 ))-18 ))>0 ))*$(( ($s%1472-18) )) )) ))>>Size2.dat    #$(($s+$O))
    ping 172.16.0.1 -s $s -c 5 -i 0.1|grep min|cut -d '/' -f 4|cut -d ' ' -f 3
        >>Result.dat
done

for ((s=5000;s<25000;s=s+100))
do
     echo $s>>Size1.dat
     echo  $(( $(( 1532*$(($s/1472)) )) +64+12+8+ $(( $(( $(( $(( $s
         %1472 ))-18 ))>0 ))*$(( ($s%1472-18) )) )) ))>>Size2.dat
     ping 172.16.0.1 -s $s -c 5 -i 0.1|grep min|cut -d '/' -f 4|cut -d ' ' -f 3
         >>Result.dat
done

for ((s=25000;s<50000;s=s+500))
do
    echo $s>>Size1.dat
    echo  $(( $(( 1532*$(($s/1472)) )) +64+12+8+ $(( $(( $(( $(( $s
        %1472 ))-18 ))>0 ))*$(( ($s%1472-18) )) )) ))>>Size2.dat
    ping 172.16.0.1 -s $s -c 5 -i 0.1|grep min|cut -d '/' -f 4|cut -d ' ' -f 3
        >>Result.dat
done

paste Size1.dat Result.dat>> Fin1.dat
paste Size2.dat Result.dat>> Fin2.dat
```

We also did a cicle to check the validity of the model (that is commented in the script)

```
#ulteriore ciclo per controllare validità modello
#for ((s=1460;s<1501;s=s+1))
#do
#   echo $s>>test1.dat
   #echo  $(( $(( 1532*$(($s/1472)) )) +64+12+8+ $(( $(( $(( $(( $s
       %1472 ))-18 ))>0 ))*$(( ($s%1472-18) )) )) ))>>test2.dat
#   ping 172.16.0.1 -s $s -c 30 -i 0.1|grep min|cut -d '/' -f 4|cut -d ' ' -f 3
    >>testresult.dat
#done

#paste test1.dat testresult.dat>> check1.dat
#paste test2.dat testresult.dat>> check2.dat
```

This is the script of the bash relative to the gnuplot implementation of the graphs. The final gnuplot implementation is a check and it is commented.

```
gnuplot <<- EOF
set output "rRTT1.png"
set terminal png
set xlabel "size[B]"
set ylabel "RTT[ms]"
set title "S-RTT graph"
plot "Fin1.dat" using 1:2 title "RTT(S)" with linespoints
EOF

gnuplot <<-EOF
set output "rRTT2.png"
set terminal png
set xlabel "size[D]"
set ylabel "RTT[ms]"
set title "D-RTT graph"
plot "Fin2.dat" using 1:2 title "RTT(D)" with linespoints
EOF

gnuplot <<- EOF
set output "rappr_CAPACITY.png"
set terminal png
set xlabel "size[D] (byte)"
set ylabel "C (bit/s)"
set title "appr_CAPACITY graph"
plot "Fin2.dat" using ($1):(4*$1/$2) title "C" with linespoints
EOF



#gnuplot <<- EOF
#set output "specific_test.png"
#set terminal png
#set xlabel "size[S]"
#set ylabel "RTT[ms]"
#set title "specific_check"
#plot "check1.dat" using 1:2 title "RTT(S)" with linespoints
#EOF
```

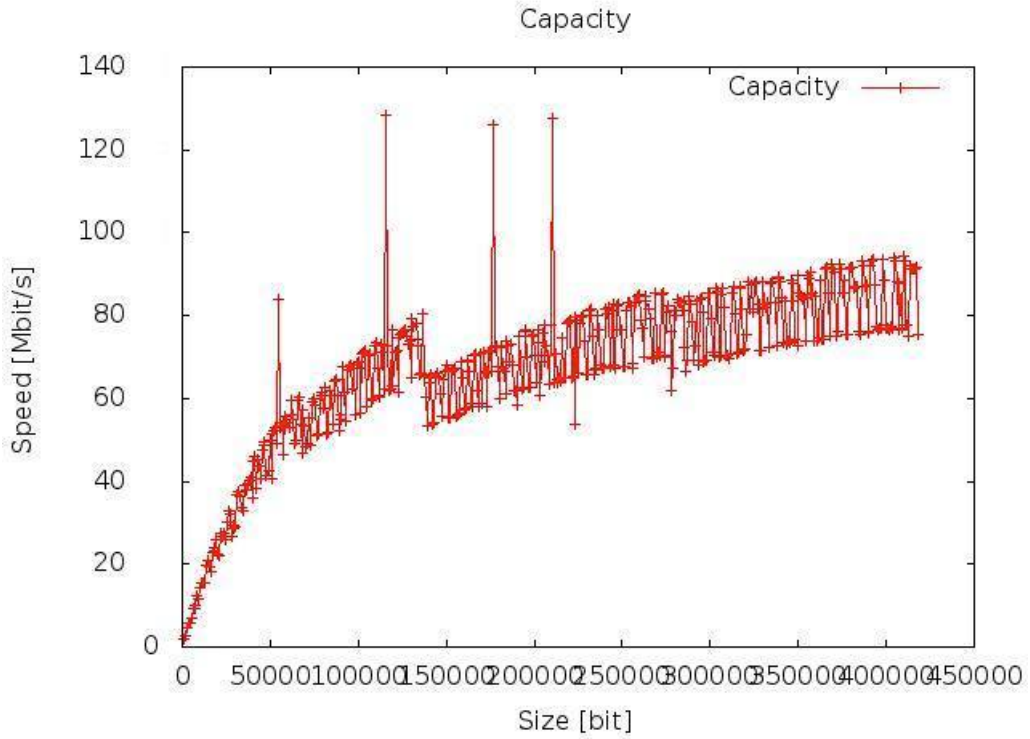


FIGURE 6 : EVOLUTION OF THE ESTIMATED CAPACITY OVER PDU SIZE

We can see that, a part from oscillation, the graph attempt to an asymptote.

Capacity has a behavior similar to a function as $C = \frac{D}{\alpha+D}$, so it has an superior orizzontal asymptote which indicates the maximum value reachable from the capacity.
As we can see in the figure above, which is an evaluation of the capacity over PDU size, while we are plotting we have some oscillations (we can see these oscillations also in the figure of the Evolution of RTT over PDU size). Those are due to a quantization's error introduced by processor's clock which of course doesn't have infinite precision, so sometimes we have approximation by excess and other times by defect adn for sure it can't be modified because the precision of the clock depens on the computer we are using.

We have also repeated the measurement of the capacity by connecting the two hosts directly and forcing the interfaces to negotiate a 10Mb/s full duplex link.

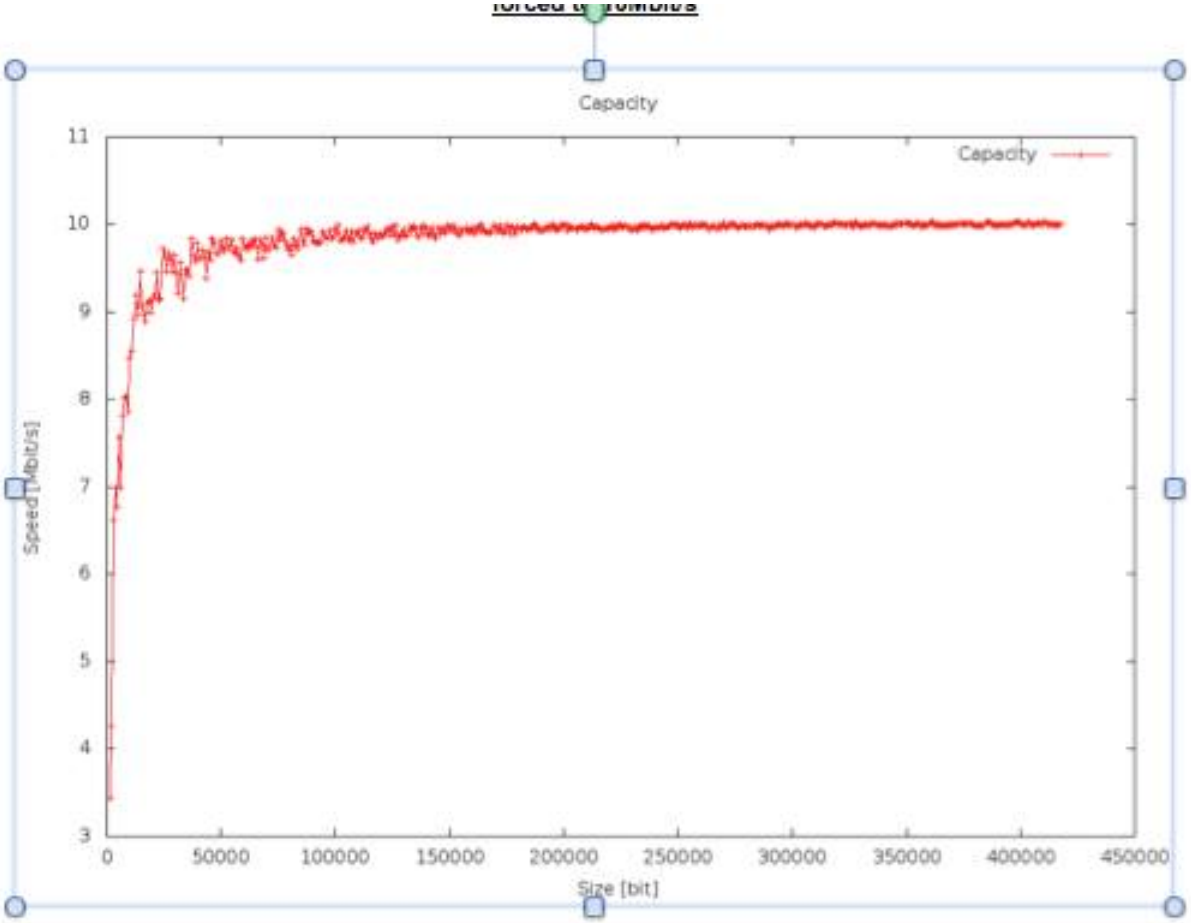


FIGURE 7 : ESTIMATED CAPACITY WITH A DIRECT LINK BETWEEN THE 2 HOST

We can see that the capacity has the same behavior of the graph showed before.

Networking Laboratory III
ANALYSIS OF THE CHARGEN SERVICE

GIULIO FRANZESE 182969 - MARCO RECENTI 180644 - TOMMASO CALIFANO 180808

ACADEMIC YEAR 2013 - 2014
PROF. MELLIA MARCO

Preparation for the experiment :

We configure the two host with the following IP addresses : 172.16.0.1/25 and 172.16.0.2/25. Before proceeding we check the active TCP or UDP connections with " netstat -t -u -n " and we see that there aren't open connections, then we have to activate chargen service by using :

gedit /etc/inetd.conf

/etc/init.d/openbsd-inetd start

Then we disable the offload capabilities of the linecard :

ethtool -K eth0 rx off

ethtool -K eth0 tx off

ethtool -K eth0 gso off

ethtool -K eth0 gro off

```
root@franz:/home/franz# ethtool -k eth0
Offload parameters for eth0:
rx-checksumming: off
tx-checksumming: off
scatter-gather: off
tcp-segmentation-offload: off
udp-fragmentation-offload: off
generic-segmentation-offload: off
generic-receive-offload: off
large-receive-offload: off
```

and we have checked it using ethtool -k, after that we can finally proceed with our experiment.

When a TCP connection from 2 host is started, the first phase of the transmission is called three-way handshake, indicating the necessity of exchanging three segments, to correctly begin it. Let's consider the case in which A want to open a connection with B :

1- A sends a segment to B where SYN flag is set to 1 and the sequence number field contains the value x which specify the initial sequence number of A

2 - B sends a segment to A with both SYN and ACK flags set to 1, the sequence number field y contains the value of B' initial sequence number and the Acknowledgment number containg the x+1 value, conferming the receipt of A' initial seq. no.

3 – A sends a segment to B, ACK flag is set to A and the Acknowledgmente number field contains the y+1 value

The three-way-handshake is necessary because we need to check if the other host is alive, the sequence number is not related to a clock of the network so a synchronization between the two host is needed, and for security reasons, since the sequence number has to be chosen randomly in order to prevent scams of important informations such as passwords.

	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.0.1	172.16.0.2	TCP	74	55001 > chargen [SYN] Seq=
2	0.000512	172.16.0.2	172.16.0.1	TCP	74	chargen > 55001 [SYN, ACK]
3	0.000542	172.16.0.1	172.16.0.2	TCP	66	55001 > chargen [ACK] Seq=

FIGURE 1 : THREE-WAY-HANDSHAKE CAPTURE

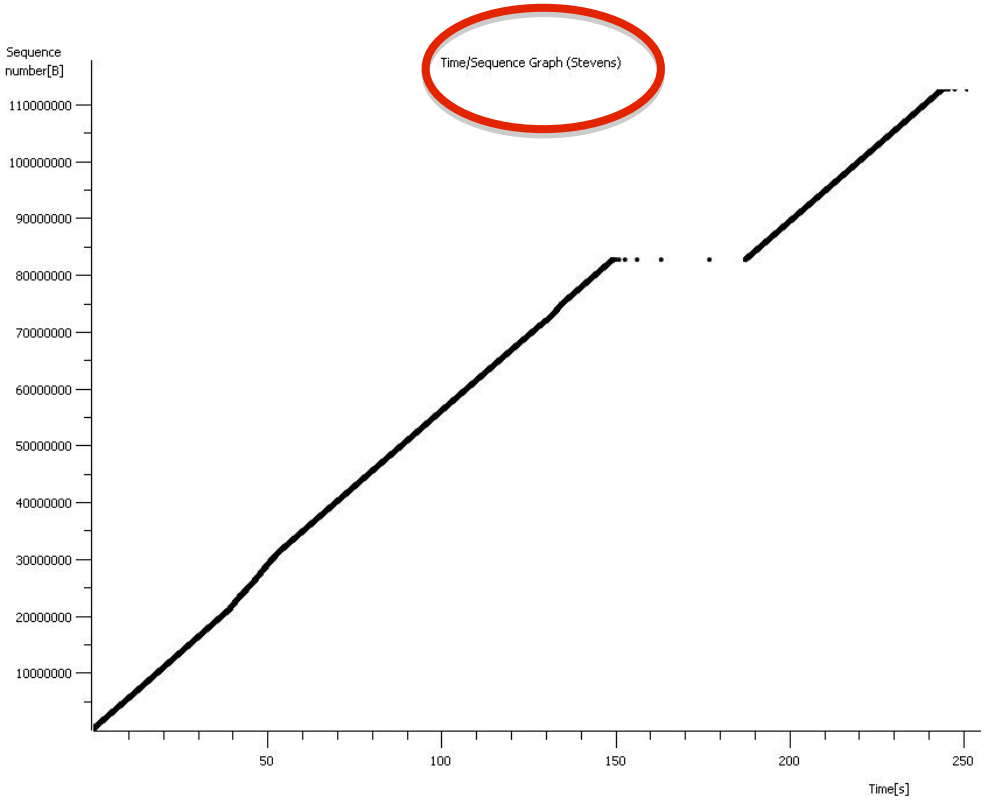


FIGURE 2 : CLIENT ACK NUMBER OVER TIME

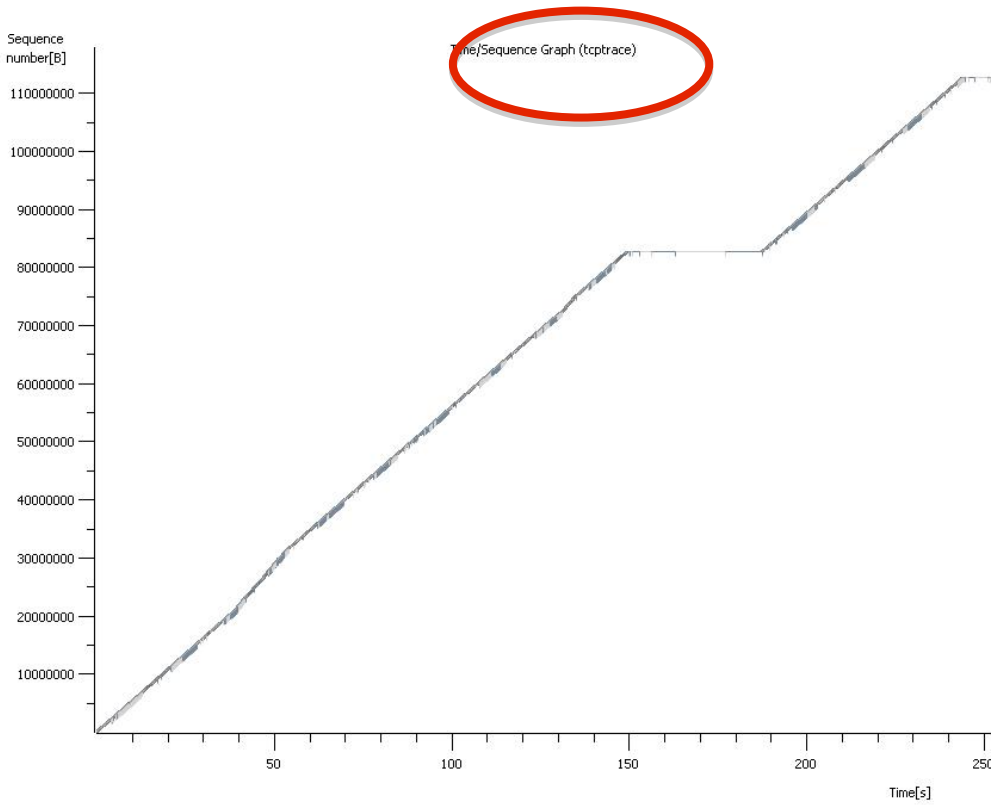


FIGURE 3 : SERVER SEQUENCE NUMBER OVER TIME

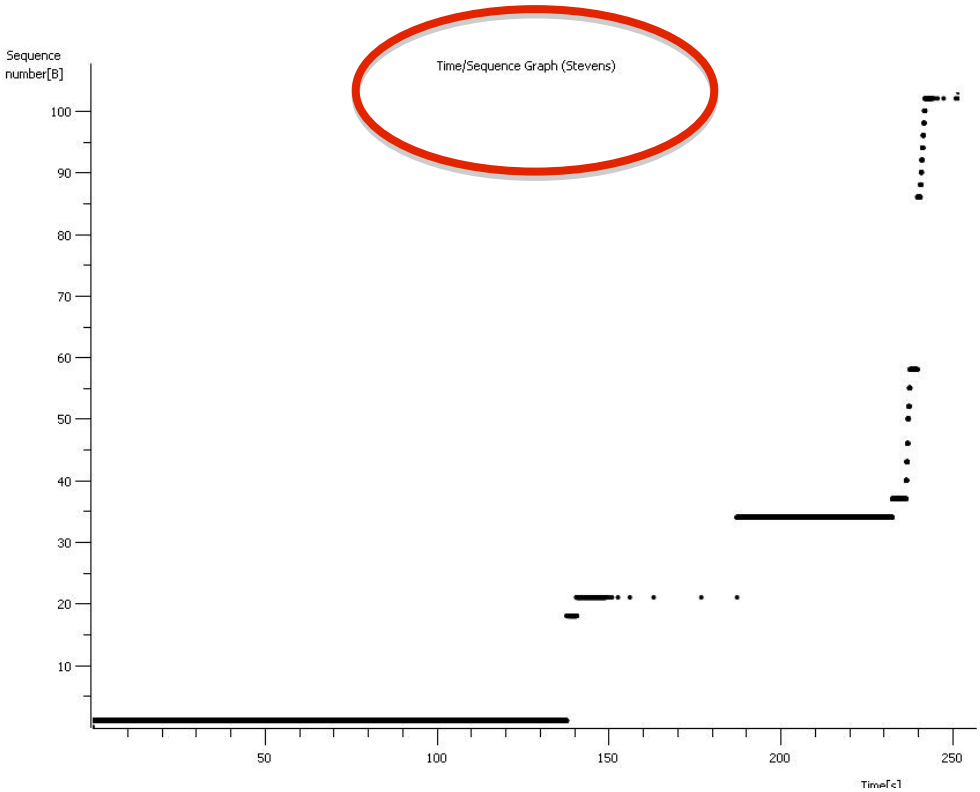


FIGURE 4 : CLIENT SEQUENCE NUMBER OVER TIME

NO

For some reasons client sequence number is not always 1 as we were expecting. This is because in some ACKs, probably due to a chargen's implementation, datas of various lengths are sent, so the client sequence number increase over time. The evolution over time of the server ACK number is the same as client sequence number.

We can also notice that, in the graph of server sequence number, the slope change in some areas; it increases for a while at around 40 second and decreas close to 130 second. This is due to maximizing and minimizing the terminal window : when it is maximazed we don't waste any time for printing the characters generated by chargen on the screen so, in the same time slot, we can process more packets; on the contrary if we minimize it, we decrease the amounts of packets sent over time.

As we were expecting both client ack number and server sequence number increase linearly except for the phase in which we have stopped transmitting pressing the escape sequence; here what we see are just some dots, whose distance increase over time, corresponding to the transmission of two packets :

-TCP ZeroWindow sent from the client to inform server that his receiving window is zero, so it's not ready to receive any packet

-TCP KeepAlive sent periodically from the server in order to keep up the connection between the 2 host; those KeepAlive have length equals to 0

On top of those we can see a "Window Full" packet too, sent from the server to inform that his transmission window is completely full

74807	149.515880	172.16.0.2	172.16.0.1	TCP	674 [TCP Window Full]
74808	149.515901	172.16.0.1	172.16.0.2	TCP	66 [TCP ZeroWindow]
74809	149.731844	172.16.0.2	172.16.0.1	TCP	66 [TCP Keep-Alive]
74810	149.731858	172.16.0.1	172.16.0.2	TCP	66 [TCP ZeroWindow]
74811	150.163853	172.16.0.2	172.16.0.1	TCP	66 [TCP Keep-Alive]
74812	150.163878	172.16.0.1	172.16.0.2	TCP	66 [TCP ZeroWindow]
74813	151.027843	172.16.0.2	172.16.0.1	TCP	66 [TCP Keep-Alive]
74814	151.027866	172.16.0.1	172.16.0.2	TCP	66 [TCP ZeroWindow]
74815	152.759818	172.16.0.2	172.16.0.1	TCP	66 [TCP Keep-Alive]
74816	152.759842	172.16.0.1	172.16.0.2	TCP	66 [TCP ZeroWindow]
74817	156.219743	172.16.0.2	172.16.0.1	TCP	66 [TCP Keep-Alive]
74818	156.219767	172.16.0.1	172.16.0.2	TCP	66 [TCP ZeroWindow]
74819	163.147638	172.16.0.2	172.16.0.1	TCP	66 [TCP Keep-Alive]
74820	163.147662	172.16.0.1	172.16.0.2	TCP	66 [TCP ZeroWindow]
74821	177.003460	172.16.0.2	172.16.0.1	TCP	66 [TCP Keep-Alive]
74822	177.003486	172.16.0.1	172.16.0.2	TCP	66 [TCP ZeroWindow]

FIGURE 5 : CAPTURE FROM WIRESHARK OF ZERO WINDOW AND KEEPALIVE PACKETS

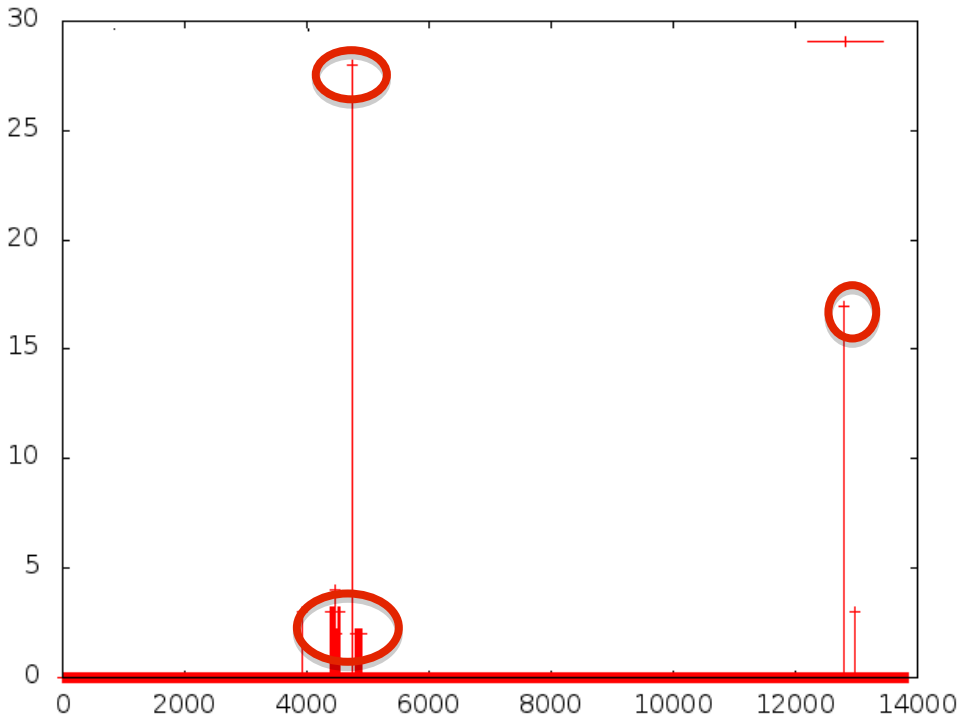


FIGURE 6 : SERVER'S PACKET LENGTH

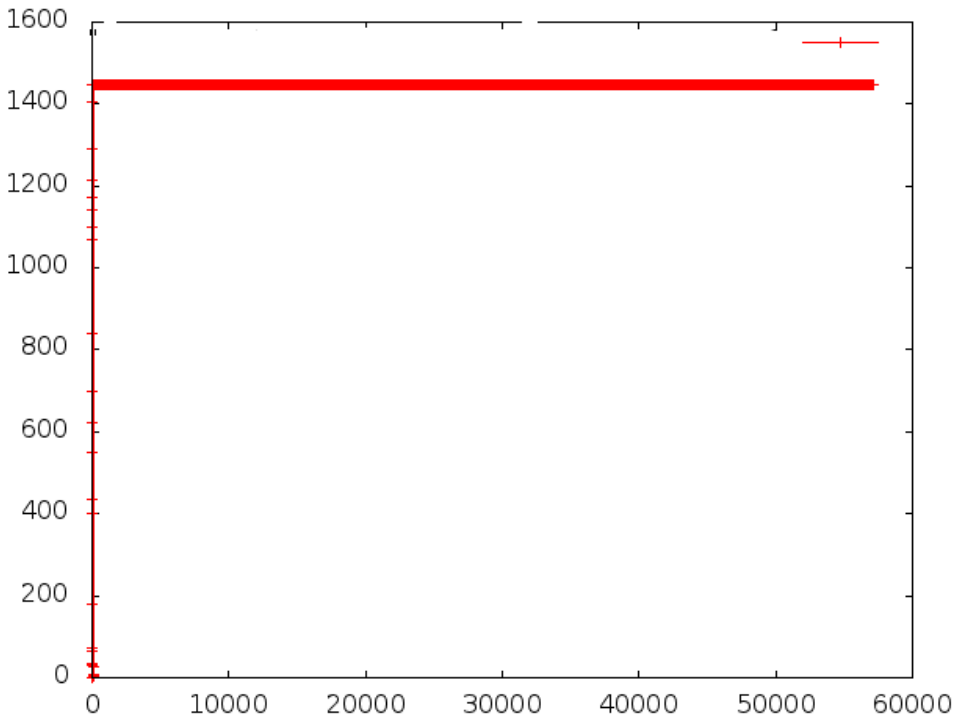


FIGURE 7 : CLIENT'S PACKET LENGTH

We can notice that, a part from for a short amount of time in the early beginning of the transmission, and for the temporal slot in which there are no transmitted packets except for KeepAlive messages, the size of the packets sent by the server is almost constant 1448 B, while the ones sent by the client, since they are just ACK, almost always have size of 0 B except for the ones containing other datas (Len !=0). We have also seen that chargen is implemented in a way that an algorithm decide to send, at certain istants, push bit to tcp layer. Since we have no control on packet size and since no standard for chargen implementation are available we can simply take note of that behaviour taking into account that a push bit force (eventually) the amount of transmitted data to be smaller than the maximum possible.

which data?

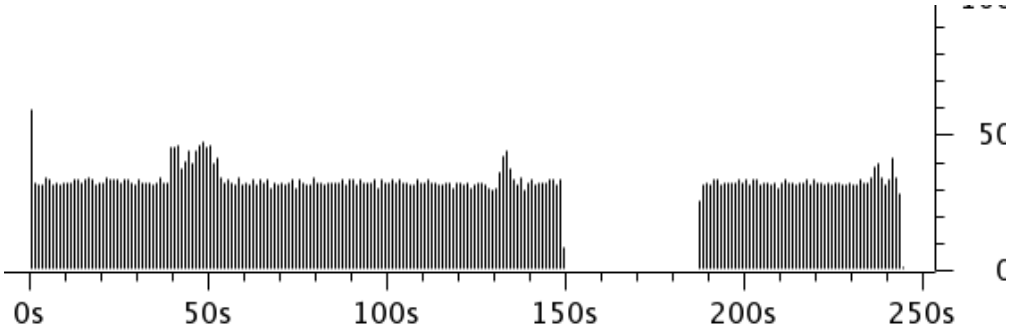


FIGURE : PUSH BIT GRAPH

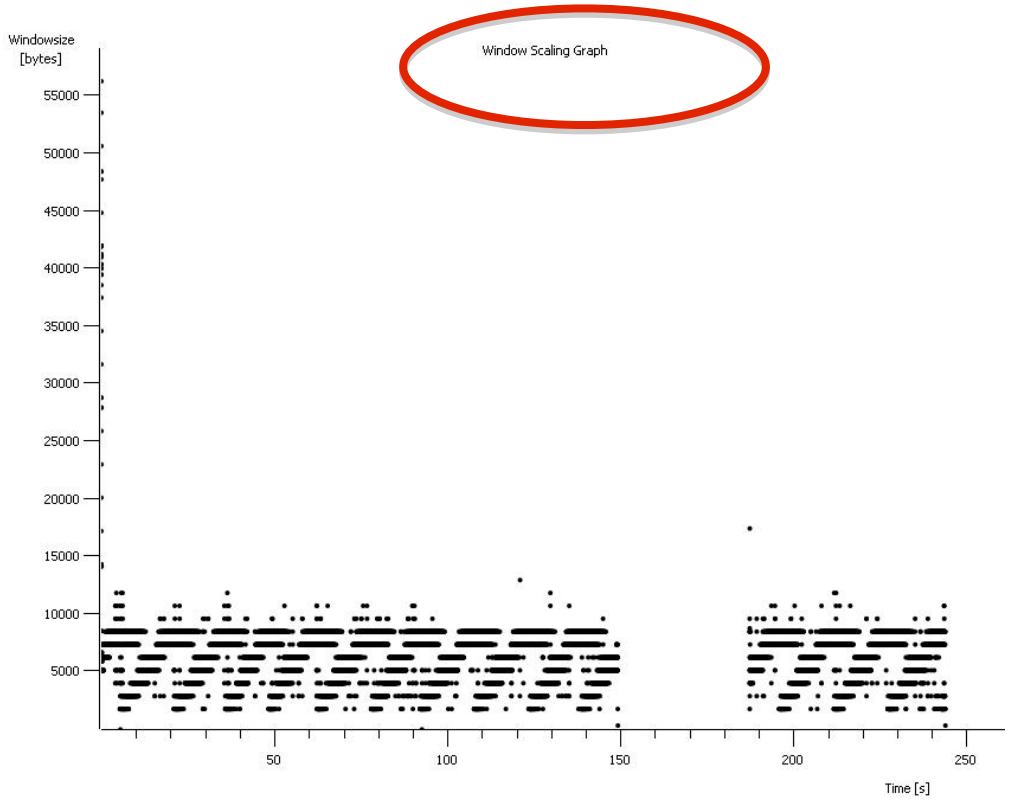


FIGURE 8 : RECEIVER WINDOW SIZE ADVERTISED BY THE CLIENT

The receiver window size advertised by the client is initially set to a very high value which drop almost istantaneously as soon as the transmission starts and then we can notice that it has a saw tooth shape, which is related to TCP's congestion control algorithms.

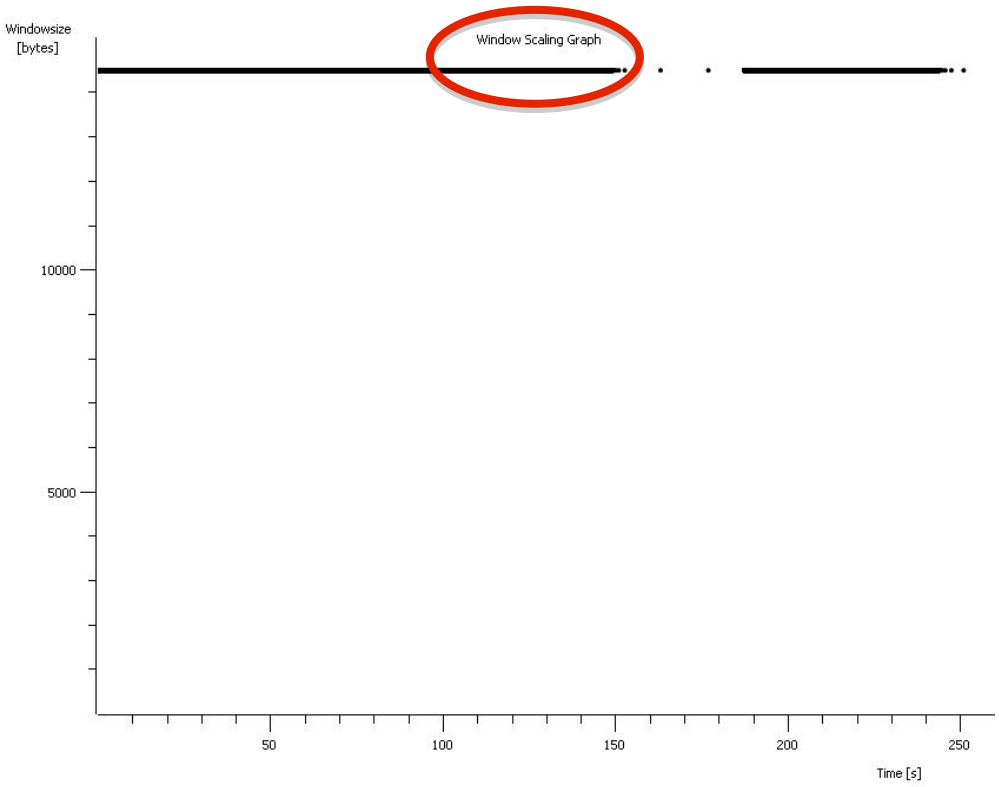


FIGURE 9 : RECEIVER WINDOW SIZE ADVERTISED BY THE SERVER

As we were expecting the receiver window size advertised by the server is at a maximum value, constant over time, since the client is just sending ACKs. In the time when we have stopped the transmission we notice some dots when there is the exchange of ZeroWindow and KeepAlive packets.

Networking Laboratory IV
NTTCP

GIULIO FRANZESE 182969 - MARCO RECENTI 180644 - TOMMASO CALIFANO 180808

ACADEMIC YEAR 2013 - 2014
PROF. MELLIA MARCO

First of all,a consideration : indipendently from the type of transmission that we are using, TCP or UDP, at first there will always be a TCP handshake between the host to ensure the data transmission

1 Transmission between 2 host

1.1 Single flow : H1 sending data to H2 (FULL-DUPLEX)

We have done the experiment both using TCP and UDP. As we were expecting in the case in which we are using UDP to transmit we have more efficiency since there are no ACKs to be sent from H2 back to H1, while using TCP, obviously, the efficiency is reduced compared to the UDP case due to ACKs.

$$TCPefficiency = \eta = \frac{MSS}{(38 + 20 + 20) + MSS} = 0,949 \tag{1}$$

$$UDPefficiency = \eta = \frac{MSS}{(38 + 20 + 8) + MSS} = 0,957 \tag{2}$$

Our experimental results are :

```
root@laboratorio:/home/laboratorio# nttcp -T 172.16.0.2
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608    6.70    0.01     10.0148   8388.6080    2048    305.63  256000.0
1  8388608    7.13    0.07      9.4130    932.0158    5796    812.97   80495.5
root@laboratorio:/home/laboratorio#
```

FIGURE 1 : EFFICIENCY AND GOODPUT USING TCP

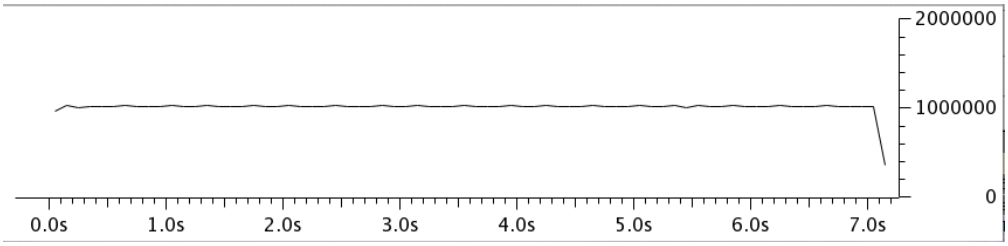


FIGURE 2 : FLOW OF DATA SENT BY H1 TO H2

```
root@laboratorio:/home/laboratorio# nttcp -T -u  172.16.0.2
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608    6.94    0.02      9.6697   2796.0862    2051    295.53   85454.8
1  8388608    7.01    0.02      9.5729   2796.2027    2049    292.28   85375.0
root@laboratorio:/home/laboratorio#
```

FIGURE 3 : EFFICIENCY AND GOODPUT USING UDP

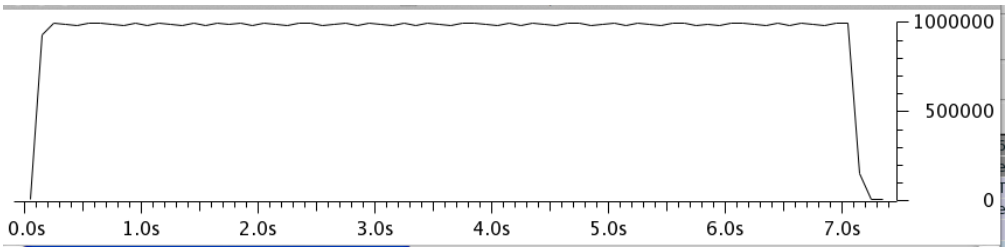


FIGURE 4 : FLOW OF DATA SENT BY H1 TO H2

Of course, in both cases, as predictable, we have no packets loss and UDP's goodput is better than TCP one. We have also noticed that if we increase the amount of transmitted data of the packet using -l 1473, due to the fragmentation we have a lower efficiency

1.2 Bidirectional flow : H1 is sending data data to H2 and viceversa (FULL-DUPLEX)

1.2.1 Both Host using TCP

$$TCPefficiency = \eta = \frac{MSS}{(38+20+20)+90(ACK)+MSS} = 0,9 \tag{3}$$

```
root@laboratorio:/home/laboratorio# nttcp -T -r 172.16.0.2 & nttcp -T 172.16.0.2
[2] 4122
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608    7.20    0.01      9.3251   8388.6080    2048    284.58  256000.0
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608    7.34    0.08      9.1478    882.9649    5796    790.07   76259.1
1  8388608    7.09    0.01      9.4698   8388.6080    2048    288.99  256000.0
1  8388608    7.42    0.08      9.0460    882.9649    5785    779.79   76114.4
[2]+  Done                    nttcp -T -r 172.16.0.2
```

FIGURE 5 : BIDIRECTIONAL TCP GOODPUT AND EFFICIENCY

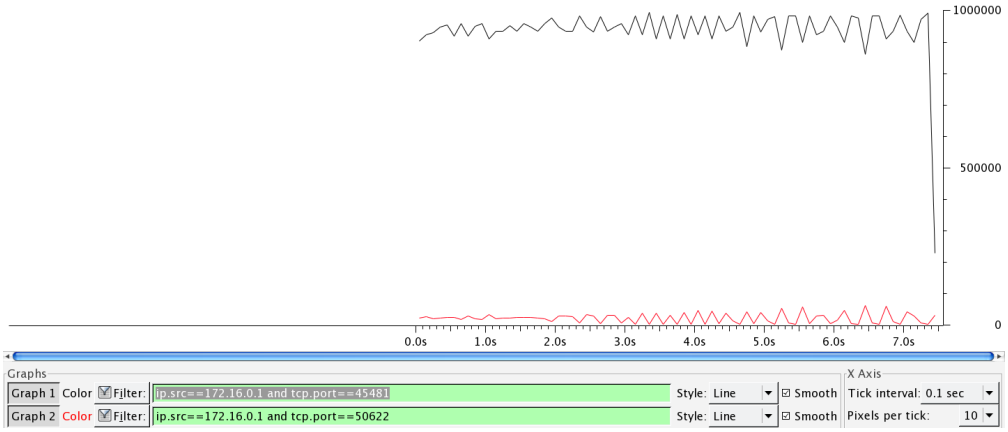


FIGURE 6 : FLOWS SEEN FROM H2

The black line represent the TCP data and the red line represent the ACKs. As we can see some of the channel's capacity is taken to transmit ACKs

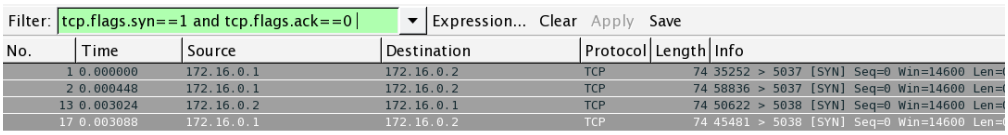


FIGURE : APERTURA CONNESSIONI CON DOPPIO FLUSSO TCP

From the capture we notice that there are 2 TCP connections opened on different ports, and this is the reason why don"t have any piggybacking.

1.2.2 Both Host using UDP

This is the case in which we have the best efficiency for both host since we don't need to send any ACK on channels, we have no packet loss and the time slot in which the data

transmission is up is the shortest one. The efficiency in this case is equal to the one in which we had only the single UDP flow.

```
root@laboratorio:/home/laboratorio# nttcp -T -r -u  172.16.0.2 & nttcp -T -u 172
.16.0.2
[2] 4221
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608    7.01    0.02      9.5730   2796.0862    2049    292.29   85371.4
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608    6.94    0.02      9.6700   2796.0862    2051    295.54   85454.8
1  8388608    7.01    0.02      9.5728   3355.2754    2049    292.28  102444.9
1  8388608    6.94    0.02      9.6697   2796.0862    2051    295.53   85454.8
[2]+  Done                    nttcp -T -r -u 172.16.0.2
root@laboratorio:/home/laboratorio#
```

FIGURE 7 : BIDIRECTIONAL UDP GOODPUT AND EFFICIENCY

1.2.3 H1 using TCP and H2 using UDP

We expect no packet loss and that UDP flow ends before TCP one who decrease his speed because he find "busy" the channel on which the ACKs need to be sent, since there is the UDP flow running. We can also notice that the time difference between then endings of the two transmission is really short.

```
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608    7.10    0.01      9.4522   8388.6080    2048    288.46  256000.0
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608    7.21    0.02      9.3013   2796.0862    2049    283.99   85371.4
1  8388608    7.38    0.08      9.0877    882.9649    5793    784.47   76219.7
root@laboratorio:/home/laboratorio# 1  8388608    7.14    0.02      9.3935   279
6.0862    2051    287.09   85454.8
```

FIGURE 8

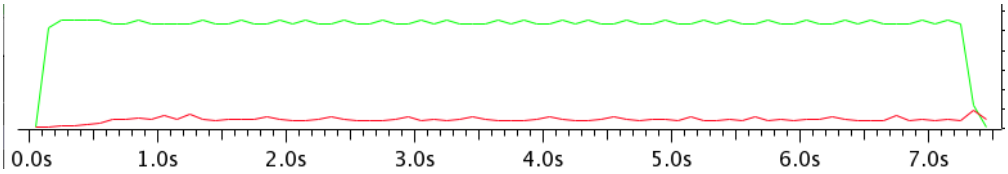


FIGURE 9 : FLOWS FROM H1 POINT OF VIEW

The green line represent the UDP data and the red line represent the ACKs

1.3 H2 and H3 are receiving datas from H1

As we were expecting we have no loss and the capacity is almost equally splitted between the two flows in both cases (TCP and UDP only).

1.3.1 H1 is using TCP

```
root@laboratorio:/home/laboratorio# nttcp -T 172.16.0.2 & nttcp -T 172.16.0.3
[2] 4281
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608   12.91    0.01      5.1975   8388.6080    2048    158.61  256000.0
1  8388608   13.63    0.08      4.9241    882.9533    5798    425.43   76284.5
root@laboratorio:/home/laboratorio#      Bytes  Real s   CPU s Real-MBit/s  CPU-
MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608   13.86    0.01      4.8427   8388.6080    2048    147.79  256000.0
1  8388608   14.26    0.08      4.7066    882.9649    5799    406.70   76298.6
```

FIGURE 10

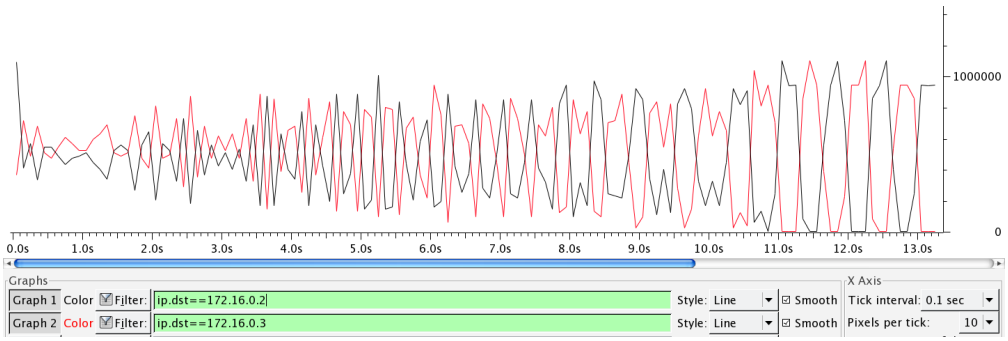


FIGURE 11 : FLOWS ON THE CHANNELS FROM H1 POINT OF VIEW

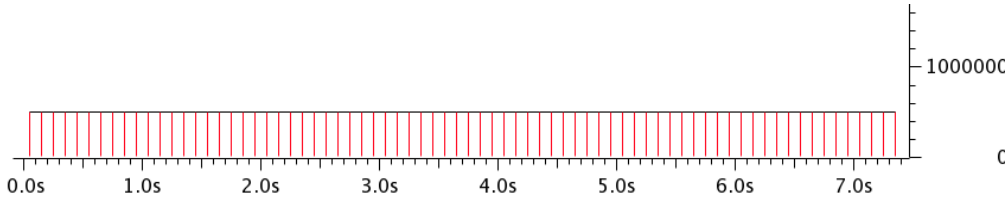


FIGURE 12 : AVERAGE TRANSMISSION ON THE CHANNELS

Using the option 'average' on the plots of the two flows we can appreciate that, in media, H1 split his transmission equally between H2 and H3.

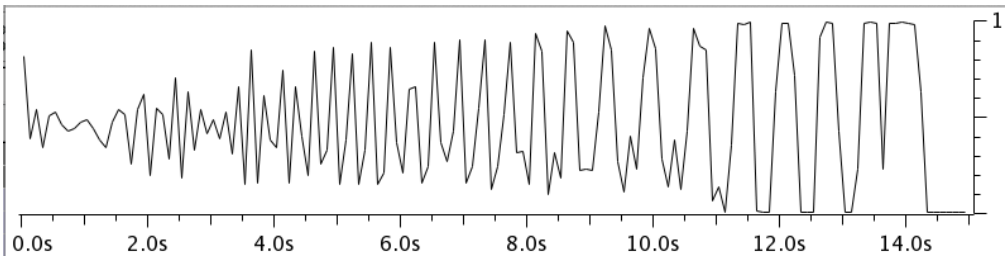


FIGURE 13 : FLOW OF DATA RECEIVED AT H2 FROM H1

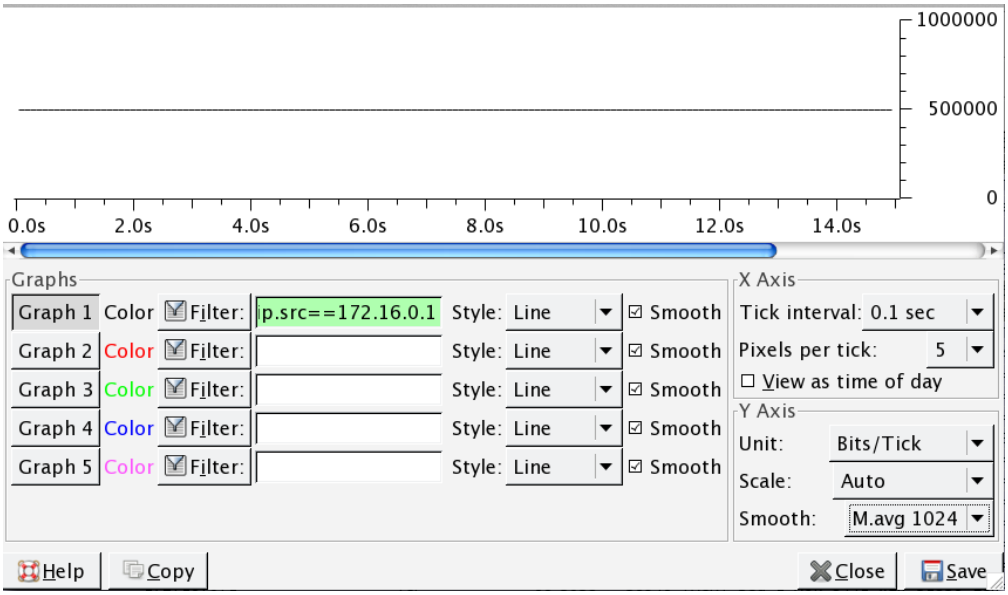


FIGURE 14 : AVARAGE FLOW OF DATA RECEIVED AT H2 FROM H1

1.3.2 H1 is using UDP

```
root@laboratorio:/home/laboratorio# nttcp -T -u 172.16.0.2 & nttcp -T -u  172.16
.0.3
[2] 4393
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608   13.88    0.02      4.8359   2796.0862    2051    147.80   85454.8
1  8388608   13.99    0.02      4.7962   2796.0862    2049    146.44   85371.4
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608   13.92    0.02      4.8216   2796.0862    2051    147.36   85454.8
1  8388608   13.92    0.02      4.8197   2796.0862    2049    147.16   85371.4
[2]+  Done                    nttcp -T -u 172.16.0.2
root@laboratorio:/home/laboratorio#
```

FIGURE 15

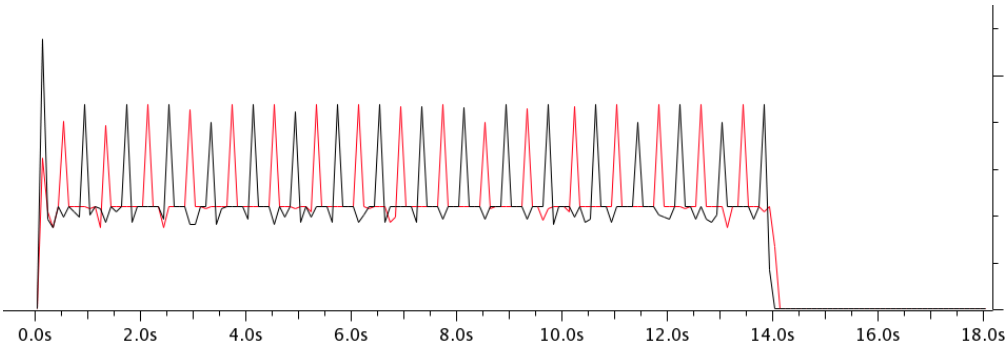


FIGURE 16 : TRANSMITTED DATA FROM H1

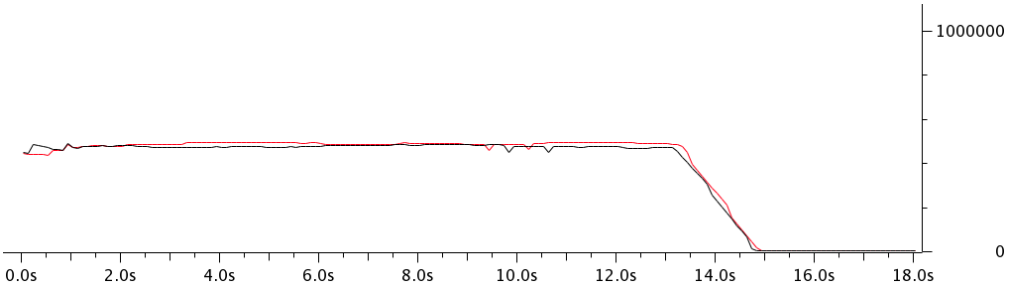


FIGURE 17 : AVARAGE TRANSMITTED DATA FROM H1

Also in this case each of the two flows occupy, in media, half of the channel capacity.

1.3.3 H1 is using TCP to H2 and UDP to H3

On the contrary with our previsions, in this case, the TCP flow ends before the UDP one and has a better efficiency. This is probably related to the decision alghorithms of the switch that give priority, for unknown reason, to the TCP flow.

```
root@laboratorio:/home/laboratorio# nttcp -T 172.16.0.2 & nttcp -T -u  172.16.0.
3
[2] 4334
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608    9.09    0.01      7.3834   8388.6080    2048    225.32  256000.0
1  8388608    9.44    0.08      7.1069    882.9649    5793    613.48   76219.7
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608   14.07    0.02      4.7701   2796.0862    2051    145.79   85454.8
1  8388608   14.14    0.02      4.7464   2796.0862    2049    144.92   85371.4
[2]+  Done                    nttcp -T 172.16.0.2
root@laboratorio:/home/laboratorio#
```

FIGURE 18

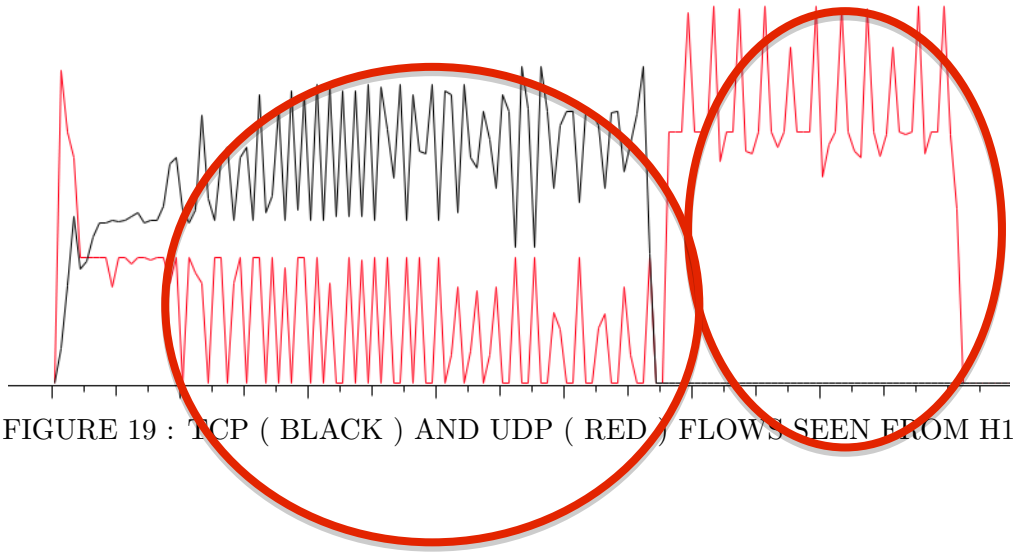


FIGURE 19 : TCP (BLACK) AND UDP (RED) FLOWS SEEN FROM H1

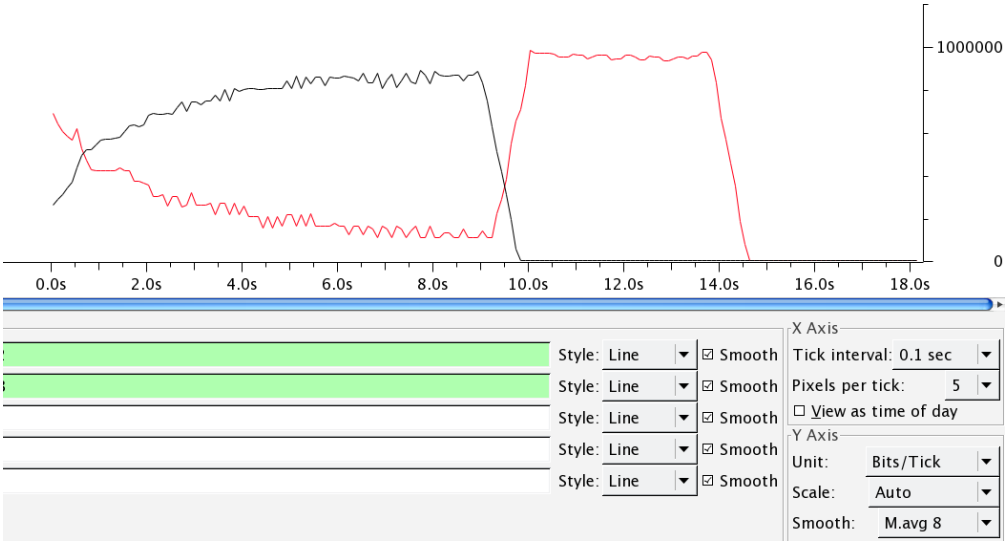


FIGURE 20 : AVARAGE TCP AND UDP FLOWS SEEN FROM H1

NO

In this figure can also appreciate the slow-start phase of TCP's congestion control algorithms.

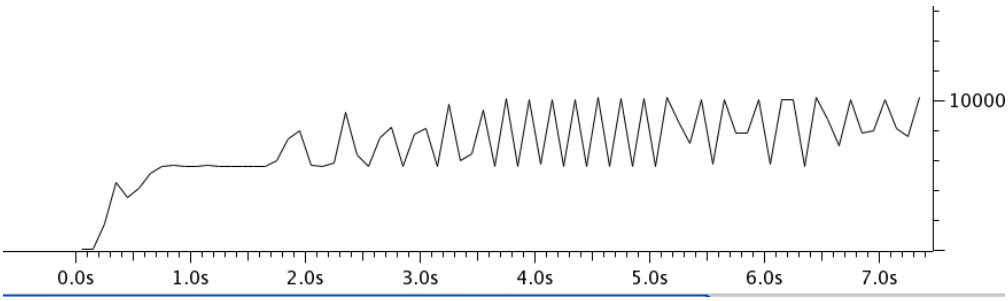


FIGURE 21 : FLOW OF DATA RECEIVED FROM H2

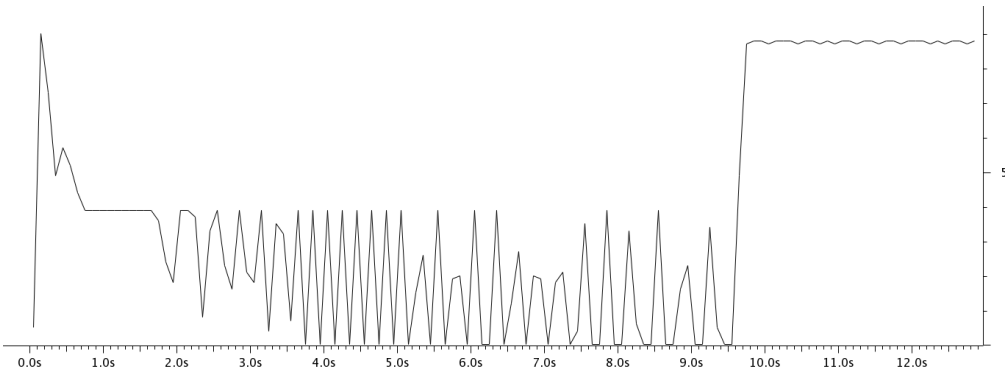


FIGURE 22 : FLOW OF DATA RECEIVED FROM H3

1.4 H2 and H3 are sending datas to H1

1.4.1 Both Host using TCP

As we were expecting we have no loss since TCP is a reliable service, but of course we pay this in terms of goodput since we need much more time to transmit the datas.

```
root@laboratorio:/home/laboratorio# nttcp -T -r  172.16.0.2 & nttcp -T  -r 172.1
6.0.3
[2] 4446
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608   13.24    0.08      5.0685    882.9649    5303    400.52   69772.6
1  8388608   13.19    0.01      5.0873   8388.6080    2048    155.25  256000.0
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608   14.26    0.08      4.7076    882.9649    5307    372.28   69825.3
1  8388608   14.22    0.01      4.7199   8388.6080    2048    144.04  256000.0
[2]+  Done                    nttcp -T -r 172.16.0.2
root@laboratorio:/home/laboratorio#
```

FIGURE 23

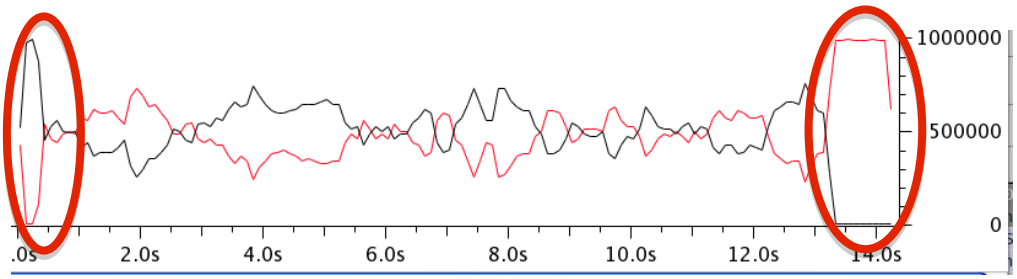


FIGURE 24 : FLOWS OF DATA RECEIVED FROM H1

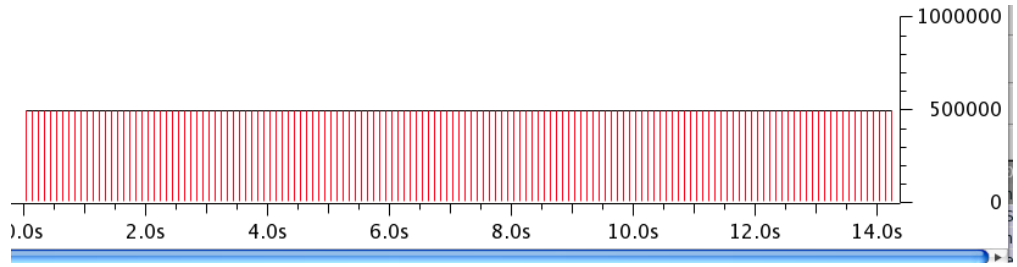


FIGURE 25 : AVARAGE FLOWS OF DATA RECEIVED FROM H1

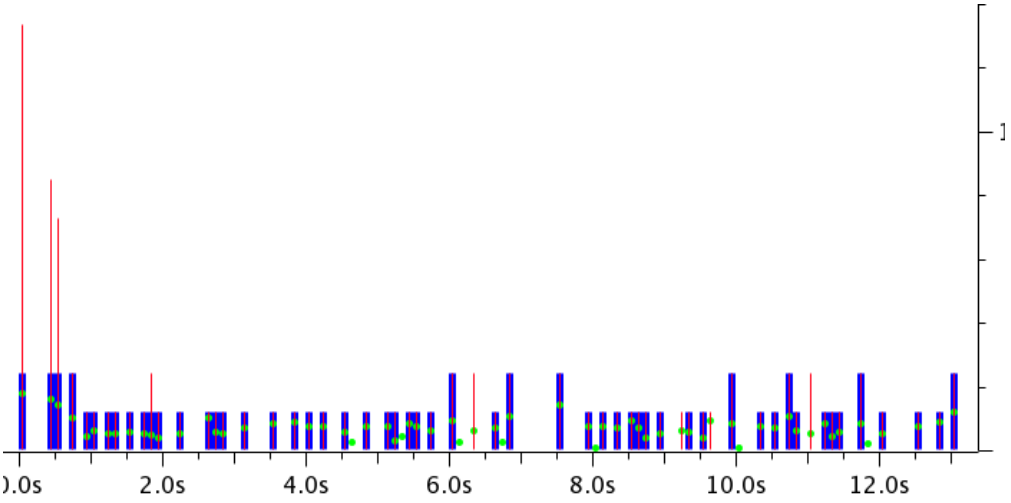


FIGURE : 26

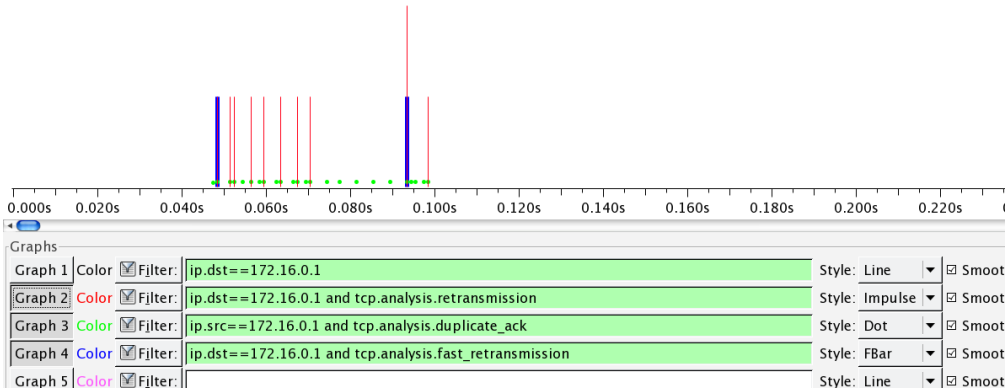


FIGURE 27 : GRAPH OF DUPLICATED ACKs, RETRANSMISSION AND FAST RETRANSMISSION

Of course in this case, we do have some packet loss while transmitting, but since TCP is a reliable service what we can see that there are duplicated ACKs, retransmissions and fast retransmission.

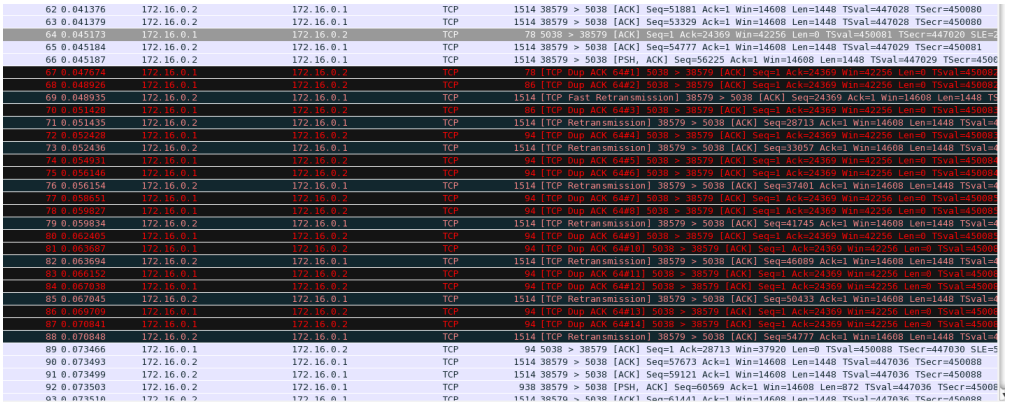


FIGURE 28 : WIRESHARK'S CAPTURE OF DUPLICATE ACKs AND FAST RETRANSMISSION

1.4.2 Both Host using UDP

In this case we were expecting a packet loss probability of p=0.5 since we have a bottleneck with 2 flows of 10Mb and a channel with capacity of 10Mb. What we notice is that this probability is way higher, in fact only a short amount of packets are sent correctly; this is because we need to fragment every packet of dimension 4096 B (standard) into three for both flows,and to obtain a correct packet at the receiver we also need to reassamble those, but the switch's decisor alghorithm is implemented in a way that this never happens. We may also say that the probability of receiving the three packets it's not even $\frac{1}{8}$ since the three events are not independent.So what we receive are the packets processed until the switch's buffer is full.

```
root@laboratorio:/home/laboratorio# nttcp -T -r -u  172.16.0.2 & nttcp -T  -r -u
  172.16.0.3
[2] 4560
     Bytes   Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l    32768     8.94    0.00      0.0293  26214.4000       9      1.01  900000.0
     Bytes   Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l    32768     8.94    0.00      0.0293  26214.4000       9      1.01  900000.0
1  8388608     6.94    0.02      9.6701   2796.0862    2051    295.54   85454.8
1  8388608     6.94    0.02      9.6697   2796.0862    2051    295.53   85454.8
root@laboratorio:/home/laboratorio#
```

FIGURE 29

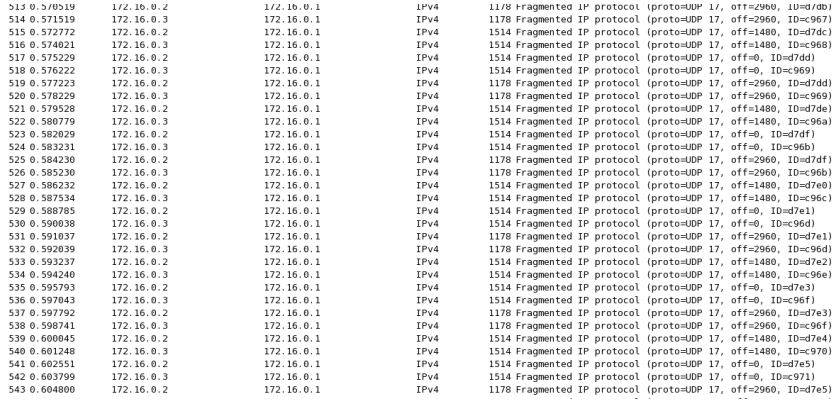


FIGURE 30 : CAPTURE FROM WIRESHARK

We can see that pratically all packets are lost since they are not reassembled at the receveir.

1.4.3 H3 is using UDP and H2 is using TCP

The two transmissions start togheter and each of them has a bit rate equal to 10 Mbit/s. The switch receives a flow of data at 20 Mbit/s, while the channel which links switch and H3 has a bitrate of 10 Mbit/s. This is the reason why there are packets lost

```
root@laboratorio:/home/laboratorio# nttcp -T -r  172.16.0.2 & nttcp -T  -r -u  1
72.16.0.3
[2] 4498
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  5771264    7.03    0.02      6.5712   2885.4517    1410    200.68   88119.5
1  8388608    6.94    0.02      9.6698   2796.0862    2051    295.53   85454.8
root@laboratorio:/home/laboratorio#      Bytes  Real s   CPU s Real-MBit/s  CPU-
MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608   13.34    0.08      5.0312    882.9649    5562    416.98   73180.4
1  8388608   13.18    0.01      5.0905   8388.6080    2048    155.35  256000.0
```

FIGURE 31

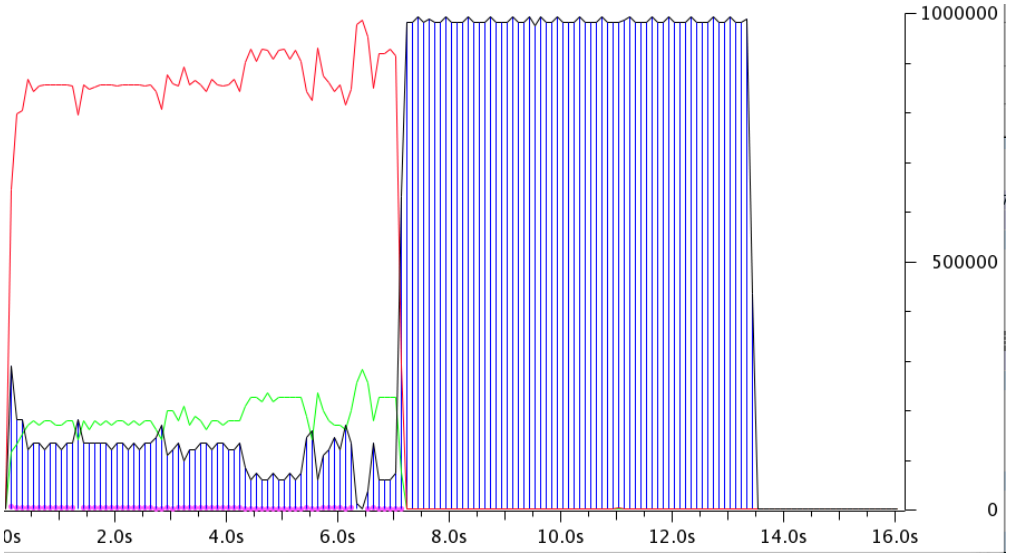


FIGURE 32 : FLOWS OF DATA RECEIVED BY H1

In this case the switch 'privileges', as we were expecting, the UDP flow but TCP'one is still occupying some of the channel capacity while the UDP flow is running, and we can see that there are a lot of duplicated ACKs in this phase. The green line represent the UDP flow of packets correctly reassembled at the receiver, but we also have to consider that every reassmbled packet is made out of 3 packet.

1.5 Full Mesh Scenarios

1.5.1 Everybody is using TCP

```
root@laboratorio:/home/laboratorio# nttcp -T -r 172.16.0.2 & nttcp -T  -r  172.1
6.0.3
[2] 4633
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608   15.18    0.08      4.4219    838.8084    5520    363.72   68995.7
1  8388608   15.09    0.01      4.4462   8388.6080    2048    135.69  256000.0
root@laboratorio:/home/laboratorio#      Bytes  Real s   CPU s Real-MBit/s  CPU-
MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608   15.69    0.08      4.2762    838.8084    5525    352.05   69058.2
1  8388608   15.63    0.01      4.2923   8388.6080    2048    130.99  256000.0
```

FIGURE 33 : H1 POINT OF VIEW

FIGURE 34 : FLOWS OF DATA RECEIVED FROM H1

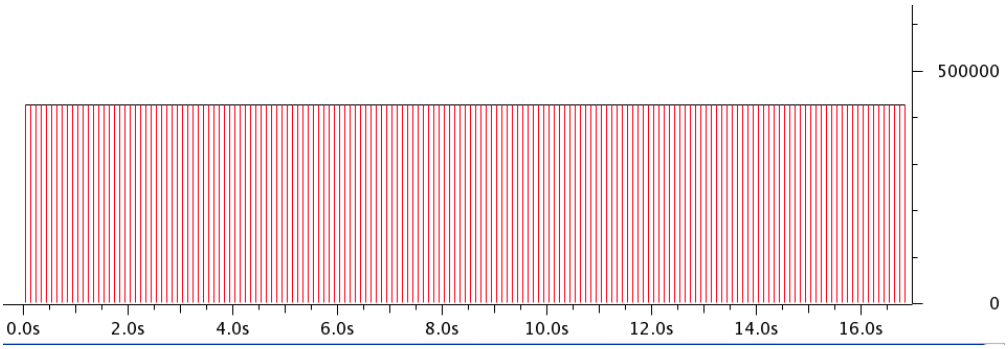


FIGURE 35 : AVARAGE FLOWS OF DATA RECEIVED FROM H1

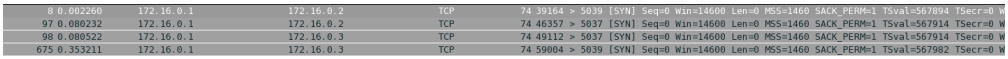


FIGURE 36 : CAPTURE OF THE OPENING OF THE CONNECTIONS

1.5.2 Everybody is using UDP

```
root@laboratorio:/home/laboratorio# nttcp -T -r -u 172.16.0.2 & nttcp -T  -r -u
172.16.0.3
[2] 4691
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  3616768   10.57    0.01      2.7385   3616.7680     884     83.67  110500.0
1  8388608    8.49    0.02      7.9003   2796.0862    2051    241.45   85454.8
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  4255744   13.92    0.01      2.4453   2837.1627    1040     74.70   86666.7
1  8388608   13.92    0.02      4.8205   2796.0862    2051    147.32   85454.8
[2]+  Done                    nttcp -T -r -u 172.16.0.2
root@laboratorio:/home/laboratorio#
```

FIGURE 37 : H1 POINT OF VIEW

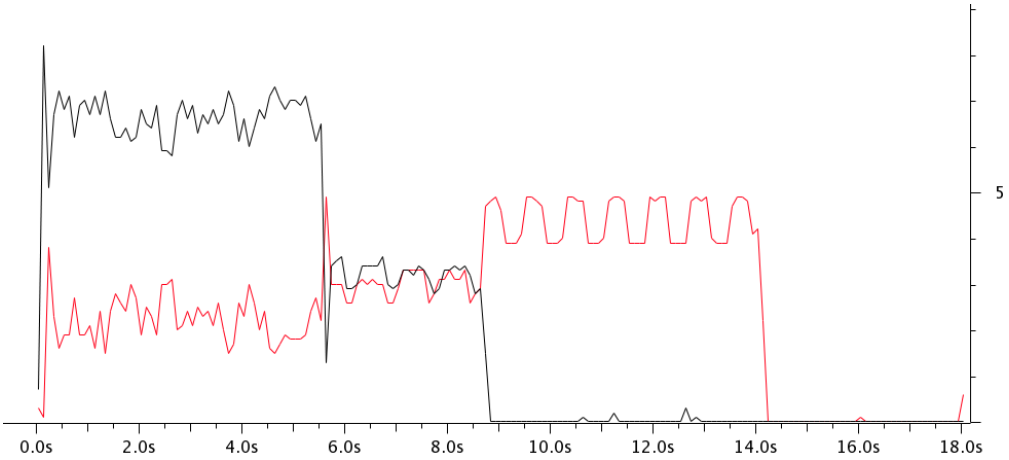


FIGURE 38 : FLOWS OF DATA RECEIVED FROM H1

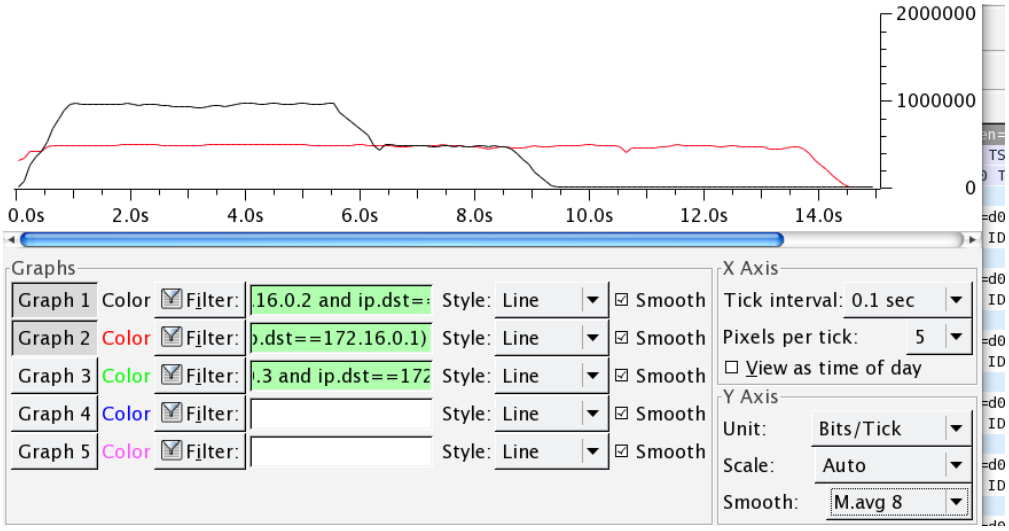


FIGURE 39

We can see that, for some time slots, if we sum the total bitrate of the two flows of data it exceed the capacity of the channel and this is the reason why we have packets losses.

1.5.3 Every host sends and receive 1 TCP flow and 1 UDP flow

```
root@laboratorio:/home/laboratorio# nttcp -T  172.16.0.2 & nttcp -T -u 172.16.0.
3
[2] 4747
     Bytes  Real s   CPU s Real-MBit/s  CPU-MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608    7.71    0.02      8.7080   2796.0862    2051    266.14   85454.8
1  8286208    7.81    0.02      8.4901   2761.9543    2024    259.22   84329.8
root@laboratorio:/home/laboratorio#      Bytes  Real s   CPU s Real-MBit/s  CPU-
MBit/s   Calls  Real-C/s   CPU-C/s
l  8388608   15.75    0.01      4.2614   8388.6080    2048    130.05  256000.0
1  8388608   15.79    0.08      4.2503    882.9649    5740    363.54   75522.3
```

FIGURE 40 : H1 Point of view

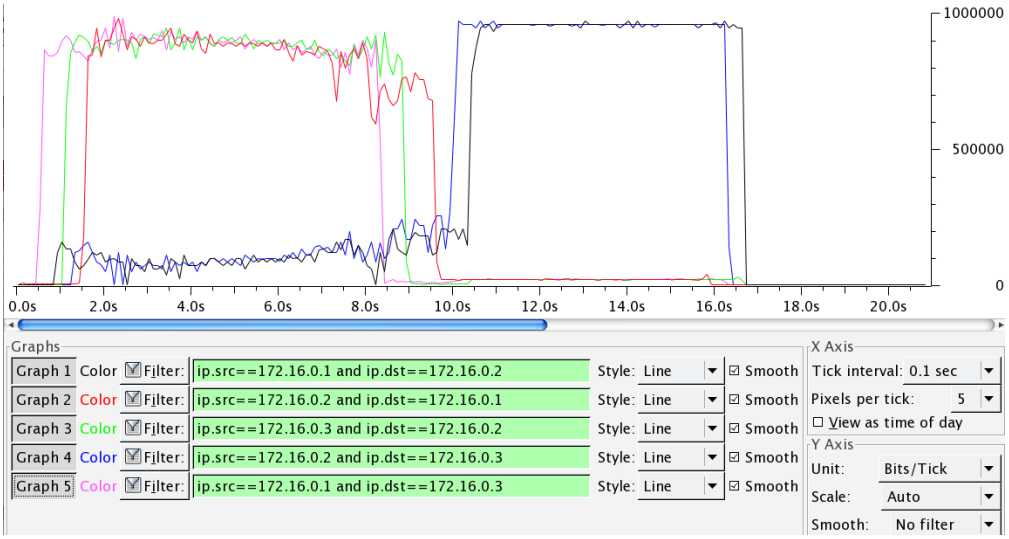


FIGURE 41 : PLOT OF THE FLOWS MERGING THE CAPTURES DONE FROM EVERY HOST

In this particular configuration we have a very low loss and describing the network's traffic is really complicated.