

POLITECNICO di TORINO

Dipartimento di Elettronica e Telecomunicazioni

COURSE SYLLABUS

NETWORK LABORATORY

Marco Mellia

E-mail: marco.mellia@polito.it

Martino Trevisan

E-mail: martino.trevisan@polito.it

May, 2022

Table of content

Networking Lab 1 – Basic ping.....	4
1. Linux and useful commands.....	4
2. Networking related commands.....	4
3. Configuration of the testbed	6
4. Report the configuration of the testbed	6
5. Connectivity check – basic ping.....	8
Networking Lab 2 – Introduction to Quagga and Cisco IOS syntax.....	9
1. The Zebra process.....	10
2. Interface configuration.....	10
3. Routing configuration.....	10
4. Visualization and management	11
Networking Lab 3 – Introduction to Wireshark.....	12
1. Wireshark filters	12
2. Simple capture session	12
3. Connectivity check – basic ping.....	13
1. Arp table management.....	13
4. Arp requests and link failure	14
5. Connectivity check – Advanced ping	14
Networking Lab 4 – Analysis of TCP	15
1. Running some service on the host	15
2. Using services from a client.....	16
3. Analysis of the ECHO service	16
4. Analysis of the CHARGEN service	18
How to make the plots	Errore. Il segnalibro non è definito.
Networking Lab 5 – Analysis of the nmap command.....	20
1. nmap.....	20
2. Using nmap.....	20
Networking Lab 6 – analysis of normal web browsing.....	21
1. Normal Web Browsing analysis	21
2. The Basic HTTP GET/Response interaction	21
Networking Lab 7 – Performance test on Ethernet links	24
5. nttcp	25

6. Iperf	26
7. Computing the goodput	27
Networking Lab 7bis – Performance test with off-the-shelf hardware	31
Experiment to consider	33
Networking Lab 8 – Performance test on WiFi	34
1. Configuration of the host	35
2. Configuration of the Access Point	35
3. Configure the WiFi USB adapter.....	35
4. Check that the WiFi link and interface are up and running.....	36
5. Performance test over WiFi.....	37
Networking Lab 9 – Impact of packet loss on Throughput	38
1. Emulating wide area network delays	39
2. TBF – Token Buffer Filter	40
3. TCP Probe	41
4. Impact of RTT and Packet Loss on Congestion Control.....	42
Networking Lab 10 – Passive monitoring	46
1. Tstat.....	46
2. Monitored objects.....	46
3. Analysis of TCP logs	47
4. Example of scripts.....	48
5. SOLUTIONS	49

Networking Lab 1 – Basic ping

The goal of this laboratory is to get used to the tools and command line mechanisms to properly configure a PC. Students will work in groups of 3 people. Each group will use three PCs, one Ethernet switch, three UTP cables and three pen drives with installed Linux. PCs must be bootstrapped from the laboratory pen drive.

1. Linux and useful commands

Linux is a UNIX based O.S. Most of the lab will require working with the Linux command line. Here is a brief list of useful commands. It is not intended to be a complete tutorial. In general, it is always useful to check the online manual that can be obtained by using the `man` command. There are plenty of tutorials you can find in the Internet to get more information.

`man <command>`: show the man page of the `<command>`. For example

`man man` shows the man page of the `man` command.

You can go up and down with the cursor arrows or page up/down keys. To exit, press “q” as quit. To look for some keyword, use the “/” – slash- then insert the string you are looking for.

The laboratory pen drive has been configured so that the user is logged in as a normal user, with account name “**laboratorio**”. This is a non-privileged user, i.e., she has no right to change the system configuration. Thus, you have to run them as “**superuser**”, or “**root**” to run most of the commands that require privileged access. This can be done by prepending the command “**sudo**” – do as superuser- to the command you would like to run. The system will ask for the user password before granting access as super user. The password is “**studente**”. For example

```
sudo ifconfig eth0 172.16.1.1 netmask 255.255.255.0
```

runs `ifconfig` as *superuser* instead of *laboratorio* user.

2. Networking related commands

- **ethtool** - Display or change Ethernet card settings. This command allows you to check and change the **physical layer configuration** of an Ethernet interface. It has a lot of options, which depends on the actual hardware and driver of the Ethernet interface.
- **ifconfig** - configure a network interface. Some useful options
 - `ifconfig`: show the configuration of the interfaces that are actually up and running
 - `ifconfig -a`: show the configuration of **all** interfaces in the system, including those that are down (not currently active)
 - `ifconfig <ethX>`: show the configuration of interface X, X=0,1,2,3,... depending on how many interfaces the system has
 - `ifconfig <ethX> <IP_ADDRESS> [netmask <MASK>] [broadcast <BROADCAST>] [MTU <MTU>]`: configure the interface `ethX` with `IP_ADDRESS`, netmask `MASK`, broadcast `BROADCAST` and MTU `MTU`.
 - `[same as before] ifconfig <ethX> <IP_ADDRESS/MASK> [broadcast <BROADCAST>] [MTU <MTU>]`: configure the interface `ethX` with `IP_ADDRESS`, netmask `MASK`, broadcast `BROADCAST` and MTU `MTU`.
 - `ifconfig <ethX> up|down`: put the interface up and running (active) or down (not active).

- **route:** show / manipulate the IP routing table. Route manipulates the kernel's IP routing tables. Its primary use is to set up static routes to specific hosts or networks via an interface after it has been configured with the `ifconfig` program. When the `add` or `del` options are used, route modifies the routing tables. Without these options, route displays the current contents of the routing tables.
 - `route add default gw <GW_ADDR>`: add the default gateway at `GW_ADDR`
 - `route add -net <NET_ADDR> Netmask <MASK> gw <GW_ADDR>`: add the `NET_ADDR/MASK` route via `GW_ADDR`.
- **arp:** manipulates or displays the kernel's IPv4 network neighbor cache. It can add entries to the table, delete one or display the current content.
 - `arp -n`: show the arp table state without resolving addresses
 - `arp -d IP_ADDR`: delete the entry in the arp table referring to `IP_ADDR`
- **netstat:** Print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships
 - `netstat -t -n`: show the TCP (-t) connections status using numerical (-n) addresses instead of trying to determine symbolic host, port or user names
 - `netstat -u`: show the UDP (-u) connections status
 - `netstat -l`: show only listening sockets. (These are omitted by default.)
- **ping:** send ICMP ECHO_REQUEST to network hosts `IP_ADDRESS`. It uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway. ECHO_REQUEST datagrams ("pings") have an IP and ICMP header, followed arbitrary number of "pad" bytes used to fill out the packet.
 - `ping -s <size> -c <count> -i <interval> IP_ADDR`: send count ICMP ECHO_REQUESTs separated by interval seconds, each of size size to `IP_ADDR`
- **traceroute** `<IP_ADDR>`: print the route packets trace to network host `IP_ADDR`. It tracks the route packets taken from an IP network on their way to a given host. It utilizes the IP protocol's time to live (TTL) field and attempts to elicit an ICMP `TIME_EXCEEDED` response from each gateway along the path to the host.

Which of the previous commands must be run as superuser?

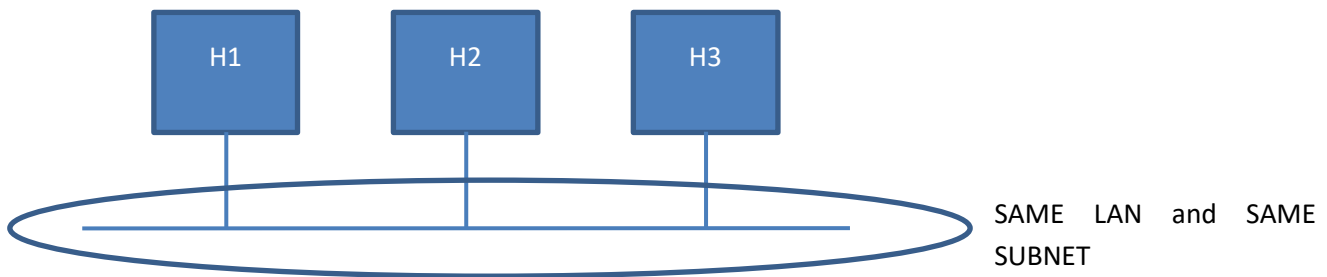
3. Configuration of the testbed

Each group must configure its testbed, i.e., the three PCs must be properly configured and interconnected to create and setup a LAN.

1. **Disconnect** the PCs from the laboratory LAN **before booting** them
2. Insert the pendrive and boot the PC
3. Connect the PCs to the switch using the cables
4. Check that the physical links are working by looking at the switch LEDs.
5. Assign IP addresses to each host in the LAN. Each group **selects private addresses from the CLASS B range (172.16.0.0/16), using as third byte the group number (172.16.XX.0)** and choose the strictest netmask that is required to host no more than **62 hosts**.

WARNING: the Ubuntu Linux Network configuration manager will periodically try con (re)-configure your LAN. Disable it from the top left menu bar (remove the tick from “Enable Networking”).

In the following, we will refer to each host as H1, H2, H3.



4. Report the configuration of the testbed

In the laboratory workbook, report the physical layer configuration of the testbed. Sketch both the physical, datalink and network layer topologies.

- 1) Use `ethtool` to check the status of the physical layer and datalink layer configuration of the linecard. How much of the provided information can you understand? Report the configuration of a linecard and summarize its state. What changes if you unplug the cable from the linecard?
- 2) Use `ifconfig` to check the status of the networking layer. How much of the provided information can you understand? Report the configuration of a linecard and summarize its state. What is the difference between “UP” and “RUNNING” for an interface?
- 3) Use the command `route` to check the status of the routing table. What happens when you configure a linecard using the `ifconfig` command?
- 4) Put the linecard “down”. What happens to the routing table? Is the IP configuration of the linecard been discarded (suggestion: use `ifconfig -a` to see all interfaces...)?
- 5) Put the linecard “up”. What happens to the routing table? What happens to the IP configuration of the linecard?
- 6) How can you remove an IP address that has been assigned to a linecard?

5. Connectivity check – basic ping

Using the ping command, check if it is possible to reach other hosts in your LAN. Run the command

```
H1: ping H2 -c 4
PING H2 (H2) X(Y) bytes of data.
Z bytes from H2: icmp_seq=1 ttl=64 time=18.6 ms
Z bytes from H2: icmp_seq=2 ttl=64 time=0.127 ms
Z bytes from H2: icmp_seq=3 ttl=64 time=0.127 ms
Z bytes from H2: icmp_seq=4 ttl=64 time=0.125 ms

--- H2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.125/4.758/18.656/8.024 ms
```

- 1) What are the numbers **X**, **Y**, **Z** ?
- 2) How can the ping program report the `icmp_seq`? Why there is such a header? What could happen if such header were not present?
- 3) What is the `ttl` value? Does it refer to the packets sent, or received? Who decides to use 64?
- 4) What does it change if you change the size of the requests using the `-s<size>` option? Which is the minimum size that allows ping to measure and report the “time” field? Why this is happening? If you were the programmer working on the implementation of a `ping` command, how would you implement the “time” measure?
- 5) What happens if H1 tries to ping a host that is not active but belonging to your subnet, e.g., H1: `ping H4`
Is there any packet that is sent on the LAN (check the LEDs on the switch)? Which packets are those?
- 6) What happens if H1 tries to ping a host that is not active and does NOT belong to your subnet, e.g., Y1? Is there any packet that is sent on the LAN (check the LEDs on the switch)? Which packets are those?

Networking Lab 2 – Introduction to Quagga and Cisco IOS syntax

We have so far used the Unix/Linux commands to check and change the system network configuration. While the fundamentals are the same, each operating system has its custom way to check and change the IP address of an interface, or the routing tables. We now get confident to the CISCO IOS syntax, which is commonly used on all CISCO devices. For this, we use the Quagga package, a routing software suite for Unix platforms, providing implementations of OSPFv2, OSPFv3, RIP v1 and v2, RIPng and BGP-4, particularly FreeBSD, Linux, Solaris and NetBSD. Quagga is a fork of GNU Zebra which was developed by Kunihiro Ishiguro. The Quagga architecture consists of a core daemon, *zebra*, which acts as an abstraction layer to the underlying Unix kernel.

Quagga daemons are configurable via a network accessible CLI (called a 'vty'). The CLI follows a style similar to that of other routing software. Each daemon can be accessed remotely, via telnet, using the specified port as indicated in the table below. Each daemon manages the configuration of a specific routing protocol. The easiest way to connect to a daemon is via telnet, on the localhost (or from any remote host once the network is configured and the host is reachable)

```
telnet localhost zebra
```

Daemon	Goals	TCP PORT
zebra	Host main configuration for IP addresses and static routing	2601
ripd	Management of the RIP protocol	2602
ospfd	Management of the OSPF protocol	2604
bgpd	Management of the BGP protocol	2605

would connect you to the zebra daemon running on your host/router.

Once connected you need to authenticate yourself and get access as a *regular user*. Authentication will be required via password verification. In the lab distribution, password is “*studente*”. Regular users can only execute visualization commands, but they are not allowed to modify the system configuration. To gain the right to modify the system config, one need to enter into *privileged mode* – which is done via the `enable` command. Pre-configured password is “*studenteADM*”. A privileged user can change the host configuration entering into the `configure terminal` mode. To exit from each state, you can use the command `exit`, or `disable`. Line prompt reminds the current mode. In summary, the important command are:

Mode	Command	Autentication	Prompt
Regular user	<code>telnet host port</code>	Pwd: <i>studente</i>	>
Privileged used	<code>enable</code>	Pwd: <i>studenteADM</i>	#
Global configuration	<code>configure terminal</code>	---	(config) #

Here is an example of the whole process:

```
... studente]# telnet localhost zebra
Trying 127.0.0.1 ...
Connected to 127.0.0.1.
Escape character is '^]'.

Hello, this is Quagga (version 0.99.24.1).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

User Access Verification
Password:
Router> enable
Password:
router# configure terminal
```

```
router(config)#
```

1. The Zebra process

Zebra allows to check and configure the static part of the IP configuration, for instance allowing to assign IP addresses to interfaces, or modifying static routes. It allows autocomplete via the TAB key, and offers an online help by pressing at any time the “?” character. Commands can be negated by prepending a “no” token

2. Interface configuration

IP addresses refer to each single interface. As such, you have to enter into interface configuration mode, and then assign/change/remove the desired IP address for the selected interface. Main commands are:

Command	Description
interface IFNAME	Select the IFNAME interface (eth0, eth1, ...) to work with
(no) ip address A.B.C.D/M	(de)assign the address A.B.C.D to the selected interface, with netmask M
(no) shutdown	(activate) deactivate the interface. By default, an interface is activated when you assign an IP address.

Example:

```
Router# configure terminal
Router(config)# interface eth0
Router(config-if)# ip address 192.168.1.230/24
Router(config-if)# show running-config
```

Current configuration:

```
!
hostname Router
password studente
enable password studenteADM
!
interface eth0
 link-detect
 ip address 192.168.1.230/24
 ipv6 nd suppress-ra
```

3. Routing configuration

Routing is a system wide configuration. To show or manage routing tables you can use the following command:

Command	Description
(no) ip route A.B.C.D/M E.F.G.H	(remove) add destination network A.B.C.D/M via E.F.G.H as next hop
(no) ip route A.B.C.D/M E.F.G.H COST	(remove) add destination network A.B.C.D/M via E.F.G.H as next hop with cost COST (used for OSPF)
(no) ip route 0.0.0.0/0 E.F.G.H	(remove) add E.F.G.H as default gateway
(no) ip route A.B.C.D/M IFNAME	(remove) add A.B.C.D/M via direct delivery through interface IFNAME

Example:

```
Router# configure terminal
Router(config)# ip route 10.0.0.0/24 192.168.1.254
Router(config)# show running-config
```

```

Current configuration:
!
hostname Router
password studente
enable password studenteADM
!
interface eth0
  link-detect
  ip address 192.168.1.90/24
  ipv6 nd suppress-ra
!
interface lo
  no link-detect
!
ip route 10.0.0.0/24 192.168.1.254
!
line vty
!
end
Router(config)#
Router# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, P - PIM, A - Babel,
       > - selected route, * - FIB route

S>* 10.0.0.0/24 [1/0] via 192.168.1.254, eth0
C>* 127.0.0.0/8 is directly connected, lo
K>* 169.254.0.0/16 is directly connected, eth0
C>* 192.168.1.0/24 is directly connected, eth0
Router(config)# no ip route 10.0.0.0/24
% Command incomplete.
Router(config)# no ip route 10.0.0.0/24 192.168.1.254
Router(config)# exit
Router# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, P - PIM, A - Babel,
       > - selected route, * - FIB route

C>* 127.0.0.0/8 is directly connected, lo
K>* 169.254.0.0/16 is directly connected, eth0
C>* 192.168.1.0/24 is directly connected, eth0

```

4. Visualization and management

It is always possible to visualize the configuration of an interface, of routing tables, and of the currently running configuration. Similarly, to make current configuration permanent, it can be saved on system configuration file `/etc/quagga/zebra.conf`. Important commands are:

Command	Description
show running config	Show the currently running configuration. Note that default are positive, i.e., no line means active
show interface IFNAME	Show the statistics of the interface IFNAME
show ip route	Show the current routing tables. Entries may be due to static routes (S), present in the kernel (K), due to direct connected interfaces (C), or inserted by some protocols (R,O,I,B,P,A). Entries in the FIB are marked with a *, while > state selected entries.
write file	Save the running config to the <code>/etc/quagga/zebra.conf</code> to make it persistent

Networking Lab 3 – Introduction to Wireshark

The goal of this lab is to understand and get used to the usage of Wireshark, a network analyzer software, or sniffer. Wireshark is program that allows the user (with the proper rights) to capture all packets that are received and sent by a network adapter, and to visualize them using advanced features. Thanks to the knowledge of protocols themselves, Wireshark can correctly decode almost all protocols headers, showing very detailed (and often too detailed) protocol information.

1. Wireshark filters

Filters are useful to limit the number of packets exposed to the user. When capturing packets in operative network, a network interface typically receives several hundreds of packets per seconds. Thus, it is necessary to select and filter those packets the network analyst is interested into.

Wireshark offers two types of filters

- **Capture filters:** packets that do not pass the filter are discarded and not captured at all
- **Display filters:** captured packets that pass the filter are visualized. Others are only temporarily hidden.

Capture filters are useful to limit the amount of packets Wireshark will capture. Wireshark uses the `libpcap` language syntax for capture filters, explained in the `tcpdump` man page. We will not use them in the course.

Display filters allow you to concentrate on the packets you are interested in, while hiding the currently uninteresting ones. They allow you to select packets by protocol, by the presence of a field, by the values of fields, or by a comparison between fields, and a lot more.

To specify a filter, simply enter it in the filter box. Experiment with it, using the GUI.

2. Simple capture session

1. Configure the testbed as usual.
2. Start capturing packets on H2 BEFORE sending any packet
3. From H1, start sending ping messages to H2 and from H2 start sending ping messages to H3
4. Stop capturing after about 60s, and stop the ping commands

Get confident with the Wireshark GUI, experimenting with the various menu voices that the program offers.

1. Define the display filter that allows to see all PDUs involving H2 and H1 only.
2. Set a coloring rule so that ICMP echo requests and ICMP echo reply messages are highlighted using different background colors
3. Consider a ICMP-req message, and check ALL protocols headers you can identify
 - a. At the Ethernet layer
 - b. At the IP layer
 - c. At the ICMP layer
 - d. At the application layer

4. **Is there any field that is missing? What are the reasons to have some headers?** What could happen if such a header were missing? For example, look at the ICMP header.
- How many bits are used for the sequence number?
 - Why you need an identifier fields?
 - How is the RTT computed? Why?

Consider ARP-req messages and ARP-rep messages and check the Ethernet headers.

- What is missing? Recall, an Ethernet payload MUST be larger or equal than 46B (and shorter than 1500B)

3. Connectivity check – basic ping

Using the ping command, check if it is possible to reach other hosts in your LAN. Run the command

```
H1: ping H2 -c 4
PING H2 (H2) X(Y) bytes of data.
Z bytes from H2: icmp_seq=1 ttl=64 time=18.6 ms
Z bytes from H2: icmp_seq=2 ttl=64 time=0.127 ms
Z bytes from H2: icmp_seq=3 ttl=64 time=0.127 ms
Z bytes from H2: icmp_seq=4 ttl=64 time=0.125 ms

--- H2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.125/4.758/18.656/8.024 ms
```

- 1) What are the numbers **X, Y, Z** ?
- 2) How can the ping program report the `icmp_seq`? Why there is such a header? What could happen if such header were not present?
- 3) What is the `ttl` value? Is it the sender or receiver TTL? Who decides to use 64?
- 4) What does it change if you change the size of the requests using the `-s<size>` option? Which is the minimum size that allows ping to measure and report the “time” field? Why this is happening? If you were the programmer working on the implementation of a ping command, how would you implement the “time” measure?
- 5) Look at the time measured for the first and other requests. Why are so different?
- 6) What happens if H1 tries to ping a host that is not active but belonging to your subnet, e.g., H1 : ping H4? Is there any packet that is sent on the LAN (check the leds on the switch)? Which packets are those?
- 7) What happens if H1 tries to ping a host which is not active and does NOT belong to your subnet, e.g., Y1? Is there any packet that is sent on the LAN (check the leds on the switch)? Which packets are those?
- 8) Check IP fragmentation. Send large messages, and eventually using the `-M` option, observe how IP packets and fragments are built. How can the receiver reassemble the packet from received fragments? How can eventually reorder them? How can it detect that all fragments have been received? In which order fragments are sent by the sender? Describe the possible algorithm the receiver implements.

4. Arp table management

Using the `arp` command, check how the system handles the ARP tables.

Suggestion: to clean up arp tables, put the interface DOWN and then UP again.

- 1) Check the status of the ARP table before and after issuing a ping command. If hosts H1 pings hosts H2, which hosts update their ARP tables?
- 2) Did host H3 update it? Why?
- 3) What happens if H1 tries to ping a host that is not active but belonging to your subnet, e.g., H4? How does the arp table of H1 change?
- 4) For how long an entry remains in the arp table? Suggestion: use the command `date` to track time and keep checking the arp entry. The PC can run simple programs, like `while true; do arp; date; sleep 1; done`
- 5) Is there any difference considering “incomplete” entries?

5. Arp requests and link failure

Delete all entries from the arp tables of H2 and H1. Start pingping H2 from H1. After about 90s, simulate a link failure by disconnecting the cable from the switch. Observe the arp messages exchanged between H2 and H1 using Wireshark

- 1) What happens during the first 90s when everything is working?
- 2) What happens if the failure lasts less than 5s? Check the arp tables of H1 and H2
- 3) What happens if the failure lasts more than 30s? Check the arp tables of H1 and H2
- 4) What happens as soon as the link is reestablished at the physical layer? CHECK THE RTT measurements of ICMP messages after the link goes up again

6. Connectivity check – Advanced ping

In this section we try to understand what happens in some strange configurations

- 1) What happens when a host pings the broadcast address?
- 2) What happens when a host pings the networking address?

Duplicate addresses: Configure your LAN so that there are two hosts with the same IP address, H1, H1', H2.

- 1) What happens when H2 pings H1? Check the arp tables of H1, H1', and H2
- 2) What happens when H1 and H1' starts pingping H2 at the same time? Check the arp tables.

Wrong Netmask: Configure your LAN so that H1 sees H2 as belonging to his subnet, but H2 sees H1 as NOT belonging to his subnet. Suggestion – use different Netmask for H1 and H2

- 1) What happens when H1 pings H2? Which packets is H1 sending? Which packets is H2 sending? How do arp tables of H1 and H2 change?
- 2) What happens when H2 pings H1? Which packets is H1 sending? Which packets is H2 sending? How do arp tables of H1 and H2 change?

Wrong Netmask and conflict with broadcast address Configure your LAN so that

- H1 has address 172.16.0.127 and Netmask 255.255.255.0
 - H2 has address 172.16.0.1 and Netmask 255.255.255.128
- 1) What happens when H1 pings H2? Which packets is H1 sending? Which packets is H2 sending? How do arp tables of H1 and H2 change?

- 2) What happens when H2 pings H1? Which packets is H1 sending? Which packets is H2 sending? How do arp tables of H1 and H2 change?

Networking Lab 4 – Analysis of TCP

The goal of this laboratory is to understand transport layer protocols, with particular attention to TCP. TCP offers a reliable service, based on a connection oriented paradigm based on a bidirectional connection. It performs flow control, congestion control, error correction, etc.

1. Running some service on the host

To use transport protocol, we need some application running at the application layer at the server, which is “listening” for possible requests from clients. To enable some basic services, the O.S. must be correctly configured. To do so, the file `/etc/inetd.conf` must be properly edited. First, check which services are actually already running on your host using the `netstat` command

1. Check using `netstat` which are the current state of both TCP and UDP. Check the man page to figure out which parameter to use.
2. Is there any service running in your PC that can be reached using TCP? Or using UDP?

`netstat` - Print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships

```
netstat [address_family_options] [--tcp|-t] [--udp|-u] [--raw|-w]
[--listening|-l] [--all|-a] [--numeric|-n] [--numeric-hosts]
[--numeric-ports] [--numeric-users] [--symbolic|-N]
[--extend|-e[--extend|-e]] [--timers|-o] [--program|-p] [--verbose|-v]
[--continuous|-c]
```

3. Using `gedit`, edit the `/etc/inetd.conf` file to enable the `echo` and `chargen` services. Do you need to be root to edit this configuration file? Do it on all hosts of your network.
4. (Re)start the `inetd` service by running the command
`/etc/init.d/openbsd-inetd restart`
5. Check again the status of the services that are now running on your host using the `netstat` command. Which ports are used by the `echo` and `chargen` service?

WARNING: To avoid some misunderstanding, we need to disable some advanced functionalities that modern linecards implement. This can be done using the `ethtool`. Check the man page to see which are those advanced features.

1. Checking the OFFLOADING capability of a linecard

```
ethtool -k ethX
```

2. Disable any OFFLOADING capability implemented by any linecard

```
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]
```

2. Using services from a client

Now that each host is running different services, you can use a client application to connect to one of the applications running on another host. We can use the `telnet` command to contact the application we are interested in on any host of your LAN. `telnet` is a remote terminal emulator that, using TCP at the transport layer, allows one accessing a service on a remote host. The `telnet` command is used for interactive communication with another host (the server) using the TELNET protocol. It begins in command mode, where it prints the `telnet` prompt ("`telnet>`"). If `telnet` is invoked with a host argument, it performs an open command implicitly; to terminate using a service, enter into the "**telnet command mode**" by pressing the escape sequence '`^]`' + RETURN (that is, press CTRL and] keys at the same time on the keyboard and then RETURN key). This will force the telnet application running locally to not send character to the server, but to interpret them as command. Use the `quit` command to exit from the telnet application.

```
telnet <host> [port]
```

will try to setup an application layer communication to `host` on port `port` using TCP at the transport layer.

1. Which well-known port do you know?
2. Which should be the command to be used from H1 to contact the echo server on H3?
3. Check the file `/etc/services`. Which information is present? Which functionality is offered by that file according to the ISO-OSI standard?

3. Analysis of the ECHO service

1. Run Wireshark and start capturing packets
2. Using telnet, open a connection from your host to another host in the LAN
 - a. Contacting the `time` service
 - b. Contacting the `echo` service. Type some strings on your keyboard, then close the connection by entering into command mode, and using the `quit` command
3. Analyze the packet level trace, identifying
 - a. the connection setup phase
 - b. the data exchange phase
 - c. the connection tear down phase.
4. How many connection request do you see?
 - a. When contacting the `echo` service, is the three-way-handshake successful? Why?
 - b. When contacting the port 1234, is the three-way-handshake successful? Why?
5. How are sequence numbers chosen during the three-way-handshake? **WARNING:** Wireshark uses relative sequence numbering. Which is the actual sequence number?
 - a. Why the client and the server perform a SYNchronization of sequence number? Why it is deprecated to always start numbering from 0?
 - b. How are TCP source port numbers chosen? Hint: open two connections from the same server to the same destination host at the same port.
6. How is it possible to multiplex/demultiplex correctly different TCP connection?
7. How are ACK numbers sent by the receiver?
8. How can the server refuse an incoming connection?
9. When does the application send data to the server? How is the text encoded?
10. How do the sequence number and ACK number evolve over time?

11. What happens if the server gets disconnected (e.g., unplug the cable) and then the client has to send data to it (e.g., type something on the terminal)?

Suggestion: get confident with wireshark functionalities to

- Follow a TCP stream
- Generate statistics about conversations
- Showing IO graphs
- Performing flow analysis

4. Analysis of the CHARGEN service

1. Run wireshark and start capturing packets
 - a. **Suggestion:** since we are NOT interested in the actual payload, you should limit the packet capture to include all TCP headers, but not all application payload – use the “snaplen” options
2. Open a connection from your host to another host in the LAN contacting the `Chargen` service.
 - a. What is the application running on the server doing?
 - b. What happens to eventual characters you type on the keyboard?
3. After some time, **maximize** the terminal windows on your screen
4. Now **minimize** the terminal window for about 10s, then bring it back to full screen.
5. Now press CTRL+C on the telnet window [this stops telnet to print characters on the screen – but the chargen server is still sending lot and lot of packets, now at line rate speed]
6. Open a new terminal window, and open a second telnet to the same server
7. Press CTRL+C on the second telnet screen
8. After about 10s, enter into Telnet command mode by pressing the escape sequence `'^]' + return` `CONTROL` and `']'` character
 - a. What is happening now? How can the client stop the server from sending packets?
 - b. Wait 10s then press “return” on the keyboard. What happens?
9. Enter again into telnet command mode, and close the connection.
10. What happens if the client gets disconnected (e.g., unplug the cable) and the server is still sending data to it?

Analyze the packet level trace

1. Identify the connection setup phase, the data exchange phase, and the connection tear down phase.
2. How are sequence number chosen during the three-way-handshake? WARNING: wireshark uses relative sequence numbering. Which is the actual sequence number?
3. How are ACK numbers sent by the receiver?
4. When the application sends data to the server? How is the payload encoded?
5. How do the sequence numbers and ACK numbers evolve over time?
6. When the terminal window is bigger/smaller, or when telnet stops printing character, how is the RWND evolving?

Plot the evolution versus time of

- The client sequence number
- The client acknowledge number
- The server sequence number
- The server acknowledgement number
- The size of the packets sent by the client

- The size of the packets sent by the sever
- The receiver window size advertised by the client
- The receiver windows size advertised by the server

Briefly comment the plots.

Networking Lab 5 – Analysis of the nmap command

1. nmap

Nmap (Network Mapper and security auditing) is a tool that was designed to rapidly **scan large networks**, although it works fine against single hosts. Nmap uses raw IP packets in novel ways to determine what hosts are available on the network, what services (application name and version) those hosts are offering, what operating systems (and OS versions) they are running, what type of packet filters/firewalls are in use, and dozens of other characteristics.

Nmap is a very complete and complicated tool, which can generate packets that abuse of the normal semantic of protocols, e.g., sending/forging a `icmp echo reply` message to a host directly without having previously received a `icmp echo requests` message. The goal of this lab is not to understand all possible usage of nmap, but rather to try to see how it is possible to abuse of protocols to infer information from a remote host. NOTE: there are legal implications on trying to access and penetrate a remote server. Do not use nmap unless you know what you are doing, and you have the right to do so.

2. Using nmap

Configure H1, H2 and H3 so that they belong to the same subnet as usual. Activate the `chargen` and `echo` services on each of them. Using wireshark, analyze the packet trace of the following commands:

1. Perform the scan of an active and inactive service on a remote host:

```
nmap IP_ADDRESS -p 7,99 [warning – no space after the comma]
```

```
sudo nmap IP_ADDRESS -p 7,99
```

2. What happens when you run the above command as `root`, or as a `unprivileged` user? Observe the sequence of sent packets. How are TCP source port chosen? Do they follow the normal behaviour? Compare the TCP segments sent with the `SYN` flag on. Which entity (at which layer) is generating those packets?
3. As above, but add the `-O` option (remote host Operating System identification)

```
sudo nmap IP_ADDRESS -p 7,99 -O
```

Observe the sequence of sent packets. How are TCP source port chosen? Do they follow the normal behaviour? How is it possible for `and APPLICATION` to generate those packets?
4. Run a complete scan of a host, looking for which services are running in the first 100 ports using the TCP and UDP ports:

```
sudo nmap IP_ADDRESS -p 1-100
```

```
sudo nmap IP_ADDRESS -p 1-100 -sU
```

5. Report and comment the plot that represent the port checked versus time. Why the TCP scan is much faster than the UDP scan? How many times nmap checks a given port using TCP? And using UDP? Why it changes the behaviour?

Suggestions: use the `-v` option to increase verbosity if needed. Use the `'time'` command to measure the system resource usage.

Networking Lab 6 – analysis of normal web browsing

1. Normal Web Browsing analysis

The goal of this laboratory is to analyse traces collected during normal web browsing operations. Starting from simple pages, we understand how DNS and HTTP work first. Next, we try to observe the normal browsing of complicated webpages. This lab is deeply inspired by the laboratory proposed in the Book “Computer Networking: A top-down approach” by Jim Kurose and Keith Ross.

Leave the PCs connected to the normal LAIB network for this experience and configure H1, H2 and H3 so that they automatically get a valid IP address from the LAB Network (i.e., let the Linux Network Configuration manager configure the network setup via DHCP).

Warning: if you cannot connect to the didattica.polito.it portal, it may be because the DNS is not properly configured in your system. Check the file `/etc/resolv.conf`. If it does not exist, try the following

```
# delete the wrong configuration

sudo rm /etc/resolv.conf

# replace with the one got from the DHCP

sudo ln -s /run/resolvconf/resolv.conf /etc
```

2. The Basic HTTP GET/Response interaction

Let's begin our exploration of HTTP by downloading a very simple HTML file - one that is very short, and contains no embedded objects. Do the following:

1. Start up your web browser
2. Start up the Wireshark packet sniffer. Enter “http” (just the letters, not the quotation marks) in the display-filter-specification window, so that only captured HTTP messages will be displayed later in the packet-listing window. Try to get rid of other HTTP messages that are not involved in the browsing session
3. Begin Wireshark packet capture.
4. Enter the following to your browser
<http://www.tstat.polito.it/wireshark-labs/HTTP-wireshark-file1.html>
Your browser should display the very simple, one-line HTML file.
5. Stop Wireshark packet capture.
6. Analyze the packet trace – HTTP analysis
 1. Is your browser running HTTP version 1.0 or 1.1? What version of HTTP is the server running?
 2. What languages (if any) does your browser indicate that it can accept to the server?
 3. What is the IP address of your computer? Of the `www.tstat.polito.it` server?
 4. What is the status code returned from the server to your browser?
 5. When was the HTML file that you are retrieving last modified at the server?
 6. How many bytes of content are being returned to your browser?
 7. Is there a second HTTP request in the trace? What is that?

7. Repeat the request, by forcing the browser to reload the page
 1. Press the reload icon. How is the response modified?
 2. Press **SHIFT** + the reload icon. What is the difference?
8. Use `telnet` to download the page:
 1. Open 4 terminal windows, and in each of them use `telnet` to connect to the same webserver via `telnet www.tstat.polito.it http`
 2. Enter the following lines

WINDOW 1
GET /wireshark-labs/HTTP-wireshark-file1.html HTTP/1.0
[return]

WINDOW 2
GET /wireshark-labs/HTTP-wireshark-file1.html HTTP/1.0
Host: www.tstat.polito.it
[return]

WINDOW 3
GET /wireshark-labs/HTTP-wireshark-file1.html HTTP/1.1
[return]

WINDOW 4
GET /wireshark-labs/HTTP-wireshark-file1.html HTTP/1.1
Host: www.tstat.polito.it
[return]
 3. Check the results on each window. What are the differences? Observe the response code from the server. What is the difference between HTTP 1.0 and HTTP 1.1?
 4. Observe the packet trace. How many TCP connections are present? Who initiates the connection tear-down? When? How are PUSH flags being used by the client and the server?
9. Repeat the analysis on the following websites using the browser and pointing to:
 1. <http://www.tstat.polito.it/wireshark-labs/HTTP-wireshark-file3.html>
 2. <http://www.tstat.polito.it/wireshark-labs/HTTP-wireshark-file5.html>
 3. http://www.tstat.polito.it/wireshark-labs/protected_pages/HTTP-wireshark-file5.html
 username: wireshark-students ; passwork: network
 4. <http://www.tstat.polito.it/wireshark-labs/wireshark-form.php>
 compile the form and submit it
10. Observe how the HTTP protocol works, and how HTML pages are processed to retrieve objects referenced in the source of the HTML code.
 1. How many TCP connections are being used? Toward which servers?
 2. How does the response code change?
 3. Why some objects fail to load?
 4. Can you observe the username and password in the trace?
 NOTE: While it may appear that your username and password are encrypted, they are simply encoded in a format known as Base64 format. Username and password are *not* encrypted! To see this, go to <http://www.motobit.com/util/base64-decoder-encoder.asp> and enter the string `d2lyZXNoYXJrLXN0dWR1bnRz` and decode. *Voila!*


5. How is timing of the protocol being defined? Use the I/O graph to display over time
 1. DNS requests in red
 2. TCP SYN packets in blue
 3. HTTP requests in green

3. Analysis of complex webpages

In this part, you will analyse real webpages offering rich and multimedia content. Modern websites include complex webpages, composed of many objects. They are images, CSS stylesheets, JavaScripts, etc. These objects are all retrieved using the HTTP(s) protocol and are automatically fetched by your browser when you load the webpages.

Each object is identified by a URL (Note: it is different from the URL of the main webpage you are loading!). The browser issues an HTTP request for each object and obtains an HTTP response. By analysing these HTTP transactions, it is possible to study many characteristics of a webpage, such as its size, its performance, the type of content it includes, how many servers are contacted, etc.

For this practice, repeat the following steps for the websites: www.tstat.polito.it www.repubblica.it and a website you choose:

1. Start the Chrome browser and open the Developer Tools (Right Click -> Inspect or CTRL+SHIFT+I)
2. Open Wireshark and record the traffic
3. Set the Developer Tools on the Network Tab
4. Visit the web page
5. Once the page has been loaded, stop the Recording clicking on the Red circle or pressing CTRL+E
6. Look at the waterfall (add more columns if needed right-clicking on the waterfall header):
 - a. How many objects have been downloaded?
 - b. Which HTTP version has been used?
 - c. Reload the page emulating a mobile device (click on the  icon on the top left of the Developer Panel). The number of objects is the same?
 - d. Reload the page? How many objects are found in the disk cache?
7. Save the Waterfall as a HAR file: Right Click on any element of the Waterfall -> Copy -> Copy All as HAR. Then, paste it into a text file. Write some simple bash scripts to print:
 - a. How many HTTP transactions have been issued
 - b. The number of objects separately per mimeType
 - c. How many servers (IP addresses) where contacted
 - d. The total size of objects. You can obtain it from the bodySize
8. Use Wireshark to watch the traffic generated by the visit:
 - a. Plot the traffic due to the DNS and HTTP(s) protocols over time
 - b. Obtain the list of domains your browser resolved during the visit
 - c. Is there any other traffic which is not related to the browser's visit?

Networking Lab 7 – Performance test on Ethernet links

The goal of this laboratory is to evaluate the performance of a file transfer over the network and observe the impact of different mechanisms on the performance. The focus is on performance obtained using TCP or UDP at the transport layer, considering scenarios of increasing complexity.

To measure the performance, we will use either the `nttcp` or the `iperf` command, described in the next sections.

We are interested in measuring what we will refer to as the “**goodput**”, that is, the speed at which *useful* data is received at the *application* layer.

Goodput = Useful **data** at the application layer / **time** to complete the transfer

This is different from the **throughput**, that is, the amount of information carried by the physical layer. The latter includes all protocol overheads, e.g., Protocol Control Information added by each layer, retransmission due to error recovery mechanisms implemented at layer 2 or layer 4, interfering traffic due to other applications sharing the link such as ARP requests, etc.

1. What is the maximum goodput you expect when using UDP at the transport layer? Compute the maximum efficiency μ_{UDP} defined as the *per packet goodput*, i.e., the fraction of useful data carried by a UDP message with respect to physical layer bits being transmitted.
2. What is the maximum goodput you expect when using TCP at the transport layer? Compute the maximum efficiency μ_{TCP} defined as the *per packet goodput*, i.e., the fraction of useful data carried by a TCP segment with respect to physical layer bits being exchanged. The latter includes all headers, acknowledgements too.
3. What changes if the link is set to HALF DUPLEX? Compute the maximum efficiency $\mu_{\text{TCP-HD}}$.

To allow a better control of the test bed, configure your Ethernet interface so that

- All **offloading** capabilities are **disabled**
- The speed is **10Mb/s** Full Duplex unless otherwise specified.

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

NOTE: the NIC may be also support the Ethernet flow control mechanism known and the **PAUSE Frame** protocol defined in the IEEE 802.3x standard. It allows the switch to send a PAUSE Frame which inhibits the transmission of further frames on the other transmitter side. This avoids congesting the switch queue by blocking the upstream sender. Try to check and eventually disable it using the `ethtool`

```
H1: ethtool -a eth1 # shows the status of the pause support
```

```
H1: ethtool -A eth1 [autoneg on|off] [rx on|off] [tx on|off]
```

Warning: this setting is critical, and not all line cards allow you to control the configuration in a proper way. Be careful.

5. `nttcp`

The `nttcp` application measures the transfer rate (and other indexes) on a TCP, UDP or UDP multicast connection. To use `nttcp` you must run the application on the local host (client) and on the remote host (server). On the server host simply start `nttcp` with the option `-i`.

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

[note: the `&` runs the application in background so that the terminal is not blocked waiting for the application to exit].

Alternatively, you may wish to configure the `/etc/inetd.conf` – see the `nttcp` and `inetd` manpage

On `H1`, `nttcp` is now listening for connections from client `nttcp` applications.

Which port is the `nttcp` server listening to?

On the client `H2` simply call `nttcp` with the name of the partner host.

```
H2: nttcp H1
```

`H2` will contact the `nttcp` server running on the `H1` and it will initiate the test. By default, the test transfers 2048 buffers of 4kByte length (a total of 8 MByte) to the partner host. The goodput performance will be measured on both sides, and the results (both local and remote) are reported on the client terminal as output. You can change parameters of the transmission via command line options.

Useful options:

- `-r` defines the **receive** transfer direction; data is sent from the partner host to the local host.
- `-t` defines the **transmit** transfer direction; data is sent from the local host to the partner host. This is the default direction.
- `-T` Print a **title line**.
- `-u` Use the **UDP** protocol instead of TCP (which is the default).
- `-n N` The **number N of buffers** will be written to the transmitting socket. It defaults to 2048.
- `-l L` The length `L` defines the **size of one buffer** written to the transmitting socket. Defaults to 4096.

For example, let `H2` receive 100.000 UDP frames with a payload of 100B from `H1`:

```
H2: nttcp -r -u -n 100000 -l 100 H1
```

1. What is the impact of the choice of the `N` and `L` parameters? Does it change if you use TCP or UDP? Why?
2. How should you choose `N` and `L` to get reliable measurements? Remember: `nttcp` is measuring the **time** spend to complete the transfer. How long should be the test to avoid eventual bias due to errors in measuring the duration of the transfer?
3. If you repeat the measurement several times, do you get the same numbers? If not, why?
4. Are the measurements reported by the local host and the remote host different? Why?
5. Does it change if `H1` transmits data to `H2` or viceversa? Why?
6. How can `nttcp` measure the time spent to *complete* the transfer? Does TCP or UDP change the scenario? Which assumption are required?

6. Iperf and iperf3

`iperf` is another tool for performing network throughput measurements. It can test either TCP or UDP goodput. As for `nttcp`, to perform an `iperf` test, the user must establish both a server (to discard traffic) and a client (to generate traffic). Note: `iperf` is the original version of the tool. Recently, `iperf3` – version 3 of the tool – has been released. We suggest using the latter.

To setup a server, run `iperf3` as below

```
H1: iperf3 -s
```

As in the case of `nttcp`, now an `iperf3` server is running on H1, and waiting for incoming connections.

1. Which port is the `iperf3` server listening to? And the original `iperf` tool?
2. How to run an `iperf` server that uses UDP instead of TCP?
3. Is it possible to have an `iperf`, `iperf3` UDP and TCP server, and a `nttcp` server at the same time?

On the client H2 simply call `iperf3` with the name of the partner host.

```
H2: iperf3 -c H1
```

This will run a test, using TCP at the transport layer, with H2 sending data to H1. The test will last 10s. This is different from `nttcp`, where you have the control on the amount of data to be sent, with `iperf3` you have the control on the time the test lasts. Note that the `iperf3` client will act as the sender by default.

Useful option:

GENERAL OPTIONS

- `-i, --interval n`: pause n seconds between periodic bandwidth reports
- `-u, --udp`: use UDP rather than TCP
- `-w, --window n[kM]`: TCP window size (socket buffer size)
- `-M, --set-mss n`: set TCP maximum segment size (MTU - 40 bytes)
- `-p, --port n`: set server port to listen on/connect to to n (default 5201)
- `-f, --format: [kmgTKMGt]` format to report k/Mbits/Gbits/Tbits – useful to force the same output format when running multiple tests under different configurations
- `-B, --bind host`: bind to the specific interface associated with address host
- `-V, --verbose`: give more detailed output
- `-J, --json`: output in JSON format
- `-s, --server`: run in server mode

CLIENT SPECIFIC OPTIONS

`-b, --bandwidth n[KM]`: set target bitrate to n bits/sec (default 1 Mbit/sec for UDP, unlimited for TCP/SCTP). If there are multiple streams (-P flag), the throughput limit is applied separately to each stream. Setting the target bitrate to 0 will disable bitrate limits (particularly useful for UDP tests). This throughput limit is implemented internally inside iperf3, and is available on all platforms.

`-c, --client <host>`: run in client mode, connecting to <host>

`-R, --reverse`: reverse the direction of a test, so that the server sends data to the client

`-t, --time n`: time in seconds to transmit for (default 10 secs)

`-l, --length n[KM]`: length of buffer to read or write. For TCP tests, the default value is 128kB. In the case of UDP, iperf3 tries to dynamically determine a reasonable sending size based on the path MTU; if that cannot be determined, it uses 1460 bytes as a sending size. For SCTP tests, the default size is 64kB.

`-P, --parallel n`: number of parallel client threads to run

`-C, --congestion <algo>`: set TCP congestion control algorithm (Linux only)

`--get-server-output`: Get the output from the server. May be in JSON format.

Iperf3 offer some nice possibilities which will allow us to have more control on the experiments. In particular, iperf3 allows one to choose the TCP congestion control the sender will use by specifying it via the `-C` option. Linux support several algorithms, among which:

- Original additive Increase, multiplicative decrease version: `Reno`
- Versions designed for large bandwidth delay product paths: `Bic`, `Cubic`, `Highspeed`, `Htcp`, `Illinois`, `Scalable`
- Versions designed for satellite links: `Hybla`
- Version that supports low priority traffic: `lp`
- Congestion control driven by delay instead of losses, or that specifically are designed for wireless links: `Vegas`, `Veno`, `Westwood`, `yeah`

See <http://interstream.com/node/1084> or <http://linuxgazette.net/135/pfeiffer.html> for more details.

Similarly, iperf3 allows one to control the TCP window size via the `-w` parameter, and to eventually use multiple parallel connections via the `-P` option. NOTE: the actual value of the window size is typically different from the one given as input – always double check which is the actual value when running an experiment.

7. Computing the goodput

Consider now the following different scenarios, with increasing complexity. For each of them

- **Predict the goodput you expect** when TCP or UDP is used at the transport layer
- In scenario where TCP and UDP flow coexists, predict the goodput you expect for each flow
- Is it possible to observe **collision** at the Data Link Layer? If so, which would be the collision probability? Suggestion: check the number of collisions before and after the test using the `ifconfig` command.
- Is it possible that **frames** are **dropped** by the switch? You could compare the number of frames sent on the sender NIC with the number of frames received on the receiver NIC via `ifconfig`.

- Is it possible that **data is lost** in the protocol stack?

After having discussed the previous questions, run the actual test and check your predictions.

How good was it? Comment on eventual disagreements. Why you got something different from what you were expecting?

- A. Single flow – FULL DUPLEX: H1 is sending data to H2
- B. Single flow – HALF DUPLEX: H1 is sending data to H2, but the linecard of H1 is set to half duplex. Does it change if H1 receives data from H2? Why? What if both linecards are in half duplex mode?
- C. Bidirectional flows – FULL DUPLEX: H1 is sending data to H2 *AND* H2 is sending data to H1. To allow “concurrent” flows, you can concatenate two `nttcp` commands, like `H1: nttcp -T H2 >tx.dat & nttcp -T -r H2 >rx.dat`, or you can use `iperf -d` option
- D. Bidirectional flows – Half DUPLEX: H1 is sending data to H2 *AND* H2 is sending data to H1. Linecard of H1 is in half duplex mode. What if other cards are in half duplex too?
- E. Two flows to the same *receiver* – FULL DUPLEX: H1 sends data to H2 *AND* H3 sends data to H2
- F. Two flows to the same *receiver* – HALF 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
- H. Two flows from the same *sender* – HALF DUPLEX: H1 sends data to H2 *AND* to H3
- I. Full mesh scenario- FULL DUPLEX: Each hosts sends data to all hosts at the same time.
- J. Full mesh scenario- HALF DUPLEX: Each hosts sends data to all hosts at the same time.

Summarize results in the tables below:

Test	Average TCP Goodput per flow		Collision probability		Loss at the application layer		Comment
	Prediction	Observed	Prediction	Observed	Prediction	Observed	
A							
B							
C							
D							
E							
F							
G							

H							
I							
J							

Table 1: Summary of TCP goodput experiments

Test	Average UDP Goodput per flow		Collision probability		Loss at the application layer		Comment
	Prediction	Observed	Prediction	Observed	Prediction	Observed	
A							
B							
C							
D							
E							
F							
G							
H							
I							
J							

Table 2: Summary of UDP goodput experiments

Test	Average TCP Goodput per flow		Average UDP Goodput per flow		Comment
	Prediction	Observed	Prediction	Observed	
C					
D					
E					
F					
G					
H					
I					
J					

Table 3: Summary of mixed TCP and UDP goodput experiments

Networking Lab 7bis – Performance test with off-the-shelf hardware

For this lab, you will use hardware you have at home, with all the limits and lack of full knowledge you can have of the actual configuration of your hardware and software. The goal is to i) build a simplified model of the setup, ii) use it to predict the expected goodput, iii) run the actual test in a thorough manner, and iv) critically compare the experiment results with the model predictions. In the following, we describe the process step by step.

We will consider two scenarios:

1. **Home network experiment:** in a local and controlled testbed: consider two hosts, H1 and H2, and a local network that connects the two hosts with a physical link. The two hosts can be two PCs, or a PC and a mobile device. Run the tests with H1 and H2.
2. **Internet experiment:** consider a local host H1 regularly connected to the internet that will act as client, while the host bigdatadb.polito.it connected in Politecnico di Torino campus network with a bond of 4 interfaces at 1Gb/s ethernet link will be a server. Run the test from the client H1 to the server on bigdatadb.polito.it.

NOTE: in the Internet scenario, the presence of **NAT and Firewalls** may make the experiment difficult. In particular, when NAT is present (as normally happens in the ADSL/FTTH/3G contracts you have), the host H1 inside the network cannot receive TCP connection and UDP packet from any host on the Internet. For TCP, you can still run the experiment by letting the application on the H1 client to open the connection to the server, and then let the server send data (remember – TCP offers a bidirectional service). Nttcp requires the server to be able to open a separate connection with the client (on port 5038). Thus, it will not work. Iperf3 instead uses the same connection opened by the client to run the test. Thus, it will work. With UDP, you may have difficulties in running a proper experiment. If this is the case – show the packet level trace – as seen from your client – that justifies the problem.

For this laboratory – it is important to consider the KISS¹- *Keep It Simple Stupid* - principle.

1. Choice and description of the physical setup of the testbed

Select a simple configuration. Prefer running the test on PCs rather than mobile devices. It is ok to use virtual machines, configuring the virtual adapter in bridged mode – which is the simplest configuration that allows the virtual host to connect to the physical network using the host adapter(s). Quit all applications running in background that could generate traffic during the experiment.

Prefer a Linux system if possible since it is easier there to control background processes and applications that can use the internet. On Windows and MacOS can be more complicated to quit all applications, especially those embedded inside the OS (e.g., cloud services, software updates, autodiscover protocols, etc.). Controlling the background services on mobile devices is even more complicated.

On mobile devices, pay attention to power saving features. Connect the device to a power supply and disable screen saving during the tests.

For the network setup, prefer wired connections, running at low speed (10Mb/s ethernet ideally – assuming the hardware allows you to control the speed correctly). If not available, consider a WiFi network. Try to be as close as possible to the Access Point – to limit interference and channel degradation. For the first setup,

¹ https://en.wikipedia.org/wiki/KISS_principle

consider disconnecting the home network from the internet to avoid interfering traffic and background applications too.

Describe the setup of the experiment. In particular, detail all the network connections, including virtual and physical bridges, like USB-Ethernet adapters, switches, access points, routers that are present in your home network along the path from H1 to H2, and from H1 to the default gateway in the second scenario.

For each link, specify the physical link speed and duplex mode. Then, identify the speed of the bottleneck link, i.e., the slowest link in the path between the client and the server. This will drive the prediction later.

2. Choice of the testing application

There are three applications that can be used to run the test. Nttcp is the oldest one, but still the most reliable one. Iperf version 2 and 3 are available on more operating systems and offer more options. However, sometimes they are buggier, and may produce inconsistent results.

You can use iperf3, eventually run it directly on your operating system – see <https://iperf.fr>. Get confident with it, and eventually compare results with other tools to check that results are consistent. Double check how the application generates data. Do some preliminary tests while capturing with Wireshark to check that everything goes as expected. Use the “Statistics => TCP stream graph” to check the sequence number evolution and throughput evolution over time. These are the same plots we did for the Chargen lab and let you check if there is congestion (sequence number slope that changes over time, with throughput curve that shows sudden drops). With UDP, choose the size of the data generated by the application wisely to avoid fragmentation. In a nutshell, set the parameters wisely.

3. Predict the expected goodput

For this,

- Describe and compute the maximum efficiency μ_{UDP} , μ_{TCP} for the considered scenario. For WiFi, consider an overall efficiency of approximately 50% for TCP, 60% for UDP.
- Discuss if you expect to see
 - (L1) **Transmission errors** at the physical layer
 - (L2) **Collisions** due to the data link layer using a shared medium
 - (L2 and L3) **Congestion** due to store and forward at switches, routers, USB/Ethernet adapters, etc.
 - (L4 and other protocols) **Signaling** messages, **retransmissions**, congestion control, etc.
 - (L7) Bottleneck at the application (e.g., CPU rate similar to Gooput)
- Predict the expected goodput, considering the maximum efficiency μ for that setup and identify of the **bottleneck link** that will constrain the goodput.

Discuss eventual sources of possible impairments that can be present (especially with WiFi, e.g., changes in the physical link rate due to changing channel conditions, interference of other devices over the same WiFi Channel, etc.)

4. Run the experiment and compare with prediction

Run the experiment. Consider the measurement as reported by the receiver (which shall measure more reliably). Repeat it at least 5 times to check if results are consistent. If not, compute the average, minimum, maximum values.

Compare them with your prediction and discuss eventual disagreements. Try to justify them (e.g., adding plots obtained with wireshark using the TCP stream graph, or I/O plot over time).

Experiment to consider

For each of scenario (home network, and internet setup) consider

1. Single flow - TCP
 - a. H1 sends data to H2 with TCP
 - b. H1 receives data from H2 with TCP
2. Single flow – UDP
 - a. H1 sends data to H2 with UDP
 - b. H1 receives data from H2 with UDP
3. Bidirectional flow [OPTIONAL]
 - a. 2xTCP: H1 receives data from H2 with 2 TCP connections
 - b. 2xUDP: H1 receives data from H2 with 2 UDP connections
 - c. TCP + UDP: H1 receives data from H2 with one TCP and one UDP flows

Note: For these scenarios, you need to run 2 iperf3 clients – as below:

```
iperf3 -c bigdatadb.polito.it > TCP1.dat & iperf3 -c bigdatadb.polito.it  
-p 5202 > TCP2.dat
```

You need also to have 2 servers running on your/bigdatadb.polito.it server, on two different ports. On bigdatadb.polito.it there are 9 servers running on ports from [5201:5209] for you.

For each scenario, predict the expected goodput, and compare it with the experimental results as described above. Document the results, eventually double checking the packet level trace and use the wireshark features to produce i/o graphs and sequence number evolution over time.

Networking Lab 8 – Performance test on WiFi

In this laboratory, we will repeat the speedtest experiments, but using a WiFi link. We need first to configure the Access Point (AP), so that each group will be using a different WiFi network, and to limit the wireless interference as much as possible.

Next, we need to configure the WiFi interface of the host to join the proper WiFi network. For this purpose, we use the `iwconfig` command which is the equivalent of the `ethtool`, but configures the wireless “link” of a WiFi interface. It is used to set the parameters of the physical layer and data link layer interface which are specific to the wireless operation (for example: the channel being used, the modulation, the physical layer speed, etc.). `iwconfig` may also be used to display those parameters, and the wireless statistics.

NOTE: not all hardware interfaces support all option. The O.S. driver may have bugs, and may not allow one to control all options.

`iwconfig` - configure a wireless network interface

Syntax: `iwconfig interface [essid X] [channel C] [rate R] [enc E] [key K]`

`essid`: Set the ESSID (or Network Name). The ESSID is used to identify the WLAN to which devices belong to.

`channel`: Set the operating channel in the device. Values are between 1 and 11 for European countries in the 2.4GHz band. See https://en.wikipedia.org/wiki/List_of_WLAN_channels for more details.

`rate/bit[rate]` : For cards supporting multiple bit rates, set the bit-rate in b/s. The bit-rate is the speed at which bits are transmitted over the medium, the user speed of the link is lower due to medium sharing and various overhead. Use `auto` to let the AP and WLAN card select automatically the best bit-rate.

`key/enc[ryption]` : Used to manipulate encryption or scrambling keys and security mode when WEP encryption is enabled. To set the current encryption key, just enter the key in hex digits as `XXXX-XXXX-XXXX-XXXX` or `XXXXXXXX`. To set a key other than the current key, prepend or append `[index]` to the key itself (this won't change which is the active key). You can also enter the key as an ASCII string by using the `s :` prefix. Passphrase is currently not supported. To change which key is the currently active key, just enter `[index]` (without entering any key value). `off` and `on` disable and enable encryption.

Example: configure the `wlan0` interface to join the network “group-01” on channel 1 and force the speed to be 11Mb/s

```
iwconfig wlan0 essid group-01 rate 11M channel 1
```

NOTE: You may also use the `iwlist` and `iwspy` commands. Check the man pages for more info. If WPA is used, you need to use the `wpa-supPLICant` command to configure the WiFi interface. We will not consider WPA in our tests.

1. Configuration of the host

- Disable Ubuntu Network Manager as always.
- Assign H1, H2, H3 addresses in the subnet 192.168.0.0/24 for interfaces ethX. Call these addresses E1, E2, E3 (as IP addresses associated to the Ethernet card)
- Warning: The AP is using address 192.168.0.1 – do not use it

2. Configuration of the Access Point

- Open the Firefox browser and connect to address 192.168.0.1
 - If Firefox is blocked, kill the process and restart it
 - From a shell, type **sudo killall firefox**
- Login using username “admin” and password “admin” [do not change the password!]
- Using the menu “Wireless” on the left, configure the WiFi network of the AP
 - Wireless Network Name: “Group-XX” where XX is the group number, e.g., 01,..., 09, 10,... note: name is case sensitive, i.e., Group-XX is different from group-XX!
 - Region: Italy
 - Mode: bg mixed [DISABLE n] – this avoids the AP using two adjacent physical channels and then reduces the interference among APs of different groups.
 - Channel: since only channels 1,6,11 are independent, we must carefully limit interference by evenly distributing groups on different channels.
 - Group (1+3i): Groups 1,4,7,10,...=> channel 1
 - Group (2+3i): Groups 2,5,8,11,...=> channel 6
 - Group (0+3i): Groups 3,6,9,12,...=> channel 11
 - Enable wireless router radio
 - Enable SSID broadcast
 - Save settings
- Using the menu “Wireless security” on the left
 - Disable Security
 - Save settings

3. Configure the WiFi USB adapter

- Insert the USB adapter and check via `ifconfig` the name of the new wireless interface (should be wlanX)
- Use the `iwconfig` to assign the wifi network name to join your group wifi wlan
`iwconfig wlanX essid Group-XX`
Assign wlanX a proper IP address so that it appears as another host in the **same LAN**, e.g., W1, and W2 (where W1 and W1 are the IP addresses associated to each WiFi card) with `ifconfig`.
 - Which addresses are you going to use?
 - Can you use addresses in the 192.168.0.0/24 subnet?
 - What if you were using another subnet?

4. Check that the WiFi link and interface are up and running

The configuration of your LAN now sees 4 hosts, some of which connected to the same LAN and subnet using multiple interfaces:

- H1 has address E1 on Ethernet interface and address W1 on the wireless interface.
- H2 has address E2 on Ethernet interface and address W2 on the wireless interface.
- H3 has address E3 on Ethernet interface.
- H4 is the switch/AP is also connected to the same subnet, with address 192.168.0.1

Check the routing tables and ARP tables. How is it possible to reach a host belonging to your subnet now?

- From H1, ping H2 and check that everything works.
 - Which interface is H1 using to reach H2? Which interface is H2 using to reply to H1?
- From H1, ping H2, by binding ping to the WiFi interface using the `-I interface address` option. From the man page of ping:
 - `-I interface address`: Set source address to specified interface address. Argument may be numeric IP address or name of device. What is the difference among the following commands?
H1: `ping -I W1 E2` or `ping -I wlan0 E2`
H1: `ping -I W1 W2` or `ping -I wlan0 W2`
 - Which addresses are contained in the ARP tables?
 - What is the MAC address seen by H1? By H2?
 - What is the RTT now? Is it stable as before, or does it varies much more? Why?
- Capture with Wireshark packets from the `wlanX` interface of H1 and H2 and check the trace.
 - Is there something different than pinging from the Ethernet interface?
 - Which packets are sent/received on this interface?
- Capture now enabling the “**monitor mode**” on the `wlan0` interface.
 - Double click on the `wlan0` interface name to bring the “Edit Interface Setting” window up
 - Enable “Capture packets in monitor mode”
 - Start capturing while not sending any traffic
 - How many IEEE 802.11 different frames do you recognize?
 - Can you see frames from other groups?

NOTE: there might be bugs in the hardware or driver which might not allow a card to disable monitor mode and return to “managed” mode. Disconnect and reconnect the USB adapter to force a hardware reset. You need then to reconfigure your WiFi interface from scratch.

5. Performance test over WiFi

Configure the testbed so that H1 and H2 are connected via WiFi to the AP, and disconnect the cables from the Ethernet ports. Leave H3 connected using the wired Ethernet. You have now W1, W2 and E3 active IP addresses. Consider now the following different scenarios.

- A. Single flow – Ethernet to WiFi: E3 sends data to W1 [Eth => WiFi]
- B. Single flow – WiFi to Ethernet: W1 sends data to E3 [WiFi => Eth]
- C. Single flow – WiFi to WiFi: W1 sends data to W2 [WiFi => WiFi]
- D. Bidirectional flow – Ethernet and WiFi - same hosts: E3 sends data to W1, and W1 sends data to E3 [Eth <==> WiFi]
- E. Bidirectional flow – Ethernet and WiFi - different hosts: E3 sends data to W1, and W2 sends data to E3 [Eth ==> WiFi (H1), Eth <== WiFi (H2)]
- F. Bidirectional flow – WiFi and WiFi: W2 sends data to W1, and W1 sends data to W2 [WiFi <==> WiFi]

For each of them:

- Predict the goodput you expect when TCP is used at the transport layer
- Predict the goodput you expect when UDP is used at the transport layer
- In scenario where TCP and UDP flow coexists, predict the goodput you expect for each flow

After having discussed the previous questions, do the actual test and check your predictions.

WARNING: given the high probability of interference at the physical layer, **repeat the test several times!**

Do the tests confirm your assumptions?

Why you got something different from what you were expecting? Summarize the results by discussing what you would expect, and what are the differences. Try to create different scenarios, and report different experiments you did. For instance, you can test and report

- Ethernet to WiFi: What changes if H3 sends data to H1, or if H3 sends data to H2?
- WiFi to Ethernet: What changes if H1 sends data to H3, or if H2 sends data to H3?
- ...
- Impact of WiFi settings (NOTE: the effectiveness of a parameter choice depends on the support of the driver and hardware WiFi interface):
 - Try forcing a given bitrate on the physical link via the `iwconfig rate` option.
 - Try changing the channel
 - Try enabling WEP encryption [`enc param`]
 - Try enabling 802.11n (if supported) [`modu param`]
 - Try changing the TX power (if supported) [`power param`]
 - Try enabling/disabling the RTS/CTS [`rts param`]
 - Try changing the number of retransmission at layer 2 [`retry param`]
 - ...

As previously said, the hardware/driver of the WiFi interface may not allow you to change the parameters. Always double check that results are consistent with your predictions. Describe the experiment design, and results you obtained. If useful, use Wireshark to collect traces and produce graphs.

Networking Lab 9 – Impact of packet loss on Throughput

In the previous laboratory, we have observed how the goodput changes when considering different scenarios, with TCP or UDP, with half/full duplex links, and with wired or WiFi links. In all cases, the RTT was very small, and the link capacity was either in the order of 10Mb/s or more. In this laboratory, instead, we consider some simple scenarios where a single TCP stream is used to transmit data, but under different network conditions. We learn how to control RTT and the packet loss, and how these affect TCP goodput.

To emulate and control the link path, we use the *Linux Traffic Control project*. Traffic control is the name given to the sets of queuing discipline (QDISC) systems and mechanisms by which packets are received and transmitted on a Linux router. This includes deciding which (and whether) packets to accept at what rate on the input of an interface and determining which packets to transmit, in what order, at what rate on the output of an interface. We use two modules:

- `netem`: Linux Network Emulator
- `TBF`: Linux Token Buffer Filter

`tc` is used to configure Traffic Control in the Linux kernel. The generic syntax to configure it is

```
tc qdisc add dev DEV root QDISC QDISC-PARAMETERS
tc qdisc change dev DEV root QDISC QDISC-PARAMETERS
tc qdisc del dev DEV root
tc qdisc show dev DEV root
```

TC COMMANDS

The following commands are available for `qdisc`, classes and filter:

`add` Add a `qdisc`, class or filter to a node. For all entities, a **parent** must be passed, either by passing its ID or by attaching directly to the `root` of a device. When creating a `qdisc` or a filter, it can be named with the **handle** parameter. A class is named with the **classid** parameter.

`delete` A `qdisc` can be deleted by specifying its handle, which may also be 'root'. All subclasses and their leaf `qdiscs` are automatically deleted, as well as any filters attached to them.

`change` Some entities can be modified 'in place'. Shares the syntax of 'add', with the exception that the handle cannot be changed and neither can the parent. In other words, change cannot move a node.

`replace` Performs a nearly atomic remove/add on an existing node id. If the node does not exist, it is created.

`show` report the status of the node id.

NetEm `qdisc`

NetEm provides Network Emulation functionality for testing protocols by emulating the properties of wide area networks. NetEm is an enhancement of the Linux traffic control facilities that allow to **add delay, packet loss**, duplication and more other characteristics to packets outgoing from a selected network interface.

1. Emulating wide area network delays

WARNING: disable offloading capabilities of the Ethernet driver since those are known to interfere with linux TC.

This is the simplest example; it just adds a fixed amount of delay to all packets going out of the local Ethernet.

```
# tc qdisc add dev eth0 root netem delay 100ms
```

Now a simple ping test to host on the local network should show an increase of 100 milliseconds. The delay is limited by the clock resolution of the kernel (HZ). On most 2.4 systems, the system clock runs at 100hz which allows delays in increments of 10ms. On 2.6, the value is a configuration parameter from 1000 to 100 hz.

You can change the delay with

```
# tc qdisc change dev eth0 root netem delay 10ms
```

Real wide area networks show variability, so it is possible to add random delay variation. Supported random variables can be specified via the following syntax:

```
DELAY := delay TIME [ JITTER [ CORRELATION ] ]  
[ distribution { uniform | normal | pareto | paretonormal } ]
```

```
# tc qdisc change dev eth0 root netem delay 100ms 10ms
```

This causes the added delay to be $100\text{ms} \pm 10\text{ms}$, according to a uniform distribution.

Delay distribution

Typically, the delay in a network is not uniform. It is common to use a something like a normal distribution to describe the variation in delay. The netem discipline can take a table to specify a non-uniform distribution.

```
# tc qdisc change dev eth0 root netem delay 100ms 20ms distribution normal
```

The actual tables (normal, pareto, paretonormal) are generated as part of the iproute2 compilation and placed in /usr/lib/tc; so it is possible with some effort to make your own distribution based on experimental data.

Packet loss

Random packet loss is specified in the 'tc' command in percent.

```
# tc qdisc change dev eth0 root netem loss 0.1%
```

This causes 1/10th of a percent (i.e 1 out of 1000) packets to be randomly dropped, according to an i.i.d. process. An optional correlation may also be added. This causes the random number generator to be not i.i.d. anymore and can be used to emulate packet burst losses.

```
# tc qdisc change dev eth0 root netem loss 0.3% 25%
```

This will cause 0.3% of packets to be lost, and each successive probability depends by a quarter on the last one, i.e., $P_{\text{loss}_n} = .25 * P_{\text{loss}_{n-1}} + .75 * \text{Random}(0.3\%)$

2. TBF – Token Buffer Filter

The Token Bucket Filter is a classless queueing discipline available for traffic control with the `tc` command. TBF is a pure shaper and never schedules traffic. It is non-work-conserving and may throttle itself, although packets are available, to ensure that the configured rate is not exceeded.

As the name implies, traffic is filtered based on the expenditure of **tokens**. Tokens roughly correspond to bytes, with the additional constraint that each packet consumes some tokens, no matter how small it is. This reflects the fact that even a zero-sized packet occupies the link for some time. On creation, the TBF is stocked with tokens which correspond to the amount of traffic that can be burst in one go. Tokens arrive at a steady rate, until the bucket is full. If no tokens are available, packets are queued, up to a configured limit. The TBF now calculates the token deficit, and throttles until the first packet in the queue can be sent. If it is not acceptable to burst out packets at maximum speed, a peakrate can be configured to limit the speed at which the bucket empties. This peak rate is implemented as a second TBF with a very small bucket, so that it doesn't burst.

This limit is caused by the fact that the kernel can only throttle for at minimum 1 'jiffy', which depends on HZ as $1/\text{HZ}$. For perfect shaping, only a single packet can get sent per jiffy - for HZ=100, this means 100 packets of on average 1000 bytes each, which roughly corresponds to 1mbit/s.

To attach a TBF with a sustained maximum rate of 5mbit/s, a 5kilobyte burst, with a pre-bucket queue size limit calculated so the TBF causes at most 10ms of latency, issue:

```
# tc qdisc add dev eth0 root tbf rate 5mbit burst 5kb latency 10ms
```

Rate control

By combining NetEm and a TBF it is then possible to emulate a link with given delay, packet loss and capacity:

```
# tc qdisc add dev eth0 root handle 1:0 netem delay 100ms
```

```
# tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 5mbit burst 5kb  
latency 10ms
```

Check on the options for buffer and limit as you might find you need bigger defaults than these (they are in bytes)

3. TCP Probe

Linux offers lots of possibilities for experimenting with TCP. The Linux kernel allows one to easily change the TCP congestion control algorithm, and to print internal TCP state variables, like congestion and receiver windows (CWND or RWND). `tcpprobe` is a kernel module that records the state of a TCP connection in response to incoming packets. To enable it, you need to load the module via the `modprobe` command. On the host you monitor, do:

```
H1: modprobe tcp_probe port=5001 full=1 bufsize=80
```

This enables the logging of TCP statistics on flows using TCP port 5001 (e.g., used by `iperf`). To check the log, you need to monitor the output that is recorded to virtual file in the `/proc/net/tcpprobe` file handle. For instance, on H1

```
cat /proc/net/tcpprobe >/tmp/tcpprobe.out & # start logging data
sleep 1 # just wait 1s to be sure the logger is started...
pid=$! # get the cat process ID to later kill it
iperf -c H2 # run the iperf test
kill $pid # now kill the logger
```

The `tcpprobe` capture file contains one line for each packet sent, according to the following syntax (warning: some extra columns may be present depending on the version of the Linux Kernel you are using)

```
[1] [2] [3] [4] [5] [6] [7] [8] [9]
0.073678 10.8.0.54:38644 192.168.1.42:5001 24 0xb6b19bb 0xb6b19bb 2 2147483647 5792
```

```
[1] Time seconds since tcpprobe was loaded
[2] Sender address:port
[3] Receiver address:port
[4] Packet length in Bytes
[5] snd_nxt: Sequence Number of the next packet to send
[6] snd_una: Sequence Number of the first unacknowledged packet
[7] snd_cwnd: Sender congestion window
[8] snd_ssthresh: Slow start threshold; -1 not known yet
[9] snd_wnd: Send Window size in Bytes
```

An event is logged **at the reception of a segment**. It works by inserting a hook into the `tcp_rcv` processing path so that the congestion window and sequence number can be captured.

This text file can be easily filtered and modified with standard tools. A common usage is to make a plot of congestion window and slow start threshold over time using `gnuplot`.

```
set data style linespoints
show timestamp
set xlabel "time (seconds)"
set ylabel "Segments (cwnd, ssthresh)"
plot "/tmp/tcpprobe.out" using 1:7 title "snd_cwnd", \
      "/tmp/tcpprobe.out" using 1:($8>=2147483647 ? 0 : $8) title
"snd_ssthresh"
```

4. Impact of RTT and Packet Loss on Congestion Control

Before doing the tests, first double check the system configuration. Verify that Ethernet card is set to 100Mb/s Full Duplex, and offloading capabilities are disabled.

In addition, by default, Linux TCP implements a caching feature that remembers the last valid slow start threshold (ssthresh) achieved in previous connections to the same host. This modifies the result of the further tests, and can causes issues when the properties of the path to the same host have changed in the meanwhile. If the test involves repeated connections, you should also turn off the route caching metrics:

```
sysctl -w net.ipv4.tcp_no_metrics_save=1
```

Check which congestion control algorithms are supported by your Linux distribution. Those are kernel modules that can be loaded on demand. Check which are available by just listing the files in

```
ls /lib/modules/`uname -r`/kernel/net/ipv4/tcp_*
```

You can then load some module using the `modprobe` tool

```
modprobe tcp_vegas #load tcp_vegas for testing
```

and enable it by default by

```
sysctl net.ipv4.tcp_congestion_control=vegas
```

To check which is the currently default TCP congestion control, use

```
sysctl net.ipv4.tcp_congestion_control
```

To check which are the currently enabled, use

```
sysctl net.ipv4.allowed_tcp_congestion_control
```

1. Which is the default congestion control on your distribution?
2. Write a script that loads all TCP congestion control modules, enables them, then turn TCP RENO as default

The goal of this laboratory is to verify the impact of RTT and packet loss (p_loss) on TCP congestion control performance. To this end, we will

- Use linux TC to control RTT and p_loss on the sender
- Use tcp_probe to log detailed statistics of TCP tests
- Use iperf to run the test
- Use gnuplot to plot results

We will consider the impact of TCP parameters:

- Maximum allowed values for the TCP congestion window (-w option on iperf)
- Usage of multiple TCP connection in parallel (-P option in iperf)
- TCP congestion control algorithm (-Z option on iperf)

The goal is to obtain plots for different values of the above system parameters:

- Application goodput versus RTT
- Application goodput versus p_loss

Here are some possible experiments to be done. For each of them, define the goals of your experiment, describe how to setup the testbed, define the parameter range, and then run the experiment. Report and comment the results. To highlight differences, you must choose a proper scenario, e.g., consider small/large RTT, with/without losses. When choosing the parameters, some ingenuity must be used to properly set the ranges, eventually increasing the experiments in regions where the changes are sharper.

Test A – impact of RTT - Plot the goodput versus RTT in [0:300]ms range for:

1. different values of the congestion window
2. increasing number of parallel TCP connections
3. different TCP congestion control algorithms

Test B – impact of p_loss - Plot the goodput versus p_loss in [0:10]% range for:

1. RTT equal to 10, 50, 100, 200ms
2. RTT equal to 50ms, and different TCP congestion control algorithms

Test C – impact of congestion control algorithm - Plot the evolution of the cwnd for different congestion control algorithms, for:

1. For small RTT (e.g., 10ms), p_loss=0 and p_loss=0.5%
2. For large RTT (e.g., 300ms), p_loss=0 and p_loss=0.5%

Test D: TCP fairness: consider the scenario RTT=50ms, p_loss=0.1%. Consider two TCP flows that compete for the link capacity. One uses RENO, the second uses any other algorithm X. Compare

1. The evolution versus time of the congestion window of each flow
2. The average goodput obtained of TCP reno and TCP X

Instead of enforcing a p_loss=0.1% using netem, you could also consider the actual losses induced by the two TCP flows that compete for capacity in a shared link: H1 and H2 send data to H3, so that congestion is created on the output port from the switch to H3 (hint: disable PAUSE frame otherwise no losses will happen).

Suggestions: Choose the appropriate duration to get a reliable test. Repeat the tests several times. Write a script to run test automatically, save results, and then use gnuplot to check what you have obtained.

Here is an example.

```

#!/bin/bash

# choose parameter
TARGET=192.168.1.112 # iperf server to contact
DELAY=3ms             # extra delay for netem
LOSS="1%"             # losses for netem
LIMIT=1000            # limit for netem
DEV=eth0              # which Ethernet card to use
TCP_WIN="512k"        # default tcp buffer size (note to self: remember to
                      # check the real one iperf will use)

# choose which TCP congestion control algorithms to test
# CongContr="bic cubic reno highspeed htcp hybla illinois lp scalable vegas
# veno westwood yeah"
CongContr="bic cubic reno vegas"

# insert the tcp_probe module and bind it to iperf port.
# Enable full logging, and limit the buffering to 80 lines to be sure
# everything will be logged
rmmod tcp_probe
modprobe tcp_probe port=5001 full=1 bufsize=80

#configure linux tc
tc qdisc add dev $DEV root netem
tc qdisc change dev $DEV root netem delay $DELAY loss $LOSS limit $LIMIT
tc qdisc show dev $DEV

# run ping to be sure that everything is in place
ping $TARGET -c 3

# By default, TCP saves various connection metrics in the route cache
# when the connection closes, so that connections established in the
# near future can use these to set initial conditions. Usually, this
# increases overall performance, but may sometimes cause performance
# degradation. If set, TCP will not cache metrics on closing
# connections.
sysctl -w net.ipv4.tcp_no_metrics_save=1

# Disable/enable selected acknowledgments (SACKS)
sysctl -w net.ipv4.tcp_sack=0

# now run the tests versus different congestion control algorithms
for CC in $CongContr
do
    echo "Testing $CC as Congestion Control"
    # start the TCP logger
    cat /proc/net/tcpprobe >tcp_data_${CC}.out &
    # wait for 1 s to let cat start properly
    sleep 1
    pid=$!
    iperf -c $TARGET -t 10 -Z $CC -w $TCP_WIN
    sleep 1
    kill $pid
done
# now plot them with gnuplot
gnuplot plot.gnu

```

```
# This is plot.gnu file

set terminal png
set out "TCP-cwnd-vs-cong-control.png"
set style data lines
set title "TCP Congestion Control"
set xlabel "time (seconds)"
set ylabel "Segments (cwnd, ssthresh)"
plot \
    "tcp_data_reno.out" using 1:7 title "snd_cwnd - reno", \
    "tcp_data_bic.out" using 1:7 title "snd_cwnd - bic", \
    "tcp_data_cubic.out" using 1:7 title "snd_cwnd - cubic", \
    "tcp_data_vegas.out" using 1:7 title "snd_cwnd - vegas"
```

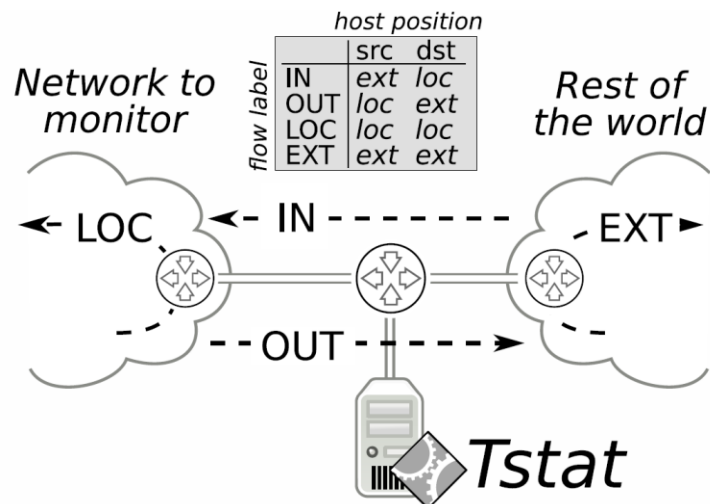
Networking Lab 10 – Passive monitoring

Network monitoring has always played a key role in understanding telecommunication networks since the pioneering time of the Internet. Today, monitoring traffic has become a key element to characterize network usage and users' activities, to understand how complex applications work, to identify anomalous or malicious behaviors. In this lab, we will start using Tstat, a free open source passive monitoring tool that has been developed in the past ten years. Started as a scalable tool to continuously monitor packets that flow on a link, Tstat has evolved into a complex application that gives to network researchers and operators the possibility to derive extended and complex measurements via advanced traffic classifier.

1. Tstat

Tstat initial design objective was to automate the collection of TCP statistics of traffic aggregate, using real time monitoring features. Over the years, Tstat evolved into a more complex tool offering rich statistics and functions. Developed in ANSI C for efficiency purposes, Tstat is today an Open Source tool that allows sophisticated multi-Gigabit per second traffic analysis to be run live using common hardware. Tstat design is highly flexible, with several plug-in modules offering different capabilities that are briefly described in the following. Being a passive tool, live monitoring of Internet links, in which all transmitted packets are observed, is the typical usage scenario. Fig.

1 sketches the common setup for a probe running Tstat: on the left there is the network to monitor, e.g., a network, that is connected to the Internet through an access link that carries all packets originated from and destined to terminals in the monitored network. The Tstat probe observes the packets and extracts the desired information. Note that this scenario is common to a wide set of passive monitoring tools.



2. Monitored objects

The basic objects that passive monitoring tools considers are the *IP packets* that are transmitted on the monitored link. *Flows* are then typically defined by grouping, according to some rules, all packets identified by the same *flowID* and that have been observed in a given time interval. A common choice is to consider

```
flowID = (ipProtoType, ipSrcAddr, srcPort, ipDstAddr, dstPort)
```

so that TCP and UDP flows are automatically considered, and tracked. For example, in case of TCP, a new flow start is commonly identified when the TCP three-way handshake is observed; similarly, its end is triggered when either the proper TCP connection teardown is seen, or no packets have been observed for some *idle* time. Similarly, in case of UDP, a new flow is identified when the first packet is observed, and it is ended after an *idle* time. As Internet conversations are generally bidirectional, the two opposite unidirectional flows (i.e., having symmetric source and destination addresses and ports) are typically grouped and tracked as connections. This allows gathering separate statistics for client-to-server and server-to-client flow, e.g., the size of HTTP client request and server HTTP response. Furthermore, the origin of information can be distinguished, so that it is possible to separate local hosts from remote hosts in the big Internet. As depicted in Fig. 1, traffic is then organized in four classes:

- **incoming traffic:** the source is remote and the destination is local;
- **outgoing traffic:** the source is local and the destination is remote;
- **local traffic:** both source and destination are local;
- **external traffic:** both source and destination are remote.

This classification allows to separately collecting statistics about incoming and outgoing traffic; for example, one could be interested in knowing how much incoming traffic is due to YouTube, and how many users access Facebook from the monitored network. The local and external cases should not be considered but in some scenarios they can be present. At packet, flow and application layers, a large set of statistics can be defined and possibly customized at the user's will.

Tstat can analyze traffic traces, extracting as much information as possible from the pure passive observation of packets. It first rebuilds flows, and, for each flow, a number of statistics are collected. Once the flow is terminated, Tstat logs it in a simple text file, in which each column is an attribute of the flow. Several "*log files*" are then created, one per type of flows (e.g., TCP, UDP, Video, etc.). A *log file* is arranged as a simple table where each column is associated to specific information and each line reports the two unidirectional flows of a connection. Different flow-level logs are available, e.g., the log of all UDP flows, or the log of all VoIP calls. The log information is a summary of the connection or flow properties. For instance, the starting time of a TCP connection, its duration, the number of sent and received packets, the observed Round Trip Time are all valuable metrics that allow to monitor the TCP performance. The file containing the log of monitored TCP connections is called `log_tcp_complete`. Table 1 reports the description of each fields, while table 2 reports the classification of "Connection Type" used in field 101. More detailed statistics can be found on the Tstat web site at <http://www.tstat.polito.it>, in the referenced papers², and on the references listed in the [Publications](#) section. As it can be seen, the amount of information exposed by Tstat is large, and allows to easily computing other metrics by combining them.

See the file "*log_tcp_complete_description.pdf*" for more details.

3. Analysis of TCP logs

In this laboratory, the PC must be normally connected to the LAIB-4 network since we will use it as normal unix terminal. Most of the analysis can be also run on any PC, possibly using other software and any operating system.

Warning: if you cannot connect to the didattica.polito.it portal, it may be because the DNS is not properly configured in your system. Check the file `/etc/resolv.conf`. If it does not exist, try the following

```
# delete the wrong configuration
sudo rm /etc/resolv.conf

# replace with the one got from the DHCP
sudo ln -s /run/resolvconf/resolv.conf /etc
```

We will consider TCP logs that have been collected from operative networks and will learn how to post-process them to extract valuable information. To this goal, simple post-processing scripts will be required, implemented using any scripting languages, like `awk`, `perl`, `python`, etc. By combining them with some

² Alessandro Finamore, Marco Mellia, Michela Meo, Maurizio M. Munafò, Dario Rossi, *Experiences of Internet Traffic Monitoring with Tstat*, IEEE Network "March/April 2011", Vol.25, No.3, pp.8-14, ISSN: 0890-8044, March/April 2011

shell tools like `grep`, `sort`, `cut`, etc., it will be easy to derive aggregated statistics from the `log_tcp_complete` file. Remember that you can always check the command man page, and you can look for support in the Internet.

For this lab, we will use the file “`log_tcp_complete.xz`” that **must be downloaded** from the <http://didattica.polito.it> portal. This log has been collected from an operative network during November 2015 and contains all TCP flows observed considering a one-hour long time interval. The file has been compressed using `xz`, an efficient compression program, which reduced its size to 339MB from the original 2.1GB. The file can be easily accessed by using `xzcat` and using a pipe to input the uncompressed data to the post processing script/command. For instance

```
xzcat log_tcp_complete.xz | head -n 10
```

will print the first 10 lines of the file on the screen.

WARNING: Although decompressing the file every time can slow down the processing, you should consider avoiding decompress the file on the laboratory computers since the system has very limited disk space! You could consider working on a decompressed file that contains only the rows or columns that you are interested into. To test and tune the scripts, you should limit the analysis to the first few lines of the log, by using the `head -n XX` command.

The typical steps to follow to get a desired figure are

1. *Define the metric* that has to be computed, e.g., “distribution of number of transmitted packets per flow”, or “average download throughput”, or “number of IP addresses serving facebook.com traffic”, etc.
2. *Define the set of flows* that are targeted, e.g., all HTTP traffic, or facebook.com flows coming from a given IP address, etc.
3. Compute the metric for the desired set of flows by writing a suitable script
4. Plot the results using for example `gnuplot`
5. Go back to 3 to refine the results
6. Go back to 1 to compute another metric and produce another plot

4. Example of scripts

For example, given the `log_tcp_complete.xz`, count

1. the number of total TCP connections
2. the number of TCP connections whose client is “local”
3. the number of TCP connections whose client is “local” and uses HTTP as L7-protocol
4. the number of TCP connections whose client is “local”, uses HTTP as L7-protocol and are directed to “*facebook.com”. Do the same for HTTPS flows
5. produce a histogram that counts, for each server IP address, the number of connections whose client is “local”, uses HTTP as L7-protocol, are directed to “*facebook.com”
6. as above, but sort the results in increasing number of flows
7. **plot the results using `gnuplot`**

5. SOLUTIONS

1. Simply count the number of rows in the file:

```
#using awk to count the number of lines
xzcat log_tcp_complete.xz | awk '{tot++} END {print "total number of
lines", tot}'
```

Other possibility:

```
xzcat log_tcp_complete.xz | awk 'END {print NR}' # recall that NR is
the current line number ...
```

or even simpler using the `wc` unix command

```
xzcat log_tcp_complete.xz | wc -l
```

2. This time we need to filter those flows whose client is marked as "internal"

```
# using awk to count the number of lines which take the value of
# "1" in the 38th column - client internal
xzcat log_tcp_complete.xz | awk '{if ($38==1) tot_int++} END
{print "number of flows initiated by internal clients", tot_int}'
```

3. We need to consider those lines in which the 38th field takes the value of 1 (client internal), and then 42nd connection type field takes the value of 1 (HTTP)

```
# using awk to count the number of lines which take
# the value of "1" in the 38th column - client internal
# the value of "1" in the 42nd column - HTTP protocol
# note: the "\" - backslash - character is useful to break a single
# line into multiple lines
xzcat log_tcp_complete.xz | \
awk '{if ($38==1 && $42==1) tot_int++}\
END {\
print "number of HTTP flows initiated by internal clients", tot_int\
}'
```

4. As above, but consider only those lines that match "facebook.com".

```
# using awk to count the number of lines which take
# the value of "1" in the 38th column - client internal
# the value of "1" in the 42nd column - HTTP protocol
# or the value of "8192" in the 42nd column - HTTPS protocol
# and match facebook.com in the line.
# let's use the "filtering" functionalities of awk!
xzcat log_tcp_complete.xz | \
awk ' /facebook.com/ {\
    if ($38==1 && $42==1) tot_int_fb_http++; \
    if ($38==1 && $42==8192) tot_int_fb_https++ \
}\
END {\
print "flows to facebook initiated by internal clients - HTTP", \
tot_int_fb_http \
print "flows to facebook initiated by internal clients - HTTPS", \
tot_int_fb_https \
}'
```

5. This time we need to count the number of flows per single IP addresses. We can use an associative array to count the number of flows directed to a given IP address. The server IP address will be used as "index" of the array.

```
# using awk to count the number of lines which take
# the value of "1" in the 38th column - client internal
# the value of "1" in the 42nd column - HTTP protocol
# or the value of "8192" in the 42nd column - HTTPS protocol
# and match facebook.com in the line.
# Count per IP address the number of flows
# so create and array this time
xzcat log_tcp_complete.xz | \
awk '/facebook.com/ {\
    if ($38==1 && $42==1) tot_int_fb_http[$15]++; \
    if ($38==1 && $42==8192) tot_int_fb_https[$15]++ \
} \
END { \
    print "#IP address\t number of flows - HTTP flows"; \
    for (addr in tot_int_fb_http) \
        print "HTTP: " addr "\t" tot_int_fb_http[addr]; \
    print "#IP address\t number of flows - HTTPS flows"; \
    for (addr in tot_int_fb_https) \
        print "HTTPS: " addr "\t" tot_int_fb_https[addr]; \
} \
}'
```

6. As above, plus simply use the sort command.

```
# using awk to count the number of lines which take
# the value of "1" in the 38th column - client internal
# the value of "1" in the 42nd column - HTTP protocol
# or the value of "8192" in the 42nd column - HTTPS protocol
# and match facebook.com in the line.
# Count per IP address the number of flows
# so create and array this time
xzcat log_tcp_complete.xz | \
awk '/facebook.com/ {\
    if ($38==1 && $42==1) tot_int_fb_http[$15]++; \
    if ($38==1 && $42==8192) tot_int_fb_https[$15]++ \
} \
END { \
    print "#IP address\t number of flows - HTTP flows"; \
    for (addr in tot_int_fb_http) \
        print "HTTP: " addr "\t" tot_int_fb_http[addr]; \
    print "#IP address\t number of flows - HTTPS flows"; \
    for (addr in tot_int_fb_https) \
        print "HTTPS: " addr "\t" tot_int_fb_https[addr]; \
} \
}' | sort -n -k2
```

A much better way to organize the work is to create an AWK scripts instead of typing it on the command line directly. This can be done by simply creating a text file that contains the AWK commands, and then running it.

The above single-line-long-command can then be simplified as follows:

Create the file “count_servers.awk” that contains

```
/facebook.com/ {
    if ($38==1 && $42==1) tot_int_fb_http++;
    if ($38==1 && $42==8192) tot_int_fb_https++
}
END {
print "flows to facebook initiated by internal clients - HTTP",
tot_int_fb_http
print "flows to facebook initiated by internal clients - HTTPS",
tot_int_fb_https
}
```

Then, run it as before

```
xzcat log_tcp_complete.xz | awk -f count_servers.awk
```

Suggestions: when testing the scripts, you can run it on a smaller portion of the file to speed up testing, e.g., considering the first 10000 lines or so. Use the command `head -n 10000` for this.

```
xzcat log_tcp_complete.xz | head -n 1000 | awk -f count_servers.awk
```

1. Modify the previous script to find which is the facebook.com most used service. What if you check “facebook” instead of “facebook.com”? (Consider the name in column 127 as “service”)
2. What are the top most used “services” in general?
3. **Write a script that counts the fraction of connections for different “connection type”. Plot the results as a histogram.**
4. **Write a script that counts the fraction of HTTP flows every 10,000 connections. Plot the results over time.**
5. **Add a second line to the above plot that reports the fraction of HTTPS connections over time.**
6. **Add a third line with the number of other protocols flows over time.**
7. **Consider a given service (e.g., /facebook/, or /whatsapp/, or /google/, or /scorecardresearch/, ...)**
 1. **Plot the number of flows per each sub-service**
 2. **Count the number of flows handled by each IP addresses matching the service. Plot the rank of IP server addresses, sorted from the one handling the most flows, to the least.**
 3. **Count the client IP addresses handled by each IP addresses matching the service. Plot the rank of IP server addresses, sorted from the one handling the most clients, to the least. Suggestion: use arrays of arrays, e.g., a[server][client] = 1...**
 4. **Count the total bytes handled by each IP addresses matching the service. Plot the rank of IP server addresses, sorted from the one handling the largest amount of traffic, to the least**
 5. **For each server IP address matching the service, extract the minimum and average RTT from the client to the server. Plot them, sorted from the one handling the most clients, to the least.**

8. What is the next script doing? Try to understand it

```
xzcat log_tcp_complete.xz | \  
head -n 1000000 | awk -f count_server_and_names.awk \  
sort -n -k2
```

Content of the "count_server_and_names.awk" file

```
{  
    if ($38==1 && $127 ~ /googlevideo/) {  
        tot_int[$15]++;  
        alreadyIn = match (name[$15], $127);  
        if (alreadyIn==0)  
            name[$15]=name[$15] ", " $127  
    }  
}  
END {  
    print "number\tIP address\tnames";  
    for (addr in tot_int) {  
        print tot_int[addr] "\t" addr "\t" name[addr]  
    }  
}
```