Computer Networks

Lab 2 - Computer Networks

up202007163@fe.up.pt Bárbara Ema Pinto Rodrigues
up202000162@fe.up.pt Igor Rodrigues Diniz
up202008462@fe.up.pt José Luís Cunha Rodrigues

December 22

Contents

1 Introduction

This report describes the work done for the second assignment of the Computer Networks class. The project is divided into two parts described below.

The first part is the development of an FTP download application. The idea is to download a single file using a TCP connection and following the FTP protocol described in RFC959.

The second phase of the assignment is about the configuration and study of a network. Through a series of experiments, we got to play with different settings and configurations of networks and implement a set of different mechanisms for them to work together (e.g. bridge, router).

2 Download application

This section describes the download application developed by explaining its structure and flow of execution.

2.1 Process

2.1.1 Introduction

The application was developed in C language and uses sockets together with a TCP connection. The execution starts by parsing out the user-provided URL through the invocation of the `parseUrl()` function. Relevant fields such as user, password, host and path are divided and stored for future use.

2.1.2 Establishing a connection

A TCP connection is then established using the provided host by calling the `openControlConnection()` predicate. This function, upon receiving the struct `url`, retrieves the host's IP address. Then, it calls the `openConnection()` function with that address so it creates a socket that allows for communication with the server. Invoking `cleanSocket()` at the end of this process is done to clear the remaining input in the socket.

2.1.3 Login

At this stage, the user-given credentials are fed to the server to perform the login by sending `user` and `pass` commands. If none are given, the application performs an anonymous login. Both responses are read with `readCode()` function.

2.1.4 Passive mode

After the first connection is established, we can now use passive mode to get the file. To do that, the server is asked to change to passive mode by sending the `pasv` command. After that, the programs parses the address and port for the new data connection. Then, using the provided port number a new connection is established by the use of the function `openConnection()`. The latter returns a file descriptor for the socket from which the data will be retrieved. This will be referred to as "data connection".

2.1.5 Getting the file

When both connections are established, the file is then requested from the server using the first connection (i.e. "control connection"). The function `getFile()` sends the command `retr [path]` so the server outputs the file to be downloaded. After this is done, the data connection is used to read the file and thus the download is complete.

2.2 Architecture

The application is structured into three modules: `main`, `connection` and `utils`.

2.2.1 main

This is the smallest module. What it does is connect all the functionalities and perform the downloading process by calling them in the right order. Figure 1 portraits a function call scheme for the application.

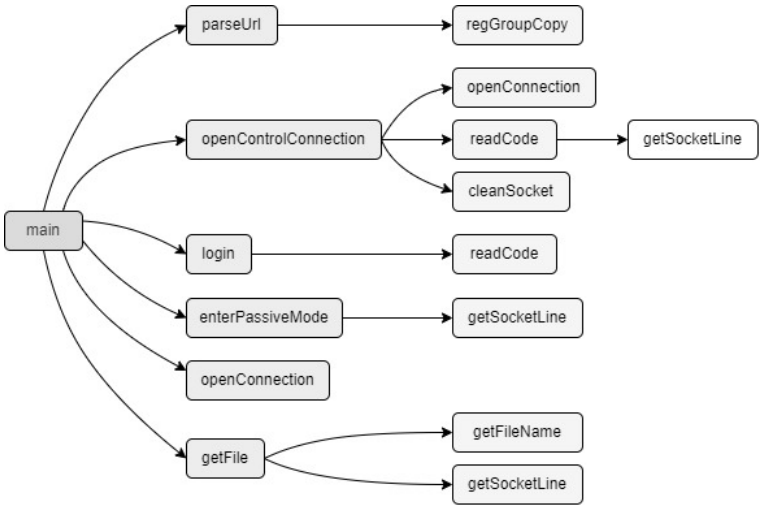


Figure 1: Download Function call graph.

2.2.2 connection

The `connection` module is responsible to define the functions related to the TCP connections. Therefore, it defines the following set of functionalities to work with the connections:

- Open connection.
- Authentication.
- Enter passive mode.
- Get a file from the server.

2.2.3 utils

The latter section provides utilities needed for the program, namely handling I/O. Its functionalities include:

- Parsing a user-give URL.
- Read a line from a socket.
- Read input from sockets and interpret its meaning.

3 Configuration and Study of a Network

3.1 Exp. 1 - Configure an IP Network

This experience aims to understand what ARP packets are and what they are for, what type of packets the ping command generates and what MAC addresses and IP addresses are. The network should be set up according to the schema in Figure 2.

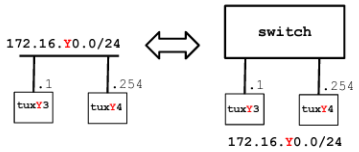


Figure 2: Experience 1 - Architecture.

3.1.1 Steps

1. We connect `tux3` and `tux4` to any of the switch's ports (except port 1) with `eth0`.
   - TUX13E0: Switch Port 3
   - TUX14E0: Switch Port 4
2. Next, configure both machines and Register the IP and MAC addresses:
   - tux3: `ifconfig eth0 172.16.10.1/24`
   - tux4: `ifconfig eth0 172.16.10.254/24`

3.1.2 Analysis

The MAC address is a unique identifier of a machine (i.e. no two machines can have the same address) and belongs to the link layer. An IP address, on the order hand, is a part of the internet layer and is used to identify a machine within a network. We can check both by the command `ifconfig` entered on the tux console.

IP	MAC	tux	ether
172.16.10.1	00:21:5a:5a:7d:16	tux3	eth0
172.16.10.254	00:c0:df:25:13:65	tux4	eth0

Table 1: IP addresses, MAC addresses and corresponding interfaces in Exp1.

If we execute ping command between `tux3` and `tux4` we verify connectivity between the computers. The ping command works by generating **ICMP packets**. It sends an ICMP request and waits for an ICMP reply (Figure 3).

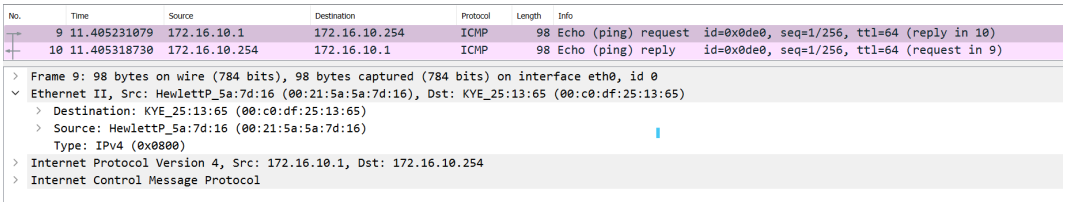


Figure 3: Target and destination addresses for ICMP packets.

For an ICMP request packet, the IP and MAC addresses will be the ones from the requesting machine, while the destination addresses will match the destination machine. For an ICMP response, these values would be the other way around (Figure 3).

The Address Resolution Protocol (ARP) aims to create a mapping between an internet layer address (IP address) and a link layer address (MAC address). This means that, given an IP address of a machine, ARP is used to find that machine's MAC address.

The `arp -a` inspects the **ARP tables** of a tux. When we execute the `arp -d` ≪`IP_address`≫ command, the ARP tables entries will be deleted. The ARP process will need to be executed before `tux3` can ping `tux4` (Figure 4).

No.	Time	Source	Destination	Protocol	Length	Info
7	11.405121079	HewlettP_5a:7d:16	Broadcast	ARP	42	Who has 172.16.10.254? Tell 172.16.10.1
8	11.405223466	KYE_25:13:65	HewlettP_5a:7d:16	ARP	60	172.16.10.254 is at 00:c0:df:25:13:65

Figure 4: ARP request and response packets.

The first line of Figure 4 represents an ARP packet sent in broadcast by `tux3`. Its MAC and IP source addresses are the same as for `tux3`. The target IP address is the same as `tux4`'s but the target MAC field is filled with zeros, as the sender does not know this value (Figure 5). The machine that recognizes its MAC address, will then send a reply to `tux3`.

```
Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: HewlettP_5a:7d:16 (00:21:5a:5a:7d:16)
    Sender IP address: 172.16.10.1
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 172.16.10.254
```

Figure 5: ARP request sent by tux3.

In turn, we can take a look into the `tux4`'s ARP response (Figure 6). This time, the source MAC and IP addresses are the ones from `tux4`, while the target's addresses are the ones from `tux3`.

```
Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: KYE_25:13:65 (00:c0:df:25:13:65)
    Sender IP address: 172.16.10.254
    Target MAC address: HewlettP_5a:7d:16 (00:21:5a:5a:7d:16)
    Target IP address: 172.16.10.1
```

Figure 6: ARP response sent by tux4.

The **Ethernet header** makes the distinction between ARP and IP packets. For IP, this is `0x0800` and `0x806` for ARP. In turn, an ICMP packet is distinguished by the IP header, which is `1`. All of these are also identified by Wireshark.

We can also determine the length of a receiving frame with Wireshark, which displays it at the length field.

The **loopback interface** is a virtual network interface that allows for clients and servers on the same host to communicate. The mechanism can be used to troubleshoot problems, perform tests, and access servers running on the same machine, among others. An example of this interface is the `localhost` on Linux.

3.2 Exp. 2 - Implement two bridges in a switch

This experience aims to create two Bridges on the switch (see Figure 7) and understand the connectivity between the machines configured in each of the subnets.

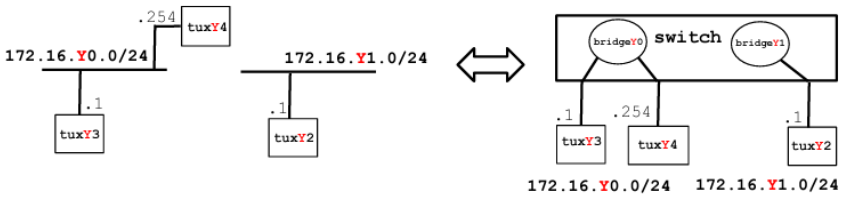


Figure 7: Experience 2 - Architecture.

3.2.1 Steps

1. Taking part of the configuration of the previous experience, now, we connect `tux2` with eth0 to the port 2 and make it configuration.
   - TUX12E0: Switch Port 2
   - `ifconfig eth0 172.16.11.1/24`
2. To create the Bridges it is necessary to modify the switch, for this we choose `tux3` and connected the serial port of that machine to the serial port of the switch console.
   - TUX13S0 - RS232 CISCO
   - CISCO RS232 - Switch MicroTik console
3. Then we open the serial port /dev/ttyS0 on `tux3` using o GTKterm, set the Baudrate to 115200 bps and make the login.
4. We create the two bridges, then, for each tux, remove the default connections to the virtual bridge and after that, we can connect the tuxs to the bridges. We will connect `tux3` and `tux4` to brigde10 and `tux2` to bridge11:
   - `/interface bridge add name = bridge10`
   - `/interface bridge add name = bridge11`
   - `/interface bridge port remove [find interface = ether2]`
   - `/interface bridge port remove [find interface = ether3]`
   - `/interface bridge port remove [find interface = ether4]`
   - `/interface bridge port add interface=ether3 bridge=10`
   - `/interface bridge port add interface=ether4 bridge=10`
   - `/interface bridge port add interface=ether2 bridge=11`
5. In order to do ping broadcasts we will need to unblock the traffic of ICMP ECHO request packets in all the machines:
   - `echo 0 > proc/sys/net/ipv4/icmp_echo_ignore_broadcasts`

3.2.2 Analysis

Bridges are created on the switch. We have chosen `tux3` to be the machine connected to the switch serial port. All the machines are connected to a **virtual bridge** in the switch by default, so we need to delete those connections first. Then, we can create both bridges and connect the machines. To respect the architecture of this experience (Figure 7) the respective port of `tux3` on the switch is connected to `bridge10` as well as the one from `tux4`. In addition, the respective port of `tux2` on the switch is

connected to bridge11.

A **broadcast ping** sends packets to every host on the broadcast domain. It allows a machine to connect with every reachable address. In `tux3`, a broadcast ping was performed (`ping -b 172.16.10.255`), which only got a response from `tux4`, as shown in figure 8.

No.	Time	Source	Destination	Protocol	Length	Info
26	42.171077885	172.16.10.0	172.16.10.255	ICMP	98	Echo (ping) request id=0x1ac0, seq=1/256, ttl=64
27	42.171277763	172.16.10.254	172.16.10.0	ICMP	98	Echo (ping) reply id=0x1ac0, seq=1/256, ttl=64
28	43.171523644	172.16.10.0	172.16.10.255	ICMP	98	Echo (ping) request id=0x1ac0, seq=2/512, ttl=64
29	43.171686019	172.16.10.254	172.16.10.0	ICMP	98	Echo (ping) reply id=0x1ac0, seq=2/512, ttl=64

Figure 8: Tux3 broadcast ping logs.

We repeated the previously described broadcast ping process but now from `tux2`. This time there was no response, as no other device is configured on bridge11 rather than `tux2` itself (Figure 9).

No.	Time	Source	Destination	Protocol	Length	Info
2	0.020124440	172.16.11.1	172.16.11.255	ICMP	98	Echo (ping) request id=0x3e05, seq=12/3072, ttl=64 (no response found!)
3	1.044125165	172.16.11.1	172.16.11.255	ICMP	98	Echo (ping) request id=0x3e05, seq=13/3328, ttl=64 (no response found!)
5	2.068117300	172.16.11.1	172.16.11.255	ICMP	98	Echo (ping) request id=0x3e05, seq=14/3584, ttl=64 (no response found!)
6	3.092122495	172.16.11.1	172.16.11.255	ICMP	98	Echo (ping) request id=0x3e05, seq=15/3840, ttl=64 (no response found!)
8	4.116115537	172.16.11.1	172.16.11.255	ICMP	98	Echo (ping) request id=0x3e05, seq=16/4096, ttl=64 (no response found!)
9	5.140121710	172.16.11.1	172.16.11.255	ICMP	98	Echo (ping) request id=0x3e05, seq=17/4352, ttl=64 (no response found!)

Figure 9: Tux2 broadcast ping logs.

A **broadcast domain** is a set of nodes, part of a network, that can all communicate with each other through broadcasting. From here we can conclude that there are two broadcast domains that correspond to `bridge10` and `bridge11` subnets with addresses `172.16.10.255` and `172.16.11.255`, respectively.

3.3 Exp. 3 - Configure a Router in Linux

This experience aims to use `tux4` as a router in order to enable communication between `tux3` and `tux2`. This will be done with the two bridges that were set up in the previous experiment.

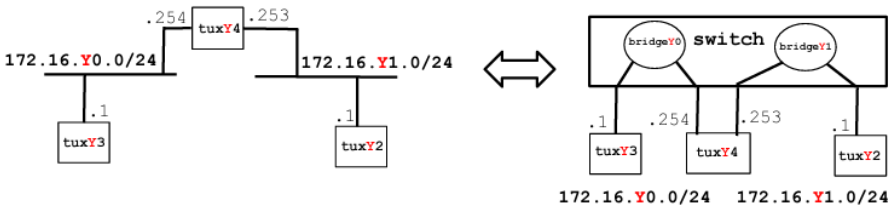


Figure 10: Experience 3 - Architecture.

3.3.1 Steps

1. Continuing with the configuration of the previous experience, we now connect and configure `tux4.eth1` on port 5.
   - TUX14E1: Switch Port 5
   - tux4: `ifconfig eth1 172.16.11.253/24`
2. To connect it to the bridge11 it is necessary to modify the switch again. So, by using the GTKterm we remove the default connection of `tux4.eth1` to the virtual bridge and then we can connect it to bridge11:
   - `/interface bridge port remove [find interface = ether5]`
   - `/interface bridge port add interface=ether5 bridge=11`
3. We have to enable IP forwarding and disable ICMP echo ignore broadcast in `tux4`:
   - `echo 1 > /proc/sys/net/ipv4/ip_forward`
   - `echo 0 > proc/sys/net/ipv4/icmp_echo_ignore_broadcasts`
4. Now we have to reconfigure `tux3` and `tux2` so that each of them can reach the other.
   - tux3: `route add -net 172.16.11.0/24 gw 172.16.10.254`
   - tux2: `route add -net 172.16.10.0/24 gw 172.16.11.253`

3.3.2 Analysis

Forwarding tables specify the gateway used to access the destination IP address whose interface is stored in the Iface column. They have a Genmask column that indicates the netmask for the destination net. These tables also contain Flags, Metric, and Ref columns which we will not describe since they aren't very relevant to our analysis. To see the routes of each tux, we can use the command `route -n` on the tux console and inspect the respective forwarding tables.

Destination	Gateway	Genmask	...	Iface
172.16.10.0	172.16.11.253	255.255.255.0	...	eth0
172.16.11.0	0.0.0.0	255.255.255.0	...	eth0

Table 2: Forwarding table for tux2.

The first line of `tux2` routes table (Table 2) is the route added by us in the step 4 and it means that to access the bridge with addresses `172.16.10.0/24` the IP `172.16.11.253` is used as a gateway. The second line has, as the gateway, the default one: `0.0.0.0`. When the gateway is all zeros, it means there is no gateway. So any packets that needs to access to the bridge with addresses `172.16.11.0/24`

don't need to be routed, as they are connected, so packets can be sent directly to the destination on the local network. Symmetrically, `tux3` routes end up being implemented with the same purpose (table 3).

Destination	Gateway	Genmask	...	Iface
172.16.10.0	0.0.0.0	255.255.255.0	...	eth0
172.16.11.0	172.16.10.254	255.255.255.0	...	eth0

Table 3: Forwarding table for tux3.

After analyzing the routes table for `tux4` (Table 4), we verify that only the default routes exist. That makes sense because `tux4` has a direct connection to both subnets, so the default gateway is sufficient to let `tux4` access both `172.16.10.0/24` and `172.16.11.0/24` addresses.

Destination	Gateway	Genmask	...	Iface
172.16.10.0	0.0.0.0	255.255.255.0	...	eth0
172.16.11.0	0.0.0.0	255.255.255.0	...	eth1

Table 4: Forwarding table for tux4.

With these routes defined, it is possible to ping, from `tux3`, all the interfaces of the other machines. When we ping `tux4.eth0` from `tux3`, it will search its ARP table for this resolution. If not, (i.e. the `tux4.eth0` IP address is not in the ARP table), `tux3` will send an ARP request to find that address.

A different procedure occurs when we ping `tux4.eth1` because it's not on `tux3`'s subnet. In this scenario, ARP will be used to find the MAC address of the router, because the destination IP address has already been deemed to not be directly reachable, so the packet must be delivered to a router that can take care of it. In this case, the packet will be sent to `tux4.eth0`, which will send it to `tux4.eth1`.

The same thing happens when we ping `tux2`. Because `tux3` and `tux2` are not in the same subnet, the packet will be sent to `tux4` first (our router), for then to be sent to `tux2` (which is on the same subnet of the router, by the `tux4.eth1`).

ARP tables are responsible for storing the address pairs that a specific device has communicated with. So, if we delete them, every time devices send packets to one another, they broadcast an ARP request to all the devices on the local subnet in order to discover the MAC address of the target device IP address. All the devices should ignore it, except the one that matches the required IP. That device will reply with its MAC address.

After deleting the tables, if we ping `tux2` from `tux3`, the described behavior occurs. Now, when `tux3` wants to send something to another tux, there's a need for an ARP Broadcast request to all the machines on its subnet. Only after `tux3` receives the reply, the MAC address for the target tux is updated on `tux3`'s ARP table. Thus, it now knows how to reach that tux for the following messages. These ARP requests can be seen in figures 11 and 12.

No.	Time	Source	Destination	Protocol	Length	Info
85	85.531128610	HewlettP_5a:7d:16	Broadcast	ARP	60	Who has 172.16.10.254? Tell 172.16.10.1
86	85.531147467	KYE_25:13:65	HewlettP_5a:7d:16	ARP	42	172.16.10.254 is at 00:c0:df:25:13:65
95	88.592467531	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) request id=0x25a2, seq=4/1024, ttl=64
96	88.592615874	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) reply id=0x25a2, seq=4/1024, ttl=63
97	89.616462459	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) request id=0x25a2, seq=5/1280, ttl=64
98	89.616603958	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) reply id=0x25a2, seq=5/1280, ttl=63
99	90.078992056	Routerbo_2b:84:84	Spanning-tree-(for-b…	STP	60	RST. Root = 32768/0/c4:ad:34:2b:84:7b Cost = 0 P
100	90.640445584	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) request id=0x25a2, seq=6/1536, ttl=64
101	90.640595394	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) reply id=0x25a2, seq=6/1536, ttl=63
102	90.688552939	KYE_25:13:65	HewlettP_5a:7d:16	ARP	42	Who has 172.16.10.1? Tell 172.16.10.254
103	90.688682704	HewlettP_5a:7d:16	KYE_25:13:65	ARP	60	172.16.10.1 is at 00:21:5a:5a:7d:16
104	91.664432969	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) request id=0x25a2, seq=7/1792, ttl=64
105	91.664614208	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) reply id=0x25a2, seq=7/1792, ttl=63

Figure 11: Logs captured in tux4 eth0.

No.	Time	Source	Destination	Protocol	Length	Info
109	88.403618392	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) request id=0x25a2, seq=1/256,
110	88.403766526	HewlettP_5a:7e:51	Broadcast	ARP	60	Who has 172.16.11.253? Tell 172.16.11.1
111	88.403773580	HewlettP_5a:7b:3f	HewlettP_5a:7e:51	ARP	42	172.16.11.253 is at 00:21:5a:5a:7b:3f
112	88.403859415	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) reply id=0x25a2, seq=1/256,
127	93.512813074	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) request id=0x25a2, seq=6/1536,
128	93.512933201	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) reply id=0x25a2, seq=6/1536,
129	93.560898848	HewlettP_5a:7b:3f	HewlettP_5a:7e:51	ARP	42	Who has 172.16.11.1? Tell 172.16.11.253
130	93.561016601	HewlettP_5a:7e:51	HewlettP_5a:7b:3f	ARP	60	172.16.11.1 is at 00:21:5a:5a:7e:51
131	94.536799202	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) request id=0x25a2, seq=7/1792,
132	94.536952294	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) reply id=0x25a2, seq=7/1792,

Figure 12: Logs captured in tux4 eth1.

An **ICMP request** contains the Ether layer, which has the target and source MAC address in it. It also contains the IP layer, which has the source and target IP and also a couple of flags included. The ARP is sent before ping messages because the fields of the MAC and IP addresses need to be filled in.

Figure 11 shows an exchange of ARP messages for `tux3` to learn the MAC address of the interface eth0 to `tux4` and vice versa, while in figure 12 there is an exchange of ARP messages for `tux2` to learn the MAC address of the interface eth1 from `tux4` and vice versa. After this exchange of ARP messages, the ICMP requests and replies have the fields of IP and MAC addresses totally filled, so it is possible to verify the communication between tux's 3 and 2 since the emitted pings obtain a response.

3.4 Exp. 4 - Configure a Commercial Router and Implement NAT

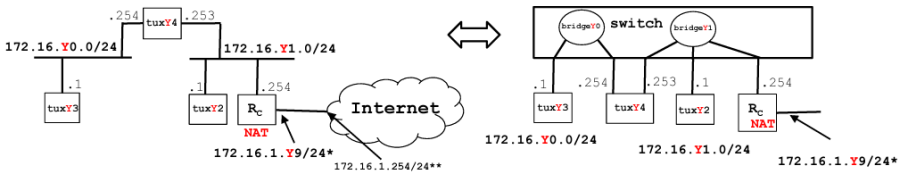


Figure 13: Experience - Architecture.

This experience aims to configure a commercial router as shown in figure 13 and establish a connection with the laboratory network. For that, we need to implement routes in the router as well as add NAT functionality to it.

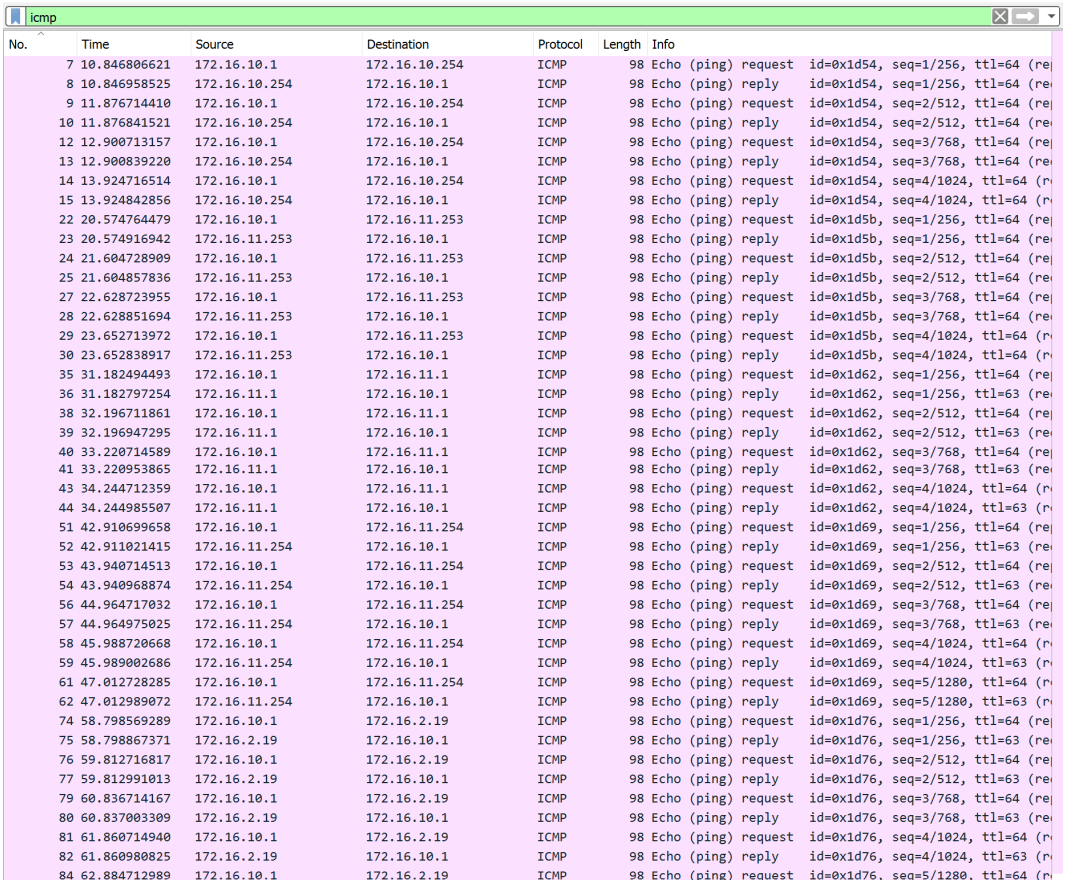
3.4.1 Steps

1. Taking advantage of the configuration of experiment 3, we only have to connect the router to the switch and the lab network. We connect ether1 of RC to the lab network on P1.1 (with NAT enabled by default) and ether2 of RC to a port on bridge11.
   - ROUTER.E2: Switch Port 6 (router - switch)
   - ROUTER.E1: Shelf Port 1 (router - lab network)
2. We connect the serial port of tux3 to the RouterMTIK port and we are able to configure the IP addresses of RC through GTKterm as well as connect it to bridge11:
   - `/ip address add address=172.16.1.19/24 interface=ether1`
   - `/ip address add address=172.16.11.254/24 interface=ether2`
   - `/interface bridge port remove [find interface = ether6]`
   - `/interface bridge port add interface=ether6 bridge=11`
3. Add the following routes to the tux's:
   - tux2: route add -net 172.16.1.0/24 gw 172.16.11.254
   - tux2: route add default gw 172.16.11.254
   - tux3: route add default gw 172.16.10.254
   - tux4: route add -net 172.16.1.0/24 gw 172.16.11.254
   - tux4: route add default gw 172.16.11.254
4. Add the following routes to RC through GtKterm (Note that 172.16.1.254 is the IP of the router of the lab I.321):
   - `/ip route add dst-address=0.0.0.0/0 gateway=172.16.1.254`
   - `/ip route add dst-address=172.16.10.0/24 gateway=172.16.11.253`

3.4.2 Analysis

In this experiment, we started by configuring the router's `eth2` interface assigned to `bridge11`. For the `eth1` interface of the router, IP `172.16.2.19` was assigned so that the connection with `netlab` network could be made.

Configuring a route in RC is done by accessing the Router's serial port with GtKterm. After all the configurations needed were done, the router's console should be accessible through `tux3` and it's now possible to add the necessary routes with the command `/ip route add dst-address=? gateway=?`, as it was done in step 4. After all the routes were added, it was possible to perform a ping from `tux3` to all other points in our network, as shown in Figure 14.



icmp

No.	Time	Source	Destination	Protocol	Length	Info
7	10.846806621	172.16.10.1	172.16.10.254	ICMP	98	Echo (ping) request id=0x1d54, seq=1/256, ttl=64 (re
8	10.846958525	172.16.10.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d54, seq=1/256, ttl=64 (re
9	11.876714410	172.16.10.1	172.16.10.254	ICMP	98	Echo (ping) request id=0x1d54, seq=2/512, ttl=64 (re
10	11.876841521	172.16.10.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d54, seq=2/512, ttl=64 (re
12	12.900713157	172.16.10.1	172.16.10.254	ICMP	98	Echo (ping) request id=0x1d54, seq=3/768, ttl=64 (re
13	12.900839220	172.16.10.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d54, seq=3/768, ttl=64 (re
14	13.924716514	172.16.10.1	172.16.10.254	ICMP	98	Echo (ping) request id=0x1d54, seq=4/1024, ttl=64 (r
15	13.924842856	172.16.10.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d54, seq=4/1024, ttl=64 (r
22	20.574764479	172.16.10.1	172.16.11.253	ICMP	98	Echo (ping) request id=0x1d5b, seq=1/256, ttl=64 (re
23	20.574916942	172.16.11.253	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d5b, seq=1/256, ttl=64 (re
24	21.604728909	172.16.10.1	172.16.11.253	ICMP	98	Echo (ping) request id=0x1d5b, seq=2/512, ttl=64 (re
25	21.604857836	172.16.11.253	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d5b, seq=2/512, ttl=64 (re
27	22.628723955	172.16.10.1	172.16.11.253	ICMP	98	Echo (ping) request id=0x1d5b, seq=3/768, ttl=64 (re
28	22.628851694	172.16.11.253	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d5b, seq=3/768, ttl=64 (re
29	23.652713972	172.16.10.1	172.16.11.253	ICMP	98	Echo (ping) request id=0x1d5b, seq=4/1024, ttl=64 (r
30	23.652838917	172.16.11.253	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d5b, seq=4/1024, ttl=64 (r
35	31.182494493	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) request id=0x1d62, seq=1/256, ttl=64 (re
36	31.182797254	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d62, seq=1/256, ttl=63 (re
38	32.196711861	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) request id=0x1d62, seq=2/512, ttl=64 (re
39	32.196947295	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d62, seq=2/512, ttl=63 (re
40	33.220714589	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) request id=0x1d62, seq=3/768, ttl=64 (re
41	33.220953865	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d62, seq=3/768, ttl=63 (re
43	34.244712359	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) request id=0x1d62, seq=4/1024, ttl=64 (r
44	34.244985507	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d62, seq=4/1024, ttl=63 (r
51	42.910699658	172.16.10.1	172.16.11.254	ICMP	98	Echo (ping) request id=0x1d69, seq=1/256, ttl=64 (re
52	42.911021415	172.16.11.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d69, seq=1/256, ttl=63 (re
53	43.940714513	172.16.10.1	172.16.11.254	ICMP	98	Echo (ping) request id=0x1d69, seq=2/512, ttl=64 (re
54	43.940968874	172.16.11.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d69, seq=2/512, ttl=63 (re
56	44.964717032	172.16.10.1	172.16.11.254	ICMP	98	Echo (ping) request id=0x1d69, seq=3/768, ttl=64 (re
57	44.964975025	172.16.11.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d69, seq=3/768, ttl=63 (re
58	45.988720668	172.16.10.1	172.16.11.254	ICMP	98	Echo (ping) request id=0x1d69, seq=4/1024, ttl=64 (r
59	45.989002686	172.16.11.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d69, seq=4/1024, ttl=63 (r
61	47.012728285	172.16.10.1	172.16.11.254	ICMP	98	Echo (ping) request id=0x1d69, seq=5/1280, ttl=64 (r
62	47.012989072	172.16.11.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d69, seq=5/1280, ttl=63 (r
74	58.798569289	172.16.10.1	172.16.2.19	ICMP	98	Echo (ping) request id=0x1d76, seq=1/256, ttl=64 (re
75	58.798867371	172.16.2.19	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d76, seq=1/256, ttl=63 (re
76	59.812716817	172.16.10.1	172.16.2.19	ICMP	98	Echo (ping) request id=0x1d76, seq=2/512, ttl=64 (re
77	59.812991013	172.16.2.19	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d76, seq=2/512, ttl=63 (re
79	60.836714167	172.16.10.1	172.16.2.19	ICMP	98	Echo (ping) request id=0x1d76, seq=3/768, ttl=64 (re
80	60.837003309	172.16.2.19	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d76, seq=3/768, ttl=63 (re
81	61.860714940	172.16.10.1	172.16.2.19	ICMP	98	Echo (ping) request id=0x1d76, seq=4/1024, ttl=64 (r
82	61.860980825	172.16.2.19	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1d76, seq=4/1024, ttl=63 (r
84	62.884712989	172.16.10.1	172.16.2.19	ICMP	98	Echo (ping) request id=0x1d76, seq=5/1280, ttl=64 (r

Figure 14: Tux3 ping to the other network interfaces.

So far, the connection from `tux2` to the eth0 interface of `tux3` is carried out through the route implemented in the first tux and described in experiment 3. To verify this implementation, that same route was removed from `tux2`, and the **traceroute** command was executed (tux2: `traceroute 172.16.20.1`), where it is verified that, as there was no route defined up to Bridge10, the router with the IP 172.16.11.254, defined as the default gateway of `tux2` was responsible for redirecting the ICMP packets to the destination as we can see in Figure 15.

No.	Time	Source	Destination	Protocol	Length	Info
4	2.498456674	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) reply id=0x76c8, seq=1/256, ttl=63
5	3.510982151	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) request id=0x76c8, seq=2/512, ttl=64
6	3.511137285	172.16.11.254	172.16.11.1	ICMP	126	Redirect (Redirect for host)
7	3.511326576	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) reply id=0x76c8, seq=2/512, ttl=63
9	4.534975355	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) request id=0x76c8, seq=3/768, ttl=64
10	4.535122667	172.16.11.254	172.16.11.1	ICMP	126	Redirect (Redirect for host)
11	4.535335567	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) reply id=0x76c8, seq=3/768, ttl=63
12	5.558977137	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) request id=0x76c8, seq=4/1024, ttl=64
13	5.559138209	172.16.11.254	172.16.11.1	ICMP	126	Redirect (Redirect for host)
14	5.559325893	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) reply id=0x76c8, seq=4/1024, ttl=63
16	6.582977021	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) request id=0x76c8, seq=5/1280, ttl=64
17	6.583135298	172.16.11.254	172.16.11.1	ICMP	126	Redirect (Redirect for host)
18	6.583319281	172.16.10.1	172.16.11.1	ICMP	98	Echo (ping) reply id=0x76c8, seq=5/1280, ttl=63
19	7.606966762	172.16.11.1	172.16.10.1	ICMP	98	Echo (ping) request id=0x76c8, seq=6/1536, ttl=64

Figure 15: Logs after removing route in tux2 when pinging from tux3 with no acceptance of ICMP redirects.

Finally, we tried to ping the laboratory router (IP `172.16.1.254`) from `tux3`. This succeeded since **NAT** is active by default. **NAT** (Network Address Translation) replaces local IP addresses in packets with a public IP address in order to establish a connection outside the network. Therefore, the router that implements NAT becomes responsible for forwarding all packets to the correct address, inside or

outside the local network.

By using NAT, the information will make it back to the respective tux using the RC public address, and not the machine's private one. So, the laboratory router ping in `tux3` is done with success only because NAT allows devices connected to the local networks (`172.16.10.0/24` or `172.16.11.0/24`), to communicate with the external one (`172.16.1.19`) as shown in Figure 16.

No.	Time	Source	Destination	Protocol	Length	Info
2	1.310729580	172.16.10.1	172.16.2.254	ICMP	98	Echo (ping) request id=0x1fad, seq=1/256, ttl=64
3	1.311245845	172.16.2.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1fad, seq=1/256, ttl=62
5	2.325117150	172.16.10.1	172.16.2.254	ICMP	98	Echo (ping) request id=0x1fad, seq=2/512, ttl=64
6	2.325543530	172.16.2.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1fad, seq=2/512, ttl=62
7	3.349113802	172.16.10.1	172.16.2.254	ICMP	98	Echo (ping) request id=0x1fad, seq=3/768, ttl=64
8	3.349584461	172.16.2.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1fad, seq=3/768, ttl=62
10	4.373122258	172.16.10.1	172.16.2.254	ICMP	98	Echo (ping) request id=0x1fad, seq=4/1024, ttl=64
11	4.373583139	172.16.2.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1fad, seq=4/1024, ttl=62
12	5.397123799	172.16.10.1	172.16.2.254	ICMP	98	Echo (ping) request id=0x1fad, seq=5/1280, ttl=64
13	5.397564635	172.16.2.254	172.16.10.1	ICMP	98	Echo (ping) reply id=0x1fad, seq=5/1280, ttl=62

Figure 16: Logs captured in tux3 when pinging the lab router.

When we use the command `/ip firewall nat disable 0` to disable the NAT and then went back to ping the lab router in `tux3`, we verified that the request was not successful, confirming the dependence of the NAT configuration on the accomplishment of this task.

We should **enable NAT** again, since it will be needed for the next experiments. To enable it, we just have to type the following commands in the RC through GtKterm console:

- `iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE`
- `iptables -A FORWARD -i eth1 -m state --state NEW,INVALID -j DROP`
- `iptables -L`
- `iptables -t nat -L`

3.5 Exp. 5 - DNS

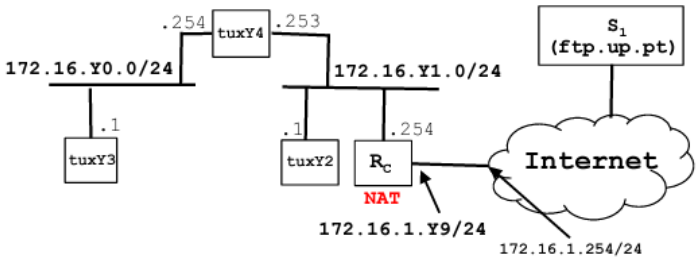


Figure 17: Experience 5 - Architecture.

The goal of this experiment is to work with DNS, understand what it is, and how it works. By configuring DNS we could observe packet exchange and better understand its behavior.

3.5.1 Steps

Before starting, make sure the setup from experience 4 is complete. Then, to configure DNS in all of the machines, we can use the following:

```
echo $'search netlab.fe.up.pt\nnameserver 172.16.1.1' > /etc/resolv.conf
```

To check if the configuration was done correctly, try to ping a server with its hostname, e.g. `ping google.com`. To complete the experience, capture the ping and save the logs.

3.5.2 Analysis

DNS was created to allow for a more user-friendly way of accessing servers. There are a lot of IP addresses and it's hard to remember them. Thus, DNS creates a mapping between hostnames and IP addresses. DNS works by querying DNS servers that hold this mapping. This way, to configure DNS we just need to set the DNS server we want to use on the machine. This can be done by adding a line to the `etc/resolv.conf` file containing the name of the server to be used and its corresponding IP address. For this experiment, we used netlab's DNS server (`172.16.2.1`).

We can now analyze the DNS packets generated after pinging `google.com` (Figure 18). Firstly, a couple of requests are made to the server. The first request is an 'A' record, that queries the IPv4 address for a specific hostname. The second one is the 'AAAA' record that requests the IPv6 address of the hostname. Both of these requests receive responses. For `google.com` the IPv4 address is `142.250.200.78` and the IPv6 address is `2a00:1450:4003:80d::200e`.

Lastly, after the first responses are obtained, a PTR record request is sent. This is the opposite of an 'A' record as it provides the domain name linked to an IP address instead of the IP address for a domain. The purpose of the PTR record is to verify that the sender matches the IP address it claims to be using.

4	5.186238114	172.16.10.1	172.16.2.1	DNS	70	Standard query 0x34f3 A google.com
5	5.186249149	172.16.10.1	172.16.2.1	DNS	70	Standard query 0x48fd AAAA google.com
6	5.186972143	172.16.2.1	172.16.10.1	DNS	86	Standard query response 0x34f3 A google.com A 142.250.200.78
7	5.186985273	172.16.2.1	172.16.10.1	DNS	98	Standard query response 0x48fd AAAA google.com AAAA 2a00:1450:4003:80d::200e
8	5.187300955	172.16.10.1	142.250.200.78	ICMP	98	Echo (ping) request id=0x20ff, seq=1/256, ttl=64 (reply in 9)
9	5.204977236	142.250.200.78	172.16.10.1	ICMP	98	Echo (ping) reply id=0x20ff, seq=1/256, ttl=112 (request in 8)
10	5.205120340	172.16.10.1	172.16.2.1	DNS	87	Standard query 0xb741 PTR 78.200.250.142.in-addr.arpa
11	5.205790745	172.16.2.1	172.16.10.1	DNS	126	Standard query response 0xb741 PTR 78.200.250.142.in-addr.arpa PTR mad07s24-in-f14.1e100.net

Figure 18: DNS packets captured on tux3.

3.6 Exp. 6 - TCP connections

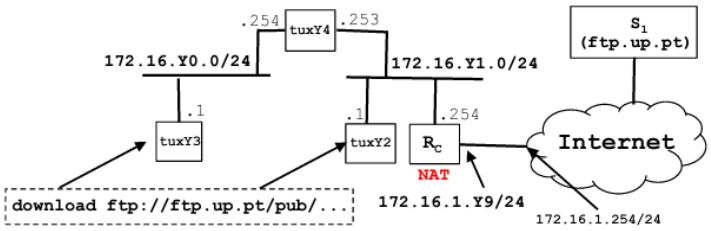


Figure 19: Experience 6 - Architecture.

After all the machines are configured according to Figure 19's schema, we can now use our download application to work with FTP.

3.6.1 Steps

This experience assumes all the steps from experiences 1 to 5 have been executed. To begin, use the application to retrieve a file and capture the logs:

```
download ftp://netlab1.fe.up.pt/pipe.txt
```

Moreover, perform a download on `tux3` followed by another in `tux2` in order to measure how the connection evolves.

```
download ftp://rcom:rcom@netlab1.fe.up.pt/files/crab.mp4
```

3.6.2 Analysis

The FTP application opens two TCP connections. The first is used as control. This is where we can send control information (e.g. login command). The second is opened to join the data connection after changing to passive mode.

A TCP connection is done in three phases. In the first phase, a connection is established. Then, the data transfer occurs in the second phase. The connection ends in phase 3.

Automatic Repeat Request (ARQ) is a mechanism used by the TCP protocol to handle errors. When a sender sends a packet, it waits for the receiver to reply with an acknowledgment. If this does not occur, the sender will re-send the packet (after a timeout) and wait for the acknowledgment. It does not send a different packet until it's assured the last one was received.

TCP most relevant fields include:

- Port numbers for sender and receiver.
- Sequence number that identifies the application data and is incremented while transmitting.
- Acknowledgment number to keep track of the number of bytes received.
- Window size that specifies the number of bytes that can be transmitted at a time.
- Checksums to verify the data integrity.

If we take a look at the logs on figure 20, it's noticeable that they describe the steps taken by the download application.

- First, there's a query to the DNS server (light blue). This is the very first step of the download.
- After the IP is retrieved, the application creates a connection to the server on port 21 (logs 4 and 5).

- What follows (in light purple) is the communication to the server done by the application (See section 2.1).
- Furthermore, we can observe another connection being open (log 22) after the server enters passive mode.
- Finally, both connections are shut down and the download is complete.

2	1.223060552	172.16.10.1	172.16.2.1	DNS	76	Standard query 0x279a A netlab1.fe.up.pt
3	1.223864143	172.16.2.1	172.16.10.1	DNS	92	Standard query response 0x279a A netlab1.fe.up.pt A 192.168.109.136
4	1.223946136	172.16.10.1	192.168.109.136	TCP	74	44360 → 21 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2976689571 TSecr=0
5	1.224771727	192.168.109.136	172.16.10.1	TCP	74	21 → 44360 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=241052495
6	1.224791981	172.16.10.1	192.168.109.136	TCP	66	44360 → 21 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=2976689571 TSecr=2410524917
7	1.226902997	192.168.109.136	172.16.10.1	FTP	100	Response: 220 Welcome to netlab-FTP server
8	1.226912286	172.16.10.1	192.168.109.136	TCP	66	44360 → 21 [ACK] Seq=1 Ack=35 Win=64256 Len=0 TSval=2976689574 TSecr=2410524920
9	1.226951117	172.16.10.1	192.168.109.136	FTP	75	Request: user rcom
10	1.227376659	192.168.109.136	172.16.10.1	TCP	66	21 → 44360 [ACK] Seq=35 Ack=10 Win=65280 Len=0 TSval=2410524920 TSecr=2976689574
11	1.227384830	172.16.10.1	192.168.109.136	FTP	67	Request:
12	1.228016682	192.168.109.136	172.16.10.1	TCP	66	21 → 44360 [ACK] Seq=35 Ack=11 Win=65280 Len=0 TSval=2410524921 TSecr=2976689574
13	1.228107126	192.168.109.136	172.16.10.1	FTP	100	Response: 331 Please specify the password.
14	1.228125215	172.16.10.1	192.168.109.136	FTP	75	Request: pass rcom
15	1.228841435	192.168.109.136	172.16.10.1	TCP	66	21 → 44360 [ACK] Seq=69 Ack=20 Win=65280 Len=0 TSval=2410524922 TSecr=2976689575
16	1.228847861	172.16.10.1	192.168.109.136	FTP	67	Request:
17	1.229415389	192.168.109.136	172.16.10.1	TCP	66	21 → 44360 [ACK] Seq=69 Ack=21 Win=65280 Len=0 TSval=2410524922 TSecr=2976689575
18	1.245406126	192.168.109.136	172.16.10.1	FTP	89	Response: 230 Login successful.
19	1.245469682	172.16.10.1	192.168.109.136	FTP	71	Request: pasv
20	1.245948233	192.168.109.136	172.16.10.1	TCP	66	21 → 44360 [ACK] Seq=92 Ack=26 Win=65280 Len=0 TSval=2410524939 TSecr=2976689592
21	1.246121019	192.168.109.136	172.16.10.1	FTP	120	Response: 227 Entering Passive Mode (192,168,109,136,187,190).
22	1.246168581	172.16.10.1	192.168.109.136	TCP	74	57994 → 48062 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=2976689593 TSecr
23	1.246814332	192.168.109.136	172.16.10.1	TCP	74	48062 → 57994 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=241052
24	1.246829906	172.16.10.1	192.168.109.136	TCP	66	57994 → 48062 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=2976689593 TSecr=2410524940
25	1.246850090	172.16.10.1	192.168.109.136	FTP	71	Request: retr
26	1.247425371	192.168.109.136	172.16.10.1	TCP	66	21 → 44360 [ACK] Seq=146 Ack=31 Win=65280 Len=0 TSval=2410524940 TSecr=2976689593
27	1.247433333	172.16.10.1	192.168.109.136	FTP	75	Request: pipe.txt
28	1.248000931	192.168.109.136	172.16.10.1	TCP	66	21 → 44360 [ACK] Seq=146 Ack=40 Win=65280 Len=0 TSval=2410524941 TSecr=2976689594
29	1.248158143	192.168.109.136	172.16.10.1	FTP	134	Response: 150 Opening BINARY mode data connection for pipe.txt (1863 bytes).
30	1.248486257	192.168.109.136	172.16.10.1	FTP-DA…	1929	FTP Data: 1863 bytes (PASV) (retr)
31	1.248496244	172.16.10.1	192.168.109.136	TCP	66	57994 → 48062 [ACK] Seq=1 Ack=1864 Win=63488 Len=0 TSval=2976689595 TSecr=2410524941
32	1.248499596	192.168.109.136	172.16.10.1	TCP	66	48062 → 57994 [FIN, ACK] Seq=1864 Ack=1 Win=65280 Len=0 TSval=2410524941 TSecr=297668
33	1.248559310	172.16.10.1	192.168.109.136	TCP	66	44360 → 21 [RST, ACK] Seq=40 Ack=214 Win=64256 Len=0 TSval=2976689595 TSecr=241052494
34	1.248573488	172.16.10.1	192.168.109.136	TCP	66	57994 → 48062 [FIN, ACK] Seq=1 Ack=1865 Win=64128 Len=0 TSval=2976689595 TSecr=241052
35	1.249097855	192.168.109.136	172.16.10.1	TCP	66	48062 → 57994 [ACK] Seq=1865 Ack=2 Win=65280 Len=0 TSval=2410524942 TSecr=2976689595

Figure 20: Logs captured while downloading a file.

In the TCP protcol, the receiver is the one that defines the window size. However, it also allows the network to redefine this value if it becomes too congested. This mechanism is called **TCP congestion control**. In short, the receiver will increase window size with every acknowledgment, until it reaches the maximum value that avoids congestion. The connection is then monitored for packet losses and the window size is changed again if necessary. This behavior was confirmed in our experiment, as described on Figure 21.

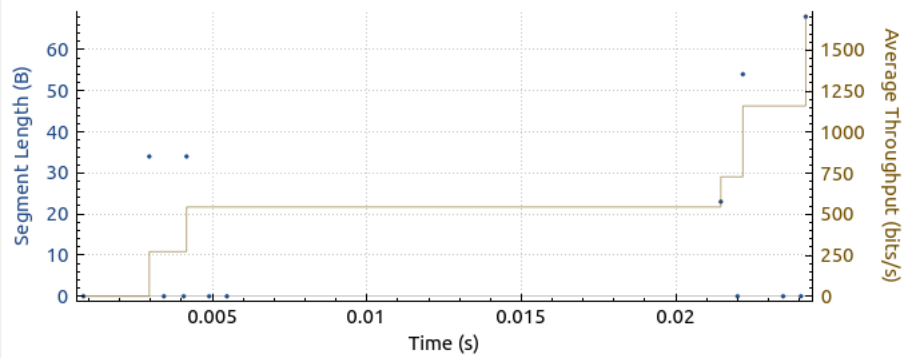


Figure 21: Throughput of the TCP connection.

TCP has a fairness rule in place that avoids uneven distribution of the network's bandwidth between connections. When the connection in `tux3` first starts, it can use all the bandwidth available, as there are no other connections. However, if we start a new transfer on `tux2`, we observe the first machine's throughput dropping to about half. This happens because of **TCP fairness**.