FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Configuration and Study of a Computer Network and Development of a File Transfer Protocol Client

Alexandre Ferreira
Paulo Saavedra
Class nº7

Licenciatura em Engenharia Informática e Computação

November 9, 2025

Configuration and Study of a Computer Network and Development of a File Transfer Protocol Client

Alexandre Ferreira

Paulo Saavedra
Class nº7

Licenciatura em Engenharia Informática e Computação

November 9, 2025

Resumo

Este relatório descreve o estudo, configuração e análise de uma rede de computadores, bem como o desenvolvimento de um cliente **FTP!** (**FTP!**). A configuração da rede foi realizada através de scripts automatizados, abrangendo a atribuição de endereços **IP!** (**IP!**), configuração de routing, **DNS!** (**DNS!**) e ligação de múltiplos dispositivos (máquinas, switch e router). O cliente **FTP!**, desenvolvido em C, implementa autenticação de utilizador, modo passivo, transferências binárias e download de ficheiros, seguindo as normas RFC959 e RFC1738. O projeto demonstra competências práticas tanto na configuração de redes como na implementação de protocolos de aplicação.

Abstract

This report describes the study, configuration, and analysis of a computer network, as well as the development of an **FTP!** (**FTP!**) client. The network setup was automated through scripts, covering **IP!** (**IP!**) address assignment, routing configuration, **DNS!** (**DNS!**), and the interconnection of multiple devices (machines, switch, and router). The **FTP!** client, developed in C, implements user authentication, passive mode, binary transfers, and file downloads, following RFC959 and RFC1738 standards. The project demonstrates practical skills in both network configuration and application-layer protocol implementation.

Contents

List of Figures

List of Tables

List of Acronyms

Chapter 1

Introduction

Briefly introduce the project objectives: network configuration and FTP client development.

Chapter 2

Part 1 – Download Application

2.1 Architecture of the Download Application

Describe the FTP client architecture, main modules, and protocol features.

2.2 Successful Download Report

Describe a successful file download, including a Wireshark screenshot of FTP packets

Chapter 3

Part 2 – Network Configuration and Analysis

3.1 Experiment 1 - Configure an IP! (IP!) Network

3.1.1 Network Architecture

For this experiment, we connected Tux103 and Tux104 to the same switch, creating a single broadcast domain. The **IP!** addresses were configured as specified in the project description, assigning `172.16.100.1` to Tux103 and `172.16.100.254` to Tux104, both within the `172.16.100.0/24` subnet.

3.1.2 IP! Configuration and Connectivity

To configure the network, we accessed the terminals of both hosts and used the `ifconfig` command to assign the **IP!** addresses and netmasks manually. We verified the configuration by inspecting the routing table with `route −n`, confirming a direct connection to the local subnet. To test connectivity and observe the address resolution process, we cleared the ARP cache using `arp −d` and immediately executed a `ping` command while capturing traffic with Wireshark, as shown in Figure **??**.

3.1.3 ARP Packets and Addressing

We observed that ARP (Address Resolution Protocol) packets are generated when a host needs to map a known Layer 3 (**IP!**) address to an unknown Layer 2 (MAC) address. Regarding the addresses, the **ARP Request** uses the Broadcast MAC address (`FF:FF:FF:FF:FF:FF`) as the destination because the sender does not yet know the target's hardware address. However, the destination **IP!** is specific to the target. The **ARP Reply** is sent via Unicast, using the specific MAC and **IP!** addresses of the requester and the target, as the identity of the host is now established.

3.1.4 Ping Command and ICMP Generation

The `ping` command generates **ICMP** (Internet Control Message Protocol) packets. Specifically, it sends "Echo Request" messages and awaits "Echo Reply" messages to test connectivity. Unlike ARP, the MAC and **IP!** addresses observed in these packets are always **Unicast**. This is because the ARP resolution has already successfully occurred, allowing the sender to encapsulate the **IP!** packet in an Ethernet frame addressed to the specific destination MAC.

3.1.5 Frame Identification and Length

To determine the type of receiving Ethernet frame, we analyzed the **EtherType** field in the header: a value of `0x0806` indicates an ARP frame, while `0x0800` indicates an **IP!** frame. To identify ICMP packets, we further inspected the **IP!** header, where the Protocol field value is `1`. The length of the receiving frame was determined by analyzing the return value of the socket read function (in the application context) or by inspecting the "Total Length" field within the **IP!** header during packet capture.

3.1.6 The Loopback Interface

We also identified the loopback interface (`lo`, **IP!** `127.0.0.1`). This is a virtual network interface that allows the system to communicate with itself. It is crucial for testing internal network stack functionality and for inter-process communication without sending data through the physical network hardware.

3.1.7 Analysis

In this experiment, we successfully configured a simple **IP!**v4 network and verified connectivity. The results demonstrated the dependency of Layer 3 communication on Layer 2 address resolution. We concluded that before any ICMP (Ping) traffic can be exchanged, the system must broadcast ARP Requests to resolve the destination MAC address. Once the ARP table is populated, subsequent communication uses efficient Unicast frames.

3.2 Experiment 2 - Implement two bridges in a switch

3.2.1 Network Architecture

In this experiment, we logically partitioned the physical switch into two distinct Layer 2 segments. We created two separate bridges: `bridge100`, connecting Tux103 and Tux104 (Subnet `172.16.100.0/24`), and `bridge101`, connecting Tux102 (Subnet `172.16.101.0/24`). This configuration effectively isolates the traffic between the two groups of hosts, despite them being connected to the same physical device.

3.2.2 Bridge Configuration

To configure the bridges on the MikroTik switch, we accessed the device via the serial console and executed the following steps:

1. We created the logical bridge interfaces using the command `/interface bridge add name=bridge100` (and similarly for bridge101).
2. We assigned the physical ports to these bridges. For example, the ports connected to Tux103 and Tux104 were removed from the default bridge and added to `bridge100` using `/interface bridge port add`.

We verified the configuration using `/interface bridge port print`, ensuring that each physical port was associated with the correct logical bridge.

3.2.3 Connectivity Verification

After configuring the bridges, we verified the connectivity and isolation using ICMP (ping) commands from Tux103.

- **Intra-Bridge Communication:** When pinging Tux104 (`172.16.100.254`), which resides on the same bridge (`bridge100`), the communication was successful with valid Echo Replies, as shown in Figure **??**.
- **Inter-Bridge Isolation:** When pinging Tux102 (`172.16.101.1`), which resides on the separate `bridge101`, the communication failed with "Destination Unreachable" messages (Figure **??**). This confirms that traffic cannot cross from one bridge to another without a Layer 3 routing device.

3.2.4 Broadcast Domains Analysis

A key objective of this experiment was to identify the number of broadcast domains and verify their isolation.

- **Number of Domains:** By creating two separate bridges (`bridge100` and `bridge101`), we created two distinct broadcast domains. A bridge forwards broadcast frames to all ports within itself, but never to ports assigned to a different bridge.
- **Verification via Logs:** We concluded this by analyzing the traffic logs during the Broadcast Ping test (`ping -b`). When Tux103 sent a broadcast packet to `172.16.100.255`:
  - **Tux104** (on the same bridge) received the broadcast request and immediately responded with a unicast Echo Reply (Figure **??**). This confirms they share the same broadcast domain
  - **Tux102** (on `bridge101`) did **not** capture any packet. This silence proves that the broadcast traffic was contained within `bridge100` and did not cross over to `bridge101`.

- We also performed a broadcast ping from Tux102 (`172.16.101.255`).
  - As expected, **no Echo Replies were received**, which confirms that Tux102 is isolated in its own broadcast domain (`bridge101`) and cannot trigger responses from hosts in `bridge100` (Tux103/104).

3.2.5 Analysis

In this experiment, we successfully segmented the network at Layer 2. The results demonstrated that while Tux103 and Tux104 could communicate directly (as they share the same bridge and **IP!** subnet), connectivity to Tux102 was impossible. This isolation occurs because the switch acts as two separate logical devices. Since there is no Layer 3 device (Router) configured to forward packets between the `172.16.100.0` and `172.16.101.0` networks, traffic cannot pass between the bridges. This experiment reinforces the concept that bridges define the boundaries of broadcast domains.

3.3 Experiment 3 - Configure a Router in Linux

3.3.1 Network Architecture

In this experiment, we connected two distinct subnets using a Linux host acting as a router.

- **Subnet 1 (172.16.100.0/24):** Contains **Tux103** (172.16.100.1) and the first interface of the router **Tux104** (172.16.100.254).
- **Subnet 2 (172.16.101.0/24):** Contains **Tux102** (172.16.101.1) and the second interface of the router **Tux104** (172.16.101.253).

Tux104 was configured to forward traffic between these two networks, effectively functioning as a Layer 3 Router.

3.3.2 Router Configuration

To transform the standard Linux host (Tux104) into a router, we executed the following configuration steps:

1. **IP! Forwarding:** We enabled packet forwarding in the kernel using the command `sysctl net.`**`IP!`**`v4.`**`IP!`**`_forward=1`. This allows the operating system to accept packets destined for other networks and forward them through the appropriate interface.
2. **ICMP Broadcasts:** We used `sysctl net.`**`IP!`**`v4.icmp_echo_ignore_broadcasts=0` to disable the ignore broadcast setting to allow the router to respond to broadcast pings for diagnostic purposes.

3.3.3 Routing Table Analysis

We analyzed the routing tables on all three hosts to ensure connectivity.

- **Routes in End Hosts (Tux103/Tux102):** We added static routes to reach the remote subnet. For example, on Tux103, we added a route to `172.16.101.0/24` using Tux104 (`172.16.100.254`) as the **Gateway**. The entry contains the **Destination Network**, the **Gateway IP!**, the **Genmask**, and the **Interface**. The flag **UG** indicates the route is Up and uses a Gateway.
- **Routes in Router (Tux104):** The router automatically has direct routes (Flag **U**) to both `172.16.100.0/24` and `172.16.101.0/24` because it has physical interfaces connected to them.
- **Forwarding Table Entry:** A forwarding table entry tells the system: "To reach network X, send the packet to Next-Hop Y via Interface Z".

3.3.4 Connectivity Verification

From Tux103, we performed a sequence of pings to verify connectivity to all relevant interfaces in the network topology:

- **Local Gateway Interface:** Ping to `172.16.100.254` (Tux104-e1). Successful (Figure **??**).
- **Remote Gateway Interface:** Ping to `172.16.101.253` (Tux104-e2). Successful (Figure **??**).
- **Remote Host:** Ping to `172.16.101.1` (Tux102). Successful with `ttl=63` in the reply, indicating the packet traversed one router hop (Figure **??**).

3.3.5 ARP Analysis

During the connectivity test (Ping from Tux103 to Tux102), we observed specific ARP behaviors that differ from the single-subnet experiment:

- **Tux103 ARP Request:** Tux103 needs to send the packet to the final destination (Tux102), but it knows Tux102 is on a remote network. Therefore, Tux103 broadcasts an ARP Request asking for the **MAC address of the Gateway (Tux104)**, not the destination host.
- **Router ARP Request:** Once Tux104 receives the packet and decides to forward it to the 172.16.101.0 network, it broadcasts a new ARP Request on that subnet asking for the **MAC address of Tux102**.

3.3.6 ICMP and Addressing Analysis

We captured the traffic on both interfaces of the router (Tux104) simultaneously to analyze the packet headers during transit.

- **Capture on Interface E1 (Subnet 100):** As shown in Figure **??**, the router receives the ICMP Echo Request from Tux103. The Destination **IP!** is `172.16.101.1` (Remote), but the frame is addressed to the router's MAC address.
- **Capture on Interface E2 (Subnet 101):** As shown in Figure **??**, the router forwards the same **IP!** packet out of interface E2.
- **Key Observation:**
  - **End-to-End IP!:** The Source **IP!** (`172.16.100.1`) and Destination **IP!** (`172.16.101.1`) remain identical in both captures.
  - **Hop-by-Hop MAC:** The Layer 2 addresses change. In E1, the destination MAC is the Router's E1 interface. In E2, the source MAC becomes the Router's E2 interface, and the destination MAC becomes Tux102.

3.3.7 Analysis

In this experiment, we successfully implemented **IP!** routing on a Linux machine. The results demonstrated that for communication between different subnets, end hosts must be configured with a Gateway. The analysis of the captured traffic highlighted the distinction between Layer 2 (MAC), which handles local delivery hop-by-hop, and Layer 3 (**IP!**), which handles end-to-end addressing. We also confirmed that enabling `IP!_forward` is mandatory for a Linux host to act as a router.

3.4 Experiment 4 - Configure a Commercial Router and Implement NAT

3.4.1 Configuring a Static Route in a Commercial Router

To configure a static route in the commercial router, we accessed its console and entered the necessary commands. We added a new static route with the destination network, subnet mask, and gateway as per the project requirements. After saving the configuration, we verified the route was correctly added by checking the routing table in the router's interface.

3.4.2 ICMP Redirection

To test **ICMP!** (**ICMP!**) redirection, we initiated ping requests from TUX2 to TUX3. Initially, the packets were routed through TUX4 as it was the shortest path to subnet 172.16.Y0.0/24. After that, as requested in the project description, we disabled redirection acceptance on TUX2, and changed the routes to force the packets to go through the commercial router. This way, we observed that TUX2 continued to send packets through TUX4, ignoring the **ICMP!** redirect messages from the commercial router. With **ICMP!** redirection disabled, after the first **ICMP!** redirect, TUX2 switched to its original routing path, demonstrating the effect of disabling **ICMP!** redirect acceptance on the routing behavior of the host.

3.4.3 Configuring Network Address Translation

As **NAT!** (**NAT!**) was already enabled by default on the commercial router, to understand its functionality, we disabled it through the router's console and observed its effects on the network communication. We then re-enabled **NAT!** to restore normal operation.

3.4.4 Network Address Translation

To understand **NAT!** functionality, we performed ping tests from TUX3 to the **FTP!** (**FTP!**) server before and after disabling **NAT!** on the commercial router. With **NAT!** enabled, TUX3 was able to successfully ping the server, as the router correctly translated the private **IP!** address of TUX3 to a public **IP!** address for communication. With **NAT!** disabled, the pings failed, indicating that the server could not reach TUX3's private **IP!** address directly. This demonstrated the importance of **NAT!** in allowing devices within a private network to communicate with external networks.

3.5 Experiment 5 - Domain Name System (DNS)

3.5.1 Configuring a DNS Service

To configure the DNS service on TUX2, TUX3, and TUX4, we modified the `/etc/resolv.conf` file on each machine to include the **IP!** address of the **FTP!** server `services.netlab.fe.up.pt`. This allowed the machines to resolve the domain name to its corresponding **IP!** address when attempting to connect to the **FTP!** server.

3.5.2 DNS Packets

DNS! (**DNS!**) packets were captured using Wireshark while performing a domain name resolution for `google.com` from TUX3. The captured packets showed the standard **DNS!** query and response process, including the query sent by TUX3 to the DNS service and the corresponding response containing the resolved **IP!** address of the server.

3.6 Experiment 6 - TCP Connections

3.6.1 TCP Connections in FTP

In order to successfully download a file using the **FTP!** client, two **TCP!** (**TCP!**) connections are established between the client and the server. The first connection is the control connection, which is used for sending commands and receiving responses. This connection is established on port 21 of the server. The second connection is the data connection, which is used for transferring files. Depending on whether active or passive mode is used, this connection can be initiated by either the client or the server on a dynamically assigned port. Every **TCP!** connection involves a three-way handshake process to establish the connection, followed by data transfer, and finally a four-way handshake to terminate the connection.

3.6.2 Automatic Repeat reQuest in TCP

ARQ! (**ARQ!**) is a fundamental mechanism used in **TCP!** to ensure reliable data transmission. It works by requiring the receiver to send acknowledgments (ACKs) back to the sender for the data packets received. If the sender does not receive an acknowledgment within a specified timeout period, it assumes that the packet was lost or corrupted and retransmits the packet. This process continues until the sender receives an acknowledgment for all sent packets, ensuring that all data is correctly received by the receiver. On duplicate ACKs, **TCP!** can also perform fast retransmissions to improve efficiency.

The following table summarizes the key **TCP!** header fields involved in the **ARQ!** mechanism and their roles:

Field	**Role in ARQ**
Sequence Number	Identifies the **first byte** in this segment. Used for ordering and retransmission.
Acknowledgment Number	Indicates **next byte expected** by the receiver. Confirms receipt of previous bytes.
Flags	Especially **ACK**, **SYN**, **FIN**. ACK is critical for ARQ.
Window Size	Flow control; tells sender how many bytes can be sent before receiving an ACK.
Checksum	Detects corrupted segments; segments failing the checksum are discarded and retransmitted.

Table 3.1: TCP Header Fields and Their Role in ARQ

3.6.3 TCP Congestion Control

TCP! employs several congestion control mechanisms to manage network congestion and ensure efficient data transmission. The primary algorithms used in **TCP!** congestion control include Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery.

The following table highlights the main **TCP!** header fields relevant to congestion control and their roles:

To measure the effects of the **TCP!** congestion control mechanism, we initiated a file download using our **FTP!** client while capturing the **TCP!** packets with Wireshark. By analyzing the captured packets, we observed the changes in the throughput during the concurrent download process, as can be seen in Figure **??**.

Field	**Role**
Sequence Number	Tracks the byte position for retransmission and ordering.
Acknowledgment Number	Confirms receipt of bytes; triggers congestion window (cwnd) increase.
Window Size (rwnd)	Receiver’s advertised window (flow control).
Flags (ACK)	ACK signals successful receipt.
Optional TCP Options	e.g., SACK (Selective ACK) can improve fast retransmit.

Table 3.2: TCP Header Fields and Their Role in Congestion Control

Chapter 4

Conclusions

Summarize findings, challenges, and skills acquired.

Chapter 5

References

Chapter 6

Annexes

6.1 Download Application Code

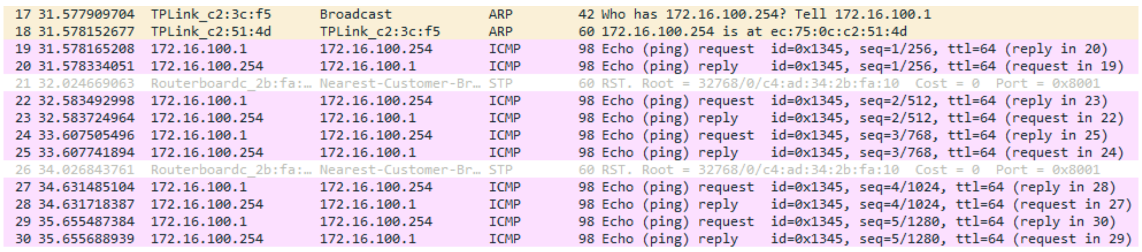
Include the source code

6.2 Configuration Commands

List scripts and manual commands used.

6.3 Captured Logs

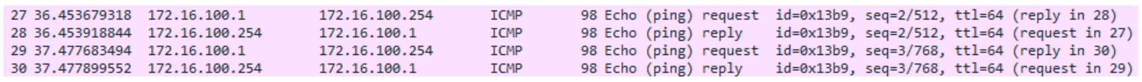
6.3.1 Experiment 1 - Configure an IP Network



```
17 31.577909704  TPLink_c2:3c:f5      Broadcast            ARP        42 Who has 172.16.100.254? Tell 172.16.100.1
18 31.578152677  TPLink_c2:51:4d      TPLink_c2:3c:f5      ARP        60 172.16.100.254 is at ec:75:0c:c2:51:4d
19 31.578165208  172.16.100.1         172.16.100.254       ICMP       98 Echo (ping) request  id=0x1345, seq=1/256, ttl=64 (reply in 20)
20 31.578334051  172.16.100.254       172.16.100.1         ICMP       98 Echo (ping) reply    id=0x1345, seq=1/256, ttl=64 (request in 19)
21 32.024669063  Routerboardc_2b:fa:… Nearest-Customer-Br… STP        60 RST. Root = 32768/0/c4:ad:34:2b:fa:10  Cost = 0  Port = 0x8001
22 32.583492998  172.16.100.1         172.16.100.254       ICMP       98 Echo (ping) request  id=0x1345, seq=2/512, ttl=64 (reply in 23)
23 32.583724964  172.16.100.254       172.16.100.1         ICMP       98 Echo (ping) reply    id=0x1345, seq=2/512, ttl=64 (request in 22)
24 33.607505496  172.16.100.1         172.16.100.254       ICMP       98 Echo (ping) request  id=0x1345, seq=3/768, ttl=64 (reply in 25)
25 33.607741894  172.16.100.254       172.16.100.1         ICMP       98 Echo (ping) reply    id=0x1345, seq=3/768, ttl=64 (request in 24)
26 34.026843761  Routerboardc_2b:fa:… Nearest-Customer-Br… STP        60 RST. Root = 32768/0/c4:ad:34:2b:fa:10  Cost = 0  Port = 0x8001
27 34.631485104  172.16.100.1         172.16.100.254       ICMP       98 Echo (ping) request  id=0x1345, seq=4/1024, ttl=64 (reply in 28)
28 34.631718387  172.16.100.254       172.16.100.1         ICMP       98 Echo (ping) reply    id=0x1345, seq=4/1024, ttl=64 (request in 27)
29 35.655487384  172.16.100.1         172.16.100.254       ICMP       98 Echo (ping) request  id=0x1345, seq=5/1280, ttl=64 (reply in 30)
30 35.655688939  172.16.100.254       172.16.100.1         ICMP       98 Echo (ping) reply    id=0x1345, seq=5/1280, ttl=64 (request in 29)
```

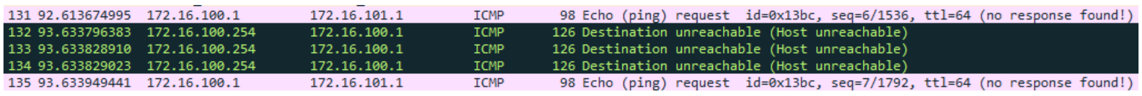
Figure 6.1: TUX103 pinging TUX104

6.3.2 Experiment 2 - Implement two bridges in a switch



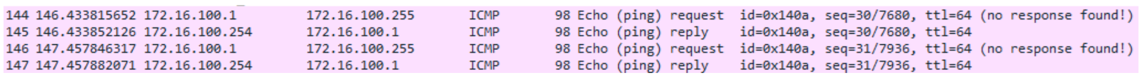
```
27 36.453679318  172.16.100.1         172.16.100.254       ICMP       98 Echo (ping) request  id=0x13b9, seq=2/512, ttl=64 (reply in 28)
28 36.453918844  172.16.100.254       172.16.100.1         ICMP       98 Echo (ping) reply    id=0x13b9, seq=2/512, ttl=64 (request in 27)
29 37.477683494  172.16.100.1         172.16.100.254       ICMP       98 Echo (ping) request  id=0x13b9, seq=3/768, ttl=64 (reply in 30)
30 37.477899552  172.16.100.254       172.16.100.1         ICMP       98 Echo (ping) reply    id=0x13b9, seq=3/768, ttl=64 (request in 29)
```

Figure 6.2: TUX103 pinging TUX104 in the same bridge



```
131 92.613674995  172.16.100.1      172.16.101.1      ICMP     98 Echo (ping) request  id=0x13bc, seq=6/1536, ttl=64 (no response found!)
132 93.633796383  172.16.100.254    172.16.100.1      ICMP    126 Destination unreachable (Host unreachable)
133 93.633828910  172.16.100.254    172.16.100.1      ICMP    126 Destination unreachable (Host unreachable)
134 93.633829023  172.16.100.254    172.16.100.1      ICMP    126 Destination unreachable (Host unreachable)
135 93.633949441  172.16.100.1      172.16.101.1      ICMP     98 Echo (ping) request  id=0x13bc, seq=7/1792, ttl=64 (no response found!)
```

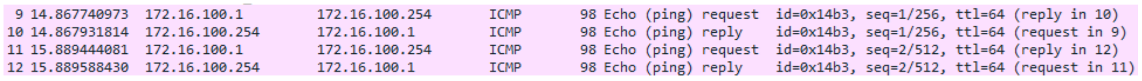
Figure 6.3: TUX103 pinging TUX102 in a different bridge



```
144 146.433815652 172.16.100.1      172.16.100.255    ICMP     98 Echo (ping) request  id=0x140a, seq=30/7680, ttl=64 (no response found!)
145 146.433852126 172.16.100.254    172.16.100.1      ICMP     98 Echo (ping) reply    id=0x140a, seq=30/7680, ttl=64
146 147.457846317 172.16.100.1      172.16.100.255    ICMP     98 Echo (ping) request  id=0x140a, seq=31/7936, ttl=64 (no response found!)
147 147.457882071 172.16.100.254    172.16.100.1      ICMP     98 Echo (ping) reply    id=0x140a, seq=31/7936, ttl=64
```

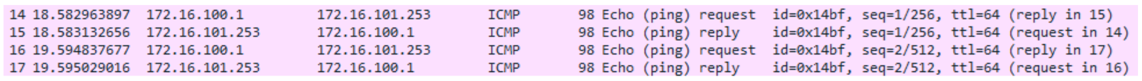
Figure 6.4: TUX104 receiving broadcast ping from TUX103

6.3.3 Experiment 3 - Configure a Router in Linux



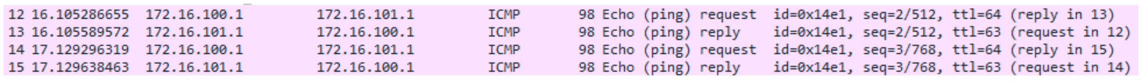
```
 9 14.867740973  172.16.100.1      172.16.100.254     ICMP      98 Echo (ping) request  id=0x14b3, seq=1/256, ttl=64 (reply in 10)
10 14.867931814  172.16.100.254    172.16.100.1       ICMP      98 Echo (ping) reply    id=0x14b3, seq=1/256, ttl=64 (request in 9)
11 15.889444081  172.16.100.1      172.16.100.254     ICMP      98 Echo (ping) request  id=0x14b3, seq=2/512, ttl=64 (reply in 12)
12 15.889588430  172.16.100.254    172.16.100.1       ICMP      98 Echo (ping) reply    id=0x14b3, seq=2/512, ttl=64 (request in 11)
```

Figure 6.5: TUX103 pinging TUX104's ether1 interface



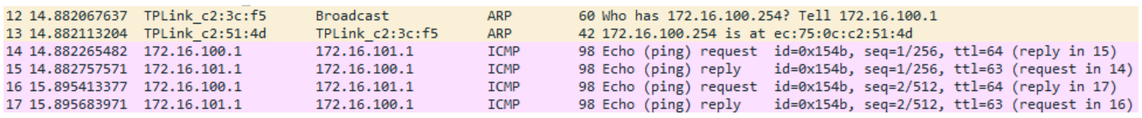
```
14 18.582963897  172.16.100.1      172.16.101.253     ICMP      98 Echo (ping) request  id=0x14bf, seq=1/256, ttl=64 (reply in 15)
15 18.583132656  172.16.101.253    172.16.100.1       ICMP      98 Echo (ping) reply    id=0x14bf, seq=1/256, ttl=64 (request in 14)
16 19.594837677  172.16.100.1      172.16.101.253     ICMP      98 Echo (ping) request  id=0x14bf, seq=2/512, ttl=64 (reply in 17)
17 19.595029016  172.16.101.253    172.16.100.1       ICMP      98 Echo (ping) reply    id=0x14bf, seq=2/512, ttl=64 (request in 16)
```

Figure 6.6: TUX103 pinging TUX104's ether2 interface



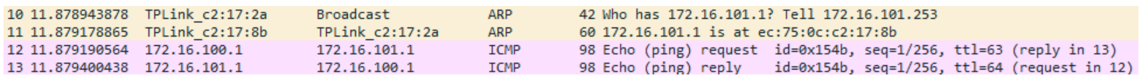
```
12 16.105286655  172.16.100.1      172.16.101.1       ICMP      98 Echo (ping) request  id=0x14e1, seq=2/512, ttl=64 (reply in 13)
13 16.105589572  172.16.101.1      172.16.100.1       ICMP      98 Echo (ping) reply    id=0x14e1, seq=2/512, ttl=63 (request in 12)
14 17.129296319  172.16.100.1      172.16.101.1       ICMP      98 Echo (ping) request  id=0x14e1, seq=3/768, ttl=64 (reply in 15)
15 17.129638463  172.16.101.1      172.16.100.1       ICMP      98 Echo (ping) reply    id=0x14e1, seq=3/768, ttl=63 (request in 14)
```

Figure 6.7: TUX103 pinging TUX102



```
12 14.882067637  TPLink_c2:3c:f5   Broadcast          ARP       60 Who has 172.16.100.254? Tell 172.16.100.1
13 14.882113204  TPLink_c2:51:4d   TPLink_c2:3c:f5    ARP       42 172.16.100.254 is at ec:75:0c:c2:51:4d
14 14.882265482  172.16.100.1      172.16.101.1       ICMP      98 Echo (ping) request  id=0x154b, seq=1/256, ttl=64 (reply in 15)
15 14.882757571  172.16.101.1      172.16.100.1       ICMP      98 Echo (ping) reply    id=0x154b, seq=1/256, ttl=63 (request in 14)
16 15.895413377  172.16.100.1      172.16.101.1       ICMP      98 Echo (ping) request  id=0x154b, seq=2/512, ttl=64 (reply in 17)
17 15.895683971  172.16.101.1      172.16.100.1       ICMP      98 Echo (ping) reply    id=0x154b, seq=2/512, ttl=63 (request in 16)
```

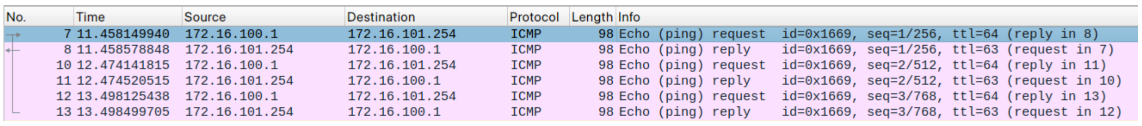
Figure 6.8: TUX104 router capturing ICMP on ether1 interface



```
10 11.878943878  TPLink_c2:17:2a   Broadcast          ARP       42 Who has 172.16.101.1? Tell 172.16.101.253
11 11.879178865  TPLink_c2:17:8b   TPLink_c2:17:2a    ARP       60 172.16.101.1 is at ec:75:0c:c2:17:8b
12 11.879190564  172.16.100.1      172.16.101.1       ICMP      98 Echo (ping) request  id=0x154b, seq=1/256, ttl=63 (reply in 13)
13 11.879400438  172.16.101.1      172.16.100.1       ICMP      98 Echo (ping) reply    id=0x154b, seq=1/256, ttl=64 (request in 12)
```

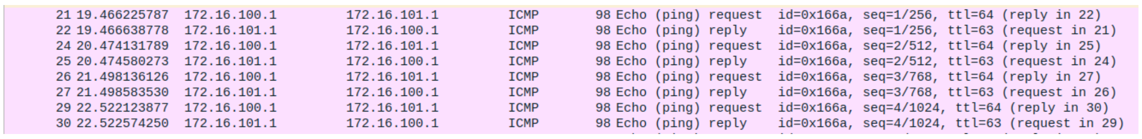
Figure 6.9: TUX104 router capturing ICMP on ether2 interface

6.3.4 Experiment 4 - Configure a Commercial Router and Implement NAT



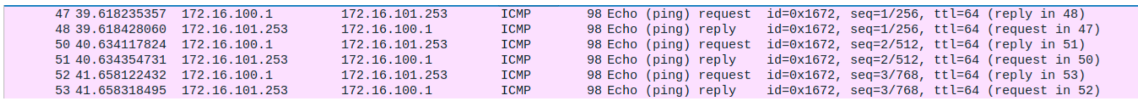
```
No.     Time          Source            Destination       Protocol Length Info
      7 11.458149940  172.16.100.1      172.16.101.254    ICMP        98 Echo (ping) request  id=0x1669, seq=1/256, ttl=64 (reply in 8)
      8 11.458578848  172.16.101.254    172.16.100.1      ICMP        98 Echo (ping) reply    id=0x1669, seq=1/256, ttl=63 (request in 7)
     10 12.474141815  172.16.100.1      172.16.101.254    ICMP        98 Echo (ping) request  id=0x1669, seq=2/512, ttl=64 (reply in 11)
     11 12.474520515  172.16.101.254    172.16.100.1      ICMP        98 Echo (ping) reply    id=0x1669, seq=2/512, ttl=63 (request in 10)
     12 13.498125438  172.16.100.1      172.16.101.254    ICMP        98 Echo (ping) request  id=0x1669, seq=3/768, ttl=64 (reply in 13)
     13 13.498499705  172.16.101.254    172.16.100.1      ICMP        98 Echo (ping) reply    id=0x1669, seq=3/768, ttl=63 (request in 12)
```

Figure 6.10: TUX3 pinging RC's ether2 interface



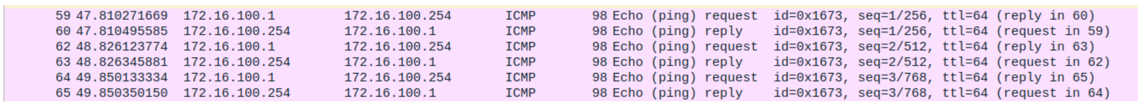
```
   21 19.466225787  172.16.100.1          172.16.101.1          ICMP       98 Echo (ping) request  id=0x166a, seq=1/256, ttl=64 (reply in 22)
   22 19.466638778  172.16.101.1          172.16.100.1          ICMP       98 Echo (ping) reply    id=0x166a, seq=1/256, ttl=63 (request in 21)
   24 20.474131789  172.16.100.1          172.16.101.1          ICMP       98 Echo (ping) request  id=0x166a, seq=2/512, ttl=64 (reply in 25)
   25 20.474580273  172.16.101.1          172.16.100.1          ICMP       98 Echo (ping) reply    id=0x166a, seq=2/512, ttl=63 (request in 24)
   26 21.498136126  172.16.100.1          172.16.101.1          ICMP       98 Echo (ping) request  id=0x166a, seq=3/768, ttl=64 (reply in 27)
   27 21.498583530  172.16.101.1          172.16.100.1          ICMP       98 Echo (ping) reply    id=0x166a, seq=3/768, ttl=63 (request in 26)
   29 22.522123877  172.16.100.1          172.16.101.1          ICMP       98 Echo (ping) request  id=0x166a, seq=4/1024, ttl=64 (reply in 30)
   30 22.522574250  172.16.101.1          172.16.100.1          ICMP       98 Echo (ping) reply    id=0x166a, seq=4/1024, ttl=63 (request in 29)
```

Figure 6.11: TUX3 pinging TUX2



```
   47 39.618235357  172.16.100.1          172.16.101.253        ICMP       98 Echo (ping) request  id=0x1672, seq=1/256, ttl=64 (reply in 48)
   48 39.618428060  172.16.101.253        172.16.100.1          ICMP       98 Echo (ping) reply    id=0x1672, seq=1/256, ttl=64 (request in 47)
   50 40.634117824  172.16.100.1          172.16.101.253        ICMP       98 Echo (ping) request  id=0x1672, seq=2/512, ttl=64 (reply in 51)
   51 40.634354731  172.16.101.253        172.16.100.1          ICMP       98 Echo (ping) reply    id=0x1672, seq=2/512, ttl=64 (request in 50)
   52 41.658122432  172.16.100.1          172.16.101.253        ICMP       98 Echo (ping) request  id=0x1672, seq=3/768, ttl=64 (reply in 53)
   53 41.658318495  172.16.101.253        172.16.100.1          ICMP       98 Echo (ping) reply    id=0x1672, seq=3/768, ttl=64 (request in 52)
```

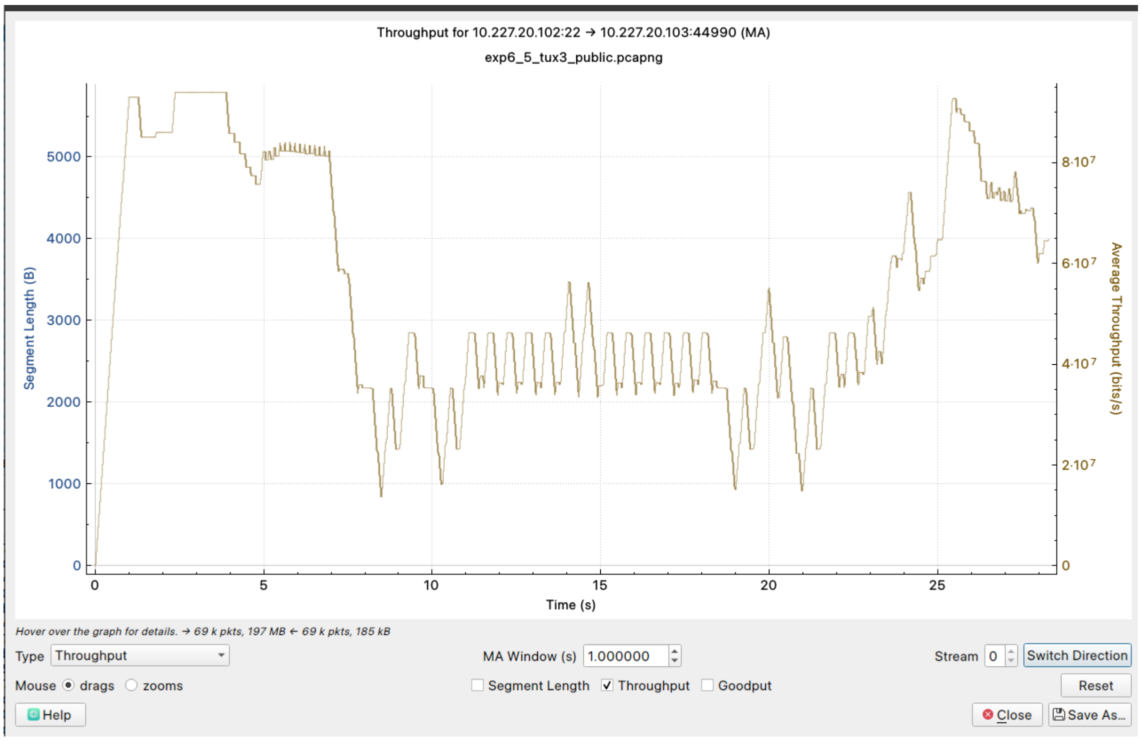
Figure 6.12: TUX3 pinging TUX4's ether2 interface



```
   59 47.810271669  172.16.100.1          172.16.100.254        ICMP       98 Echo (ping) request  id=0x1673, seq=1/256, ttl=64 (reply in 60)
   60 47.810495585  172.16.100.254        172.16.100.1          ICMP       98 Echo (ping) reply    id=0x1673, seq=1/256, ttl=64 (request in 59)
   62 48.826123774  172.16.100.1          172.16.100.254        ICMP       98 Echo (ping) request  id=0x1673, seq=2/512, ttl=64 (reply in 63)
   63 48.826345881  172.16.100.254        172.16.100.1          ICMP       98 Echo (ping) reply    id=0x1673, seq=2/512, ttl=64 (request in 62)
   64 49.850133334  172.16.100.1          172.16.100.254        ICMP       98 Echo (ping) request  id=0x1673, seq=3/768, ttl=64 (reply in 65)
   65 49.850350150  172.16.100.254        172.16.100.1          ICMP       98 Echo (ping) reply    id=0x1673, seq=3/768, ttl=64 (request in 64)
```

Figure 6.13: TUX3 pinging TUX4's ether1 interface

6.3.5 Experiment 6 - TCP Connections



Throughput for 10.227.20.102:22 → 10.227.20.103:44990 (MA)

exp6_5_tux3_public.pcapng

Hover over the graph for details. → 69 k pkts, 197 MB ← 69 k pkts, 185 kB

Type: Throughput

MA Window (s): 1.000000

Stream: 0

Mouse: ● drags ○ zooms

☐ Segment Length ✓ Throughput ☐ Goodput

Figure 6.14: Changes in Throughput during concurrent FTP download