RCOM

LEIC - 2024/2025

2nd Laboratory Work

Configuration and Study of a Network

Authors:

Bruno Ferreira up202207863@fe.up.pt

Pedro Marinho up202206854@fe.up.pt

Summary

This project involves developing a file download application that utilizes the FTP protocol and TCP/IP for file transfer. It also requires setting up and configuring a computer network to test the application in a real-world environment. The primary goal of the project is to understand how the FTP protocol and TCP/IP work together to enable file transfers over a network. Additionally, it aims to teach how to set up, configure, and troubleshoot a network to support such operations. Throughout the project, we focus on learning how to configure and set up the network to support the application. By the end, we gain practical experience in both developing the application and setting up the network.

Introduction

This project has two main objectives: developing a download application and setting up a network. The download application was created using the FTP (*File Transfer Protocol*) with TCP (*Transmission Control Protocol*) connections through *sockets*. The network setup in the laboratory aims to run the download application using two bridges connected within a *switch*.

The report is divided into three main sections: Part 1 - Development of a download application, Part 2 - Configuration and study of a computer network, and Conclusions. In Part 1, we will discuss the architecture and results of the download application. In Part 2, we will provide a detailed analysis of the network configuration experiments. Finally, we will conclude the report by reflecting on the objectives we achieved.

Part 1 - Development of a download application

Application Architecture

The application's architecture follows a client-server model, implementing the FTP protocol as defined in RFC959 to facilitate the download of a single file from a specified server, supporting URLs with the syntax outlined in RFC1738.

The program's implementation consists of a primary source file, `download.c`, which contains the main logic and entry point, and a header file, `download.h`, used for defining macros and data structures.

1. *download.c*
   Usage and Responsibilities:
   - Verifies and analyses the given arguments in the command line. Awaits for an URL formatted according to the RFC1738.
   - Configures the FTP connection details using URL with the info username, password, host, ip, filename and url path.
   - Orchestrates the program's instructions in the *main* function, calling the secondary functions to establish the connection, authenticate, guarantee binary format, go into passive mode, transfer the file and close the connection.

2. *download.h*
   Data Structures and function implementations
   - Data struct URL: Stores the information parsed at the beginning of the FTP, preserving the ip, username, password, host, url path and filename.
   - Data struct response: Stores the message and the code for each response given by the server.
   - Includes functions such as `create_socket()`, for establishing the connection, `get_ip()` for decoding the DNS ip, `parse_URL()` to extract the url information, `close_socket()` for closing the connection, `receiveResponse()` to read the servers response, utilizing `readUntilNewLine()` and `readCode()` as helper functions, `showResponse()` to display in the terminal the servers response, `reset_response()` to clean the current responses element, `writeMessage()` to send the command to the server, `calculate_new_port()` to get the new ip and port give by the server, `read_file()` and `getFileSize()` to download the file.

The *Download* program follows a simple and well structured flux, ensuring the system runs smoothly without any issues. The program runs as described below:

1. **Initialization**:
   Parse the given data through the command line, storing the login credentials, the ip, host, filename and the url path.
2. **Establish the FTP connection**:
   The program establishes the connection using the specified port and ip. Once established the connection to the FTP server, the program executes the authentication of the user, sending its credentials.
3. **File transfer**:
   Before the program solicits the server to pass to passive mode, it tells the server to switch to binary mode, getting in the process the file size. After that, having requested the server to be in passive mode, the program receives information from the server to create a new socket for data transfer. Having that socket created, this new connection is utilized to download the requested file, with the specified url.
4. **Termination**:
   After the file transfer conclusion, both created connections are terminated.

Download Report

To run the following program, compile it using *make* or the command *gcc download.c -o download* to generate the binary program and run it by typing **./download <url>** with an URL with this format: ftp://[<user>:<password>@]<host>/<url-path>.
We have tested many inputs with anonymous and non anonymous mode, as well as incorrect urls and what not, to ensure the robustness of our program. Dare say the program has shown no signs of inefficiency.
In the following image we show a successful download via our program.

```
pedro@PedriniPC:/mnt/c/Users/pmigu/Faculdade/3Ano/1semestre/RCOM/tmp/FEUP-RCOM/proj-2$ ./download ftp://ftp.up.pt/pub/gnu/emacs/elisp-manual-21-2.8.tar.gz
User: anonymous
Password: password
Host: ftp.up.pt
URL Path: pub/gnu/emacs/elisp-manual-21-2.8.tar.gz
Filename: elisp-manual-21-2.8.tar.gz
Host name  : mirrors.up.pt
IP Address : 193.137.29.15
Creating socket
Socket created
220-Welcome to the University of Porto's mirror archive (mirrors.up.pt)
220----------------------------------------------------------------
220-
220-All connections and transfers are logged. The max number of connections is 200.
220-
220-For more information please visit our website: http://mirrors.up.pt/
220-Questions and comments can be sent to mirrors@uporto.pt
220-
220-
220
------------------------------------------------------------
Connected to server

user anonymous
331 Please specify the password.

pass password
230 Login successful.

type I
200 Switching to Binary mode.

pasv
227 Entering Passive Mode (193,137,29,15,202,4).

retr pub/gnu/emacs/elisp-manual-21-2.8.tar.gz
150 Opening BINARY mode data connection for pub/gnu/emacs/elisp-manual-21-2.8.tar.gz (2455995 bytes).
Filename: elisp-manual-21-2.8.tar.gz
Receiving file...
[================================================] 100.00%
File received and written to elisp-manual-21-2.8.tar.gz
226 Transfer complete.
Transfer completed successfully

quit
221 Goodbye.
```

Part 2 - Configuration and study of a computer network

Exp 1 - Configure an IP Network

The goal of this experiment is to configure the IP addresses of two computers so that they can communicate with each other. We used the ifconfig command to configure the IP addresses of both machines. After configuring the eth0 interfaces and adding the necessary routes, we used the ping command to verify the connectivity between them.

-> What are the ARP packets and what are they used for?

ARP (Address Resolution Protocol) packets are used to map a known IP address to a MAC address in a local network. When a device knows the IP address of another device but needs to find its MAC address to send a packet, it sends an ARP request. The device with the matching IP address responds with an ARP reply, providing its MAC address.

-> What are the MAC and IP addresses of ARP packets and why?

In an ARP request packet, the source IP address is the IP address of the requesting device, and the destination IP address is the target IP address that the sender wants to resolve. The source MAC address is the MAC address of the requesting device, and the destination MAC address is initially set to "broadcast" (FF:FF:FF:FF:FF:FF), since the sender does not yet know the MAC address of the target. In an ARP reply, the source and destination roles are reversed: the source MAC address is that of the target device, and the destination MAC address is the original sender.

-> What packets does the ping command generate?

The ping command generates ICMP (Internet Control Message Protocol) Echo Request packets to check the availability of a device on the network. If the target device is reachable, it responds with an ICMP Echo Reply packet.

-> What are the MAC and IP addresses of the ping packets?

For a ping request, the source IP address is the IP address of the device sending the ping, and the destination IP address is the IP address of the device being pinged. The source MAC address is the MAC address of the sender's network interface, and the destination MAC address is either the MAC address of the target device (if it is known) or the MAC address of the router (if the target is not directly reachable).

-> How to determine if a receiving Ethernet frame is ARP, IP, ICMP?

To determine the type of a receiving Ethernet frame, you need to check the Ethernet frame's EtherType field:

- If the EtherType is 0x0806, the frame is an ARP packet.
- If the EtherType is 0x0800, the frame is an IP packet.
- If the frame is an IP packet and the protocol field is 0x01, it is an ICMP packet (ICMP is a protocol within IP).

-> How to determine the length of a receiving frame?

To determine the length of a receiving Ethernet frame, check the total frame size. Ethernet frames range from **64 bytes** (minimum) to **1518 bytes** (or **1522 bytes** with VLAN tagging). The **EtherType** field (Ethernet II) indicates the protocol type, while the **Length/Type** field (IEEE 802.3) shows the payload length.

-> What is the loopback interface and why is it important?

The loopback interface is a virtual network interface (often referred to as lo) used by the computer to communicate with itself. The IP address of the loopback interface is typically 127.0.0.1 (or ::1 for IPv6). The loopback interface is important for testing network-related software and services locally without requiring an actual network connection. It allows the system to verify that the network stack is functioning properly.

Exp 2 - Implement two bridges in a switch

The goal of this experiment is to create two virtual LANs (VLANs), bridge110 and bridge111, and assign tux3 and tux4 to the first VLAN (bridge110) and tux2 to the second VLAN (bridge111). We used the ifconfig command to configure the interfaces and the /interface bridge command to first remove the existing ports where the TUX devices were connected. Then, we created two bridges and added the respective ports to each bridge.

-> How to configure bridge110?

To configure a bridge, it was necessary to enter the switch's configuration console and execute the command /interface bridge add name=bridge110. Then, the ports connected to tux3 and tux4 are removed from the default bridge (/interface bridge port remove [find interface=ether1], where 1 is the port identifier) and added to the respective bridges (/interface bridge port add bridge=bridge110 interface=ether1). This process is repeated for bridge111, resulting in two separate subnets.

-> How many broadcast domains are there? How can you conclude it from the logs?

In this experiment, we ran the ping command from tux3 to tux2, and it failed, as expected, since there is no route between the VLANs. Next, we performed a broadcast ping from tux3 using the command ping -b 172.16.110.255, where a response from tux4 was anticipated, as both are in the same subnet.

After this, the same process was repeated from tux2. In this case, no response was received because no device, other than tux2 itself, is configured on bridge111. Thus, it can be concluded that there are two separate broadcast domains with addresses 172.16.110.255 and 172.16.111.255.

Exp 3 - Configure a Router in Linux

The goal of this experiment is to configure tux4 as a router between the two subnets created in the previous experiment, enabling communication between tux3 and tux2. To achieve this, tux4 is connected to Ethernet interface 1, and this interface is added to the subnet where tux2 is located. In this experiment we used the ifconfig command to configure the eth2 in the tux4 and then we connected the tux4 to the bridge 111 using the /interface bridge command. After that we activated the IP forwarding and deactivated the ICMP echo-ignore-broadcast. Finally we use the route add -net command to create the routes in order for the tux2 and tux3 to be connected.

-> What routes are there in the tuxes? What are their meaning?

- **tuxY3**:
  - 172.16.110.0/24 (directly connected to eth1 of tuxY3, in the subnet of bridge110).
  - 172.16.111.0/24 (via 172.16.110.254 as the gateway, which is tuxY4's eth1 IP address).
- **tuxY2**:
  - 172.16.111.0/24 (directly connected to eth1 of tuxY2, in the subnet of bridge111).
  - 172.16.110.0/24 (via 172.16.111.253 as the gateway, which is tuxY4's eth2 IP address).
- **tuxY4** (the router):
  - 172.16.110.0/24 (directly connected to eth1).
  - 172.16.111.0/24 (directly connected to eth2).

The **routes** represent the way each machine knows how to reach another network or device. For instance, tuxY3 knows that to reach devices in 172.16.111.0/24, it must send packets to tuxY4's 172.16.110.254 IP address, and tuxY2 knows that to reach devices in 172.16.110.0/24, it must send packets to tuxY4's 172.16.111.253 IP address.

-> What information does an entry of the forwarding table contain?

An entry in the **forwarding table** (also known as the routing table) typically contains:

- **Destination Network**: The target IP address or subnet to which packets are directed (e.g., 172.16.110.0/24).
- **Gateway**: The IP address of the next-hop router that should handle packets destined for that network (e.g., 172.16.110.254 for tuxY3 or 172.16.111.253 for tuxY2).
- **Interface**: The network interface through which the packets should be sent to reach the destination (e.g., eth1 or eth2 for tuxY4).
- **Metric**: The cost of reaching the destination (optional in simple routing configurations).

This table helps the machine know where to forward packets based on their destination IP.

-> What ARP messages, and associated MAC addresses, are observed and why?

ARP messages are crucial for resolving IP addresses to MAC addresses, especially when there is communication between different subnets.

- **ARP requests** will be sent when tuxY3 or tuxY2 need to resolve the MAC address of the router (tuxY4) in their subnet:
    - From tuxY3, an ARP request will be sent for 172.16.110.254 (tuxY4's IP on bridge110).
    - From tuxY2, an ARP request will be sent for 172.16.111.253 (tuxY4's IP on bridge111).
- **ARP replies** will be sent by tuxY4 in response, providing its MAC addresses for the respective interfaces:
    - For 172.16.110.254, tuxY4 will reply with the MAC address of eth1.
    - For 172.16.111.253, tuxY4 will reply with the MAC address of eth2.

These ARP messages are observed to ensure that the machines know how to forward packets to the correct network interface.

-> What ICMP packets are observed and why?

During this experiment, **ICMP Echo Request and Echo Reply packets** are observed. These packets are part of the **ping** tests to verify connectivity between tuxY3, tuxY2, and tuxY4:

- **ICMP Echo Request** packets are sent when tuxY3 pings tuxY2 or tuxY4.
- **ICMP Echo Reply** packets are sent back from the destination machine (tuxY2 or tuxY4) if they are reachable.

These ICMP packets are essential for checking if the network is properly configured and if the router (tuxY4) is correctly forwarding packets between the two subnets.

-> What are the IP and MAC addresses associated to ICMP packets and why?

- **ICMP Echo Request**:
    - **Source IP address**: The IP address of the sender (e.g., 172.16.110.1 for tuxY3).
    - **Destination IP address**: The IP address of the target (e.g., 172.16.111.253 for tuxY2, or 172.16.110.254 for tuxY4).
    - **Source MAC address**: The MAC address of the sender's network interface (e.g., tuxY3's eth1 MAC address).
    - **Destination MAC address**: The MAC address of the destination device. If the packet is being routed, this will be the MAC address of the router's relevant interface (e.g., tuxY4's eth1 or eth2).
- **ICMP Echo Reply**:
    - **Source IP address**: The IP address of the machine replying (e.g., 172.16.111.253 for tuxY2, or 172.16.110.254 for tuxY4).
    - **Destination IP address**: The IP address of the machine that sent the original Echo Request.
    - **Source MAC address**: The MAC address of the device sending the reply (e.g., tuxY2's or tuxY4's eth1 or eth2 MAC address).
    - **Destination MAC address**: The MAC address of the machine that initiated the Echo Request (e.g., tuxY3's eth1 MAC address).

Exp 4 - Configure a Commercial Router and Implement NAT

The goal of this experiment is to configure a commercial router and implement Network Address Translation (NAT). This configuration allows multiple devices within a private network to share a single public IP address when accessing external networks, such as the internet. By setting up NAT, the router handles the translation of private IP addresses to a public one and vice versa, ensuring smooth communication between internal devices and external networks. In this experiment, after connecting the router to the switch we used the /interface bridge command to add the router to the bridge111. Then the /ip command was used to configure the router IP address and the necessary routes to all tux devices, so that they are able to communicate with the router.

-> How to configure a static route in a commercial router?

To configure a static route on a commercial router, we connect the S0 serial cable from the Tux4 to the MTik input of the router, allowing us to configure it via GTKTerm. In GTKTerm, we add the route using the command: /ip route add dst-address=172.16.110.0/24 gateway=172.16.111.253.

-> What are the paths followed by the packets, with and without ICMP redirect enabled, in the experiments carried out and why?

- **Without ICMP Redirect Enabled**: When ICMP redirects are disabled, packets are routed according to the static routing table. In this case, packets from tuxY2 going to tuxY3 would go through tuxY4, as specified in the routing table. The path for ICMP packets from tuxY2 to tuxY3 would first hit tuxY4 (the default router for tuxY2), which will forward the packets to tuxY3 based on its own routing table.
- **With ICMP Redirect Enabled**: ICMP redirects inform the sender that there is a better route available to reach the destination directly. When ICMP redirects are enabled, tuxY2 may receive a redirect message from tuxY4 or RC, instructing it to send packets directly to tuxY3, instead of going through tuxY4. This reduces unnecessary hops and optimizes routing for subsequent packets.

-> How to configure NAT in a commercial router?

To configure NAT (Network Address Translation) on a commercial router, follow these general steps:

- **Access the Router Configuration Interface**: Login via the CLI or web interface.
- **Identify the Interfaces**: Determine which interface is connected to the internal network (private IP range) and which one connects to the external network (public IP range).
- **Enable NAT**:
    - For source NAT (SNAT), configure the router to translate the internal (private) IP addresses to the router's public IP address when accessing external networks.
    - For destination NAT (DNAT), configure the router to translate external IP addresses back to internal private IP addresses for incoming connections.

This would enable NAT for outgoing traffic from the internal network to the external network, we use the : /ip firewall nat enable command.

-> What does NAT do?

NAT (Network Address Translation) allows multiple devices on a private network to share a single public IP address for internet access. It modifies IP addresses in packet

headers, translating private IPs to the router's public IP when leaving the network and vice versa when returning. NAT also enhances security by hiding internal IP addresses from external networks, preventing direct access to internal devices.

-> What happens when tuxY3 pings the FTP server with the NAT disabled? Why?

When NAT is disabled, tuxY3 will attempt to access the FTP server using its private IP address. Since the NAT is no longer translating the private IP address into the router's public IP address, the packets destined for the FTP server might not be routed correctly, especially if the FTP server is outside the private network.

Without NAT, tuxY3's private IP address may not be recognized or routable by the external network, and thus tuxY3 will not be able to communicate with the FTP server. The communication would fail, as the network's routing tables would not know how to handle the private IP address outside the local network.

Disabling NAT essentially breaks the ability of devices in the private network to access external resources because there is no translation between private and public IP addresses.

Exp 5 - DNS

The goal of this experiment is to configure the DNS (Domain Name System) on tux3, tux4, and tux2. A DNS server is used to translate hostnames into their corresponding IP addresses. In this case, the server services.netlab.fe.up.pt contains a database with public IP addresses and their associated hostnames. This configuration enables the systems to resolve domain names to IP addresses, allowing devices to communicate using human-readable names instead of numeric IP addresses.

-> How to configure the DNS service in a host?

To configure the DNS service, it is necessary to edit the resolv.conf file located at /etc/resolv.conf on the tux host. This file needs to be edited to include the following information: echo "nameserver 10.227.20.3" > /etc/resolv.conf. After this configuration, it will be possible to access the internet from the tux machines.

-> What packets are exchanged by DNS and what information is transported?

DNS exchanges two types of packets: the **DNS query packet** and the **DNS response packet**. The **DNS query packet** is sent from the client to the DNS server and contains the hostname (e.g., "www.google.com") that the client wants to resolve into an IP address. The **DNS response packet** is sent from the server to the client and contains the resolved IP address for the queried hostname. This exchange allows the client to convert a human-readable hostname into a machine-readable IP address.

Exp 6 - TCP connections

Once the network was fully configured, the download application developed in Part 1 was used as a case study. The test was successfully conducted by transferring a file from an FTP server, and the application performed as expected.

-> How many TCP connections are opened by your FTP application?

The FTP application typically opens **two TCP connections**:

- **Control connection** (usually on port 21) for sending commands and receiving responses.

- **Data connection** (on a dynamically assigned port) for transferring the actual file data.

-> In what connection is transported the FTP control information?

The **FTP control information** is transported over the **control connection** (typically on TCP port 21). This connection is used to send FTP commands such as USER, PASS, TYPE, PASV, RETR (retrieve) and QUIT as well as to send responses from the server.

-> What are the phases of a TCP connection?

1. **Connection Establishment** (Three-way handshake):
    - The client sends a **SYN** packet to initiate the connection.
    - The server responds with a **SYN-ACK** packet.
    - The client sends a **final ACK** to confirm the connection.
2. **Data Transfer**:
    - After the connection is established, data is transferred between the client and server using the reliable TCP stream.
3. **Connection Termination** (Four-way handshake):
    - The client or server sends a **FIN** packet to terminate the connection.
    - The other party responds with an **ACK** to acknowledge receipt of the FIN.
    - The closing party then sends a **FIN** to fully close the connection.
    - The final **ACK** packet is exchanged to confirm termination.

-> How does the ARQ TCP mechanism work? What are the relevant TCP fields? What relevant information can be observed in the logs?

The **ARQ (Automatic Repeat reQuest) mechanism** in TCP ensures reliable data transmission by using **acknowledgments** and **retransmissions**.

- **Relevant TCP fields** include:
    - **Sequence Number**: Identifies the position of the first byte of data in the segment.
    - **Acknowledgment Number**: Indicates the next byte expected from the other side.
    - **Flags**: Especially **ACK**, indicating acknowledgment.
    - **Window Size**: The amount of data the receiver is willing to accept.

In the logs, you can observe retransmitted packets, acknowledgment packets, and the flow of data. If there is a loss, the ARQ mechanism will trigger retransmission.

-> How does the TCP congestion control mechanism work? What are the relevant fields. How did the throughput of the data connection evolve along the time? Is it according to the TCP congestion control mechanism?

The **TCP congestion control mechanism** works by dynamically adjusting the data transfer rate to avoid overloading the network. It uses algorithms such as **Slow Start**, **Congestion Avoidance**, **Fast Retransmit**, and **Fast Recovery** to adjust the window size and retransmit lost packets.

- **Relevant TCP fields**:
    - **Window Size**: This is adjusted according to the network's congestion status.

- **Congestion Window (cwnd)**: Determines how much data can be sent before receiving an acknowledgment.

Over time, the throughput typically starts small (slow start), increases gradually (congestion avoidance), and may decrease if packet loss occurs (indicating congestion). In Wireshark, you can observe the window size and retransmissions, which are indicators of the congestion control mechanism at work.

The throughput should follow the TCP congestion control curve, starting slow and increasing over time until it reaches a point where packet loss may reduce the rate.

-> Is the throughput of a TCP data connections disturbed by the appearance of a second TCP connection? How?

Yes, the throughput of a TCP data connection can be disturbed by the appearance of a second TCP connection, particularly if both connections share the same network resources (bandwidth). When the second connection starts, the available bandwidth is divided between both, which can lead to a **reduction in throughput** for the first connection. This happens because TCP congestion control mechanisms try to avoid network congestion by adjusting the window size based on the available bandwidth, causing the throughput to decrease as more traffic is introduced into the network. This effect is more noticeable if both connections are competing for the same network path.

Conclusions

In conclusion, we successfully met the objectives of this project by developing a file download application using FTP and TCP/IP, and setting up a network to test it. Through the development process, we gained valuable insights into how FTP and TCP/IP protocols enable file transfers. Configuring and troubleshooting the network further enhanced our understanding of networking concepts. Overall, the project allowed us to strengthen our skills in both application development and network setup, providing practical experience and laying a solid foundation for future work in software development and networking.

Appendix I - Source Code

download.h

```h
#ifndef  download_H
#define  download_H


#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>
#include <regex.h>
#include <fcntl.h>

#define MAX_LEN 500

#define SERVER_PORT 21
#define CODE_SIZE 4
#define WELCOME_CODE 220
#define USER_CODE 331
#define PASSWORD_CODE 230
#define PASSIVE_CODE 227
#define MIN_RETREIVE_CODE 100
#define MAX_RETREIVE_CODE 199
#define RETREIVE_CODE_WITH_SIZE 150
#define MIN_TRANSFER_CODE 200
#define BINARY_CODE 200
#define GOODBYE_CODE 221

typedef struct
{
    char ip[16];
    char user[MAX_LEN];
    char password[MAX_LEN];
    char host[MAX_LEN];
    char url_path[MAX_LEN];
    char filename[MAX_LEN];
} URL;

typedef struct
{
    char message[1024];
    int code;

} response;


int create_socket(char *ip,int port);
int get_ip(char* hostname, URL *url);
int parse_URL(URL *url,char *url_str);
int close_socket(int sockfd);
int receiveResponse(int socketfd, response *res);
int readUntilNewline(int socketfd, char *buf);
int readCode(int socketfd, char *code);
void showResponse(response *res);
void reset_response(response *newMessage);
int writeMessage(int sockfd, response *message);
int calculate_new_port(char *passiveMsg, URL url);
int readfile(int sockfd, char *filename, long long file_size);
long long getFileSize(char * message);

#endif
```

download.c

```c
#include "download.h"


int main(int argc, char *argv[]) {

    if(argc != 2) {
        perror("Invalid call to download.");
        return -1;
    }

    URL url;
    memset(&url, 0, sizeof(URL));

    if(parse_URL(&url, argv[1]) == -1) {
        perror("parse_URL()");
        return -1;
    }
    printf("User: %s\n", url.user);
    printf("Password: %s\n", url.password);
    printf("Host: %s\n", url.host);
    printf("URL Path: %s\n", url.url_path);
    printf("Filename: %s\n", url.filename);

    if(get_ip(url.host, &url) == -1) {
        perror("get_ip()");
        return -1;
    }
    int sockfd;
    printf("Creating socket\n");
    if((sockfd =  create_socket(url.ip, SERVER_PORT)) == -1) {
        perror("create_socket()");
        return -1;
    }
    printf("Socket created\n");

    response newMessage;

    reset_response(&newMessage);

    int ret = receiveResponse(sockfd,&newMessage);

    if(ret == -1) {
        if(close_socket(sockfd) == -1) return -1;
        perror("receiveResponse()");
        return -1;
    }

    if(newMessage.code != WELCOME_CODE) {
        if(close_socket(sockfd) == -1) return -1;
        perror("Server did not respond with 220.");
        return -1;
    }
    printf("------------------------------------------------------------\n");
    printf("Connected to server\n");
    /*send username*/

    reset_response(&newMessage);
    newMessage.code = USER_CODE;
    snprintf(newMessage.message, sizeof(newMessage.message), "user %s\r\n", url.user);

    if(!writeMessage(sockfd,&newMessage)){
        if(close_socket(sockfd) == -1) return -1;
        perror("Failed to send username");
        return -1;
    }

    /*Send password*/

    reset_response(&newMessage);
    newMessage.code = PASSWORD_CODE;
    snprintf(newMessage.message, sizeof(newMessage.message), "pass %s\r\n", url.password);

    if(!writeMessage(sockfd,&newMessage)){
        if(close_socket(sockfd) == -1) return -1;
        perror("Failed to send password");
        return -1;
    }
    /*Send binary format*/

    reset_response(&newMessage);
    newMessage.code = BINARY_CODE;
    strncpy(newMessage.message, "type I\r\n", sizeof(newMessage.message));

    if(!writeMessage(sockfd,&newMessage)){
```

```
    if(!writeMessage(sockfd,&newMessage)){
        if(close_socket(sockfd) == -1) return -1;
        perror("Failed to pass binary format");
        return -1;
    }


    /*Send passive mode*/

    reset_response(&newMessage);
    newMessage.code = PASSIVE_CODE;
    strncpy(newMessage.message, "pasv\r\n", sizeof(newMessage.message));

    if(!writeMessage(sockfd,&newMessage)){
        if(close_socket(sockfd) == -1) return -1;
        perror("Failed to activate passivemode");
        return -1;
    }
    int port2;

    if((port2 = calculate_new_port(newMessage.message,url)) == - 1){
        if(close_socket(sockfd) == -1) return -1;
        perror("Failed to calculate port");
        return -1;
    }
    int sockfd2;
    if ((sockfd2 = create_socket(url.ip,port2)) == -1){
        perror("Failed to create socket");
        return -1;
    }

    /*Send retreive*/
    reset_response(&newMessage);
    newMessage.code = RETREIVE_CODE_WITH_SIZE;
    snprintf(newMessage.message, sizeof(newMessage.message), "retr %s\r\n", url.url_path);

    if(!writeMessage(sockfd,&newMessage)){
        if(close_socket(sockfd) == -1) return -1;
        perror("Failed to retreive");
        return -1;
    }
    long long fileSize = 1;
    printf("Filename: %s\n",url.filename);
    if (newMessage.code == RETREIVE_CODE_WITH_SIZE) {
        fileSize = getFileSize(newMessage.message);
    }

    if(fileSize == -1){
        perror("Failed to get the file size");
        return -1;
    }

    if(readfile(sockfd2,url.filename,fileSize) == -1){
        if(close_socket(sockfd) == -1) return -1;
        return -1;
    }

    reset_response(&newMessage);

    ret = receiveResponse(sockfd,&newMessage);

    if(ret == -1) {
        if(close_socket(sockfd) == -1) return -1;
        perror("receiveResponse()");
        return -1;
    }

    if(newMessage.code < MIN_TRANSFER_CODE) {
        if(close_socket(sockfd) == -1) return -1;
        perror("Transfer went wrong.");
        return -1;
    }

    printf("Transfer completed successfully\n");

    reset_response(&newMessage);
    newMessage.code = GOODBYE_CODE;
    strncpy(newMessage.message, "quit\r\n", sizeof(newMessage.message));

    if(!writeMessage(sockfd,&newMessage)){
        if(close_socket(sockfd) == -1) return -1;
        perror("Failed to say goodbye");
        return -1;
    }

    if(close_socket(sockfd) == -1) return -1;
    if(close_socket(sockfd2) == -1) return -1;
    return 0;
```

```c
}

void reset_response(response *res) {
    if (res == NULL) return; // Safety check
    res->code = 0; // Reset the response code
    memset(res->message, 0, sizeof(res->message)); // Clear the message buffer
}

int create_socket(char *ip,int port) {

    int sockfd;
    struct sockaddr_in server_addr;

    /*server address handling*/
    bzero((char *) &server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(ip);    /*32 bit Internet address network byte ordered*/
    server_addr.sin_port = htons(port);        /*server TCP port must be network byte ordered */

    /*open a TCP socket*/
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket()");
        return -1;
    }
    /*connect to the server*/
    if (connect(sockfd,
                (struct sockaddr *) &server_addr,
                sizeof(server_addr)) < 0) {
        perror("connect()");
        return -1;
    }



    return sockfd;
}


int get_ip(char* hostname, URL *url) {
    struct hostent *h;

    if ((h = gethostbyname(hostname)) == NULL) {
        perror("gethostbyname()");
        return -1;
    }

    printf("Host name  : %s\n", h->h_name);
    printf("IP Address : %s\n", inet_ntoa(*((struct in_addr *) h->h_addr)));
    strncpy(url->ip, inet_ntoa(*((struct in_addr *) h->h_addr)), sizeof(url->ip) - 1);
    return 0;
}


int parse_URL(URL *url, char *url_str) {
    // Regular expression to match the URL format
    const char *regex_pattern = "^ftp://(([^:@/]+)(:([^@/]+))?@)?([^/]+)/(.+)$";

    regex_t regex;
    regmatch_t matches[7];  // There are 7 capture groups in the regex

    // Compile the regular expression
    if (regcomp(&regex, regex_pattern, REG_EXTENDED) != 0) {
        fprintf(stderr, "Failed to compile regex.\n");
        return -1;
    }

    // Execute the regular expression on the URL string
    if (regexec(&regex, url_str, 7, matches, 0) != 0) {
        fprintf(stderr, "Invalid URL format.\n");
        regfree(&regex);
        return -1;
    }

    // Extract user and password (if present)
    if (matches[1].rm_so != -1) {
        // If user and password are present
        if (matches[2].rm_so != -1) {
            strncpy(url->user, url_str + matches[2].rm_so, matches[2].rm_eo - matches[2].rm_so);
            url->user[matches[2].rm_eo - matches[2].rm_so] = '\0';
        } else {
            strncpy(url->user, "anonymous", sizeof(url->user) - 1);
        }

        if (matches[4].rm_so != -1) {
            strncpy(url->password, url_str + matches[4].rm_so, matches[4].rm_eo - matches[4].rm_so);
            url->password[matches[4].rm_eo - matches[4].rm_so] = '\0';
        } else {
            strncpy(url->password, "password", sizeof(url->password) - 1); // No password
```

```
            strncpy(url->password, "password", sizeof(url->password) - 1); // No password
        }
    } else {
        strncpy(url->user, "anonymous", sizeof(url->user) - 1);   // No user
        strncpy(url->password, "password", sizeof(url->password) - 1); // No password
    }

    // Extract host
    if (matches[5].rm_so != -1) {
        strncpy(url->host, url_str + matches[5].rm_so, matches[5].rm_eo - matches[5].rm_so);
        url->host[matches[5].rm_eo - matches[5].rm_so] = '\0';
    } else {
        url->host[0] = '\0'; // Handle missing host
    }

    // Extract URL path
    if (matches[6].rm_so != -1) {
        strncpy(url->url_path, url_str + matches[6].rm_so, matches[6].rm_eo - matches[6].rm_so);
        url->url_path[matches[6].rm_eo - matches[6].rm_so] = '\0';
    } else {
        url->url_path[0] = '\0'; // Handle missing URL path
    }

    // Extract filename from the path (if exists)
        char *last_slash = strrchr(url->url_path, '/');
    if (last_slash && *(last_slash + 1) != '\0') {
        // There is a filename after the last slash
        strncpy(url->filename, last_slash + 1, MAX_LEN - 1);
        url->filename[MAX_LEN - 1] = '\0';
    } else {
        // No filename, use the entire path as the filename
        strncpy(url->filename, url->url_path, MAX_LEN - 1);
        url->filename[MAX_LEN - 1] = '\0';
    }

    // Set IP to an empty string (no parsing logic for IP)
    url->ip[0] = '\0';

    // Free the regex memory
    regfree(&regex);

    return 0;
}

int close_socket(int sockfd) {
    if(close(sockfd) < 0) {
        perror("Error closing socket");
        return -1;
    }
    return 0;
}


int receiveResponse(int socketfd, response *res) {
    int is_num = 1;
    while (1) {

        if (readUntilNewline(socketfd, res->message) == -1) return -1; // Error reading until newline

        showResponse(res);

        // check if its the code is a number
        for (int i = 0; i < 3; i++) {
            if (!(res->message[i] >= '0' && res->message[i] <= '9')) {
                is_num = 0;
                break;
            }
        }

       // Check if its only the code without -
        if ((res->message[3] == ' ') && is_num) {
            res->code = atoi(res->message); // Convert the code to an integer
            return 0;
        }

        reset_response(res);
        // Read until a newline character is encountered
        is_num = 1;
    }

    return 0;
}

int readCode(int socketfd, char *code) {
    for (int j = 0; j < CODE_SIZE; j++) {
        int res = read(socketfd, &code[j], 1);
        if (res < 0) return -1; // Handle read error
    }
```

```c
    return 0; // Successfully read the code
}

// Function to read data until a newline character is encountered
int readUntilNewline(int socketfd, char *buf) {
    int res;
    int i = 0;
    do {
        res = read(socketfd, &buf[i++], 1);
        if (res < 0) return -1; // Handle read error
    } while (buf[i - 1] != '\n');

    return 0; // Successfully read until newline
}


// Message formating is weird
void showResponse(response *res) {
    printf("%s", res->message);
}


int writeMessage(int sockfd, response *message){
    if (sockfd < 0) {
        perror("Invalid socket descriptor");
        return -1;
    }

    if (strlen(message->message) == 0) {
        fprintf(stderr, "Invalid message to write\n");
        return -1;
    }

    printf("\n%s", message->message);

    size_t bytes_sent = 0;
    size_t to_send = strlen(message->message);

    // Write the message in chunks if necessary
    while (bytes_sent < to_send) {
        ssize_t result = write(sockfd, message->message + bytes_sent, to_send - bytes_sent);
        if (result < 0) {
            perror("Error writing to socket");
            return -1;
        }
        bytes_sent += result;
    }


    // Receive the server's response
    response res;
    memset(&res, 0, sizeof(res));

    if (receiveResponse(sockfd, &res) == -1) {
        perror("receiveResponse failed");
        return -1;
    }


    strncpy(message->message, res.message, sizeof(message->message) - 1);
    message->message[sizeof(message->message) - 1] = '\0';

    if (message->code ==  RETREIVE_CODE_WITH_SIZE) {
        message->code = res.code;
        return res.code >= MIN_RETREIVE_CODE && res.code <= MAX_RETREIVE_CODE;
    }

    return res.code == message->code;
}

int calculate_new_port(char *passiveMsg, URL url){
    int ip1, ip2, ip3, ip4, port1, port2;

    int parsed = sscanf(passiveMsg, "%*[^(](%d,%d,%d,%d,%d,%d)", &ip1, &ip2, &ip3, &ip4, &port1, &port2);

    if (parsed != 6) {
        perror("Failed to parse passive message");
        return -1;
    }

    // Compare the extracted IP with the provided IP (url.ip should be in the same format as the parsed IP)
    // Assuming url.ip is in the format "ip1.ip2.ip3.ip4"
    int url_ip1, url_ip2, url_ip3, url_ip4;
    if (sscanf(url.ip, "%d.%d.%d.%d", &url_ip1, &url_ip2, &url_ip3, &url_ip4) != 4) {
        perror("Failed to parse URL IP");
        return -1;
    }

```

```c
    // If the IPs don't match, return an error
    if (ip1 != url_ip1 || ip2 != url_ip2 || ip3 != url_ip3 || ip4 != url_ip4) {
        perror("IP addresses do not match");
        return -1;
    }

    int port = port1*256 + port2;

    return port;
}


int readfile(int sockfd, char *filename, long long file_size) {
    int file_fd;
    off_t bytes_received = 0;
    ssize_t res;

    // Open the file in write-only mode. Create it if it doesn't exist.
    // Set permissions to rw-r--r-- (user can read/write, group and others can read).
    if ((file_fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644)) < 0) {
        perror("Open file");
        return -1;
    }

    // Buffer for receiving data from the socket
    char buf[1024];  // Larger buffer to read multiple bytes at once

    printf("Receiving file...\n");

    // Loop to read from the socket and write to the file
    while (1) {
        // Read data from the socket
        res = read(sockfd, buf, sizeof(buf));

        if (res == 0) {
            // End of transmission
            break;
        }
        if (res < 0) {
            // Error reading from the socket
            perror("Failed to read from socket");
            close(file_fd);
            return -1;
        }

        // Write the received data into the file
        if (write(file_fd, buf, res) < 0) {
            // Error writing to the file
            perror("Failed to write to file");
            close(file_fd);
            return -1;
        }

        // Update bytes received
        bytes_received += res;
        // Print loading bar based on the total file size
        float progress = (float)bytes_received / file_size * 100; // Progress based on actual file size
        int bar_width = 50; // Width of the progress bar
        int pos = bar_width * progress / 100;
        // Ensure that progress does not exceed 100%
        if (progress > 100.0f) {
            progress = 100.0f;
        }
        printf("[");

        for (int i = 0; i < bar_width; ++i) {
            if (i < pos) {
                printf("=");
            } else if (i == pos) {
                printf(">");
            } else {
                printf(" ");
            }
        }
        printf("] %.2f%%\r", progress);
        fflush(stdout);  // Flush to update the progress bar immediately
    }

    printf("\nFile received and written to %s\n", filename);

    // Close the file
    close(file_fd);
    return 0;
}

long long getFileSize(char * message) {
    long long bytes = -1;

```

```c
long long getFileSize(char * message) {
    long long bytes = -1;


    char *start = strrchr(message, '(');
    if (start != NULL) {
        char *end = strchr(start, ')');
        if (end != NULL) {

            *end = '\0';


            if (sscanf(start + 1, "%lld", &bytes) == 1) {
                return bytes;
            } else {
                printf("Falha ao extrair o número de bytes\n");
            }
        }
    }
    return -1;  // Return -1
}
```

Appendix II - Part 2 Steps & Commands

Exp 1 - Configure an IP Network

1. Connect the eth1 interface of TUXY3 and TUXY4 to the switch (White Box).
2. Configure the IP addresses for the eth1 interface on each PC:

```
Unset
$ ifconfig eth1 up                  # Activate the interface
$ ifconfig eth1 172.16.110.1/24       # For TUXY3
$ ifconfig eth1 172.16.110.254/24     # For TUXY4
```

3. Use ifconfig to check the values (IP and MAC).
4. To verify connectivity between the two computers, use ping (sends packets and waits for a response):

```
Unset
$ ping 172.16.110.254 -c 6    # For TUXY3, the flag limits the number of packets sent
$ ping 172.16.110.1 -c 6      # For TUXY4
```

5. To inspect the Forwarding Table and ARP Table (on TUXY3):

```
Unset
$ route -n      # Forwarding Table
$ arp -a        # ARP Table (ARP translates IP to MAC address, it should display: ?
(172.16.110.1) at <MAC> on ...)
```

6. To delete an entry from the ARP table (on TUXY3):

```
Unset
$ arp -d 172.16.110.254   # Remove entries associated with TUXY4
```

7. Start Wireshark on TUXY3's eth1 interface.
8. On TUXY3, ping TUXY4:

```
Unset
$ ping 172.16.110.254 -c 6    # You should observe 6 packets
```

9. You should see packets using the following protocols:
    - ARP -> 2 packets: the first one from the source requesting the MAC address for a given IP, and the second one is the destination's response, sending its MAC address to the source.
    - ICMP -> Used for exchanging information and errors.

Exp 2 - Implement two bridges in a switch

1. Configure eth1 on TUXY2:

```
Unset
$ ifconfig eth1 up
$ ifconfig eth1 172.16.111.1/24
```

2. Create two bridges on the switch (TUXY2):

    - First, reset the switch configuration:

```
Unset
# Open GKTerm and set the baudrate to 115200.
> admin
> (password is empty)
> /system reset-configuration
> y
```

    - Second, create the two bridges:

```
Unset
> /interface bridge add name=bridge110
> /interface bridge add name=bridge111
```

- Third, remove the ports where the TUX computers are connected (assuming TUXY2, TUXY3, and TUXY4 are connected to interfaces ether1, ether2, and ether3 of the switch, respectively):

```
Unset
> /interface bridge port remove [find interface=ether2]
> /interface bridge port remove [find interface=ether3]
> /interface bridge port remove [find interface=ether4]
```

- Fourth, add the ports (for bridge110, connect TUXY3 and TUXY4; for bridge111, connect only TUXY2):

```
Unset
> /interface bridge port add bridge=bridge110 interface=ether3
> /interface bridge port add bridge=bridge110 interface=ether4
> /interface bridge port add bridge=bridge111 interface=ether2
```

- Fifth, to ensure everything is correct, run:

```
Unset
> /interface bridge port print
```

3. Start packet capture on TUXY3.
4. On TUXY3, ping TUXY4 and TUXY2:

```
Unset
$ ping 172.16.110.254   # Ping TUXY4 -> This will work (there is a bridge between TUXY3 and TUXY4)
$ ping 172.16.111.1     # Ping TUXY2 -> "connect: Network is unreachable"
```

5. Stop the captures and start capturing on all TUX computers.
6. On TUXY3, send a broadcast:

```
Unset
$ ping -b 172.16.110.255    # TUXY4 will receive the broadcast, but TUXY2 will not (same reason as above)
```

7. Repeat steps 5 and 6, but this time on TUXY2, broadcasting to the subnet ping -b 172.16.111.0/24:

```
Unset
$ ping -b 172.16.111.255    # Neither TUXY4 nor TUXY3 will receive the broadcast (same reason as above)
```

Exp 3 - Configure a Router in Linux

1. Activate eth2 on TUXY4:

```
Unset
$ ifconfig eth2 up
$ ifconfig eth2 172.16.111.253/24
```

2. Add TUXY4 to bridge 111 (connect to the switch as done previously):

```
Unset
> /interface bridge port remove [find interface=ether5]
> /interface bridge port add bridge=bridge111 interface=ether5
```

3. Enable IP forwarding and disable ICMP echo-ignore-broadcast:

```
Unset
# 1. Enable IP forwarding. This allows the computer to function as a router, forwarding packets to other subnets.
$ echo 1 > /proc/sys/net/ipv4/ip_forward
# 2. Disable ICMP echo-ignore-broadcast. This allows the PC to respond to broadcast messages.
$ echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
```

4. Check the IP and MAC addresses for eth1 and eth2 on TUXY4:

```
Unset
$ ifconfig      # The IP address should correspond to each subnet, and the MAC addresses should be different.
```

5. Add routes between TUXY2 and TUXY3 to connect them (the routes pass through TUXY4, which acts as a router):

```
Unset
$ route add -net 172.16.110.0/24 gw 172.16.111.253  # For TUXY2 (<destination subnet> gw <TUX's IP>)
$ route add -net 172.16.111.0/24 gw 172.16.110.254  # For TUXY3
# This means: to reach the subnet, the gateway (gw) must be used.
```

6. Check the routes on all three TUX machines:

```
Unset
$ route -n
```

7. Start capturing on TUXY3 and ping the other networks:

```
Unset
$ ping 172.16.110.254   # Must work
$ ping 172.16.111.253   # Must work
$ ping 172.16.111.1     # Must work
```

8. Stop capturing on TUXY3 and start capturing on TUXY4 for both eth1 and eth2.
9. Clear ARP table entries on the TUX machines:

```
Unset
$ arp -d 172.16.111.253 # TUXY2
$ arp -d 172.16.110.254 # TUXY3
$ arp -d 172.16.110.1   # TUXY4
$ arp -d 172.16.111.1   # TUXY4
```

10. On TUXY3, ping TUXY2 and check the logs for both Wireshark instances on TUXY4:

```
Unset
$ ping 172.16.111.1
```

Exp 4 - Configure a Commercial Router and Implement NAT

1. Connect ether1 of the router to the 111.12 strip.
2. Connect ether2 of the router to the switch and add it to bridge111:

```
Unset
> /interface bridge port remove [find interface=ether6]
> /interface bridge port add bridge=bridge111 interface=ether6
```

3. Switch the cable from the switch to the router????
4. Connect to the router from TUXY2 using GKTerm:

```
Unset
# Serial Port: /dev/ttyS0
# Baudrate: 115200
# Username: admin
# Password: <ENTER>
```

5. Reset the configuration:

```
Unset
> /system reset-configuration
```

6. Configure the router's IP address:

```
Unset
> /ip address add address=172.16.1.111/24 interface=ether1  # Connect eth1 to the RCOM network
> /ip address add address=172.16.111.254/24 interface=ether2 # Connect to bridge111
```

7. Add default routes on the router and TUX machines:

```
Unset
> /ip route add dst-address=172.16.110.0/24 gateway=172.16.111.253  # Router console (Packets for subnet 172.16.110 must go through TUXY4)
# For some reason, the command below was not used in the screenshots:
> /ip route add dst-address=0.0.0.0/0 gateway=172.16.1.111         # Router console (Associate 0.0.0.0 addresses with the port connecting to the FTP Server)
## DO NOT USE DEFAULTS = 172.16.1.0/24
```

```
$ route add -net 172.16.1.0/24 gw 172.16.111.254 # TUXY2 (Connect to the router)
$ route add -net 172.16.1.0/24 gw 172.16.110.254 # TUXY3 (Connect to TUXY4, which in turn has its default route connected to the router)
$ route add -net 172.16.1.0/24 gw 172.16.111.254 # TUXY4 (Connect to the router)
```

8. Verify if TUXY3 can ping other TUX machines and the router (It should work):

```
Unset
$ ping 172.16.110.254 # Ping TUXY4
$ ping 172.16.111.1   # Ping TUXY2
$ ping 172.16.111.254 # Ping Router
```

9. On TUXY2:
    - Disable accept_redirects:

```
Unset
$ echo 0 > /proc/sys/net/ipv4/conf/eth1/accept_redirects
$ echo 0 > /proc/sys/net/ipv4/conf/all/accept_redirects
```

    - Remove the route to the 172.16.110.0/24 network passing through TUXY4:

```
Unset
$ route del -net 172.16.110.0 gw 172.16.111.253 netmask 255.255.255.0
```

    - Ping TUXY3 and observe the packet path using Wireshark (it will go through the router):

```
Unset
$ ping 172.16.110.1 # Uses TUXY2's default gateway (goes to the router) -> Router forwards to TUXY4 -> TUXY4 forwards to TUXY3
```

    - Perform a traceroute to TUXY3:

```
Unset
$ traceroute -n 172.16.110.1
```

    - Add the route from TUXY2 to TUXY4 back and perform another traceroute:

```
$ route add -net 172.16.110.0/24 gw 172.16.111.253
$ traceroute -n 172.16.110.1
```

- Enable accept_redirects:

```
$ echo 1 > /proc/sys/net/ipv4/conf/eth0/accept_redirects # Shouldn't this be eth1???
$ echo 1 > /proc/sys/net/ipv4/conf/all/accept_redirects
```

10. On TUXY3, ping the FTP Server (it should work):

```
$ ping 172.16.1.10
```

11. Disable NAT on the router:

```
> /ip firewall nat disable 0
```

12. On TUXY4, ping the server (172.16.1.10) and observe what happens (it won't work because packets to 172.16.0.0/16 networks are interpreted as local and routers will drop them):

```
$ ping 172.16.1.10
```

13. Re-enable NAT:

```
> /ip firewall nat enable 0
```

Exp 5 - DNS

1. Enable DNS on all TUX machines:

```
Unset

$ echo "nameserver 10.227.20.3" > /etc/resolv.conf
```

2. Ping a website using its hostname and observe the packets in Wireshark (TUXY2):

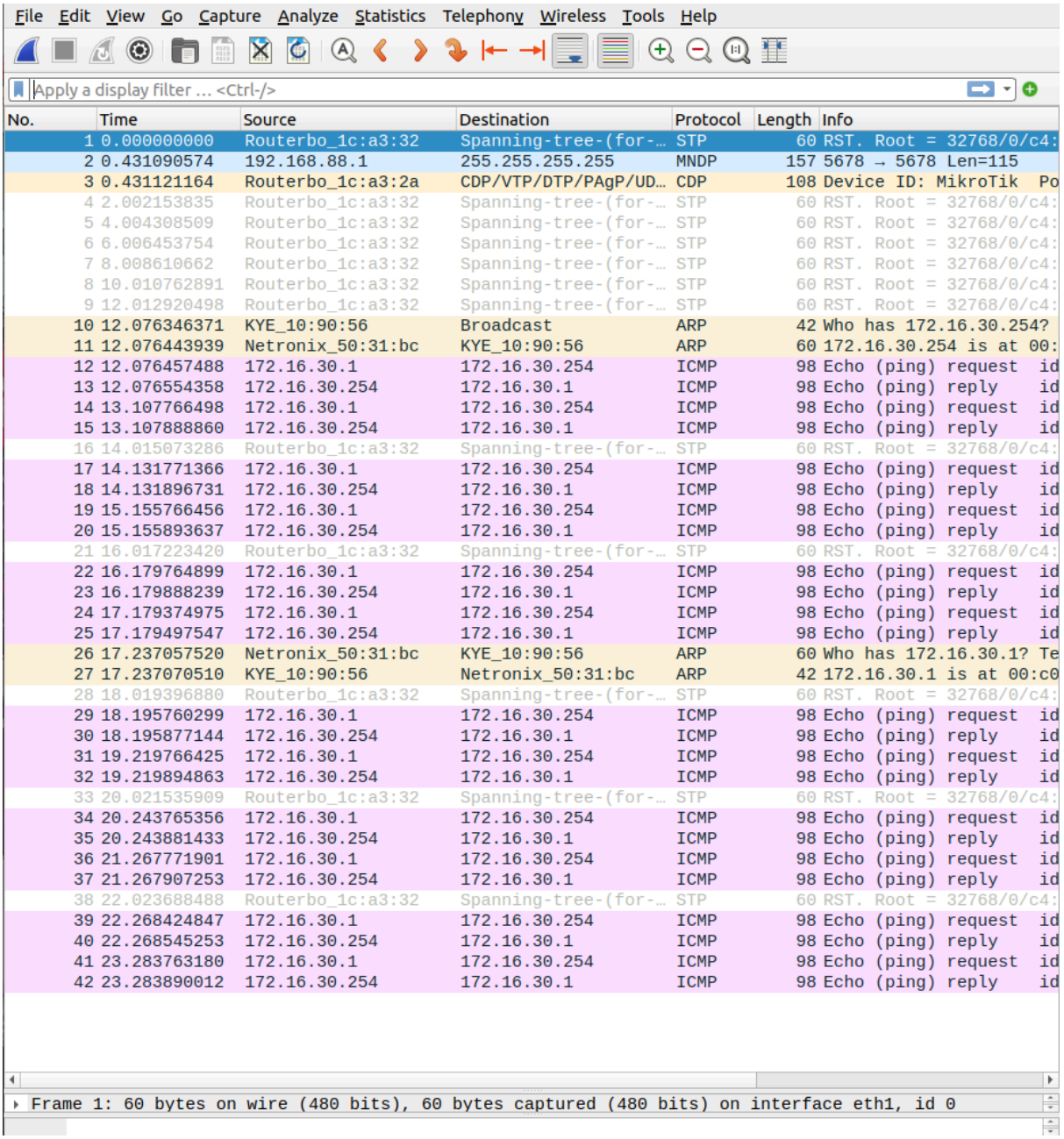
```
Unset

$ ping google.com
```

Exp 6 - TCP connections

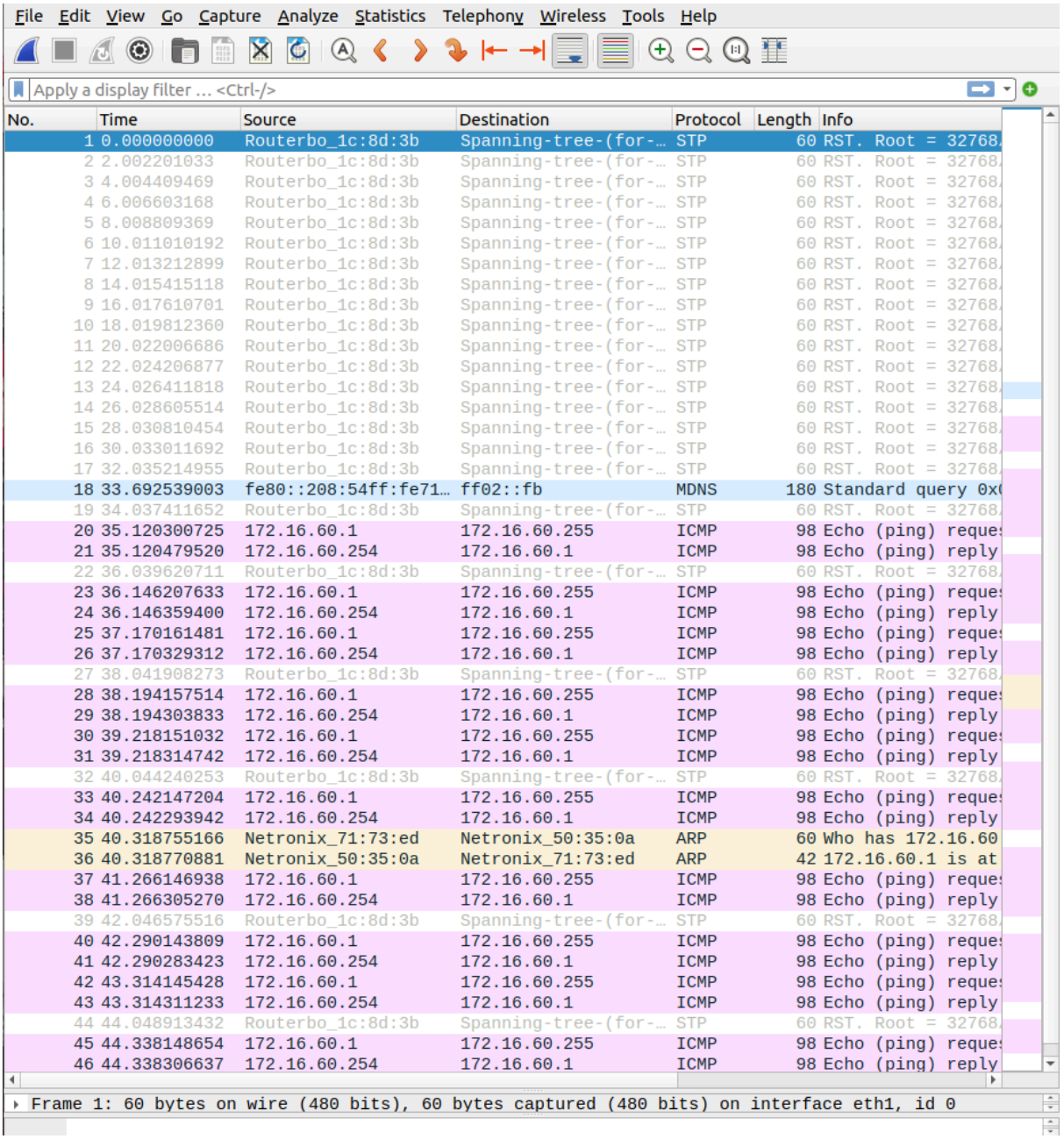
1. Start capturing on TUXY3 and run the program.
2. Repeat step 1, but start transferring on TUXY2 at the same time and observe in Wireshark.

Appendix II - Part 2 Wireshark

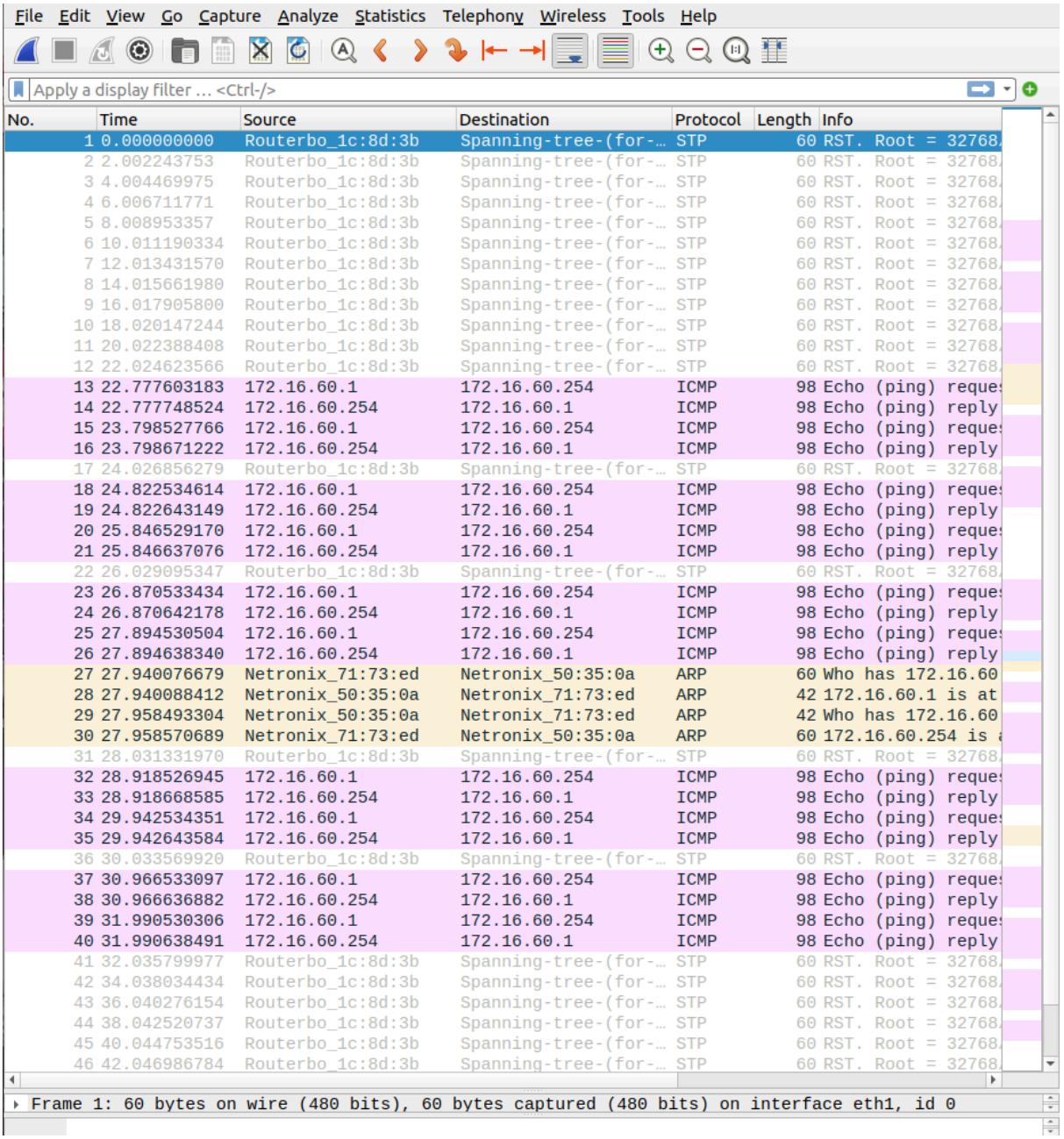
Exp 1

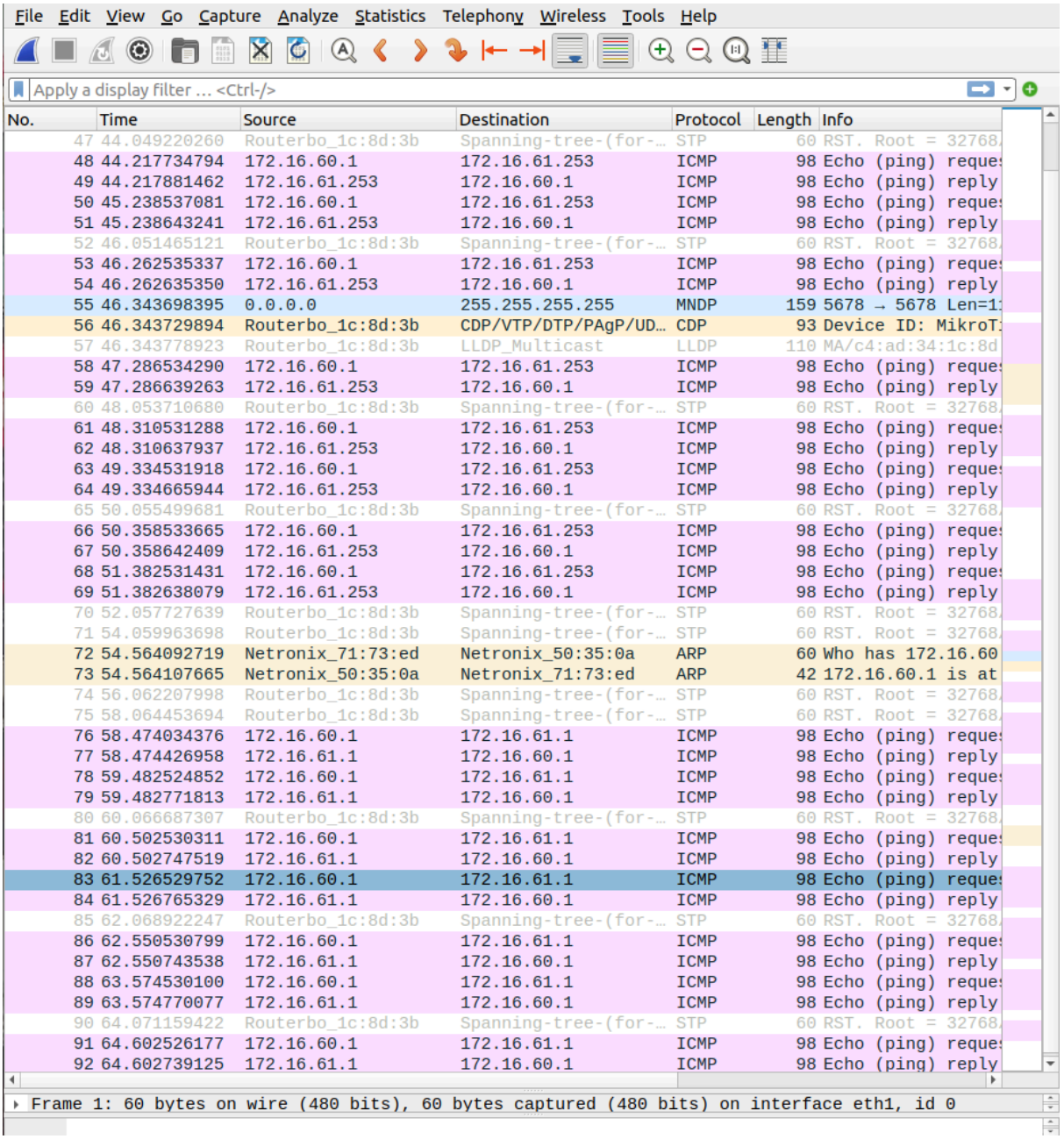


Exp 2

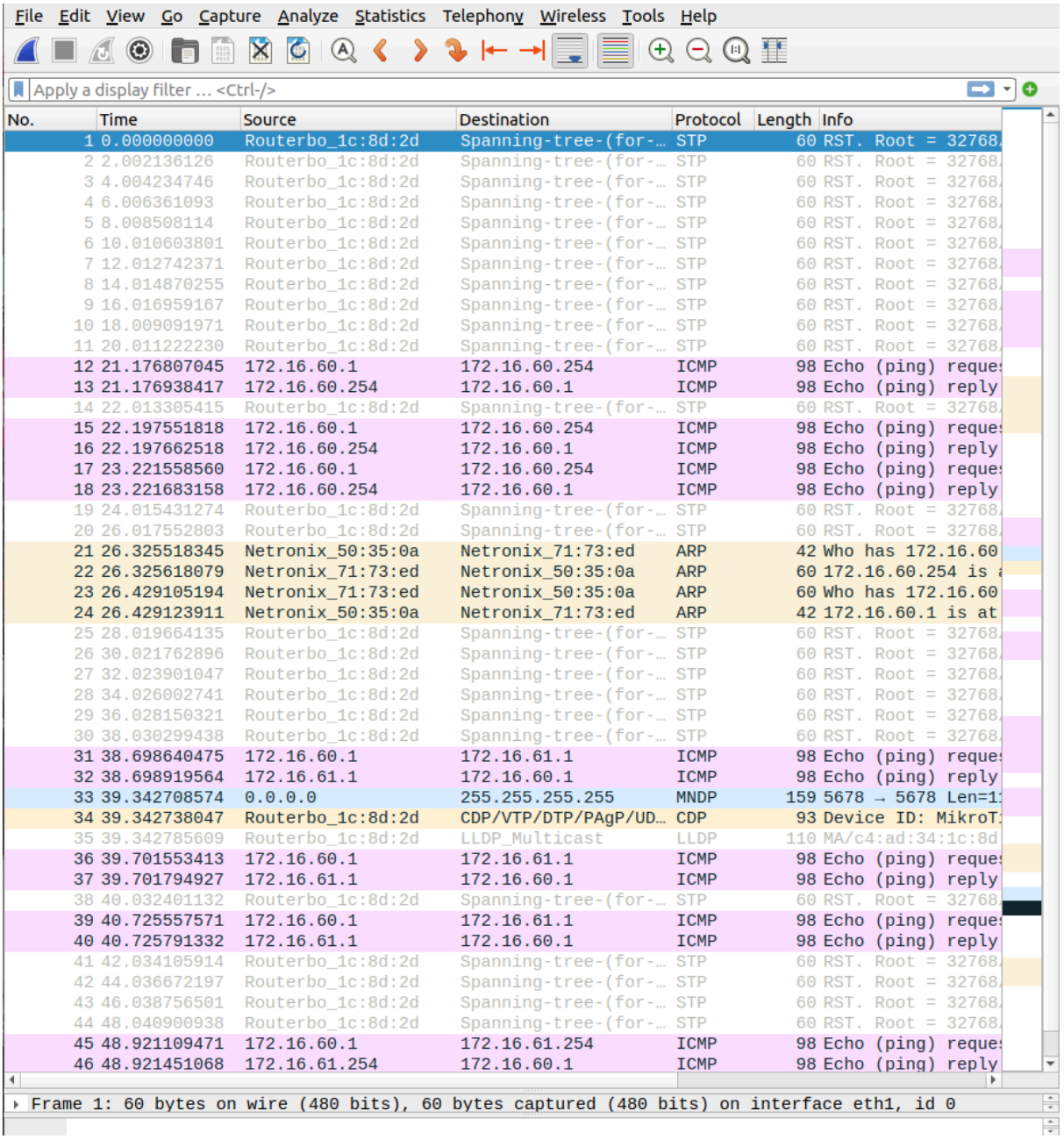

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
No. Time Source Destination Protocol Length Info
1 0.000000000 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
2 2.002201033 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
3 4.004409469 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
4 6.006603168 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
5 8.008809369 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
6 10.011010192 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
7 12.013212899 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
8 14.015415118 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
9 16.017610701 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
10 18.019812360 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
11 20.022006686 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
12 22.024206877 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
13 24.026411818 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
14 26.028605514 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
15 28.030810454 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
16 30.033011692 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
17 32.035214955 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
18 33.692539003 fe80::208:54ff:fe71… ff02::fb MDNS 180 Standard query 0x0
19 34.037411652 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
20 35.120300725 172.16.60.1 172.16.60.255 ICMP 98 Echo (ping) reque
21 35.120479520 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
22 36.039620711 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
23 36.146207633 172.16.60.1 172.16.60.255 ICMP 98 Echo (ping) reque
24 36.146359400 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
25 37.170161481 172.16.60.1 172.16.60.255 ICMP 98 Echo (ping) reque
26 37.170329312 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
27 38.041908273 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
28 38.194157514 172.16.60.1 172.16.60.255 ICMP 98 Echo (ping) reque
29 38.194303833 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
30 39.218151032 172.16.60.1 172.16.60.255 ICMP 98 Echo (ping) reque
31 39.218314742 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
32 40.044240253 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
33 40.242147204 172.16.60.1 172.16.60.255 ICMP 98 Echo (ping) reque
34 40.242293942 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
35 40.318755166 Netronix_71:73:ed Netronix_50:35:0a ARP 60 Who has 172.16.60
36 40.318770881 Netronix_50:35:0a Netronix_71:73:ed ARP 42 172.16.60.1 is at
37 41.266146938 172.16.60.1 172.16.60.255 ICMP 98 Echo (ping) reque
38 41.266305270 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
39 42.046575516 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
40 42.290143809 172.16.60.1 172.16.60.255 ICMP 98 Echo (ping) reque
41 42.290283423 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
42 43.314145428 172.16.60.1 172.16.60.255 ICMP 98 Echo (ping) reque
43 43.314311233 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
44 44.048913432 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768/
45 44.338148654 172.16.60.1 172.16.60.255 ICMP 98 Echo (ping) reque
46 44.338306637 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface eth1, id 0

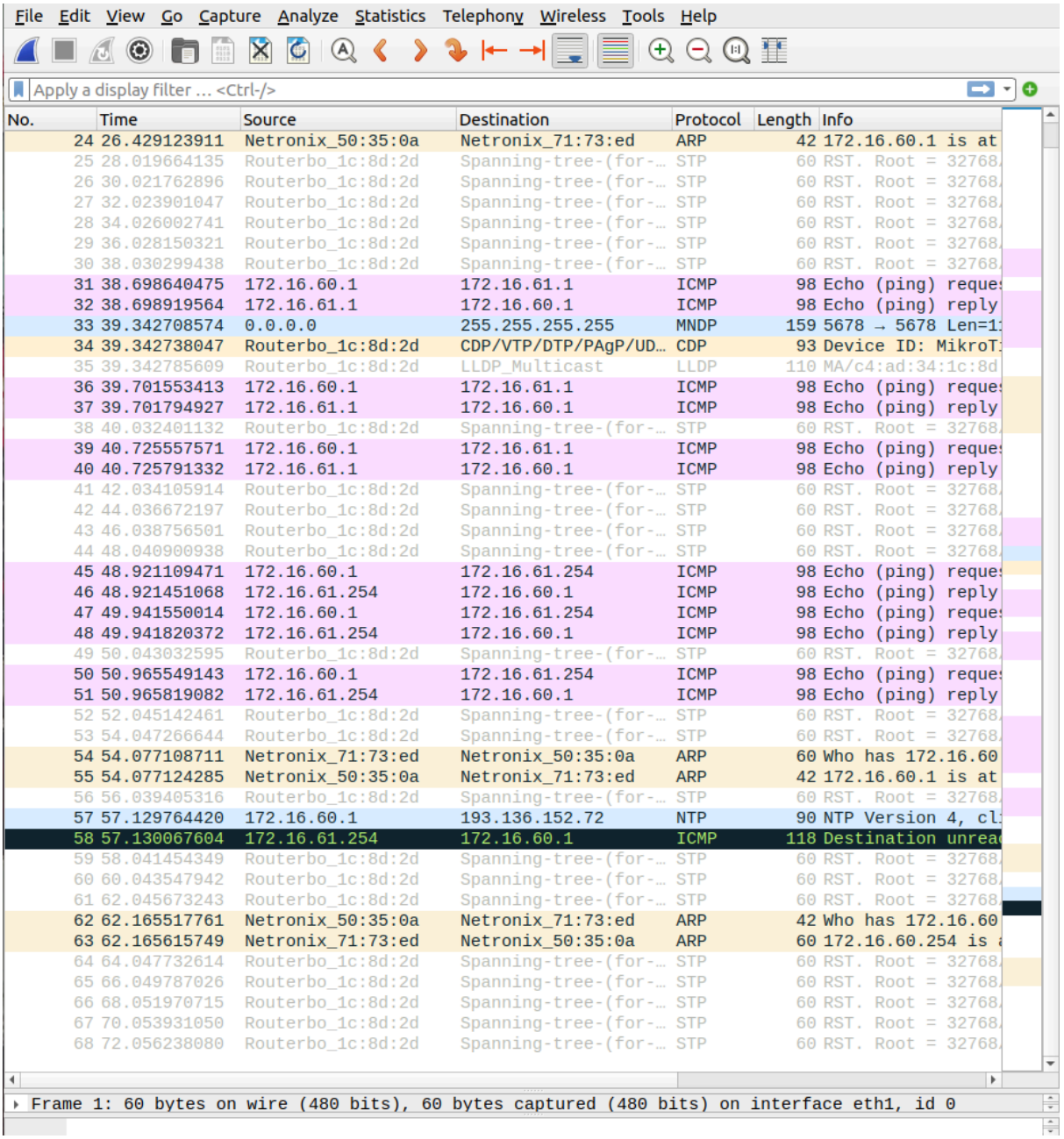
Exp 3


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
No. Time Source Destination Protocol Length Info
1 0.000000000 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
2 2.002243753 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
3 4.004469975 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
4 6.006711771 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
5 8.008953357 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
6 10.011190334 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
7 12.013431570 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
8 14.015661980 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
9 16.017905800 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
10 18.020147244 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
11 20.022388408 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
12 22.024623566 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
13 22.777603183 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
14 22.777748524 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
15 23.798527766 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
16 23.798671222 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
17 24.026856279 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
18 24.822534614 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
19 24.822643149 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
20 25.846529170 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
21 25.846637076 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
22 26.029095347 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
23 26.870533434 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
24 26.870642178 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
25 27.894530504 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
26 27.894638340 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
27 27.940076679 Netronix_71:73:ed Netronix_50:35:0a ARP 60 Who has 172.16.60
28 27.940088412 Netronix_50:35:0a Netronix_71:73:ed ARP 42 172.16.60.1 is at
29 27.958493304 Netronix_50:35:0a Netronix_71:73:ed ARP 42 Who has 172.16.60
30 27.958570689 Netronix_71:73:ed Netronix_50:35:0a ARP 60 172.16.60.254 is
31 28.031331970 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
32 28.918526945 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
33 28.918668585 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
34 29.942534351 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
35 29.942643584 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
36 30.033569920 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
37 30.966533097 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
38 30.966636882 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
39 31.990530306 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
40 31.990638491 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
41 32.035799977 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
42 34.038034434 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
43 36.040276154 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
44 38.042520737 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
45 40.044753516 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
46 42.046986784 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768
Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface eth1, id 0


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
No. Time Source Destination Protocol Length Info
47 44.049220260 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768,
48 44.217734794 172.16.60.1 172.16.61.253 ICMP 98 Echo (ping) reque
49 44.217881462 172.16.61.253 172.16.60.1 ICMP 98 Echo (ping) reply
50 45.238537081 172.16.60.1 172.16.61.253 ICMP 98 Echo (ping) reque
51 45.238643241 172.16.61.253 172.16.60.1 ICMP 98 Echo (ping) reply
52 46.051465121 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768,
53 46.262535337 172.16.60.1 172.16.61.253 ICMP 98 Echo (ping) reque
54 46.262635350 172.16.61.253 172.16.60.1 ICMP 98 Echo (ping) reply
55 46.343698395 0.0.0.0 255.255.255.255 MNDP 159 5678 → 5678 Len=1
56 46.343729894 Routerbo_1c:8d:3b CDP/VTP/DTP/PAgP/UD… CDP 93 Device ID: MikroT
57 46.343778923 Routerbo_1c:8d:3b LLDP_Multicast LLDP 110 MA/c4:ad:34:1c:8d
58 47.286534290 172.16.60.1 172.16.61.253 ICMP 98 Echo (ping) reque
59 47.286639263 172.16.61.253 172.16.60.1 ICMP 98 Echo (ping) reply
60 48.053710680 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768,
61 48.310531288 172.16.60.1 172.16.61.253 ICMP 98 Echo (ping) reque
62 48.310637937 172.16.61.253 172.16.60.1 ICMP 98 Echo (ping) reply
63 49.334531918 172.16.60.1 172.16.61.253 ICMP 98 Echo (ping) reque
64 49.334665944 172.16.61.253 172.16.60.1 ICMP 98 Echo (ping) reply
65 50.055499681 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768,
66 50.358533665 172.16.60.1 172.16.61.253 ICMP 98 Echo (ping) reque
67 50.358642409 172.16.61.253 172.16.60.1 ICMP 98 Echo (ping) reply
68 51.382531431 172.16.60.1 172.16.61.253 ICMP 98 Echo (ping) reque
69 51.382638079 172.16.61.253 172.16.60.1 ICMP 98 Echo (ping) reply
70 52.057727639 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768,
71 54.059963698 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768,
72 54.564092719 Netronix_71:73:ed Netronix_50:35:0a ARP 60 Who has 172.16.60
73 54.564107665 Netronix_50:35:0a Netronix_71:73:ed ARP 42 172.16.60.1 is at
74 56.062207998 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768,
75 58.064453694 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768,
76 58.474034376 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
77 58.474426958 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
78 59.482524852 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
79 59.482771813 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
80 60.066687307 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768,
81 60.502530311 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
82 60.502747519 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
83 61.526529752 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
84 61.526765329 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
85 62.068922247 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768,
86 62.550530799 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
87 62.550743538 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
88 63.574530100 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
89 63.574770077 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
90 64.071159422 Routerbo_1c:8d:3b Spanning-tree-(for-… STP 60 RST. Root = 32768,
91 64.602526177 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
92 64.602739125 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface eth1, id 0

Exp 4


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
No. Time Source Destination Protocol Length Info
1 0.000000000 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
2 2.002136126 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
3 4.004234746 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
4 6.006361093 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
5 8.008508114 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
6 10.010603801 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
7 12.012742371 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
8 14.014870255 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
9 16.016959167 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
10 18.009091971 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
11 20.011222230 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
12 21.176807045 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
13 21.176938417 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
14 22.013305415 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
15 22.197551818 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
16 22.197662518 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
17 23.221558560 172.16.60.1 172.16.60.254 ICMP 98 Echo (ping) reque
18 23.221683158 172.16.60.254 172.16.60.1 ICMP 98 Echo (ping) reply
19 24.015431274 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
20 26.017552803 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
21 26.325518345 Netronix_50:35:0a Netronix_71:73:ed ARP 42 Who has 172.16.60
22 26.325618079 Netronix_71:73:ed Netronix_50:35:0a ARP 60 172.16.60.254 is
23 26.429105194 Netronix_71:73:ed Netronix_50:35:0a ARP 60 Who has 172.16.60
24 26.429123911 Netronix_50:35:0a Netronix_71:73:ed ARP 42 172.16.60.1 is at
25 28.019664135 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
26 30.021762896 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
27 32.023901047 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
28 34.026002741 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
29 36.028150321 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
30 38.030299438 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
31 38.698640475 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
32 38.698919564 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
33 39.342708574 0.0.0.0 255.255.255.255 MNDP 159 5678 → 5678 Len=1
34 39.342738047 Routerbo_1c:8d:2d CDP/VTP/DTP/PAgP/UD… CDP 93 Device ID: MikroT
35 39.342785609 Routerbo_1c:8d:2d LLDP_Multicast LLDP 110 MA/c4:ad:34:1c:8d
36 39.701553413 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
37 39.701794927 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
38 40.032401132 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
39 40.725557571 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
40 40.725791332 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
41 42.034105914 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
42 44.036672197 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
43 46.038756501 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
44 48.040900938 Routerbo_1c:8d:2d Spanning-tree-(for-… STP 60 RST. Root = 32768
45 48.921109471 172.16.60.1 172.16.61.254 ICMP 98 Echo (ping) reque
46 48.921451068 172.16.61.254 172.16.60.1 ICMP 98 Echo (ping) reply
Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface eth1, id 0


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
No. Time Source Destination Protocol Length Info
24 26.429123911 Netronix_50:35:0a Netronix_71:73:ed ARP 42 172.16.60.1 is at
25 28.019664135 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
26 30.021762896 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
27 32.023901047 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
28 34.026002741 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
29 36.028150321 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
30 38.030299438 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
31 38.698640475 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
32 38.698919564 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
33 39.342708574 0.0.0.0 255.255.255.255 MNDP 159 5678 → 5678 Len=1
34 39.342738047 Routerbo_1c:8d:2d CDP/VTP/DTP/PAgP/UD... CDP 93 Device ID: MikroT
35 39.342785609 Routerbo_1c:8d:2d LLDP_Multicast LLDP 110 MA/c4:ad:34:1c:8d
36 39.701553413 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
37 39.701794927 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
38 40.032401132 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
39 40.725557571 172.16.60.1 172.16.61.1 ICMP 98 Echo (ping) reque
40 40.725791332 172.16.61.1 172.16.60.1 ICMP 98 Echo (ping) reply
41 42.034105914 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
42 44.036672197 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
43 46.038756501 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
44 48.040900938 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
45 48.921109471 172.16.60.1 172.16.61.254 ICMP 98 Echo (ping) reque
46 48.921451068 172.16.61.254 172.16.60.1 ICMP 98 Echo (ping) reply
47 49.941550014 172.16.60.1 172.16.61.254 ICMP 98 Echo (ping) reque
48 49.941820372 172.16.61.254 172.16.60.1 ICMP 98 Echo (ping) reply
49 50.043032595 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
50 50.965549143 172.16.60.1 172.16.61.254 ICMP 98 Echo (ping) reque
51 50.965819082 172.16.61.254 172.16.60.1 ICMP 98 Echo (ping) reply
52 52.045142461 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
53 54.047266644 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
54 54.077108711 Netronix_71:73:ed Netronix_50:35:0a ARP 60 Who has 172.16.60
55 54.077124285 Netronix_50:35:0a Netronix_71:73:ed ARP 42 172.16.60.1 is at
56 56.039405316 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
57 57.129764420 172.16.60.1 193.136.152.72 NTP 90 NTP Version 4, cl
58 57.130067604 172.16.61.254 172.16.60.1 ICMP 118 Destination unrea
59 58.041454349 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
60 60.043547942 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
61 62.045673243 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
62 62.165517761 Netronix_50:35:0a Netronix_71:73:ed ARP 42 Who has 172.16.60
63 62.165615749 Netronix_71:73:ed Netronix_50:35:0a ARP 60 172.16.60.254 is
64 64.047732614 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
65 66.049787026 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
66 68.051970715 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
67 70.053931050 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
68 72.056238080 Routerbo_1c:8d:2d Spanning-tree-(for-... STP 60 RST. Root = 32768/
Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface eth1, id 0