Computer Networks
2nd Lab

Class 11 Group 11

L.EIC 24/25

Redes de Computadores

Pedro Santos up202205900@fe.up.pt

Rodrigo de Sousa up202205751@fe.up.pt

Table of Contents

Summary

The second lab work for RCOM focuses on the Development of an FTP Download Application and the Creation of a Network between 3 Computers (TUXs). The project aims to enhance understanding of application protocols, network architectures, and diagnostic tools.

Introduction

This document is a detailed report of a laboratory assignment split into two main parts. The first involves developing an FTP client application adhering to RFC959 and RFC1738 standards. The second part explores configuring and analyzing network setups using Linux tools, routers, and switches, while documenting insights using tools like Wireshark. Each section emphasizes learning objectives, configurations, and findings to solidify concepts in TCP/IP and network protocols.

Part 1 - Download Application

Architecture

The FTP application is structured around the FTP protocol's core functionalities as defined in RFC959. It provides a streamlined process for downloading files, focusing on robustness and ease of debugging. The application's architecture follows these main steps:

1. **URL Parsing**: Extracting components such as username, password, host, resource path, and file name from the user-provided FTP URL.
2. **Control Connection**: Establishing a TCP connection to the FTP server for control commands using a dedicated socket.
3. **Authentication**: Authenticating the user with the FTP server via the USER and PASS commands.
4. **Passive Mode**: Switching to passive mode (PASV) to retrieve the server's IP address and port for data transfer.
5. **File Retrieval**: Sending a RETR command to request the specified file from the server.
6. **Data Transfer**: Using a separate data connection to download the file and save it locally.
7. **Error Handling**: Implementing centralized error codes and descriptive messages to assist debugging.

Core Design Features

- **ANSI Color-Coded Logs**: Enhances user interaction by providing colored outputs for errors, warnings, and information.
- **Centralized Error Management**: Uses enumerated error codes for streamlined error reporting.
- **Reusable Functions**: Modular functions for URL parsing, socket management, and protocol interactions.

Successful Download

We were given three example files we could use to test if our application was running correctly, and it was.

The URL we tested here is ftp://ftp.up.pt/pub/gnu/emacs/elisp-manual-21-2.8.tar.gz.

The application parsed the URL, established connections, authenticated, and entered passive mode. At the end, the requested file was retrieved and saved successfully.

[See “Picture I - Download Wireshark” at the Appendix II]

As we can observe in the Wireshark Logs, the client requests the Login as anonymous with the password anonymous. It then requests Passive Mode and Retrieves the file, starting the transfer of FTP Data.

Part 2 - Network Configuration

Experiment 1: Configure an IP Network

Our main goals during this experiment were to configure IP addresses on the network interfaces, test the connectivity between two machines and analyze ARP and ICMP traffic. With this in mind we started by connecting the two machines (TUX3 and TUX4) using a switch in the 172.16.110.0/24 subnet. On TUX3 we entered the following command: ifconfig eth1 172.16.110.1/24, and on TUX4 this one: ifconfig eth1 172.16.110.254/24. We can now test the connectivity by using ping 172.16.110.254 on TUX3. To analyze ARP and ICMP traffic we use Wireshark and start capturing packets on TUX3.eth1 during the ping runtime. This is what we see:

[See “Picture II - Exp 1 Wireshark” at the Appendix II]

As we can see in “Picture II - Exp 1 Wireshark”, an ARP request was sent and the MAC address of tux4 was resolved. The ICMP Echo and Reply also present in the Wireshark Capture confirms the successful exchange of ICMP packets between TUX3 and TUX4.

Experiment 2: Implement Two Bridges in a Switch

During experiment 2 we want to create two separate broadcast domains using bridges and validate the configuration by analyzing traffic, again using Wireshark. We start by connecting TUX2, TUX3 and TUX4 to a switch configured using two bridges (bridge110 connects TUX3 and TUX4; bridge111 connects TUX2).

To achieve this, we physically connect the TUXs to the switch and using GKTerm create the bridges with these commands: /interface bridge add name=bridge110 and /interface bridge add name=bridge111. We then need to remove the default ports (/interface bridge port remove [find interface=ether2]; /interface bridge port remove [find interface=ether3]; /interface bridge port remove [find interface=ether4]) and add ports to our new bridge110 (/interface bridge port add bridge=bridge110 interface=ether3; /interface bridge port add bridge=bridge110 interface=ether4) and bridge111 (/interface bridge port add bridge=bridge111 interface=ether2). Now that we have all the setup ready, we can test the connectivity and capture traffic using Wireshark.

[See “Picture III - Exp 2 Pinging TUX2 and TUX4” at the Appendix II]

The broadcast packets sent from TUX3 were visible on our bridge110, but not on bridge111. This means we can communicate within the same bridge successfully.

[See “Picture IV - Exp 2 Pinging the Broadcast Address” at the Appendix II]

Experiment 3: Configure a Router in Linux

In this experiment we aim to transform TUX4 into a router and verify connectivity between TUX2 and TUX3 through it. We start by connecting the two subnets (TUX3 and TUX4 in 172.16.110.0/24; TUX2 and TUX4 in 172.16.111.0/24). Then, in TUX4's terminal we enter two commands: one to enable IP Forwarding (echo 1 > /proc/sys/net/ipv4/ip_forward) and another to disable ICMP Echo Ignore for Broadcasts (echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts). We now need to configure the routes, so we change to TUX2 and run the following command: route add -net 172.16.110.0/24 gw 172.16.111.253. In TUX3 we also need a route, so we type this into the terminal: route add -net 172.16.111.0/24 gw 172.16.110.254

Now we can test the connectivity by pinging TUX2 via the router, from TUX3. We do this by, while on TUX3, using this command: ping 172.16.111.1. We also used Wireshark to analyze traffic during this experiment and the results can be found in the appendices (images V to VII)

[See "Picture V - Exp 3 Pinging TUX4.0" at the Appendix II]

[See "Picture VI - Exp 3 Pinging TUX4.1" at the Appendix II]

[See "Picture VII - Exp 3 Pinging TUX2" at the Appendix II]

These images show that the pings worked. That means that the routes are all set up correctly and that the packets are being forwarded by TUX4 to TUX2.

Experiment 4: Configure a Commercial Router and Implement NAT

Our goals during experiment 4 are to configure a commercial router (RC from now on), verify the packet flow across subnets and to understand the effects of enabling and disabling ICMP redirects. For this setup we started by connecting each TUX to a switch. TUX2 was connected to Switch2 by using this command: ifconfig eth1 172.16.111.1/24, TUX3 to Switch3 (ifconfig eth1 172.16.110.1/24) and TUX4 to Switch4 (ifconfig eth1 172.16.110.254/24). We also need to configure the RC, and to do so we enter the following commands in GKTerm: /system reset-configuration; login: admin; password: ; /ip address add address=172.16.1.119/24 interface=ether1; /ip address add address=172.16.111.254/24 interface=ether2. We then navigate the various TUXs to run some commands. In TUX2 (route add -net 172.16.1.0/24 gw 172.16.111.254), TUX3 (route add -net 172.16.1.0/24 gw 172.16.110.254) and TUX4 (route add -net 172.16.1.0/24 gw 172.16.111.254). We can now go back to GKTerm to finish the setup with the following lines: /ip route add dst-address=172.16.110.0/24 gateway=172.16.111.253 and /ip route add dst-address=0.0.0.0/0 gateway=172.16.1.254.

Now that the setup is complete we can connect the RC's ether1 to P11.12 and ether2 to the bridge111. Then we need to reset the router's configuration by entering /system reset-configuration into GKTerm. After this we need to configure the default routes for our router, which can be done by once again running some commands in GKTerm (/ip route add dst-address=172.16.110.0/24 gateway=172.16.111.253; /ip route add dst-address=0.0.0.0/0 gateway=172.16.1.254). Now we can test the connectivity using some ping commands in TUX3: ping 172.16.110.254; ping 172.16.111.1 and ping 172.16.111.254. If we have no issues at this point we should switch to TUX2 and disable ICMP redirect acceptance with

these commands: echo 0 > /proc/sys/net/ipv4/conf/eth0/accept_redirects and echo 0 > /proc/sys/net/ipv4/conf/all/accept_redirects. Now we need to remove the route through TUX4 in TUX2 using the route del -net 172.16.110.0 gw 172.16.111.253 netmask 255.255.255.0 command. We then run ping 172.16.120.1 on TUX2, followed by traceroute -n 172.16.110.1. Then we need to route add -net 172.16.110.0/24 gw 172.16.111.254; traceroute -n 172.16.110.1 and ping 172.16.110.1 to verify everything is working as intended. To finalize we just need to run sudo ip route del 172.16.110.0 via 172.16.111.254 and route add -net 172.16.110.0/24 gw 172.16.111.253. We can now enable redirects again by running these two commands: echo 1 > /proc/sys/net/ipv4/conf/eth0/accept_redirects and echo 1 > /proc/sys/net/ipv4/conf/all/accept_redirects. To disable and enable NAT temporarily we can use these commands on GKTerm respectively: /ip firewall nat disable 0 and /ip firewall nat enable 0. All of this is documented in the appendices, from image IX to XIII.

[See “Picture VIII - Exp 4 Traceroute 1” at the Appendix II]

[See “Picture IX - Exp 4 Traceroute 2” at the Appendix II]

[See “Picture XI - Exp 4 Traceroute 3” at the Appendix II]

[See “Picture XII - Exp 4 Traceroute 4” at the Appendix II]

[See “Picture XIII - Exp 4 Wireshark” at the Appendix II]

By analyzing the traceroutes taken, we came to the conclusions these conclusions:

The Router successfully forwarded the packets between the subnets and the NAT connections worked for public-facing IPs.

With the NAT enabled, it pings to the FTP successfully and the packets were correctly translated. However, without the NAT, the packets failed.

Disabling redirects made the router the gateway, with a correct forwarding of packets once again.

Experiment 5: DNS

In this experiment we want to configure the DNS service on client machines, use the DNS to resolve hostnames to IP addresses and analyze the DNS packets exchanged between the client and the server. Our DNS server (netlab.fe.up.pt) has the IP 10.227.20.3 and our TUXs are already configured to query it. In each TUX we need to run the following two commands: nameserver 10.227.20.3 and nameserver 172.16.1.1. To test if it works as intended we can ping www.google.com (check image XIV on the appendices). We can now start capturing the exchange of packets using Wireshark to observe the data.

[See “Picture XIV - Exp 5 Pinging Google” at the Appendix II]

As we can see in “Picture XIV - Exp 5 Pinging Google”, the experiment was a success and we got the Source and Destination of each DNS correctly: We were able to get to Google’s IP Address and send packets, as well as receive them.

Experiment 6: TCP Connections

For the final experiment our goals are to analyze the phases of a TCP connection, observe TCP mechanisms and understand the impact of concurrent TCP connections on throughput. TUX3 acts as the FTP client during this experiment, downloading a file from ftp.netlab.fe.up.pt, while TUX2 will start a download midway through the ongoing transfer on TUX3. To start the download process on TUX3, we simply need to compile and run our application, using ./download ftp://ftp.netlab.fe.up.pt/pub/{filename} (or make {filename} on our application). We can now start the capture on Wireshark to see the statistics.

[See “Picture XV - Exp 6 Transfering Ubuntu” at the Appendix II]

When we start a download on TUX2 midway through a download on TUX3, we can see that the bandwidth is split evenly between the two connections.

[See “Picture XVI - Exp 6 Multi-transfering Ubuntu” at the Appendix II]

Conclusions

This lab helped us understand the process of creation of network applications, as well as the issues related to the configuration of networks. Part 1 reestablished the basic concepts of application layer protocols by implementing an FTP client. Part 2 emphasized network layer settings and tools like Wireshark that are needed for monitoring the network.

References

- RFC959: File Transfer Protocol.
- RFC1738: Uniform Resource Locators.
- Lab Guide (uploaded document).

Appendix I - Source Code

```c
#include <stdio.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

// Constants
#define FTP_PORT 21
#define MAX_LENGTH 512

// ANSI color codes for colored output
#define RESET "\033[0m"
#define RED "\033[31m"
#define GREEN "\033[32m"
#define YELLOW "\033[33m"
#define BLUE "\033[34m"
#define MAGENTA "\033[35m"
#define CYAN "\033[36m"
#define WHITE "\033[37m"

// URL format: ftp://[<user>:<password>@]<host>/<url-path>
typedef struct URL {
    char user[MAX_LENGTH];      // Username for authentication
    char password[MAX_LENGTH];  // Password for authentication
    char host[MAX_LENGTH];      // Hostname or IP address of the
FTP server
    char resource[MAX_LENGTH];  // Path to the resource on the
server
    char file[MAX_LENGTH];      // Name of the file to download
    char ip[MAX_LENGTH];        // IP address of the server
} URL;


// Centralized error codes
typedef enum {
```

```
    ERR_SUCCESS = 0,
    ERR_SOCKET_FAIL,
    ERR_AUTH_FAIL,
    ERR_PASV_FAIL,
    ERR_RETR_FAIL,
    ERR_FILE_SAVE_FAIL,
    ERR_PARSE_FAIL,
} ErrorCode;


// Error handling and messages
void handle_error(ErrorCode code, const char *msg) {
    const char *error_messages[] = {
        "Success",
        "Socket creation or connection failed",
        "Authentication failed",
        "Failed to enter passive mode",
        "Failed to retrieve file",
        "Failed to save file",
        "URL parsing failed"
    };

    fprintf(stderr, RED "Error [%d]: %s\n" RESET, code,
error_messages[code]);
    if (msg) {
        fprintf(stderr, "Details: " YELLOW "%s\n" RESET, msg);
    }

    exit(code);
}


// Function to print parsed URL details
void print_url_info(const URL *url) {
    printf("\n");
    printf(CYAN "URL Details:\n" RESET);
    printf("User: " GREEN "%s\n" RESET, url->user);
    printf("Password: " GREEN "%s\n" RESET, url->password);
    printf("Host: " YELLOW "%s\n" RESET, url->host);
    printf("Resource Path: " BLUE "%s\n" RESET, url->resource);
    printf("File Name: " MAGENTA "%s\n" RESET, url->file);
    printf("Server IP: " WHITE "%s\n" RESET, url->ip);
    printf("\n");
```

```
}


// Parse the URL into its components
int parse_url(const char *input, URL *url) {
    memset(url->user, 0, MAX_LENGTH);
    memset(url->password, 0, MAX_LENGTH);

    if (strncmp(input, "ftp://", 6) != 0) {
        handle_error(ERR_PARSE_FAIL, "Invalid URL format.");
    }

    const char *url_start = input + 6;
    const char *at_sign = strchr(url_start, '@');
    const char *slash = strchr(url_start, '/');

    if (!slash) {
        handle_error(ERR_PARSE_FAIL, "Invalid URL format.");
    }
    if (at_sign && at_sign < slash) {
        const char *colon = strchr(url_start, ':');
        if (colon && colon < at_sign) {
            strncpy(url->user, url_start, colon - url_start);
            strncpy(url->password, colon + 1, at_sign - colon -
1);
        } else {
            strncpy(url->user, url_start, at_sign - url_start);
        }
        url_start = at_sign + 1;
    } else {
        strcpy(url->user, "anonymous");
        strcpy(url->password, "anonymous");
    }

    strncpy(url->host, url_start, slash - url_start);
    strcpy(url->resource, slash + 1);
    const char *last_slash = strrchr(url->resource, '/');
    if (last_slash) {
        strcpy(url->file, last_slash + 1);
    } else {
        strcpy(url->file, url->resource);
    }
```

```
        struct hostent *host_info = gethostbyname(url->host);
        if (!host_info) {
            handle_error(ERR_PARSE_FAIL, "Failed to resolve
hostname.");
        }
        strcpy(url->ip, inet_ntoa(*(struct in_addr
*)host_info->h_addr));

        return 0;
    }


    // Create a socket and connect to the serve
    int create_socket(char *ip, int port) {
        int sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd < 0) return ERR_SOCKET_FAIL;

        struct sockaddr_in server_addr;
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(port);
        inet_pton(AF_INET, ip, &server_addr.sin_addr);

        if (connect(sockfd, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
            close(sockfd);
            return ERR_SOCKET_FAIL;
        }
        return sockfd;
    }


    // Read the server response
    int read_response(int sockfd, char *buffer, size_t buffer_size) {
        memset(buffer, 0, buffer_size);
        char temp[MAX_LENGTH];
        int total_bytes = 0;
        int is_multiline = 0;
        char code[4] = {0};

        while (1) {
            int bytes_received = recv(sockfd, temp, sizeof(temp) - 1,
0);
            if (bytes_received < 0) {
```

```
                perror("recv");
                handle_error(ERR_SOCKET_FAIL, "Failed to read server
response.");
            }
            temp[bytes_received] = '\0';
            strncat(buffer, temp, buffer_size - strlen(buffer) - 1);
            total_bytes += bytes_received;
            if (total_bytes == bytes_received) {
                strncpy(code, buffer, 3);
                code[3] = '\0';
                if (buffer[3] == '-') {
                    is_multiline = 1;
                }
            }

            char *last_line = strrchr(buffer, '\n');
            if (last_line) {
                if (!is_multiline && strncmp(last_line + 1, code, 3)
== 0) {
                    break;
                } else if (is_multiline && strncmp(last_line + 1,
code, 3) == 0 && last_line[4] == ' ') {
                    break;
                }
            }
        }
        return total_bytes;
    }


    // Authenticate the user with the server
    int authenticate(int sockfd, char *user, char *password) {
        char buffer[MAX_LENGTH];

        while (1) {
            int bytes_received = recv(sockfd, buffer, sizeof(buffer) -
1, 0);
            if (bytes_received < 0) {
                perror("recv welcome message");
                return ERR_AUTH_FAIL;
            }
            buffer[bytes_received] = '\0';
            printf(CYAN "Welcome Message:\n" RESET "%s\n", buffer);
```

```
            if (strstr(buffer, "220 ") != NULL) {
                break;
            }
        }

        sprintf(buffer, "USER %s\r\n", user);
        if (send(sockfd, buffer, strlen(buffer), 0) < 0) {
            perror("send USER");
            return ERR_AUTH_FAIL;
        }

        int bytes_received = recv(sockfd, buffer, sizeof(buffer) - 1,
0);
        if (bytes_received < 0) {
            perror("recv USER response");
            return ERR_AUTH_FAIL;
        }
        buffer[bytes_received] = '\0';
        printf(CYAN "USER Response:\n" RESET "%s\n", buffer);

        if (strstr(buffer, "331") == NULL) {
            fprintf(stderr, "Unexpected USER response: %s\n", buffer);
            return ERR_AUTH_FAIL;
        }

        sprintf(buffer, "PASS %s\r\n", password);
        if (send(sockfd, buffer, strlen(buffer), 0) < 0) {
            perror("send PASS");
            return ERR_AUTH_FAIL;
        }

        bytes_received = recv(sockfd, buffer, sizeof(buffer) - 1, 0);
        if (bytes_received < 0) {
            perror("recv PASS response");
            return ERR_AUTH_FAIL;
        }
        buffer[bytes_received] = '\0';
        printf(CYAN "PASS Response:\n" RESET "%s\n", buffer);

        // LOGIN SUCCESSFUL
        if (strstr(buffer, "230") == NULL) {
            fprintf(stderr, "Authentication failed: %s\n", buffer);
```

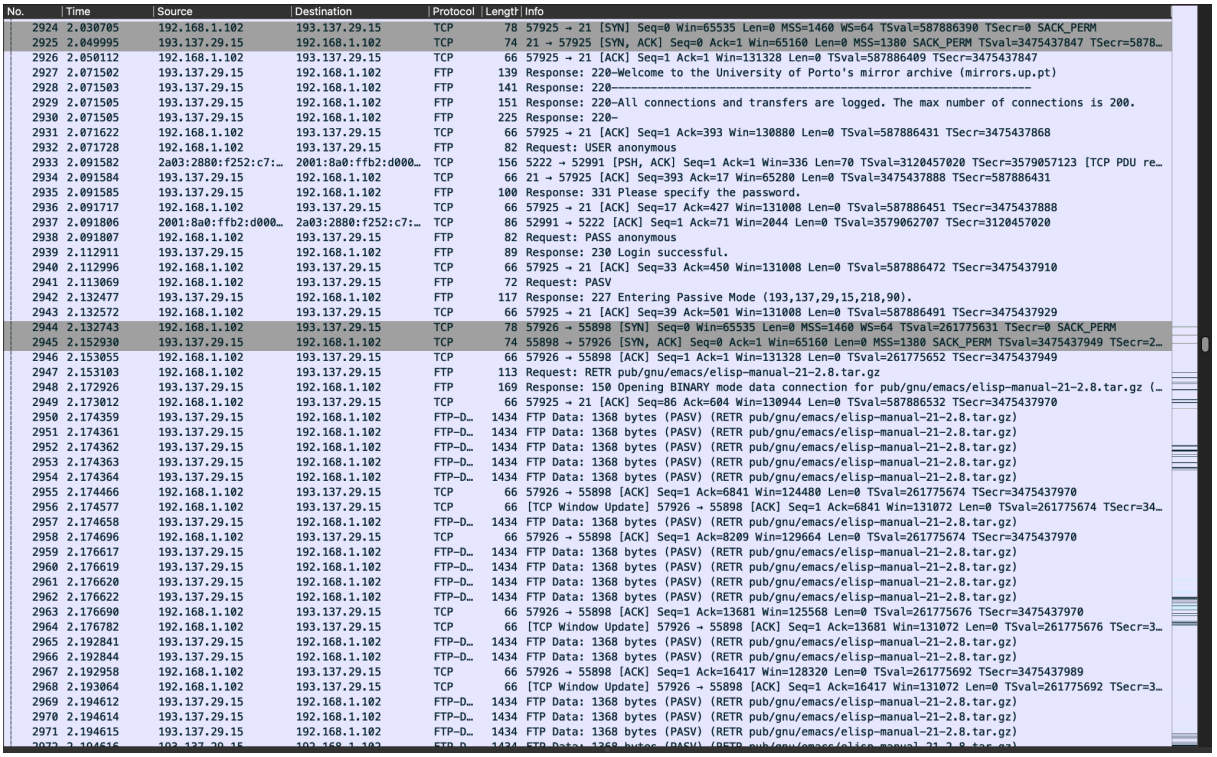
```
            return ERR_AUTH_FAIL;
        }

        return ERR_SUCCESS;
    }


    // Enter passive mode and get the IP and port for data transfer
    int enter_passive_mode(int sockfd, char *ip, int *port) {
        char buffer[MAX_LENGTH];

        if (send(sockfd, "PASV\r\n", strlen("PASV\r\n"), 0) < 0) {
            perror("send PASV");
            return ERR_PASV_FAIL;
        }

        while (1) {
            int bytes_received = recv(sockfd, buffer, sizeof(buffer) -
1, 0);
            if (bytes_received < 0) {
                perror("recv PASV response");
                return ERR_PASV_FAIL;
            }
            buffer[bytes_received] = '\0';
            printf(CYAN "PASV Response:\n" RESET "%s\n", buffer);

            if (strstr(buffer, "227") != NULL) {
                int h1, h2, h3, h4, p1, p2;
                if (sscanf(buffer, "227 Entering Passive Mode
(%d,%d,%d,%d,%d,%d)", &h1, &h2, &h3, &h4, &p1, &p2) == 6) {
                    sprintf(ip, "%d.%d.%d.%d", h1, h2, h3, h4);
                    *port = (p1 * 256) + p2;
                    printf(CYAN "Passive Mode Port:\n" RESET "%d\n",
(p1 * 256) + p2);
                    printf("\n");
                    return ERR_SUCCESS;
                } else {
                    fprintf(stderr, "Failed to parse PASV response:
%s\n", buffer);
                    return ERR_PASV_FAIL;
                }
            }
        }
```

```
    }


    // Request the file from the server
    int request_file(int sockfd, char *resource) {
        char buffer[MAX_LENGTH];
        sprintf(buffer, "RETR %s\r\n", resource);
        send(sockfd, buffer, strlen(buffer), 0);
        recv(sockfd, buffer, sizeof(buffer), 0);
        printf(CYAN "Server response:\n" RESET "%s\n", buffer);
        return strstr(buffer, "150") || strstr(buffer, "125") ?
ERR_SUCCESS : ERR_RETR_FAIL;
    }


    // Download the file from the server
    int download_file(int sockfd, char *file) {
        FILE *fp = fopen(file, "wb");
        if (!fp) return ERR_FILE_SAVE_FAIL;

        char buffer[MAX_LENGTH];
        ssize_t bytes_received;
        long total_bytes = 0;

        // Spinny
        char spinner[] = "|/-\\";
        int spinner_index = 0;

        while ((bytes_received = recv(sockfd, buffer, sizeof(buffer),
0)) > 0) {
            fwrite(buffer, 1, bytes_received, fp);
            total_bytes += bytes_received;

            printf("\r[" YELLOW "%c" RESET "] %ld bytes downloaded",
spinner[spinner_index], total_bytes);
            fflush(stdout);

            spinner_index = (spinner_index + 1) % 4;
        }
        fclose(fp);

        // CHECK
```

```
            printf("\r[" GREEN "✔" RESET "] %ld bytes downloaded\n",
total_bytes);
            printf("\n");

            return ERR_SUCCESS;
        }


        // Main function to download a file from an FTP server
        int main(int argc, char *argv[]) {
            if (argc != 2) {
                handle_error(ERR_PARSE_FAIL, "Usage: ./download
ftp://[<user>:<password>@]<host>/<url-path>");
            }

            struct URL url;
            if (parse_url(argv[1], &url) != 0) {
                handle_error(ERR_PARSE_FAIL, "Failed to parse URL.");
            }

            print_url_info(&url);

            int control_socket = create_socket(url.ip, FTP_PORT);
            if (control_socket < 0) {
                handle_error(ERR_SOCKET_FAIL, "Failed to establish control
connection.");
            }

            if (authenticate(control_socket, url.user, url.password) !=
ERR_SUCCESS) {
                close(control_socket);
                handle_error(ERR_AUTH_FAIL, "Authentication failed.");
            }

            char pasv_ip[MAX_LENGTH];
            int pasv_port;
            if (enter_passive_mode(control_socket, pasv_ip, &pasv_port) !=
ERR_SUCCESS) {
                close(control_socket);
                handle_error(ERR_PASV_FAIL, "Failed to enter passive
mode.");
            }
```

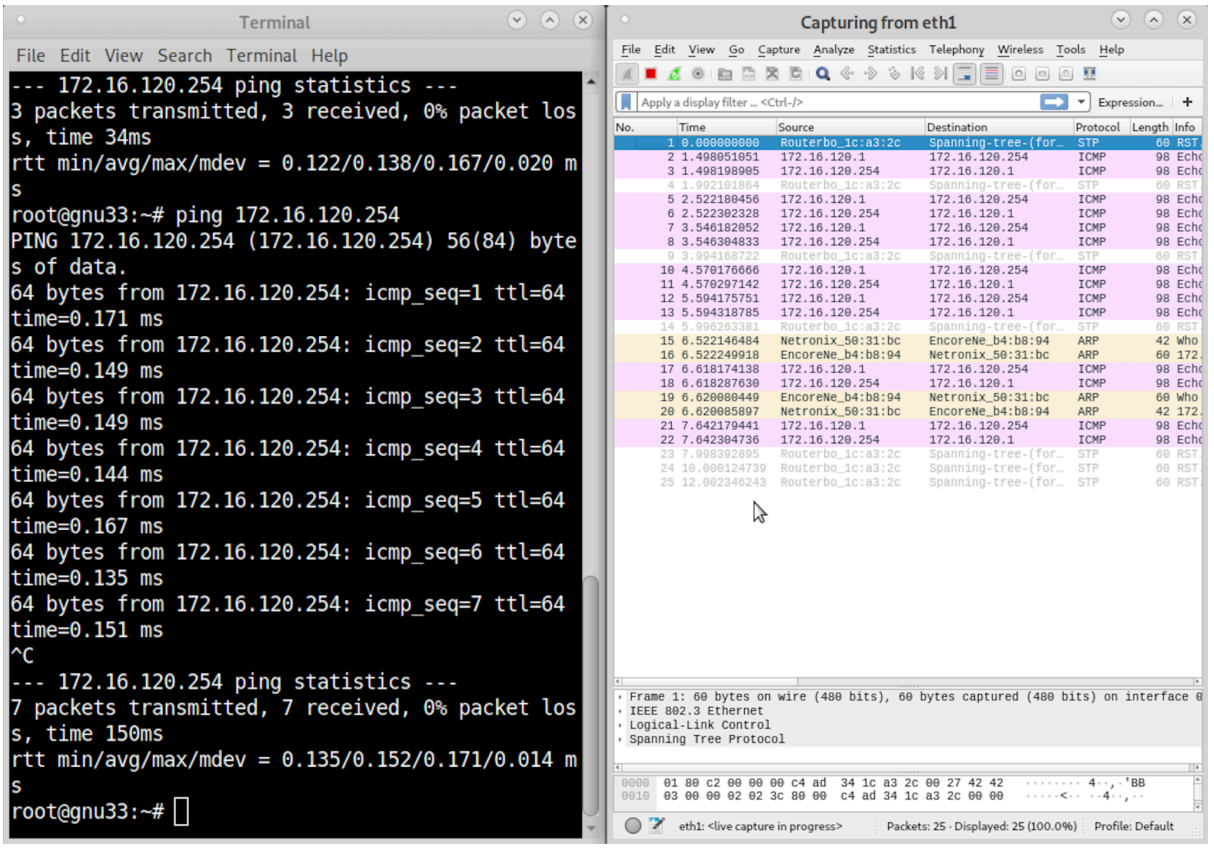
```
        int data_socket = create_socket(pasv_ip, pasv_port);
        if (data_socket < 0) {
            close(control_socket);
            handle_error(ERR_SOCKET_FAIL, "Failed to establish data
connection.");
        }

        if (request_file(control_socket, url.resource) != ERR_SUCCESS)
{
            close(control_socket);
            close(data_socket);
            handle_error(ERR_RETR_FAIL, "Failed to request file.");
        }

        if (download_file(data_socket, url.file) != ERR_SUCCESS) {
            close(control_socket);
            close(data_socket);
            handle_error(ERR_FILE_SAVE_FAIL, "Failed to save file.");
        }

        send(control_socket, "QUIT\r\n", 6, 0);
        char buffer[MAX_LENGTH];
        recv(control_socket, buffer, sizeof(buffer), 0);
        printf(CYAN "Server response:\n" RESET "%s\n", buffer);

        close(data_socket);
        close(control_socket);

        printf(GREEN "File '%s' downloaded successfully.\n" RESET ,
url.file);
        return ERR_SUCCESS;
    }
```

Appendix II - Pictures



No.	Time	Source	Destination	Protocol	Length	Info
2924	2.030705	192.168.1.102	193.137.29.15	TCP	78	57925 → 21 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64 TSval=587886390 TSecr=0 SACK_PERM
2925	2.049995	193.137.29.15	192.168.1.102	TCP	74	21 → 57925 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1380 SACK_PERM TSval=3475437847 TSecr=5878…
2926	2.050112	192.168.1.102	193.137.29.15	TCP	66	57925 → 21 [ACK] Seq=1 Ack=1 Win=131328 Len=0 TSval=587886409 TSecr=3475437847
2927	2.071502	193.137.29.15	192.168.1.102	FTP	139	Response: 220-Welcome to the University of Porto's mirror archive (mirrors.up.pt)
2928	2.071503	193.137.29.15	192.168.1.102	FTP	141	Response: 220--
2929	2.071505	193.137.29.15	192.168.1.102	FTP	151	Response: 220-All connections and transfers are logged. The max number of connections is 200.
2930	2.071505	193.137.29.15	192.168.1.102	FTP	225	Response: 220-
2931	2.071622	192.168.1.102	193.137.29.15	TCP	66	57925 → 21 [ACK] Seq=1 Ack=393 Win=130880 Len=0 TSval=587886431 TSecr=3475437868
2932	2.071728	192.168.1.102	193.137.29.15	FTP	82	Request: USER anonymous
2933	2.091582	2a03:2880:f252:c7:…	2001:8a0:ffb2:d000…	TCP	156	5222 → 52991 [PSH, ACK] Seq=1 Ack=1 Win=336 Len=70 TSval=3120457020 TSecr=3579057123 [TCP PDU re…
2934	2.091584	193.137.29.15	192.168.1.102	TCP	66	21 → 57925 [ACK] Seq=393 Ack=17 Win=65280 Len=0 TSval=3475437888 TSecr=587886431
2935	2.091585	193.137.29.15	192.168.1.102	FTP	100	Response: 331 Please specify the password.
2936	2.091717	192.168.1.102	193.137.29.15	TCP	66	57925 → 21 [ACK] Seq=17 Ack=427 Win=131008 Len=0 TSval=587886451 TSecr=3475437888
2937	2.091806	2001:8a0:ffb2:d000…	2a03:2880:f252:c7:…	TCP	86	52991 → 5222 [ACK] Seq=1 Ack=71 Win=2044 Len=0 TSval=3579062707 TSecr=3120457020
2938	2.091807	192.168.1.102	193.137.29.15	FTP	82	Request: PASS anonymous
2939	2.112911	193.137.29.15	192.168.1.102	FTP	89	Response: 230 Login successful.
2940	2.112996	192.168.1.102	193.137.29.15	TCP	66	57925 → 21 [ACK] Seq=33 Ack=450 Win=131008 Len=0 TSval=587886472 TSecr=3475437910
2941	2.113069	192.168.1.102	193.137.29.15	FTP	72	Request: PASV
2942	2.132477	193.137.29.15	192.168.1.102	FTP	117	Response: 227 Entering Passive Mode (193,137,29,15,218,90).
2943	2.132572	192.168.1.102	193.137.29.15	TCP	66	57925 → 21 [ACK] Seq=39 Ack=501 Win=131008 Len=0 TSval=587886491 TSecr=3475437929
2944	2.132743	192.168.1.102	193.137.29.15	TCP	78	57926 → 55898 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64 TSval=261775631 TSecr=0 SACK_PERM
2945	2.152930	193.137.29.15	192.168.1.102	TCP	74	55898 → 57926 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1380 SACK_PERM TSval=3475437949 TSecr=2…
2946	2.153055	192.168.1.102	193.137.29.15	TCP	66	57926 → 55898 [ACK] Seq=1 Ack=1 Win=131328 Len=0 TSval=261775652 TSecr=3475437949
2947	2.153103	192.168.1.102	193.137.29.15	FTP	113	Request: RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz
2948	2.172926	193.137.29.15	192.168.1.102	FTP	169	Response: 150 Opening BINARY mode data connection for pub/gnu/emacs/elisp-manual-21-2.8.tar.gz (…
2949	2.173012	192.168.1.102	193.137.29.15	TCP	66	57925 → 21 [ACK] Seq=86 Ack=604 Win=130944 Len=0 TSval=587886532 TSecr=3475437970
2950	2.174359	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2951	2.174361	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2952	2.174362	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2953	2.174363	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2954	2.174364	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2955	2.174466	192.168.1.102	193.137.29.15	TCP	66	57926 → 55898 [ACK] Seq=1 Ack=6841 Win=124480 Len=0 TSval=261775674 TSecr=3475437970
2956	2.174577	192.168.1.102	193.137.29.15	TCP	66	[TCP Window Update] 57926 → 55898 [ACK] Seq=1 Ack=6841 Win=131072 Len=0 TSval=261775674 TSecr=34…
2957	2.174658	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2958	2.174696	192.168.1.102	193.137.29.15	TCP	66	57926 → 55898 [ACK] Seq=1 Ack=8209 Win=129664 Len=0 TSval=261775674 TSecr=3475437970
2959	2.176617	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2960	2.176619	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2961	2.176620	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2962	2.176622	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2963	2.176690	192.168.1.102	193.137.29.15	TCP	66	57926 → 55898 [ACK] Seq=1 Ack=13681 Win=125568 Len=0 TSval=261775676 TSecr=3475437970
2964	2.176782	192.168.1.102	193.137.29.15	TCP	66	[TCP Window Update] 57926 → 55898 [ACK] Seq=1 Ack=13681 Win=131072 Len=0 TSval=261775676 TSecr=3…
2965	2.192841	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2966	2.192844	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2967	2.192958	192.168.1.102	193.137.29.15	TCP	66	57926 → 55898 [ACK] Seq=1 Ack=16417 Win=128320 Len=0 TSval=261775692 TSecr=3475437989
2968	2.193064	192.168.1.102	193.137.29.15	TCP	66	[TCP Window Update] 57926 → 55898 [ACK] Seq=1 Ack=16417 Win=131072 Len=0 TSval=261775692 TSecr=3…
2969	2.194612	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2970	2.194614	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)
2971	2.194615	193.137.29.15	192.168.1.102	FTP-D…	1434	FTP Data: 1368 bytes (PASV) (RETR pub/gnu/emacs/elisp-manual-21-2.8.tar.gz)

Picture I - Download Wireshark



```
Terminal
File  Edit  View  Search  Terminal  Help
--- 172.16.120.254 ping statistics ---
3 packets transmitted, 3 received, 0% packet los
s, time 34ms
rtt min/avg/max/mdev = 0.122/0.138/0.167/0.020 m
s
root@gnu33:~# ping 172.16.120.254
PING 172.16.120.254 (172.16.120.254) 56(84) byte
s of data.
64 bytes from 172.16.120.254: icmp_seq=1 ttl=64
time=0.171 ms
64 bytes from 172.16.120.254: icmp_seq=2 ttl=64
time=0.149 ms
64 bytes from 172.16.120.254: icmp_seq=3 ttl=64
time=0.149 ms
64 bytes from 172.16.120.254: icmp_seq=4 ttl=64
time=0.144 ms
64 bytes from 172.16.120.254: icmp_seq=5 ttl=64
time=0.167 ms
64 bytes from 172.16.120.254: icmp_seq=6 ttl=64
time=0.135 ms
64 bytes from 172.16.120.254: icmp_seq=7 ttl=64
time=0.151 ms
^C
--- 172.16.120.254 ping statistics ---
7 packets transmitted, 7 received, 0% packet los
s, time 150ms
rtt min/avg/max/mdev = 0.135/0.152/0.171/0.014 m
s
root@gnu33:~#
```

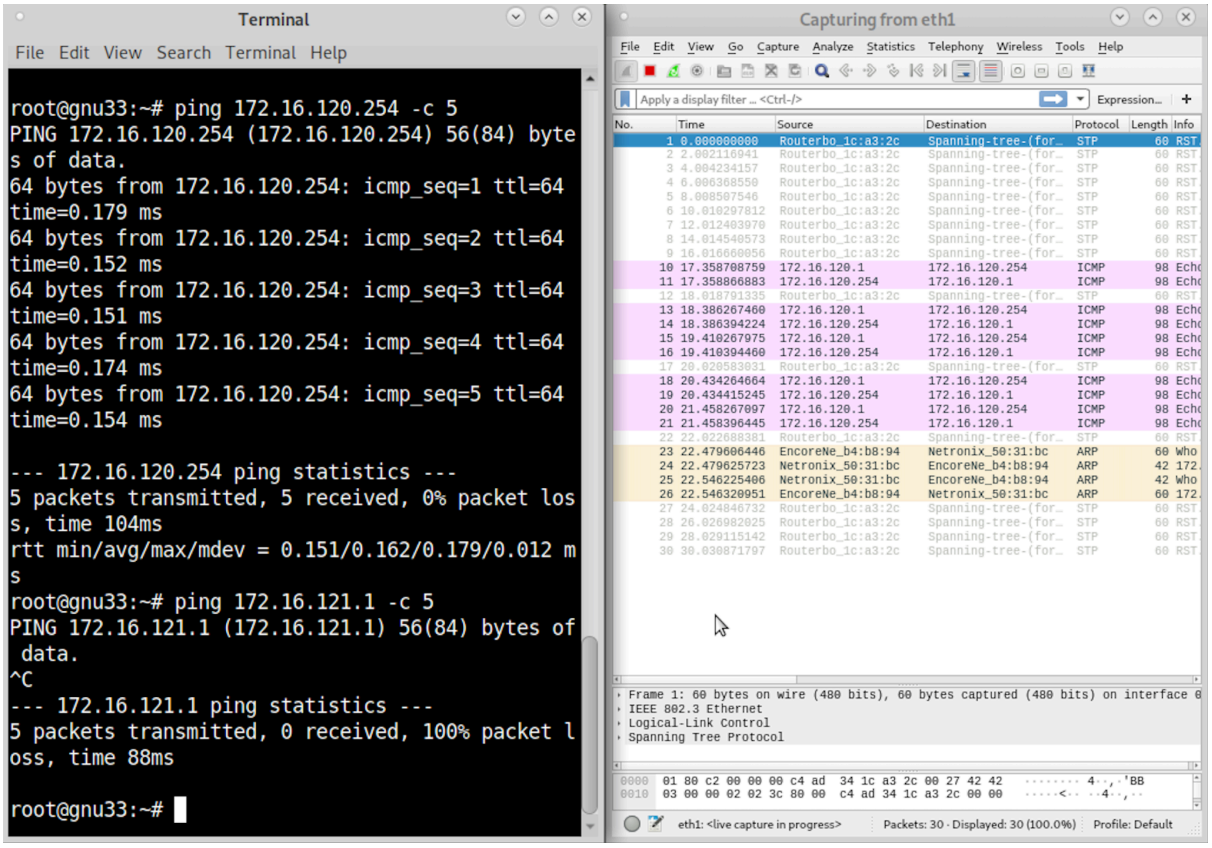
```
Capturing from eth1
File  Edit  View  Go  Capture  Analyze  Statistics  Telephony  Wireless  Tools  Help
Apply a display filter … <Ctrl-/>                                   Expression…  +
```

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	Routerbo_1c:a3:2c	Spanning-tree-(for…	STP	60	RST
2	1.498051051	172.16.120.1	172.16.120.254	ICMP	98	Echo
3	1.498198905	172.16.120.254	172.16.120.1	ICMP	98	Echo
4	1.992181864	Routerbo_1c:a3:2c	Spanning-tree-(for…	STP	60	RST
5	2.522180456	172.16.120.1	172.16.120.254	ICMP	98	Echo
6	2.522302328	172.16.120.254	172.16.120.1	ICMP	98	Echo
7	3.546182052	172.16.120.1	172.16.120.254	ICMP	98	Echo
8	3.546304833	172.16.120.254	172.16.120.1	ICMP	98	Echo
9	3.984168722	Routerbo_1c:a3:2c	Spanning-tree-(for…	STP	60	RST
10	4.570176666	172.16.120.1	172.16.120.254	ICMP	98	Echo
11	4.570297142	172.16.120.254	172.16.120.1	ICMP	98	Echo
12	5.594175751	172.16.120.1	172.16.120.254	ICMP	98	Echo
13	5.594318785	172.16.120.254	172.16.120.1	ICMP	98	Echo
14	5.996263381	Routerbo_1c:a3:2c	Spanning-tree-(for…	STP	60	RST
15	6.522146484	Netronix_50:31:bc	EncoreNe_b4:b8:94	ARP	42	Who
16	6.522249918	EncoreNe_b4:b8:94	Netronix_50:31:bc	ARP	60	172.
17	6.618174138	172.16.120.1	172.16.120.254	ICMP	98	Echo
18	6.618287630	172.16.120.254	172.16.120.1	ICMP	98	Echo
19	6.620080449	EncoreNe_b4:b8:94	Netronix_50:31:bc	ARP	60	Who
20	6.620085897	Netronix_50:31:bc	EncoreNe_b4:b8:94	ARP	42	172.
21	7.642179441	172.16.120.1	172.16.120.254	ICMP	98	Echo
22	7.642304736	172.16.120.254	172.16.120.1	ICMP	98	Echo
23	7.998392895	Routerbo_1c:a3:2c	Spanning-tree-(for…	STP	60	RST
24	10.000124739	Routerbo_1c:a3:2c	Spanning-tree-(for…	STP	60	RST
25	12.002346243	Routerbo_1c:a3:2c	Spanning-tree-(for…	STP	60	RST

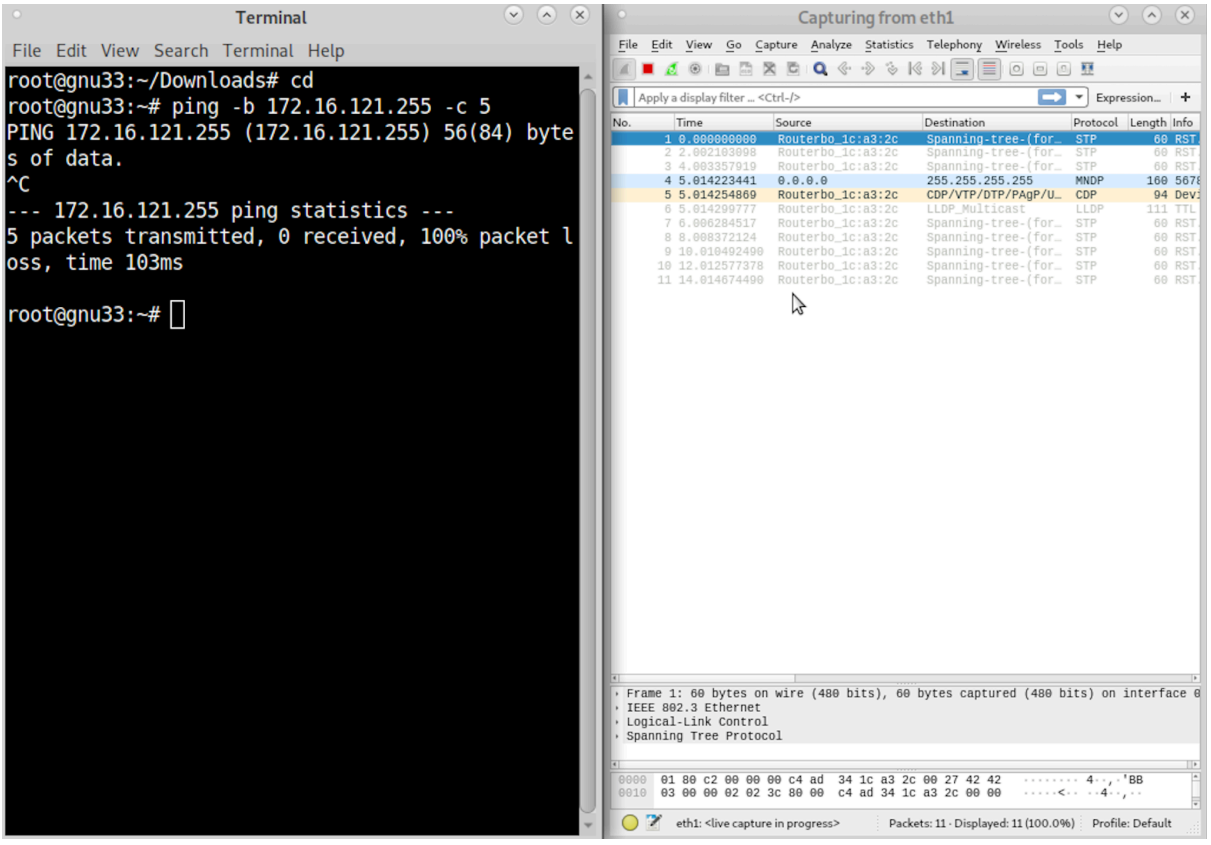
```
▸ Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
▸ IEEE 802.3 Ethernet
▸ Logical-Link Control
▸ Spanning Tree Protocol

0000  01 80 c2 00 00 00 c4 ad  34 1c a3 2c 00 27 42 42   ........ 4..,.'BB
0010  03 00 00 02 02 3c 80 00  c4 ad 34 1c a3 2c 00 00   .....<.. ..4..,..
```

```
eth1: <live capture in progress>   Packets: 25 · Displayed: 25 (100.0%)   Profile: Default
```

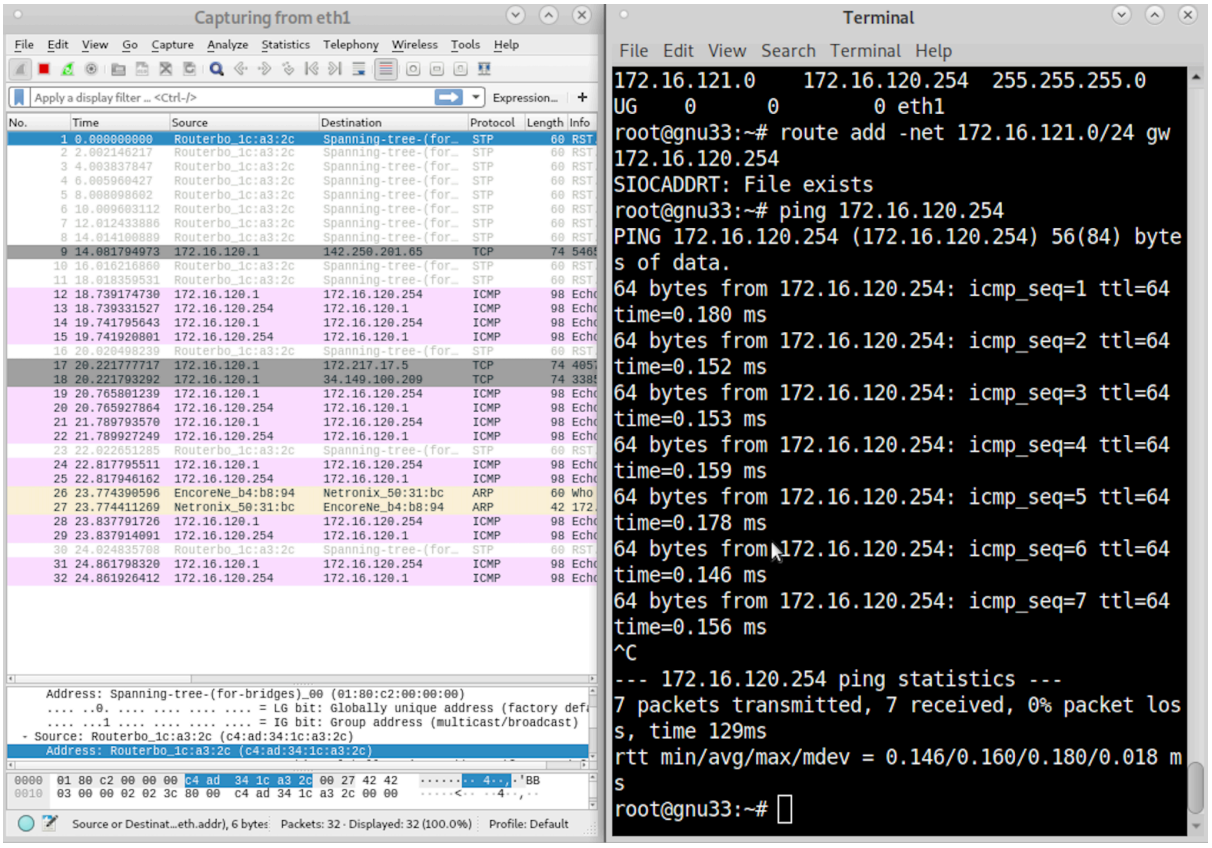
Picture II - Exp 1 Wireshark



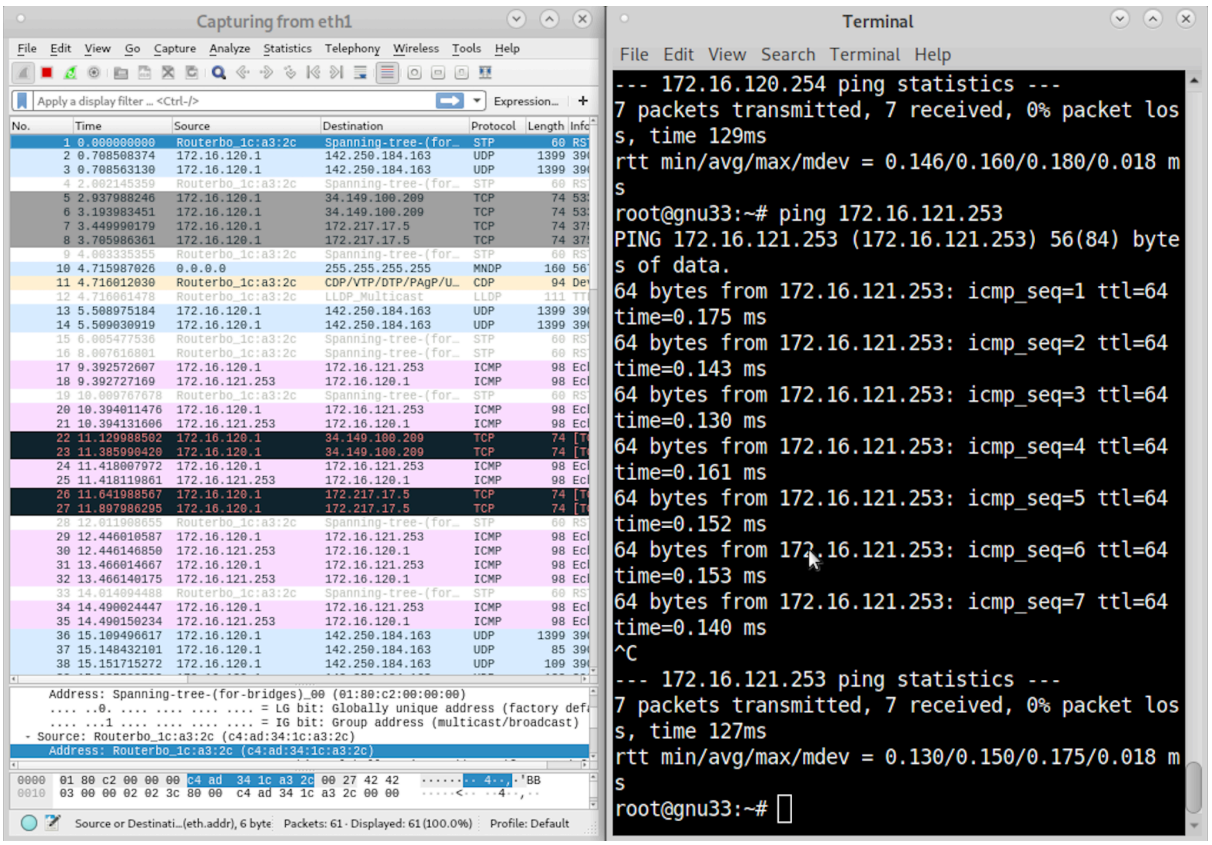
Picture III - Exp 2 Pinging TUX2 and TUX4



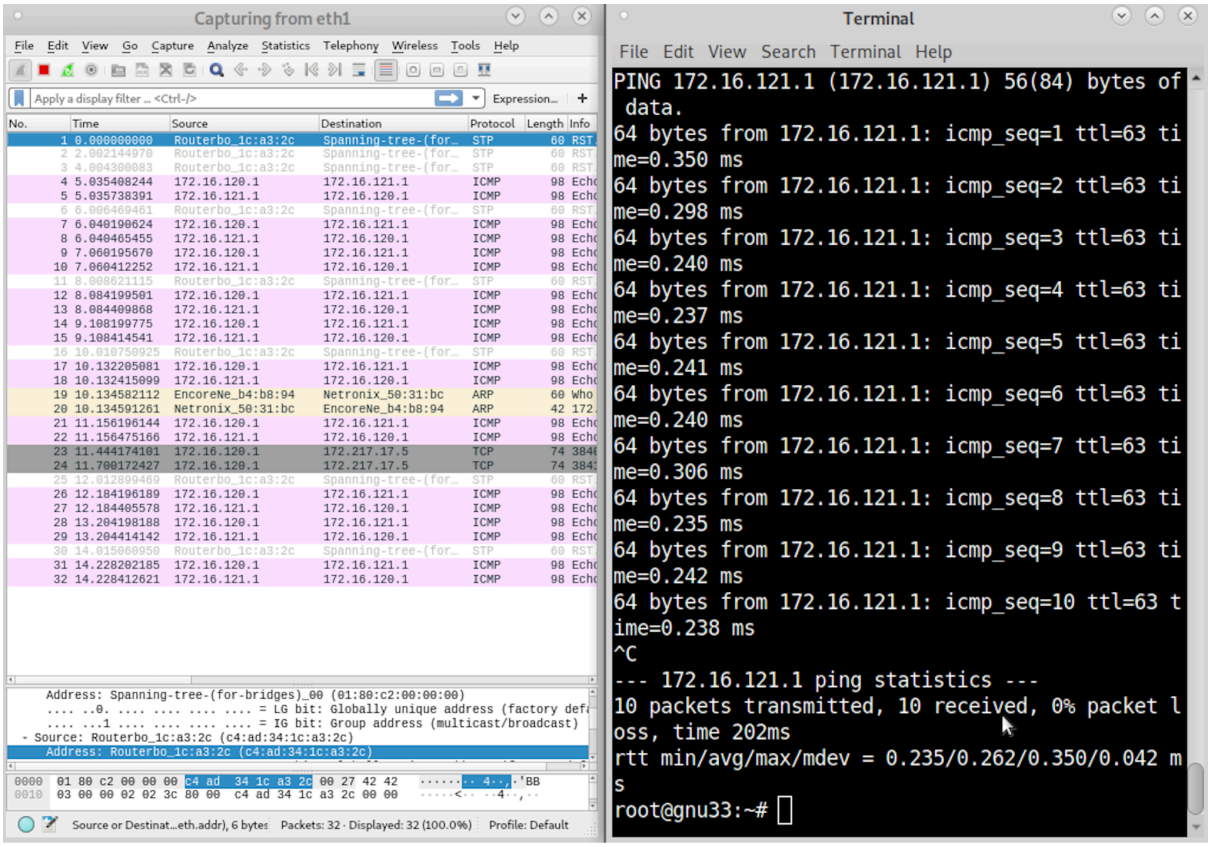
Picture IV - Exp 2 Pinging the Broadcast Address



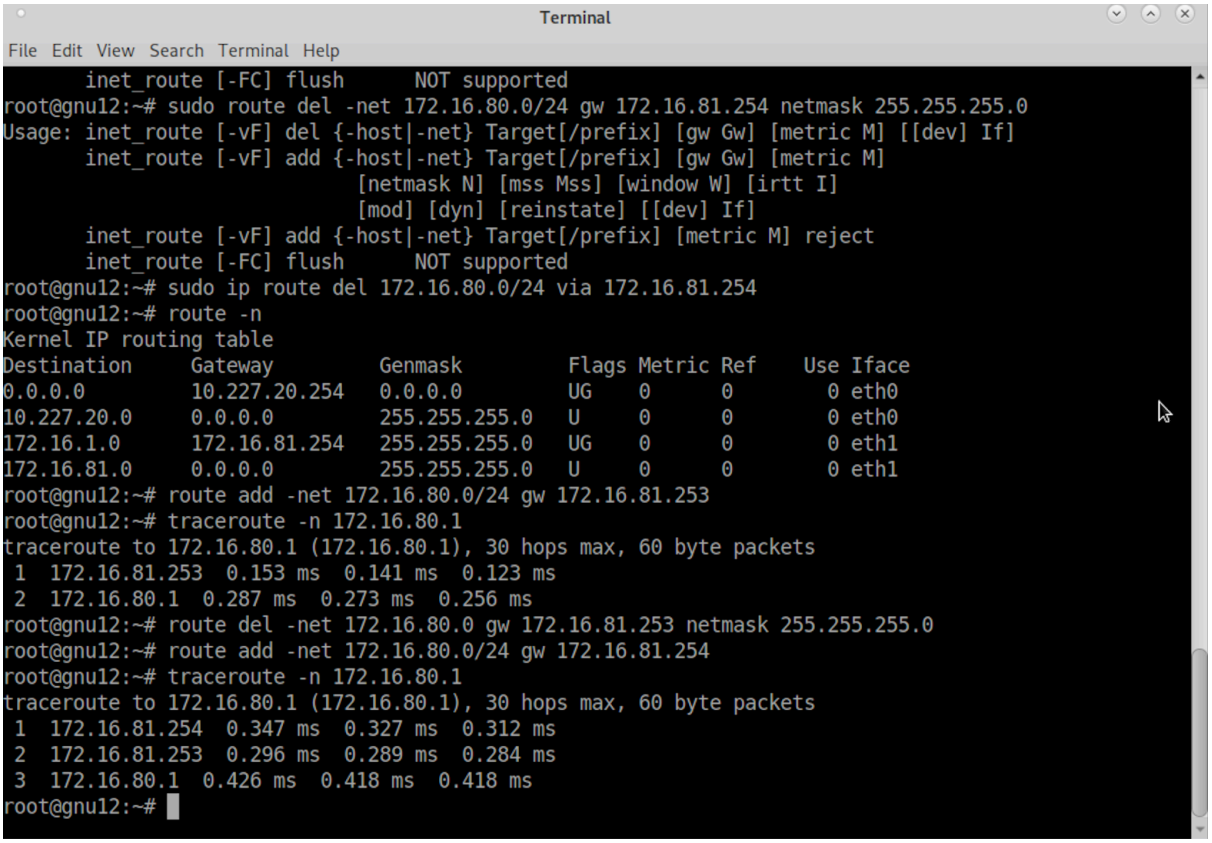
Picture V - Exp 3 Pinging TUX4.0



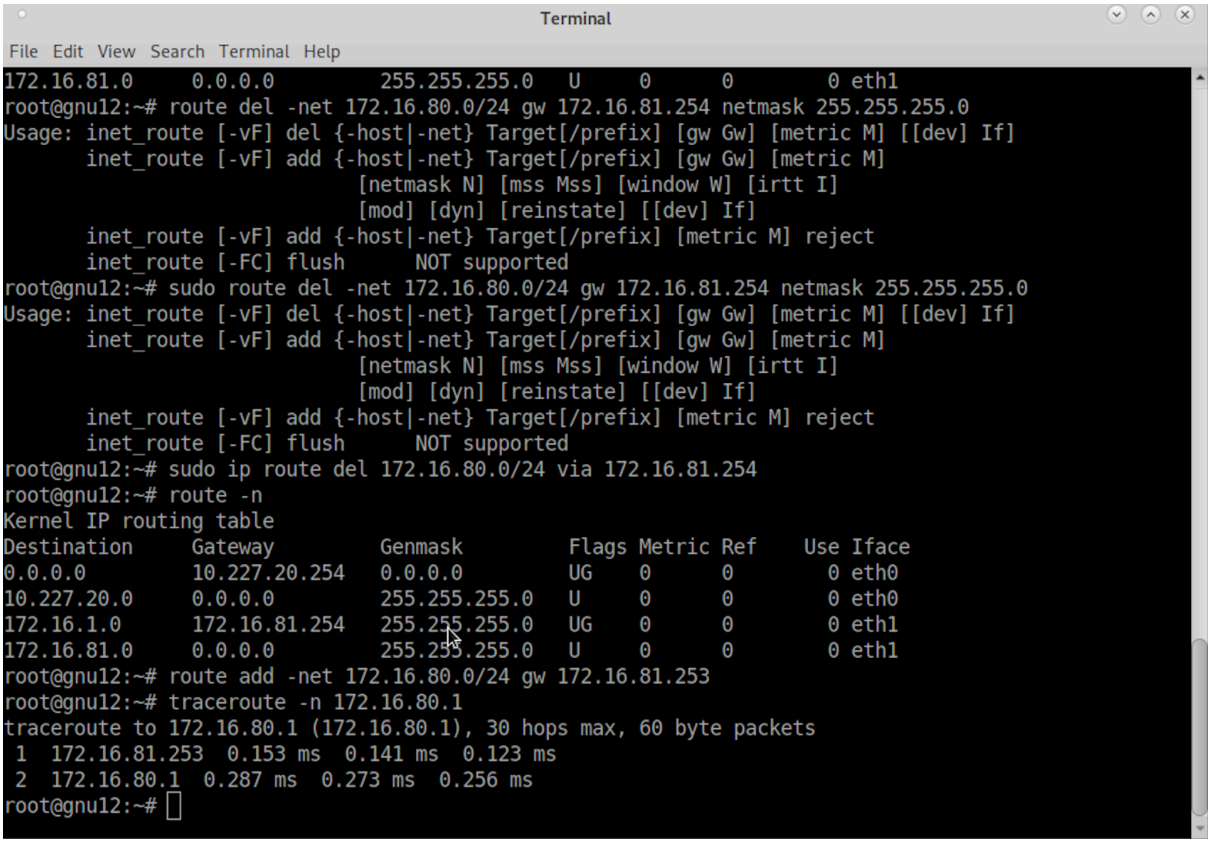
Picture VI - Exp 3 Pinging TUX4.1



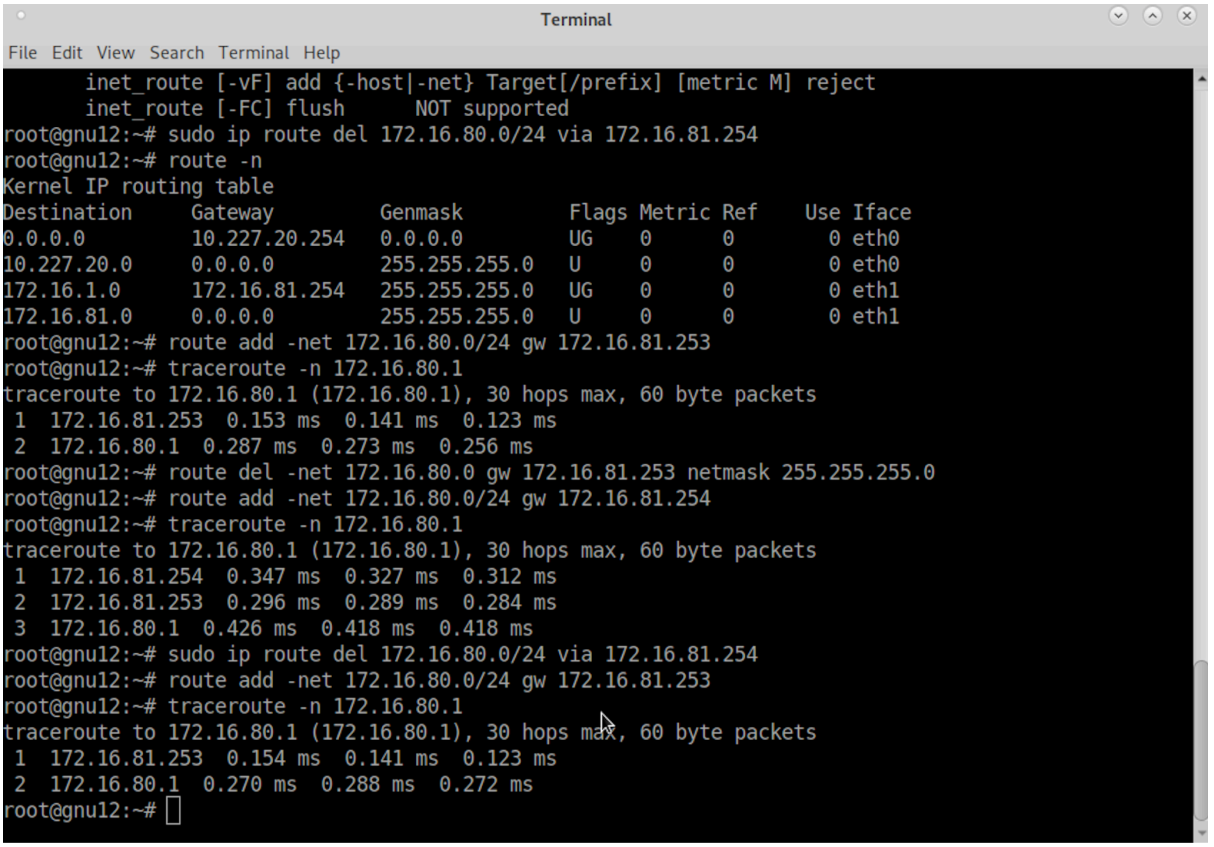
Picture VII - Exp 3 Pinging TUX2



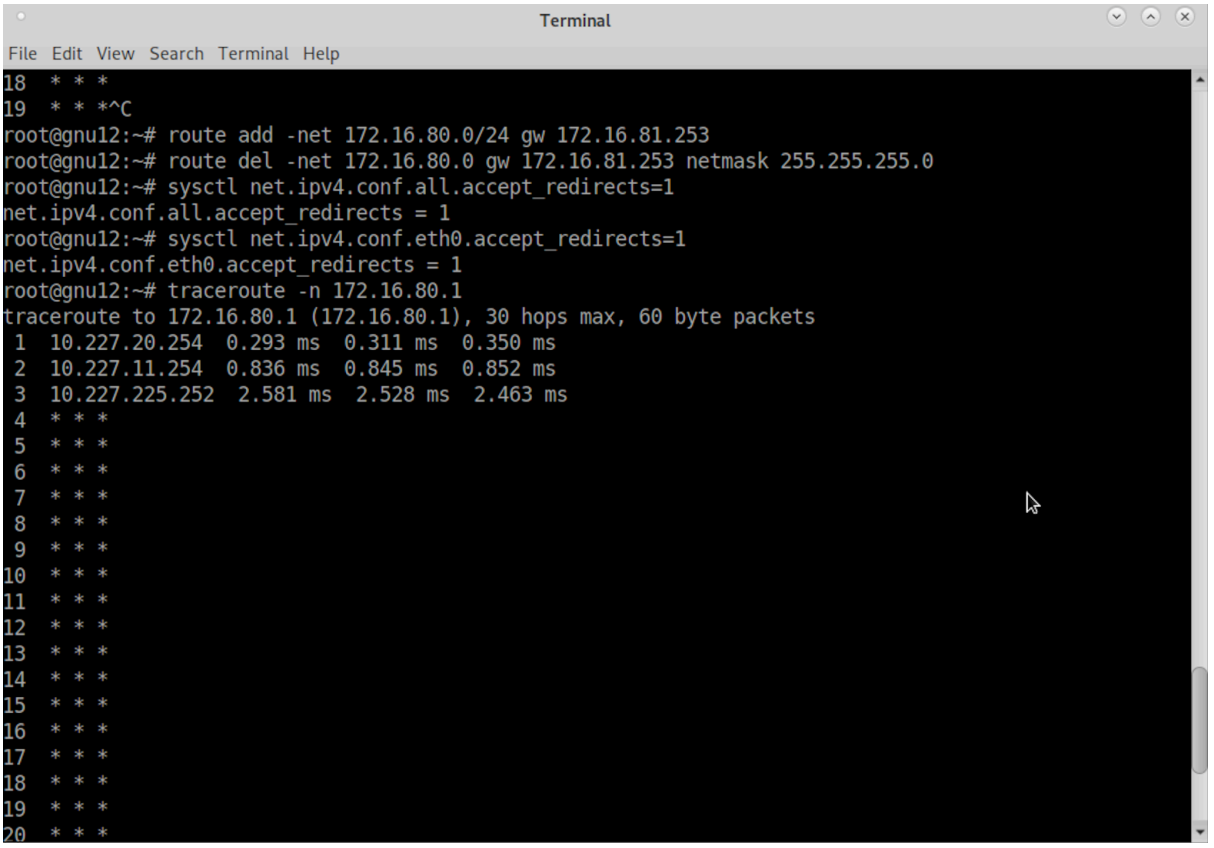
Picture IX - Exp 4 Traceroute 1



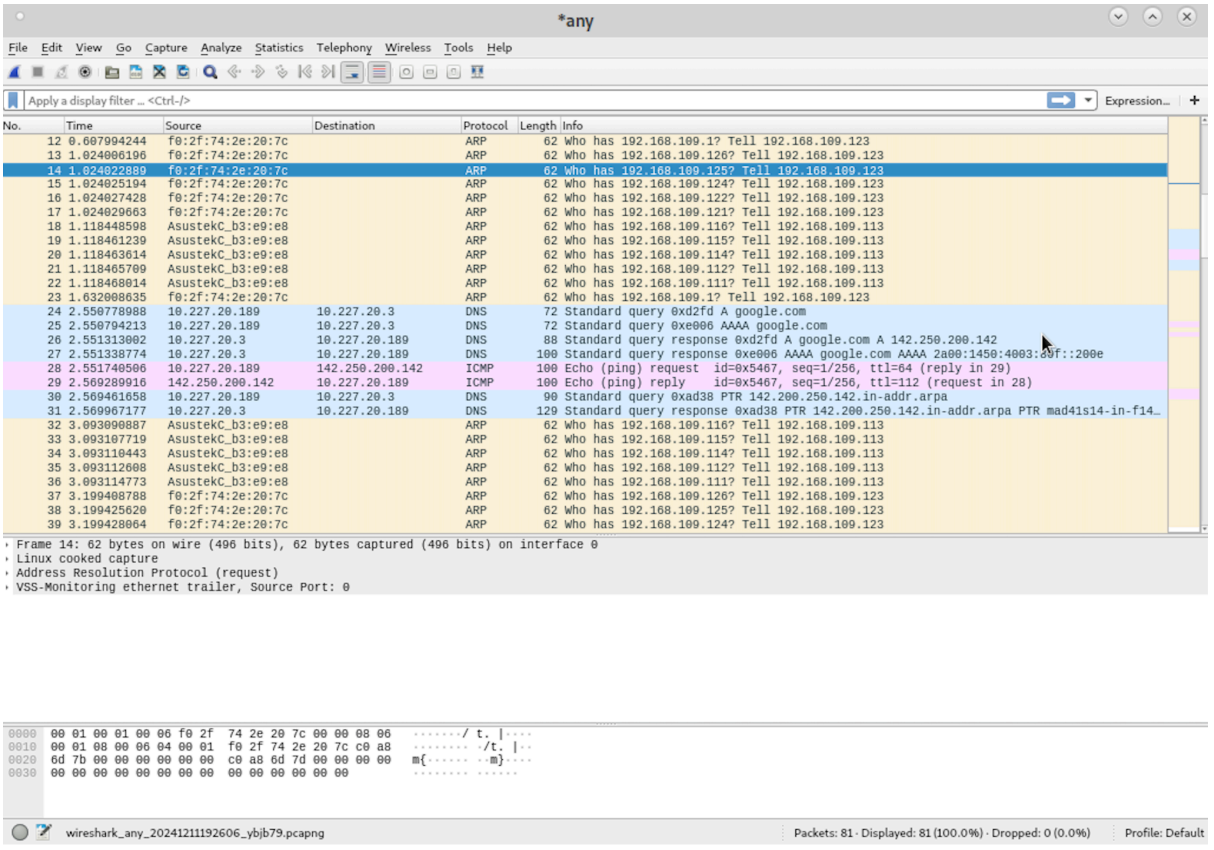
Picture X - Exp 4 Traceroute 2



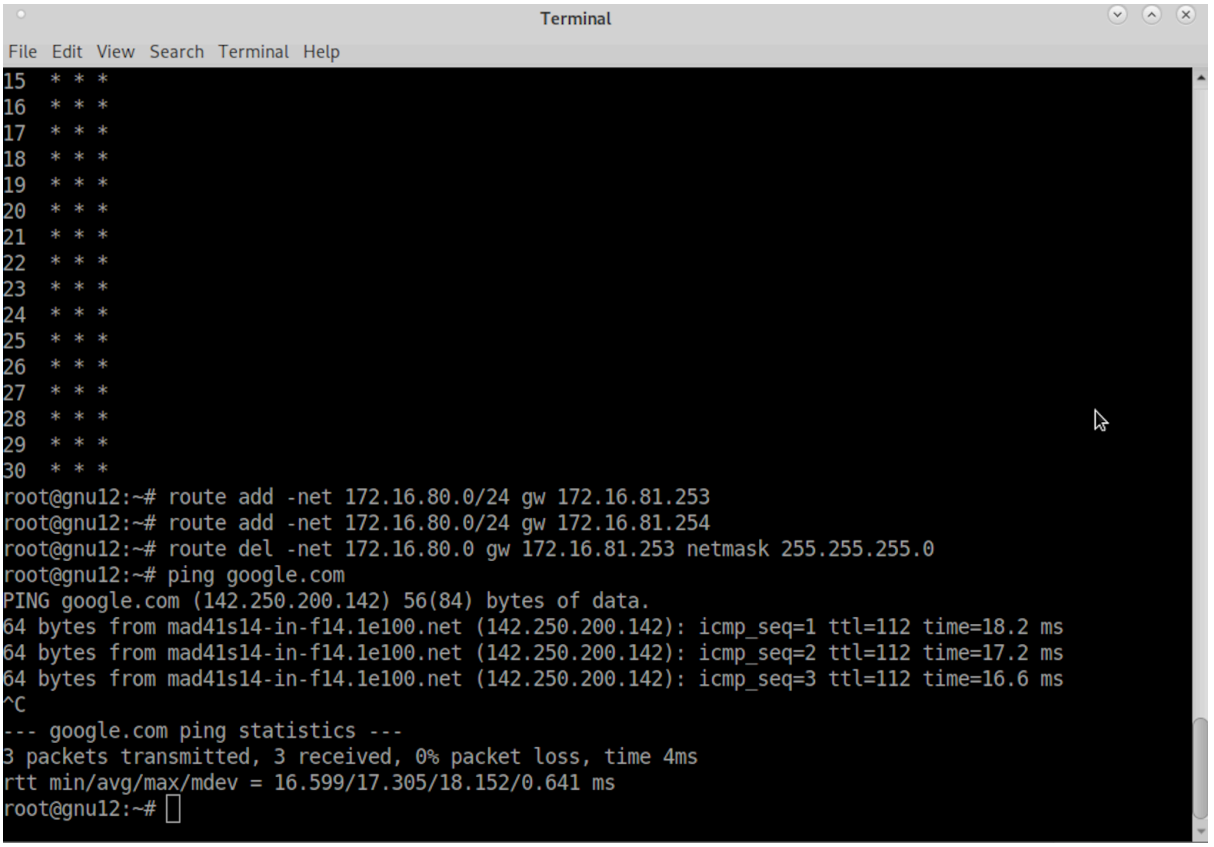
Picture XI - Exp 4 Traceroute 3



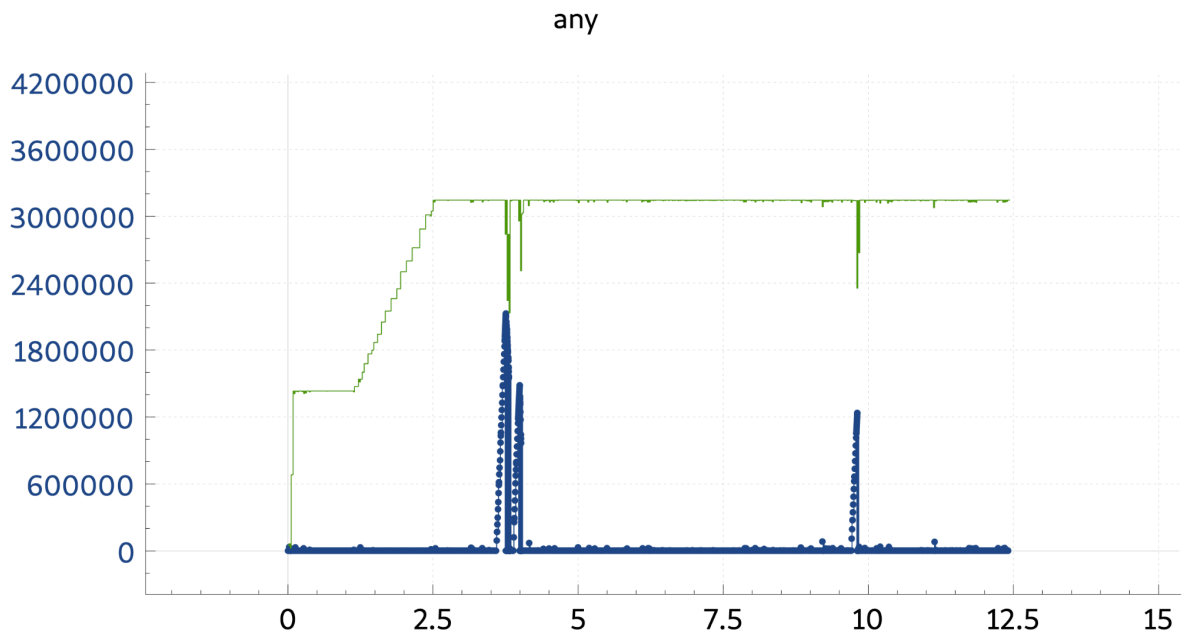
Picture XII - Exp 4 Traceroute 4



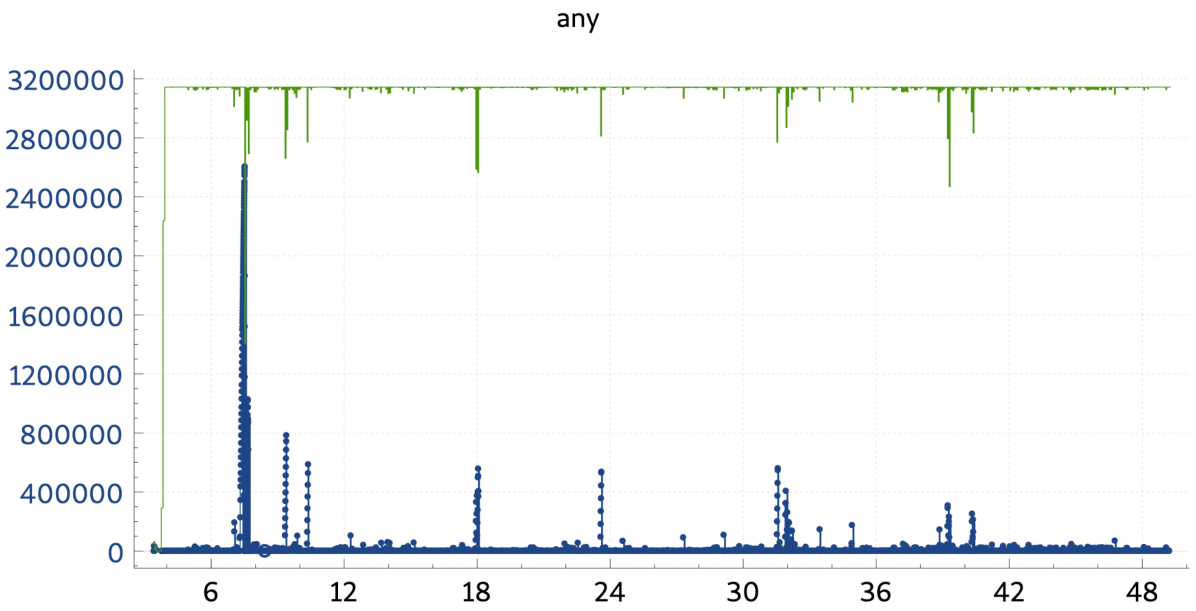
Picture XIII - Exp 4 Wireshark



Picture XIV - Exp 5 Pinging Google



Picture XV - Exp 6 Transfering Ubuntu



Picture XVI - Exp 6 Multi-Transfering Ubuntu

Appendix III - Questions

Experiment 1: Configure an IP Network

1. What are the ARP packets and what are they used for?

ARP (Address Resolution Protocol) packets are used to map an IP address to a MAC address within a local network. A device sends an ARP request to discover the MAC address of another device, and the target device responds with an ARP reply containing its MAC address.

2. What are the MAC and IP addresses of ARP packets and why?

In the ARP Request Packet, the source IP is the IP address of the requesting device, the destination IP is the IP address being queried, the source MAC address is the MAC address of the requesting device and the destination MAC address is the broadcast address (FF:FF:FF:FF:FF:FF).

In the ARP Reply Packet, the source IP is the IP address of the responding device, the destination IP is the IP address of the requesting device, the source MAC address is the MAC address of the responding device and the destination MAC address is the MAC address of the requesting device.

3. What packets does the ping command generate?

The “ping” command generates ICMP (Internet Control Message Protocol) packets: the ICMP Echo Request, sent by the source device to the target device and the ICMP Echo Reply, sent by the target device in response to the Echo Request.

4. What are the MAC and IP addresses of the ping packets?

In the ICMP Echo Request, the source addresses are the addresses of the source device and the destination addresses are the addresses of the target device.

In the ICMP Echo Reply, the source addresses are the addresses of the responding device and the destination addresses are the addresses of the initiating device.

5. How to determine if a receiving Ethernet frame is ARP, IP, or ICMP?

We can inspect the **EtherType field** in the Ethernet frame header: if it's "0x0806" then it is ARP, if it's "0x0800" then it is IPv4, for both IP and ICMP. To distinguish ICMP, we can inspect the **Protocol field** in the IP header, which will be 1 for ICMP.

6. How to determine the length of a receiving frame?

The Ethernet frame length is determined by inspecting the **Total Length field** in the IP header or by viewing the frame size in Wireshark.

7. What is the loopback interface and why is it important?

The loopback interface is a virtual network interface used for self-communication within a device, typically with the IP "127.0.0.1". It is very important because it allows testing of software and network configurations locally and does not require physical network hardware.

Experiment 2: Implement Two Bridges in a Switch

1. How to configure bridgeY0?

To create the bridge we open the Mikrotik console on the Switch and run the command "/interface bridge add name=bridge110", with Y being the number of the station. We then remove ports from the default bridge: "/interface bridge port remove [find interface=ether3]" "/interface bridge port remove [find interface=ether4]" and add ports to "bridge110": "/interface bridge port add bridge=bridge110 interface=ether3" "/interface bridge port add bridge=bridge110 interface=ether4"

2. How many broadcast domains are there? How can you conclude it from the logs?

There are two broadcast domains: one for "bridge110" containing "TUX3" and "TUX4" and one for "bridge111" containing "TUX2". Broadcast packets (e.g., ARP requests) sent from one bridge are not observed on the other, as seen in Wireshark logs.

Experiment 3: Configure a Router in Linux

1. What routes are there in the tuxes? What are their meanings?

Each tux (`TUX2`, `TUX3`, `TUX4`) has routing entries for directly connected subnets and default gateways:

- “172.16.110.0/24”: Routes traffic within the subnet.

- “default”: Directs traffic to other subnets via the router.

2. What information does an entry of the forwarding table contain?

The “Destination Network” is the target network/subnet, the “Gateway” is the next hop for the destination. The “Interface” is the local network interface for forwarding, the “Metric” is the priority of the route if multiple routes exist and “Flags” indicate status (e.g., `U` for up, `G` for gateway).

3. What ARP messages, and associated MAC addresses, are observed and why?

The ARP Request is sent to discover the MAC address for a specific IP and the Reply responds with the MAC address for the queried IP.

The source MAC, from the querying device and the destination MAC is Broadcast (for request) or specific MAC (for reply).

4. What ICMP packets are observed and why?

The ICMP Echo Request, sent by a source to check connectivity and theICMP Echo Reply, sent back by the target to confirm receipt. They are observed when testing connectivity using `ping`.

5. What are the IP and MAC addresses associated with ICMP packets and why?

For the Echo Request, the sources of the initiating device and the destinations of the target. For the Echo Reply, the sources of the responding device and the destinations of the initiator.

This is because the ICMP uses IP for addressing and Ethernet for frame delivery.

Experiment 4: Configure a Commercial Router and Implement NAT

1. How to configure a static route in a commercial router?

Use the router's console to add a route. We used this command ”/ip route add dst-address=172.16.110.0/24 gateway=172.16.111.253”

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

With ICMP Redirect Enabled, if a router detects a more efficient route, it sends an ICMP redirect message to the source device, updating its routing table. Packets follow the updated path as directed by the ICMP redirect.

Without ICMP Redirect, the devices rely strictly on their static routing tables. Packets follow the predefined paths, even if a more efficient route exists.

3. How to configure NAT in a commercial router?

We used the following commands on the router console: “/ip firewall nat enable 0” to enable the NAT and “/ip firewall nat disable 0” to disable it.

4. What does NAT do?

NAT (Network Address Translation) allows multiple devices in a private network to access external networks (e.g., the internet) using a single public IP address. It translates private IP addresses (e.g., 172.16.111.1) to the router’s public IP address for outbound traffic and maps responses back to the originating device.

5. What happens when tuxY3 pings the FTP server with NAT disabled? Why?

The ping fails because the private IP (172.16.110.1) is not routable on the internet. Without NAT, the private IP address is sent directly, and upstream routers discard the packet as they do not know how to route private IPs.

Experiment 5: DNS

1. How to configure the DNS service in a host?

Edit the `/etc/resolv.conf` file and add the DNS server addresses. For example. We used: nameserver 10.227.20.3 \n nameserver 172.16.1.1

2. What packets are exchanged by DNS, and what information is transported?

In the case of the DNS Query, they are sent by the client to the DNS server, requesting the IP address associated with a hostname, contains the hostname being queried. In the case of the DNS Response, they are sent by the DNS server, providing the requested IP address or an error message if the hostname cannot be resolved, contains the resolved IP address, TTL (time-to-live), or an error code (e.g., NXDOMAIN for nonexistent domain).

Experiment 6: TCP Connections

1. How many TCP connections are opened by your FTP application?

Two TCP connections are established: The “Control Connection”, used for sending commands like `USER`, `PASS`, `PASV`, and `RETR` and the “Data Connection” established during passive mode (`PASV`) for transferring the requested file.

2. In what connection is the FTP control information transported?

The “Control Connection” transports FTP commands and server responses.

3. What are the phases of a TCP connection?

The three phases are “Connection Establishment”, which uses a 3-way handshake (`SYN`, `SYN-ACK`, `ACK`) to establish a reliable connection; “Data Transfer”, which sends and receives data packets with acknowledgment (ACK) and the “Connection Termination”, that uses a 4-way handshake (`FIN`, `ACK`, `FIN`, `ACK`) to close the connection.

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

TCP uses the Automatic Repeat reQuest (ARQ) mechanism to ensure reliable data transmission. Lost or corrupted packets are retransmitted based on acknowledgment (ACK) or timeout.

The Relevant Fields are three: “Sequence Number”, tracking the order of data packets; “Acknowledgement Number”, confirming receipt of packets and “Window Size” managing flow control by limiting the number of unacknowledged packets.

The Logs give us the retransmissions observed when ACKs are delayed or missing.

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

TCP adjusts the congestion window size dynamically based on network conditions, using algorithms like slow start, congestion avoidance, and fast recovery.

The Relevant Fields are the “Congestion Window” (implicit in the sender's behavior) and Retransmissions indicate congestion or packet loss.

The Throughput initially increases during the slow start phase, stabilizes during congestion avoidance, and decreases during packet loss or congestion.

6. Is the throughput of a TCP data connection disturbed by the appearance of a second TCP connection? How?

Yes, the throughput of the first connection decreases when a second connection starts. This happens because both connections share the same network resources, leading to competition for bandwidth. TCP's fairness mechanism ensures equal distribution of available bandwidth among connections so they are 50/50.