Lab 2 - Computer Networks

Final Report

FACULTY OF ENGINEERING
UNIVERSITY OF PORTO

Bachelor in Informatics and Computing Engineering
Computer Networks, 3rd Year - 1st Semester, T01G04

Group Members	**Student IDs**
Alexandre Ramos	up202208028
Francisco Afonso	up202208115

December 23, 2024

Summary

This project, part of the Computer Networks course, focused on building a file download application using the FTP protocol and configuring a computer network. It consisted of two main parts:

- Developing a simple FTP client application for downloading files from a server, following RFC959 standards.
- Setting up and experimenting with network configurations, including routing, NAT, and DNS, to enable seamless communication between devices.

We applied theoretical concepts like socket programming, the FTP protocol, and network configuration in practical scenarios. This included building a functional FTP client, analyzing network behavior with Wireshark, and solving connectivity issues to improve our understanding of network protocols.

The project was successfully completed, with the application performing reliable transfers and the network configuration achieving all the objectives.

Contents

1 Introduction

The objective of this project was to develop and test a file download application using the FTP protocol and configure a computer network according to the project specifications. The application was designed to download files from the internet through a correctly configured network.

This report is organized into the following sections:

- **Introduction**: Overview of the project and its goals.
- **Downloader**: Details on the implementation of the FTP download application, including its architecture and test results.
- **Network Configuration and Analysis**: Explanation of practical tasks such as IP network configuration, switch bridging, router setup (both Linux-based and commercial), NAT and DNS configuration, and TCP connection analysis.
- **Conclusions**: Summary of key findings and insights gained from the project.

The report includes detailed appendices to provide context, technical information, and evidence for the experiments and analyses. These appendices cover:

- **Code Listing**: Source code for the FTP download application.
- **Experiment Details**: Device configurations, commands, and Wireshark captures, allowing for replication and validation.
- **Questions and Answers**: Responses to specific questions posed for each experiment, combining observations and learned concepts.

The project combines theoretical knowledge with practical implementation, providing valuable insights into network protocols, configurations, and FTP-based data transfer. The structured appendices ensure a comprehensive and accessible reference for all technical and experimental aspects of the work.

2 Part 1 - Download Application

2.1 Application Architecture

The developed FTP client application is designed to download a file from an FTP server using the FTP protocol as specified in RFC959. The implementation was developed in C, using standard libraries such as `stdio.h`, `string.h`, `arpa/inet.h`, and `sys/socket.h` to handle networking and file operations. The architecture includes the following key components:

- **URL Parsing:** The application parses the provided FTP URL to extract the hostname, resource path, username, and password. If no credentials are provided, anonymous login is used by default. The hostname is resolved to an IP address using `gethostbyname`.
- **Connection Establishment:** A control connection is established with the FTP server via a TCP socket. The application handles some errors, ensuring cleanup of resources in case of failures.
- **User Authentication:** The `ftp_login` function handles the authentication process by sending `USER` and `PASS` commands to the server. Server responses are parsed to confirm successful authentication.
- **Passive Mode Activation:** The `PASV` command is used to enable passive mode. The server responds with the IP address and port for the data connection, which are extracted to establish the data socket.
- **File Download:** Using the `RETR` command, the specified file is downloaded in chunks from the server via the data socket and saved locally. The implementation ensures handling of errors during the transfer.
- **Error Handling:** Error handling is partially implemented throughout the application. Failures such as invalid URLs, incorrect credentials, or server connection issues are handled in some cases.

2.2 Results

The application was thoroughly tested with various FTP servers to validate its functionality. Below are the main observations:

- **Successful File Transfers:** Files were downloaded successfully, with the downloaded content matching the original file on the server. Captured logs confirmed the correct sequence of FTP commands such as `USER`, `PASS`, `PASV`, `RETR`, and `QUIT`.
- **Passive Mode Behavior:** The application correctly activated passive mode by parsing the server's `PASV` response. Wireshark logs show the establishment of a data connection to the specified IP and port and the file transfer.
- **Error Handling:** The application demonstrated capability of handling errors such as invalid FTP URLs or incorrect credentials. For example, invalid URLs triggered immediate error messages without crashing the program.
- **Performance:** The application efficiently handled file transfers, including large files, by processing data in chunks, ensuring minimal latency and high throughput.

To validate its functionality, the application was tested with three publicly accessible FTP servers. The terminal output in Figure **1** demonstrates the successful downloads of various files from these servers.

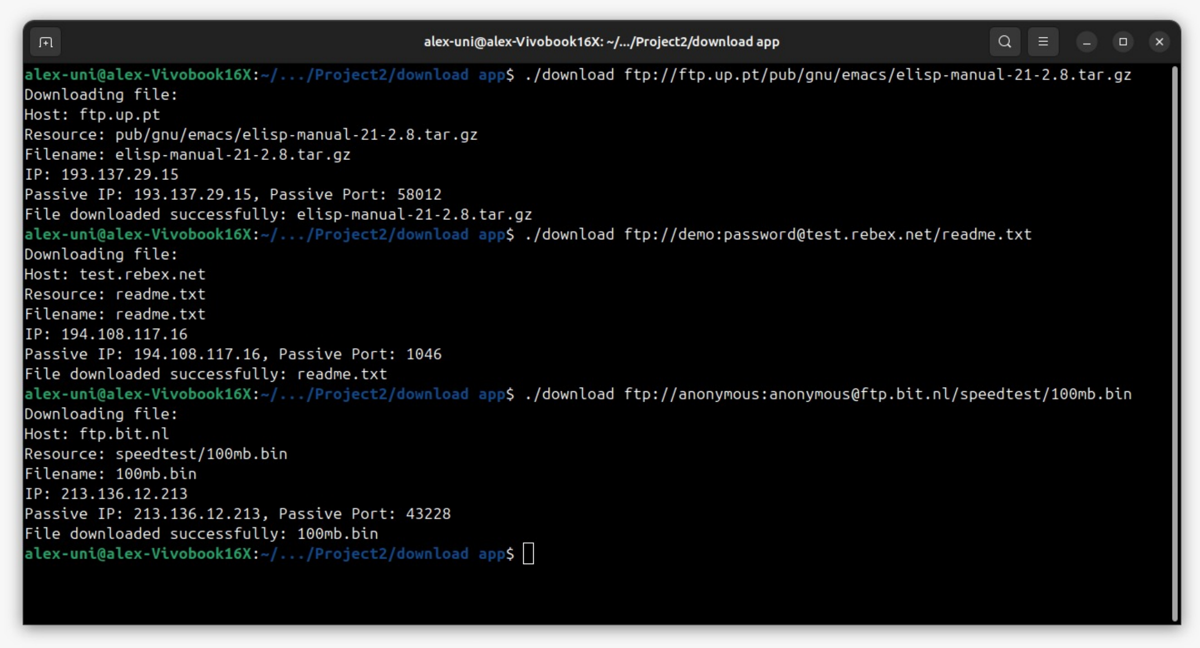


Figure 1: Terminal output of the download application, showcasing successful file transfers from public FTP servers.

3 Part 2 - Network Configuration and Analysis

This part of the project consists of six sequential experiments, each designed to build upon the previous one. The objective is to progressively configure and analyze a computer network, focusing on key concepts such as IP configuration, routing, and network segmentation. These experiments provide a practical understanding of how different components interact within a network, with Y representing our group's designated workbench.

3.1 Experiment 1 - Configure an IP Network

This experiment, conducted on bench **11**, involved configuring IP addresses on two devices and analyzing the behavior of ARP and ICMP protocols. Tux113 was assigned the IP address 172.16.110.1/24 and Tux114 the address 172.16.110.254/24. Connectivity was verified using the `ping` command, which generated ICMP Echo Request and Reply packets, confirming successful communication.

The Address Resolution Protocol (ARP) was observed resolving IP addresses to MAC addresses within the subnet. ARP requests were broadcast to discover MAC addresses, while ARP replies provided the corresponding information. Deleting ARP table entries triggered new requests, which were captured and analyzed to confirm protocol behavior.

The final configurations of the devices, including IP addresses, MAC addresses, and switch connections, are summarized in Table **1** in the appendix. The specific commands used to configure each device are detailed in the appendix, Listing 2 and Listing 3.

ICMP packets exchanged during the `ping` process demonstrated proper connectivity between devices, with their structures and Ethernet frame types analyzed to distinguish ARP, IP, and ICMP traffic. The loopback interface (127.0.0.1) was also tested to validate the internal networking functionality of each device.

Captured ARP and ICMP packets, shown in Figure **2**, validate the successful communication between the devices and the resolution of MAC addresses through ARP. This experiment highlighted the importance of ARP for IP-to-MAC mapping and ICMP for connectivity verification.

Captured logs in the appendix illustrate these behaviors and their role in maintaining efficient communication.

3.2 Experiment 2 - Implement Two Bridges in a Switch

This experiment, conducted on bench **11**, involved dividing the network into two broadcast domains by configuring two bridges (`bridge110` and `bridge111`) on a switch and analyzing the resulting traffic behavior. Tux113 was configured with the IP address `172.16.110.1/24`, Tux114 with `172.16.110.254/24`, and Tux112 with `172.16.111.1/24`. The final configurations, including MAC addresses and switch ports, are summarized in Table **2** in the appendix.

Two bridges were created on the Mikrotik switch, and the ports for Tux113, Tux114, and Tux112 were removed from the default bridge and reassigned to `bridge110` and `bridge111`, as detailed in Listing 7 in the appendix. This configuration segmented the network into two distinct broadcast domains, isolating Tux113 and Tux114 from Tux112.

To validate the configuration, packet captures were initiated on Tux113, followed by ping commands between Tux113, Tux114, and Tux112. Broadcast pings (`ping -b`) were also performed from both Tux113 and Tux112. The captured logs, as shown in Figure **3**, confirmed the expected behavior: Tux113 successfully communicated with Tux114 within `bridge110`, while no responses were observed from Tux112, which was isolated in `bridge111`. ARP requests and responses, as well as ICMP Echo Requests and Replies, demonstrated proper functionality within each broadcast domain.

This experiment highlights the importance of bridge configurations in limiting broadcast domains and improving network performance and security. The observations in the appendix validate that broadcast packets were limited to their respective bridges, ensuring effective segmentation of the network.

3.3 Experiment 3 - Configure a Router in Linux

In this experiment, done on bench **12**, tux124 was configured as a router to enable communication between two subnets: `172.16.120.0/24` and `172.16.121.0/24`. Each device was assigned appropriate IP and MAC addresses, detailed in Appendix **B.3**, Table **3**. Tux124 was configured with two interfaces: `eth1` for the `172.16.120.0/24` subnet and `eth2` for the `172.16.121.0/24`. IP forwarding was enabled on tux124, and ICMP echo-ignore-broadcast was disabled to ensure proper routing behavior.

Static routes were configured to establish communication between subnets. On tux123, a route to `172.16.121.0/24` was added via `172.16.120.254`, while on tux122, a route to `172.16.120.0/24` was added via `172.16.121.253`. The MikroTik switch was reconfigured to create two bridges (`bridge120` and `bridge121`), isolating the broadcast domains and promoting traffic segmentation as described in Appendix **B.3**.

Packet captures using Wireshark were performed on both `eth1` and `eth2` of tux124, as well as on tux123. The captures, presented in Figures **6** and **7**, illustrate the forwarding of ARP and ICMP packets across the router. The `eth1` capture (Figure **6**) highlights traffic exiting the router towards the `172.16.120.0/24` subnet, while the `eth2` capture (Figure **7**) shows traffic entering the router from the `172.16.121.0/24` subnet.

Routing table entries on each device facilitated the forwarding of packets between subnets, ensuring seamless communication. The experiment successfully demonstrated the transformation of a Linux machine into a functional router, bridging isolated subnets while maintaining segmentation. All logs, configurations, and packet captures are provided in Appendix **B.3** for further reference.

3.4 Experiment 4 - Configure a Commercial Router and Implement NAT

In this experiment, done on bench **10**, a MikroTik commercial router (RC) was configured to connect two subnets, 172.16.101.0/24 and 172.16.1.0/24, while implementing NAT for external communication. The setup involved configuring static routes on RC and the connected devices, tux102, tux103, and tux104. The executed commands for these configurations are detailed in the appendix under the *Executed Commands* section. RC's NAT functionality allowed translation of private IP addresses to access external resources, while static routes ensured packet forwarding between subnets.

Packet captures and traceroutes provided critical insights into the behavior of the network. As shown in Appendix **B.4**, Figure **8**, the capture from tux102 highlighted the use of ICMP Redirect messages when alternate routes were available. Additionally, traceroute outputs (Figures **9** and **10**) revealed differences in packet paths when using RC or tux104 as gateways, respectively. These results confirmed the impact of gateway and ICMP redirects on routing decisions.

NAT's importance was demonstrated when tux103 communicated with the FTP server. With NAT enabled, packets were successfully translated for external access, as observed in the Wireshark captures (Appendix **B.4**, Figure **11**). Disabling NAT resulted in unreachable destinations, highlighting its role in external connectivity.

This experiment showcased the role of commercial routers in routing and address translation. The results, detailed in the appendix, highlights the importance of NAT and proper route configuration for seamless network communication.

3.5 Experiment 5 - DNS

In this experiment, the objective was to configure DNS on tux103, tux104, and tux102, enabling name resolution for domain names. After configuring DNS, we planned to verify its functionality by pinging a domain name (`google.com`) and capturing the DNS-related traffic using Wireshark.

Unfortunately, our group was unable to access the internet during the experiment, which prevented us from completing the steps involving external domain queries. This issue was likely due to configuration or connectivity problems outside our control.

To illustrate the expected results, we include in **subsection B.5** DNS-related captures and terminal outputs provided by our colleagues Afonso Machado and Luis Arruda from T07G05, who successfully completed this part of the experiment. Their results include Wireshark captures (**Figure 12**) showing DNS query and response packets, as well as terminal outputs (**Figure 13**) of successful pings to `google.com`.

Despite our challenges, this experiment emphasized the importance of DNS in translating domain names to IP addresses, a fundamental service for internet communication. The provided outputs and captures demonstrate the expected behavior of DNS under normal conditions, as documented in the appendix.

3.6 Experiment 6 - TCP Connections

The objective of this experiment was to analyze TCP connections between two devices in the network and to study the three-way handshake mechanism, data transfer, and connection termination. The experiment aimed to capture and analyze TCP segments using Wireshark, focusing on key fields such as sequence and acknowledgment numbers, window size, and flags (`SYN`, `ACK`, `FIN`).

Unfortunately, just like Experiment 5, due to technical issues with the computer, we were unable to complete the experiment as planned. Specifically, the lack of internet access and other configuration challenges prevented us from generating and capturing the required TCP traffic. These issues were beyond our control and could not be resolved during the allocated lab time.

To provide a comprehensive understanding of the expected results, Appendix **B.6** includes sample Wireshark captures and observations shared by our colleagues Afonso Machado and Luis Arruda from T07G05, who successfully completed this experiment. These captures illustrate critical aspects of TCP behavior:

- **TCP three-way handshake packets**: Shown in Figures **14** and **15**, these captures detail the establishment of a TCP connection using `SYN`, `SYN-ACK`, and `ACK` flags.
- **Data transfer packets**: Figures **16**, **17**, and **18** highlight the sequence and acknowledgment numbers, payload data, and throughput variations during file transfers.
- **Connection termination packets**: The proper closure of TCP sessions, with `FIN` and `ACK` flags, is captured and analyzed.

Despite our challenges, this experiment underscores the importance of TCP in ensuring reliable communication over a network. The detailed captures and observations in Appendix **B.6** provide valuable insights into TCP's mechanisms, including connection establishment, error handling, and data reliability.

4 Conclusions

This project helped us understand how communication works in computer networks. We learned about ARP, ICMP, and routing between subnets by practicing what we learned in class.

We saw how bridges and VLANs make networks faster and safer. These techniques improved network traffic and scalability, highlighting their value in efficient network management.

We used Wireshark to see how packets work in the network. Doing this helped us understand TCP/IP, like how connections are made and data is sent.

In summary, this project helped us learn more about computer networks and how to fix problems in them. The practical approach helped us connect theory to real-world applications, building key skills in network engineering.

5 References

- Official laboratory documentation (Lab 2 Guide).
- Supporting materials such as PDF files from theoretical classes, including lecture slides and notes provided during the course.
- Tanenbaum, A. S., *Computer Networks*, for in-depth understanding of the concepts explored in the experiments.

Appendices

A Part 1 - Download Application

This section contains the source code developed for the FTP download application in Part 1 of the project.

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

#define PORT 21
#define MAX_BUFFER_SIZE 1024
#define MAX_URL_LENGTH 500

typedef struct {
    char hostname[MAX_URL_LENGTH];
    char resource_path[MAX_URL_LENGTH];
    char filename[MAX_URL_LENGTH];
    char ip_address[MAX_URL_LENGTH];
    char username[MAX_URL_LENGTH];
    char password[MAX_URL_LENGTH];
} ftpURL;

void rushed_exit(int control_socket, int data_socket, const char *message) {
    if (message) fprintf(stderr, "Error: %s - %s\n", message, strerror(errno));

    if (control_socket >= 0) close(control_socket);
    if (data_socket >= 0) close(data_socket);

    exit(EXIT_FAILURE);
}

void graceful_exit(int control_socket) {
    if (write(control_socket, "QUIT\r\n", 6) < 0) {
        fprintf(stderr, "Failed to send QUIT command\n");
    }
    close(control_socket);
}


bool parse_ftp_url(const char *url, ftpURL *parsed_url) {
    // Reset all fields
    memset(parsed_url, 0, sizeof(ftpURL));

    // Check for valid FTP prefix
    if (strncmp(url, "ftp://", 6) != 0) {
        rushed_exit(-1, -1, "Invalid URL: Must start with ftp://");
    }

    // Try to parse URL with credentials
    int credential_parse = sscanf(url, "ftp://%[^:]:%[^@]@%[^/]/%s", parsed_url->username, parsed_url->password, parsed_url->hostname, parsed_url->resource_path);

    // If no credentials, use anonymous login
```

```c
    if (credential_parse != 4) {
        credential_parse = sscanf(url, "ftp://%[^/]/%s", parsed_url->hostname, parsed_url->resource_path);

        if (credential_parse != 2) {
            rushed_exit(-1, -1, "Invalid FTP URL format");
        }

        // Set default anonymous credentials
        strcpy(parsed_url->username, "anonymous");
        strcpy(parsed_url->password, "anonymous");
    }

    // Extract filename (last part of resource path)
    char *last_slash = strrchr(url, '/');
    if (last_slash) {
        strcpy(parsed_url->filename, last_slash + 1);
    } else {
        strcpy(parsed_url->filename, "downloaded_file");
    }

    // Resolve hostname to IP
    struct hostent *host = gethostbyname(parsed_url->hostname);
    if (host == NULL) {
        herror("Hostname resolution failed");
        return false;
    }

    // Convert IP to string
    strcpy(parsed_url->ip_address, inet_ntoa(*((struct in_addr *) host->h_addr_list[0])));

    return true;
}


int create_socket(const char *ip, int port) {
    struct sockaddr_in server_addr;
    bzero((char *) &server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(ip);
    server_addr.sin_port = htons(port);

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        rushed_exit(-1, -1, "Socket creation failed");
    }

    if (connect(sockfd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0) {
        rushed_exit(-1, -1, "Connection failed");
    }

    return sockfd;
}

void ftp_login(int control_socket, const char* username, const char* password) {
    char buffer[MAX_BUFFER_SIZE];
    int response_code;

    // Send username
    snprintf(buffer, sizeof(buffer), "USER %s\r\n", username);
```

```c
    if (write(control_socket, buffer, strlen(buffer)) < 0) {
        rushed_exit(control_socket, -1, "Failed to send username");
    }

    // Read username response
    while (read(control_socket, buffer, sizeof(buffer)) > 0) {
        if (sscanf(buffer, "%d", &response_code) == 1 && response_code == 331) {
            break;
        }
    }

    // Send password
    snprintf(buffer, sizeof(buffer), "PASS %s\r\n", password);
    if (write(control_socket, buffer, strlen(buffer)) < 0) {
        rushed_exit(control_socket, -1, "Failed to send password");
    }

    // Read password response
    while (read(control_socket, buffer, sizeof(buffer)) > 0) {
        if (sscanf(buffer, "%d", &response_code) == 1 && response_code == 230) {
            break;
        }
    }
}

void activate_passive_mode(int control_socket, char *passive_ip, int *passive_port) {
    char buffer[MAX_BUFFER_SIZE];
    int response_code, byte1, byte2, byte3, byte4, byte5, byte6;

    for (int attempt = 0; attempt < 10; attempt++) {
        if (write(control_socket, "PASV\r\n", 6) < 0) {
            rushed_exit(control_socket, -1, "Failed to send PASV command");
        }

        // Read the response line by line
        while (read(control_socket, buffer, sizeof(buffer)) > 0) {
            char *pasv_response = strstr(buffer, "227 Entering Passive Mode");
            if (pasv_response && sscanf(pasv_response, "227 Entering Passive Mode (%d,%d,%d,%d,%d,%d)", &byte1, &byte2, &byte3, &byte4, &byte5, &byte6) == 6) {
                snprintf(passive_ip, MAX_URL_LENGTH, "%d.%d.%d.%d", byte1, byte2, byte3, byte4);
                *passive_port = (byte5 * 256) + byte6;
                printf("Passive IP: %s, Passive Port: %d\n", passive_ip, *passive_port);
                return;
            }
        }

        fprintf(stderr, "Retrying PASV due to parsing failure: %s\n", buffer);
    }

    rushed_exit(control_socket, -1, "Failed to activate passive mode after retries");
}

void download_file(int control_socket, int data_socket, const char *filename) {
    FILE *file = fopen(filename, "wb");
    if (!file) {
        rushed_exit(control_socket, data_socket, "Unable to create local file")
```

```c
            ;
    }

    char buffer[MAX_BUFFER_SIZE];
    ssize_t bytes_read;

    // Download file in chunks
    while ((bytes_read = read(data_socket, buffer, sizeof(buffer))) > 0) {
        if (fwrite(buffer, 1, bytes_read, file) != (size_t)bytes_read) {
            fclose(file);
            rushed_exit(control_socket, data_socket, "Failed to write to the file properly");
        }
    }
    if (bytes_read < 0) {
        fclose(file);
        rushed_exit(control_socket, data_socket, "read() function returned an error");
    }

    fclose(file);
    close(data_socket);  // Close data socket after file transfer

    // Verify download completion
    char response[MAX_BUFFER_SIZE];
    int response_code;

    while (read(control_socket, response, sizeof(response)) > 0) {
        if (sscanf(response, "%d", &response_code) == 1 && response_code == 226) {
            return;
        }
    }

    // If we exit the loop without success
    rushed_exit(control_socket, -1, response);
}


int main(int argc, char *argv[]) {
    // Validate command-line arguments
    if (argc != 2) {
        fprintf(stderr, "Usage: %s ftp://[user:pass@]host/path/to/file\n", argv[0]);
        return EXIT_FAILURE;
    }

    // Parse URL
    ftpURL url;
    if (!parse_ftp_url(argv[1], &url)) {
        return EXIT_FAILURE;
    }

    // Print parsed URL details
    printf("Downloading file:\n");
    printf("Host: %s\n", url.hostname);
    printf("Resource: %s\n", url.resource_path);
    printf("Filename: %s\n", url.filename);
    printf("IP: %s\n", url.ip_address);

    // Control socket connection
    int control_socket = create_socket(url.ip_address, PORT);
    char response[MAX_BUFFER_SIZE];
```

```c
    int response_code;

    // Initial server connection
    while (read(control_socket, response, sizeof(response)) > 0) {
        if (sscanf(response, "%d", &response_code) == 1 && response_code == 220) {
            break;
        }
    }

    // Login
    ftp_login(control_socket, url.username, url.password);

    // Passive mode
    char passive_ip[MAX_URL_LENGTH];
    int passive_port;
    activate_passive_mode(control_socket, passive_ip, &passive_port);

    // Data socket connection
    int data_socket = create_socket(passive_ip, passive_port);

    // Request resource
    char request[MAX_BUFFER_SIZE];
    snprintf(request, sizeof(request), "RETR %s\r\n", url.resource_path);
    if (write(control_socket, request, strlen(request)) < 0) {
        rushed_exit(control_socket, data_socket, "Failed to send RETR command");
    }

    // Verify resource request
    while (read(control_socket, response, sizeof(response)) > 0) {
        if (sscanf(response, "%d", &response_code) == 1 && (response_code == 150 || response_code == 125)) {
            break;
        }
    }

    // Download file
    download_file(control_socket, data_socket, url.filename);

    // Close connections
    graceful_exit(control_socket);

    printf("File downloaded successfully: %s\n", url.filename);
    return EXIT_SUCCESS;
}
```

Listing 1: FTP Download Application Code

B Part 2 - Network Configuration and Analysis

B.1 Experiment 1 - Configure an IP Network

This appendix provides the supporting information for Experiment 1, done on bench **11**, including the commands executed, the final configurations of the devices, and the captures from Wireshark.

Final Configurations

The table below summarizes the final configurations of the devices used in this experiment.

TUX	IP Address	MAC Address	Switch Port (Ether)	Interface (ETH)
3	172.16.110.1	00:08:54:50:35:0a	6	1
4	172.16.110.254	00:08:54:71:73:ed	5	1

Table 1: Final configurations of devices in Experiment 1.

Executed Commands

The following commands were executed on each device to configure the network.

tux113

```
systemctl restart networking
ifconfig eth1 up
ifconfig eth1 172.16.110.1/24
```

Listing 2: Commands executed on tux113

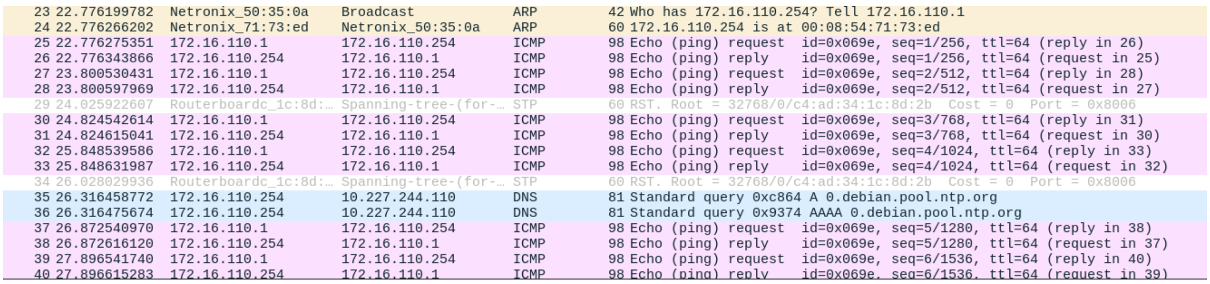
tux114

```
systemctl restart networking
ifconfig eth1 up
ifconfig eth1 172.16.110.254/24
```

Listing 3: Commands executed on tux114

Wireshark Screenshot

The following screenshot illustrates the ARP and ICMP packets captured during the experiment, confirming successful communication between the devices.



23	22.776199782	Netronix_50:35:0a	Broadcast	ARP	42	Who has 172.16.110.254? Tell 172.16.110.1
24	22.776266202	Netronix_71:73:ed	Netronix_50:35:0a	ARP	60	172.16.110.254 is at 00:08:54:71:73:ed
25	22.776275351	172.16.110.1	172.16.110.254	ICMP	98	Echo (ping) request id=0x069e, seq=1/256, ttl=64 (reply in 26)
26	22.776343866	172.16.110.254	172.16.110.1	ICMP	98	Echo (ping) reply id=0x069e, seq=1/256, ttl=64 (request in 25)
27	23.800530431	172.16.110.1	172.16.110.254	ICMP	98	Echo (ping) request id=0x069e, seq=2/512, ttl=64 (reply in 28)
28	23.800597969	172.16.110.254	172.16.110.1	ICMP	98	Echo (ping) reply id=0x069e, seq=2/512, ttl=64 (request in 27)
29	24.025922607	Routerboardc_1c:8d:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:8d:2b Cost = 0 Port = 0x8006
30	24.824542614	172.16.110.1	172.16.110.254	ICMP	98	Echo (ping) request id=0x069e, seq=3/768, ttl=64 (reply in 31)
31	24.824615041	172.16.110.254	172.16.110.1	ICMP	98	Echo (ping) reply id=0x069e, seq=3/768, ttl=64 (request in 30)
32	25.848539586	172.16.110.1	172.16.110.254	ICMP	98	Echo (ping) request id=0x069e, seq=4/1024, ttl=64 (reply in 33)
33	25.848631987	172.16.110.254	172.16.110.1	ICMP	98	Echo (ping) reply id=0x069e, seq=4/1024, ttl=64 (request in 32)
34	26.028029936	Routerboardc_1c:8d:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:8d:2b Cost = 0 Port = 0x8006
35	26.316458772	172.16.110.254	10.227.244.110	DNS	81	Standard query 0xc864 A 0.debian.pool.ntp.org
36	26.316475674	172.16.110.254	10.227.244.110	DNS	81	Standard query 0x9374 AAAA 0.debian.pool.ntp.org
37	26.872540970	172.16.110.1	172.16.110.254	ICMP	98	Echo (ping) request id=0x069e, seq=5/1280, ttl=64 (reply in 38)
38	26.872616120	172.16.110.254	172.16.110.1	ICMP	98	Echo (ping) reply id=0x069e, seq=5/1280, ttl=64 (request in 37)
39	27.896541740	172.16.110.1	172.16.110.254	ICMP	98	Echo (ping) request id=0x069e, seq=6/1536, ttl=64 (reply in 40)
40	27.896615283	172.16.110.254	172.16.110.1	ICMP	98	Echo (ping) reply id=0x069e, seq=6/1536, ttl=64 (request in 39)

Figure 2: Wireshark Capture: ARP and ICMP packets in Experiment 1.

B.2 Experiment 2 - Implement Two Bridges in a Switch

This appendix provides the supporting information for Experiment 2, done on bench **11**, including the commands executed, the final configurations of the devices, and the captures from Wireshark.

Final Configurations

The table below summarizes the final configurations of the devices used in this experiment.

TUX	IP Address	MAC Address	Switch Port (Ether)	Interface (ETH)
2	172.16.111.1	00:c0:df:08:d5:98	12	1
3	172.16.110.1	00:08:54:50:35:0a	6	1
4	172.16.110.254	00:08:54:71:73:ed	5	1

Table 2: Final configurations of devices in Experiment 2.

Executed Commands

The following commands were executed on each device to configure the network.

tux113

```
systemctl restart networking
ifconfig eth1 up
ifconfig eth1 172.16.110.1/24
```

Listing 4: Commands executed on tux113

tux114

```
systemctl restart networking
ifconfig eth1 up
ifconfig eth1 172.16.110.254/24
```

Listing 5: Commands executed on tux114

tux112

```
systemctl restart networking
ifconfig eth1 up
ifconfig eth1 172.16.111.1/24
```

Listing 6: Commands executed on tux112

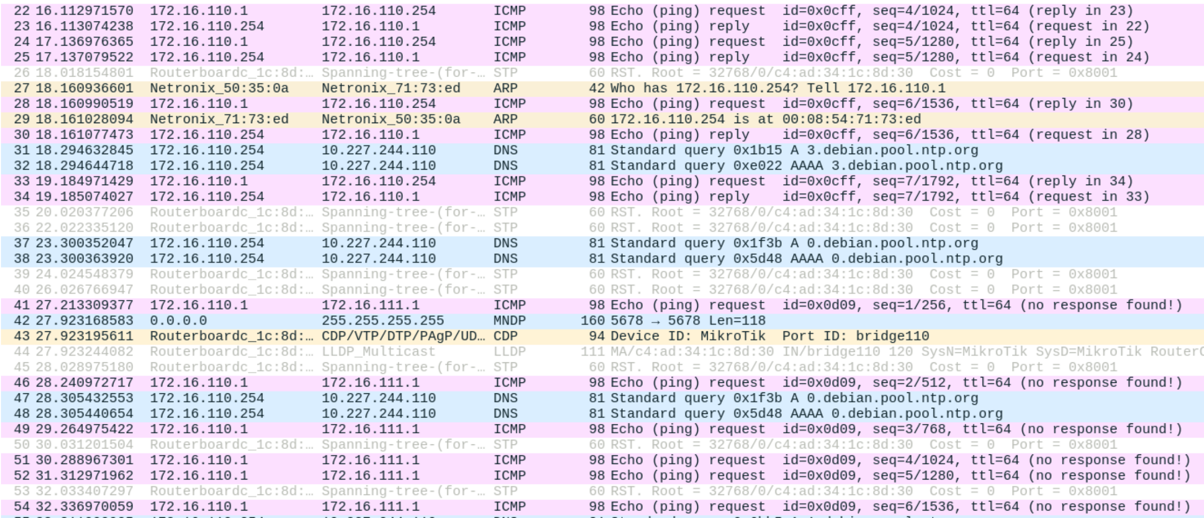
MicroTik Switch

```
/system reset-configuration

username: admin
pass:

/interface bridge host print

/interface bridge add name=bridge110
/interface bridge add name=bridge111

/interface bridge port remove [find interface=ether5]
/interface bridge port remove [find interface=ether6]
/interface bridge port remove [find interface=ether12]

/interface bridge port add bridge=bridge110 interface=ether5
/interface bridge port add bridge=bridge110 interface=ether6
/interface bridge port add bridge=bridge111 interface=ether12

/interface bridge host print
```

Listing 7: Commands executed on the MicroTik Switch

Wireshark Screenshot

The following screenshot illustrates the ARP and ICMP packets captured during the experiment. It demonstrates that Tux3 (‘172.16.110.1‘) successfully communicates with Tux4 (‘172.16.110.254‘), as they are within the same bridge (Bridge110). However, Tux3 cannot communicate with Tux2 (‘172.16.111.1‘), as it is isolated in a different bridge (Bridge111).



22	16.112971570	172.16.110.1	172.16.110.254	ICMP	98	Echo (ping) request id=0x0cff, seq=4/1024, ttl=64 (reply in 23)
23	16.113074238	172.16.110.254	172.16.110.1	ICMP	98	Echo (ping) reply id=0x0cff, seq=4/1024, ttl=64 (request in 22)
24	17.136976365	172.16.110.1	172.16.110.254	ICMP	98	Echo (ping) request id=0x0cff, seq=5/1280, ttl=64 (reply in 25)
25	17.137079522	172.16.110.254	172.16.110.1	ICMP	98	Echo (ping) reply id=0x0cff, seq=5/1280, ttl=64 (request in 24)
26	18.018154801	Routerboardc_1c:8d:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:8d:30 Cost = 0 Port = 0x8001
27	18.160936601	Netronix_50:35:0a	Netronix_71:73:ed	ARP	42	Who has 172.16.110.254? Tell 172.16.110.1
28	18.160990519	172.16.110.1	172.16.110.254	ICMP	98	Echo (ping) request id=0x0cff, seq=6/1536, ttl=64 (reply in 30)
29	18.161028094	Netronix_71:73:ed	Netronix_50:35:0a	ARP	60	172.16.110.254 is at 00:08:54:71:73:ed
30	18.161077473	172.16.110.254	172.16.110.1	ICMP	98	Echo (ping) reply id=0x0cff, seq=6/1536, ttl=64 (request in 28)
31	18.294632845	172.16.110.254	10.227.244.110	DNS	81	Standard query 0x1b15 A 3.debian.pool.ntp.org
32	18.294644718	172.16.110.254	10.227.244.110	DNS	81	Standard query 0xe022 AAAA 3.debian.pool.ntp.org
33	19.184971429	172.16.110.1	172.16.110.254	ICMP	98	Echo (ping) request id=0x0cff, seq=7/1792, ttl=64 (reply in 34)
34	19.185074027	172.16.110.254	172.16.110.1	ICMP	98	Echo (ping) reply id=0x0cff, seq=7/1792, ttl=64 (request in 33)
35	20.020377206	Routerboardc_1c:8d:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:8d:30 Cost = 0 Port = 0x8001
36	22.022335120	Routerboardc_1c:8d:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:8d:30 Cost = 0 Port = 0x8001
37	23.300352047	172.16.110.254	10.227.244.110	DNS	81	Standard query 0x1f3b A 0.debian.pool.ntp.org
38	23.300363920	172.16.110.254	10.227.244.110	DNS	81	Standard query 0x5d48 AAAA 0.debian.pool.ntp.org
39	24.024548379	Routerboardc_1c:8d:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:8d:30 Cost = 0 Port = 0x8001
40	26.026766947	Routerboardc_1c:8d:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:8d:30 Cost = 0 Port = 0x8001
41	27.213309377	172.16.110.1	172.16.111.1	ICMP	98	Echo (ping) request id=0x0d09, seq=1/256, ttl=64 (no response found!)
42	27.923168583	0.0.0.0	255.255.255.255	MNDP	160	5678 → 5678 Len=118
43	27.923195611	Routerboardc_1c:8d:…	CDP/VTP/DTP/PAgP/UD…	CDP	94	Device ID: MikroTik Port ID: bridge110
44	27.923244082	Routerboardc_1c:8d:…	LLDP_Multicast	LLDP	111	MA/c4:ad:34:1c:8d:30 IN/bridge110 120 SysN=MikroTik SysD=MikroTik Router(
45	28.028975180	Routerboardc_1c:8d:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:8d:30 Cost = 0 Port = 0x8001
46	28.240972717	172.16.110.1	172.16.111.1	ICMP	98	Echo (ping) request id=0x0d09, seq=2/512, ttl=64 (no response found!)
47	28.305432553	172.16.110.254	10.227.244.110	DNS	81	Standard query 0x1f3b A 0.debian.pool.ntp.org
48	28.305440654	172.16.110.254	10.227.244.110	DNS	81	Standard query 0x5d48 AAAA 0.debian.pool.ntp.org
49	29.264975422	172.16.110.1	172.16.111.1	ICMP	98	Echo (ping) request id=0x0d09, seq=3/768, ttl=64 (no response found!)
50	30.031201504	Routerboardc_1c:8d:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:8d:30 Cost = 0 Port = 0x8001
51	30.288967301	172.16.110.1	172.16.111.1	ICMP	98	Echo (ping) request id=0x0d09, seq=4/1024, ttl=64 (no response found!)
52	31.312971962	172.16.110.1	172.16.111.1	ICMP	98	Echo (ping) request id=0x0d09, seq=5/1280, ttl=64 (no response found!)
53	32.033407297	Routerboardc_1c:8d:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:8d:30 Cost = 0 Port = 0x8001
54	32.336970059	172.16.110.1	172.16.111.1	ICMP	98	Echo (ping) request id=0x0d09, seq=6/1536, ttl=64 (no response found!)

Figure 3: Wireshark Capture: ARP and ICMP packets demonstrating bridge isolation.

B.3 Experiment 3 - Configure a Router in Linux

This appendix provides the supporting information for Experiment 3, done on bench **12**, including the commands executed, the final configurations of the devices, and the captures from Wireshark.

Final Configurations

The table below summarizes the final configurations of the devices used in this experiment.

TUX	IP Address	MAC Address	Switch Port (Ether)	Interface (ETH)
2	172.16.121.1	00:c0:df:10:90:56	9	1
3	172.16.120.1	00:08:54:50:31:bc	7	1
4	172.16.120.254	00:e0:7d:b4:b8:94	8	1
4	172.16.121.253	00:08:54:57:fa:89	10	2

Table 3: Final configurations of devices in Experiment 3.

Executed Commands

The following commands were executed on each device to configure the network.

tux123

```
systemctl restart networking
ifconfig eth1 up
ifconfig eth1 172.16.120.1/24
route add -net 172.16.121.0/24 gw 172.16.120.254
```

Listing 8: Commands executed on tux123

tux124

```
systemctl restart networking
ifconfig eth1 up
ifconfig eth1 172.16.120.254/24
ifconfig eth2 up
ifconfig eth2 172.16.121.253/24
sysctl net.ipv4.ip_forward=1
sysctl net.ipv4.icmp_echo_ignore_broadcasts=0
```

Listing 9: Commands executed on tux124

tux122

```
systemctl restart networking
ifconfig eth1 up
ifconfig eth1 172.16.121.1/24
route add -net 172.16.120.0/24 gw 172.16.121.253
```

Listing 10: Commands executed on tux122

MicroTik Switch

```
/system reset-configuration

username: admin
pass:

/interface bridge host print

/interface bridge add name=bridge120
/interface bridge add name=bridge121

/interface bridge port remove [find interface=ether7]
/interface bridge port remove [find interface=ether8]
/interface bridge port remove [find interface=ether9]
/interface bridge port remove [find interface=ether10]

/interface bridge port add bridge=bridge120 interface=ether7
/interface bridge port add bridge=bridge120 interface=ether8
/interface bridge port add bridge=bridge121 interface=ether9
/interface bridge port add bridge=bridge121 interface=ether10

/interface bridge host print
```

Listing 11: Commands executed on the MicroTik Switch

Wireshark Screenshots

The following screenshots illustrate the ARP and ICMP packets captured during the experiment, confirming communication both within the same bridge and across different subnets through the configured router on Tux4.

Communication Within the Same Bridge

30	20.204965017	172.16.120.1	172.16.120.254	ICMP	98	Echo (ping) request id=0x2553, seq=8/2048, ttl=64 (reply in 31)
31	20.205082560	172.16.120.254	172.16.120.1	ICMP	98	Echo (ping) reply id=0x2553, seq=8/2048, ttl=64 (request in 30)
32	21.228963767	172.16.120.1	172.16.120.254	ICMP	98	Echo (ping) request id=0x2553, seq=9/2304, ttl=64 (reply in 33)
33	21.229109945	172.16.120.254	172.16.120.1	ICMP	98	Echo (ping) reply id=0x2553, seq=9/2304, ttl=64 (request in 32)

Figure 4: Wireshark Capture: ICMP packets demonstrating communication within the same bridge (Tux3 and Tux4).

Communication Across Subnets

62	38.604982512	172.16.120.1	172.16.121.253	ICMP	98	Echo (ping) request id=0x2560, seq=7/1792, ttl=64 (reply in 63)
63	38.605108226	172.16.121.253	172.16.120.1	ICMP	98	Echo (ping) reply id=0x2560, seq=7/1792, ttl=64 (request in 62)
64	39.628989434	172.16.120.1	172.16.121.253	ICMP	98	Echo (ping) request id=0x2560, seq=8/2048, ttl=64 (reply in 65)
65	39.629112494	172.16.121.253	172.16.120.1	ICMP	98	Echo (ping) reply id=0x2560, seq=8/2048, ttl=64 (request in 64)
66	40.044751006	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:30 Cost = 0 Port = 0x8001
67	40.652996077	172.16.120.1	172.16.121.253	ICMP	98	Echo (ping) request id=0x2560, seq=9/2304, ttl=64 (reply in 68)
68	40.653120464	172.16.121.253	172.16.120.1	ICMP	98	Echo (ping) reply id=0x2560, seq=9/2304, ttl=64 (request in 67)
69	41.676979532	172.16.120.1	172.16.121.253	ICMP	98	Echo (ping) request id=0x2560, seq=10/2560, ttl=64 (reply in 70)
70	41.677129132	172.16.121.253	172.16.120.1	ICMP	98	Echo (ping) reply id=0x2560, seq=10/2560, ttl=64 (request in 69)
71	42.046982782	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:30 Cost = 0 Port = 0x8001
72	42.700979820	172.16.120.1	172.16.121.253	ICMP	98	Echo (ping) request id=0x2560, seq=11/2816, ttl=64 (reply in 73)
73	42.701111540	172.16.121.253	172.16.120.1	ICMP	98	Echo (ping) reply id=0x2560, seq=11/2816, ttl=64 (request in 72)
74	44.049221053	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:30 Cost = 0 Port = 0x8001
75	46.051452968	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:30 Cost = 0 Port = 0x8001
76	48.053694522	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:30 Cost = 0 Port = 0x8001
77	50.055935308	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:30 Cost = 0 Port = 0x8001
78	50.324326022	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x256a, seq=1/256, ttl=64 (reply in 79)
79	50.324590022	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x256a, seq=1/256, ttl=63 (request in 78)
80	51.340976924	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x256a, seq=2/512, ttl=64 (reply in 81)
81	51.341233800	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x256a, seq=2/512, ttl=63 (request in 80)
82	52.058178258	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:30 Cost = 0 Port = 0x8001
83	52.364968063	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x256a, seq=3/768, ttl=64 (reply in 84)
84	52.365176818	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x256a, seq=3/768, ttl=63 (request in 83)
85	53.388968001	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x256a, seq=4/1024, ttl=64 (reply in 86)
86	53.389177106	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x256a, seq=4/1024, ttl=63 (request in 85)

Figure 5: Wireshark Capture: ICMP packets demonstrating communication across subnets via the router (Tux3 and Tux2).

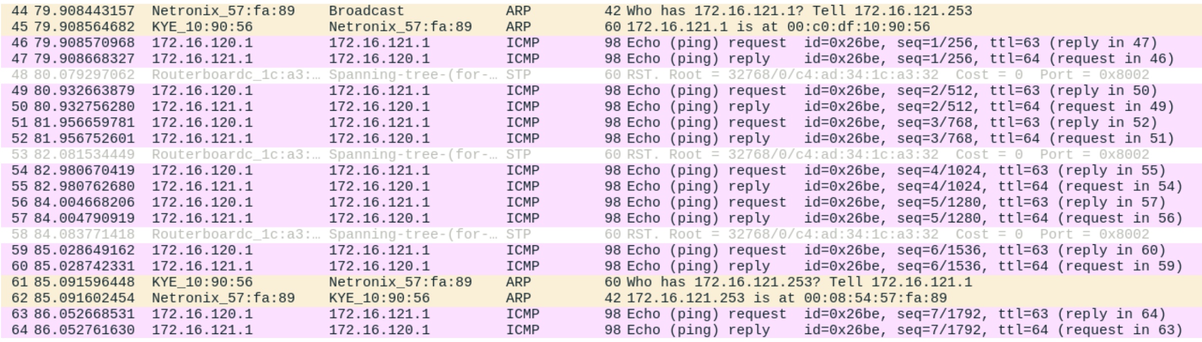
The following screenshots illustrate the ARP and ICMP packets captured on both interfaces of Tux4 (`eth1` and `eth2`), confirming the forwarding of packets across subnets during the experiment.

Interface eth1 (Egress to Subnet 172.16.120.0/24)

48	81.910506269	Netronix_50:31:bc	Broadcast	ARP	60	Who has 172.16.120.254? Tell 172.16.120.1
49	81.910529457	Netronix_b4:b8:94	Netronix_50:31:bc	ARP	42	172.16.120.254 is at 00:e0:7d:b4:b8:94
50	81.910637642	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=1/256, ttl=64 (reply in 51)
51	81.910882368	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=1/256, ttl=63 (request in 50)
52	82.081538481	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:30 Cost = 0 Port = 0x8002
53	82.934855012	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=2/512, ttl=64 (reply in 54)
54	82.934971648	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=2/512, ttl=63 (request in 53)
55	83.958849866	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=3/768, ttl=64 (reply in 56)
56	83.958968388	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=3/768, ttl=63 (request in 55)
57	84.083776148	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:30 Cost = 0 Port = 0x8002
58	84.982856803	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=4/1024, ttl=64 (reply in 59)
59	84.982979096	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=4/1024, ttl=63 (request in 58)
60	86.006857174	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=5/1280, ttl=64 (reply in 61)
61	86.007007474	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=5/1280, ttl=63 (request in 60)
62	86.086013815	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:30 Cost = 0 Port = 0x8002
63	86.945956469	Netronix_b4:b8:94	Netronix_50:31:bc	ARP	42	Who has 172.16.120.1? Tell 172.16.120.254
64	86.946074990	Netronix_50:31:bc	Netronix_b4:b8:94	ARP	60	172.16.120.1 is at 00:08:54:50:31:bc
65	87.030843717	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=6/1536, ttl=64 (reply in 66)
66	87.030956372	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=6/1536, ttl=63 (request in 65)
67	88.054857498	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=7/1792, ttl=64 (reply in 68)
68	88.054977627	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=7/1792, ttl=63 (request in 67)

Figure 6: Wireshark Capture: ARP and ICMP packets exiting Tux4 via eth1.

Interface eth2 (Ingress from Subnet 172.16.121.0/24)



44	79.908443157	Netronix_57:fa:89	Broadcast	ARP	42	Who has 172.16.121.1? Tell 172.16.121.253
45	79.908564682	KYE_10:90:56	Netronix_57:fa:89	ARP	60	172.16.121.1 is at 00:c0:df:10:90:56
46	79.908570968	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=1/256, ttl=63 (reply in 47)
47	79.908668327	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=1/256, ttl=64 (request in 46)
48	80.079297062	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:32 Cost = 0 Port = 0x8002
49	80.932663879	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=2/512, ttl=63 (reply in 50)
50	80.932756280	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=2/512, ttl=64 (request in 49)
51	81.956659781	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=3/768, ttl=63 (reply in 52)
52	81.956752601	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=3/768, ttl=64 (request in 51)
53	82.081534449	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:32 Cost = 0 Port = 0x8002
54	82.980670419	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=4/1024, ttl=63 (reply in 55)
55	82.980762680	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=4/1024, ttl=64 (request in 54)
56	84.004668206	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=5/1280, ttl=63 (reply in 57)
57	84.004790919	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=5/1280, ttl=64 (request in 56)
58	84.083771418	Routerboardc_1c:a3:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:32 Cost = 0 Port = 0x8002
59	85.028649162	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=6/1536, ttl=63 (reply in 60)
60	85.028742331	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=6/1536, ttl=64 (request in 59)
61	85.091596448	KYE_10:90:56	Netronix_57:fa:89	ARP	60	Who has 172.16.121.253? Tell 172.16.121.1
62	85.091602454	Netronix_57:fa:89	KYE_10:90:56	ARP	42	172.16.121.253 is at 00:08:54:57:fa:89
63	86.052668531	172.16.120.1	172.16.121.1	ICMP	98	Echo (ping) request id=0x26be, seq=7/1792, ttl=63 (reply in 64)
64	86.052761630	172.16.121.1	172.16.120.1	ICMP	98	Echo (ping) reply id=0x26be, seq=7/1792, ttl=64 (request in 63)

Figure 7: Wireshark Capture: ARP and ICMP packets entering Tux4 via eth2.

B.4 Experiment 4 - Configure a Commercial Router and Implement NAT

This appendix provides the supporting information for Experiment 4, done on bench **10**, including the commands executed, the final configurations of the devices, and the captures from Wireshark as well as Traceroutes.

Final Configurations

The table below summarizes the final configurations of the devices used in this experiment.

TUX/RC	IP Address	MAC Address	Switch Port (Ether)	Interface (ETH)
2	172.16.101.1	00:c0:df:13:20:1d	9	1
3	172.16.100.1	00:c0:df:13:20:10	5	1
4	172.16.100.254	00:c0:df:25:43:bc	6	1
4	172.16.101.253	00:01:02:9f:7e:9c	10	2
RC	172.16.101.254	-	11	-

Table 4: Final configurations of devices in Experiment 4.

Executed Commands

The following commands were executed on each device, the MikroTik Router, and the MikroTik Switch to configure the network and implement NAT.

tux103

```
systemctl restart networking
ifconfig eth1 up
ifconfig eth1 172.16.100.1/24
route add -net 172.16.101.0/24 gw 172.16.100.254
```

Listing 12: Commands executed on tux103

tux104

```
systemctl restart networking
ifconfig eth1 up
ifconfig eth1 172.16.100.254/24
ifconfig eth2 up
ifconfig eth2 172.16.101.253/24
sysctl net.ipv4.ip_forward=1
sysctl net.ipv4.icmp_echo_ignore_broadcasts=0
```

Listing 13: Commands executed on tux104

tux102

```
systemctl restart networking
ifconfig eth1 up
ifconfig eth1 172.16.101.1/24
route add -net 172.16.100.0/24 gw 172.16.101.253
```

Listing 14: Commands executed on tux102

MikroTik Switch

```
/system reset-configuration

/interface bridge add name=bridge100
/interface bridge add name=bridge101

/interface bridge port remove [find interface=ether5]
/interface bridge port remove [find interface=ether6]
/interface bridge port remove [find interface=ether9]
/interface bridge port remove [find interface=ether10]
/interface bridge port remove [find interface=ether11]

/interface bridge port add bridge=bridge100 interface=ether5
/interface bridge port add bridge=bridge100 interface=ether6
/interface bridge port add bridge=bridge101 interface=ether9
/interface bridge port add bridge=bridge101 interface=ether10
/interface bridge port add bridge=bridge101 interface=ether11
```

Listing 15: Commands executed on the MikroTik Switch

MikroTik Router

```
/system reset-configuration

/ip address add address=172.16.1.109/24 interface=ether1
/ip address add address=172.16.101.254/24 interface=ether2
/ip route add dst-address=172.16.100.0/24 gateway=172.16.101.253
/ip route add dst-address=0.0.0.0/0 gateway=172.16.1.254
```

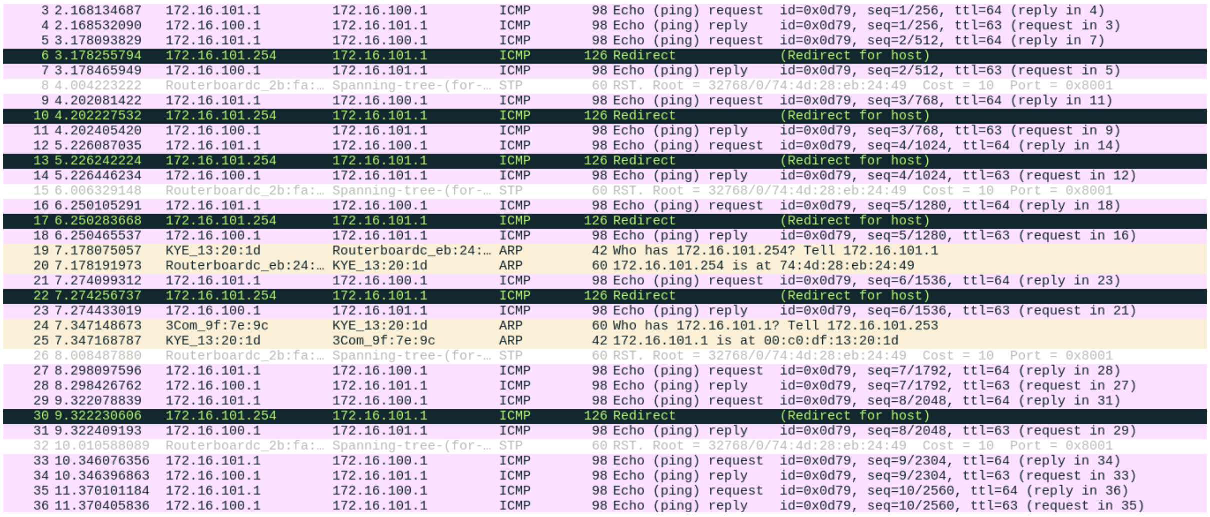
Listing 16: Commands executed on the MikroTik Router

Wireshark Screenshots and Traceroutes

This section provides the packet captures from tux102 and tux103, showcasing their interactions and connectivity during the experiment.

tux102 The capture from tux102 illustrates the routes and packet exchanges observed on this device. Key highlights include:

- ICMP Echo Requests and Replies: Demonstrating communication within the network.
- Redirect Messages: Validating the use of alternate routes via the configured router.
- ARP Requests and Replies: Confirming MAC address resolution between devices.



3	2.168134687	172.16.101.1	172.16.100.1	ICMP	98	Echo (ping) request id=0x0d79, seq=1/256, ttl=64 (reply in 4)
4	2.168532090	172.16.100.1	172.16.101.1	ICMP	98	Echo (ping) reply id=0x0d79, seq=1/256, ttl=63 (request in 3)
5	3.178093829	172.16.101.1	172.16.100.1	ICMP	98	Echo (ping) request id=0x0d79, seq=2/512, ttl=64 (reply in 7)
6	3.178255794	172.16.101.254	172.16.101.1	ICMP	126	Redirect (Redirect for host)
7	3.178465949	172.16.100.1	172.16.101.1	ICMP	98	Echo (ping) reply id=0x0d79, seq=2/512, ttl=63 (request in 5)
8	4.004223222	Routerboardc_2b:fa:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/74:4d:28:eb:24:49 Cost = 10 Port = 0x8001
9	4.202081422	172.16.101.1	172.16.100.1	ICMP	98	Echo (ping) request id=0x0d79, seq=3/768, ttl=64 (reply in 11)
10	4.202227532	172.16.101.254	172.16.101.1	ICMP	126	Redirect (Redirect for host)
11	4.202405420	172.16.100.1	172.16.101.1	ICMP	98	Echo (ping) reply id=0x0d79, seq=3/768, ttl=63 (request in 9)
12	5.226087035	172.16.101.1	172.16.100.1	ICMP	98	Echo (ping) request id=0x0d79, seq=4/1024, ttl=64 (reply in 14)
13	5.226242224	172.16.101.254	172.16.101.1	ICMP	126	Redirect (Redirect for host)
14	5.226446234	172.16.100.1	172.16.101.1	ICMP	98	Echo (ping) reply id=0x0d79, seq=4/1024, ttl=63 (request in 12)
15	6.006329148	Routerboardc_2b:fa:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/74:4d:28:eb:24:49 Cost = 10 Port = 0x8001
16	6.250105291	172.16.101.1	172.16.100.1	ICMP	98	Echo (ping) request id=0x0d79, seq=5/1280, ttl=64 (reply in 18)
17	6.250283668	172.16.101.254	172.16.101.1	ICMP	126	Redirect (Redirect for host)
18	6.250465537	172.16.100.1	172.16.101.1	ICMP	98	Echo (ping) reply id=0x0d79, seq=5/1280, ttl=63 (request in 16)
19	7.178075057	KYE_13:20:1d	Routerboardc_eb:24:…	ARP	42	Who has 172.16.101.254? Tell 172.16.101.1
20	7.178191973	Routerboardc_eb:24:…	KYE_13:20:1d	ARP	60	172.16.101.254 is at 74:4d:28:eb:24:49
21	7.274099312	172.16.101.1	172.16.100.1	ICMP	98	Echo (ping) request id=0x0d79, seq=6/1536, ttl=64 (reply in 23)
22	7.274256737	172.16.101.254	172.16.101.1	ICMP	126	Redirect (Redirect for host)
23	7.274433019	172.16.100.1	172.16.101.1	ICMP	98	Echo (ping) reply id=0x0d79, seq=6/1536, ttl=63 (request in 21)
24	7.347148673	3Com_9f:7e:9c	KYE_13:20:1d	ARP	60	Who has 172.16.101.1? Tell 172.16.101.253
25	7.347168787	KYE_13:20:1d	3Com_9f:7e:9c	ARP	42	172.16.101.1 is at 00:c0:df:13:20:1d
26	8.008487880	Routerboardc_2b:fa:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/74:4d:28:eb:24:49 Cost = 10 Port = 0x8001
27	8.298097596	172.16.101.1	172.16.100.1	ICMP	98	Echo (ping) request id=0x0d79, seq=7/1792, ttl=64 (reply in 28)
28	8.298426762	172.16.100.1	172.16.101.1	ICMP	98	Echo (ping) reply id=0x0d79, seq=7/1792, ttl=63 (request in 27)
29	9.322078839	172.16.101.1	172.16.100.1	ICMP	98	Echo (ping) request id=0x0d79, seq=8/2048, ttl=64 (reply in 31)
30	9.322230606	172.16.101.254	172.16.101.1	ICMP	126	Redirect (Redirect for host)
31	9.322409193	172.16.100.1	172.16.101.1	ICMP	98	Echo (ping) reply id=0x0d79, seq=8/2048, ttl=63 (request in 29)
32	10.010588089	Routerboardc_2b:fa:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/74:4d:28:eb:24:49 Cost = 10 Port = 0x8001
33	10.346076356	172.16.101.1	172.16.100.1	ICMP	98	Echo (ping) request id=0x0d79, seq=9/2304, ttl=64 (reply in 34)
34	10.346396863	172.16.100.1	172.16.101.1	ICMP	98	Echo (ping) reply id=0x0d79, seq=9/2304, ttl=63 (request in 33)
35	11.370101184	172.16.101.1	172.16.100.1	ICMP	98	Echo (ping) request id=0x0d79, seq=10/2560, ttl=64 (reply in 36)
36	11.370405836	172.16.100.1	172.16.101.1	ICMP	98	Echo (ping) reply id=0x0d79, seq=10/2560, ttl=63 (request in 35)

Figure 8: Wireshark Capture: ICMP packets, redirects, and ARP messages on tux102.

Additionally, the traceroute results on tux102 provide further insights into the routing behavior and path taken by packets:

- Gateway via RC: A traceroute using RC as the gateway demonstrates correct forwarding through the router.
- Gateway via tux104: An alternate traceroute using tux104 as the gateway shows differences in routing paths.

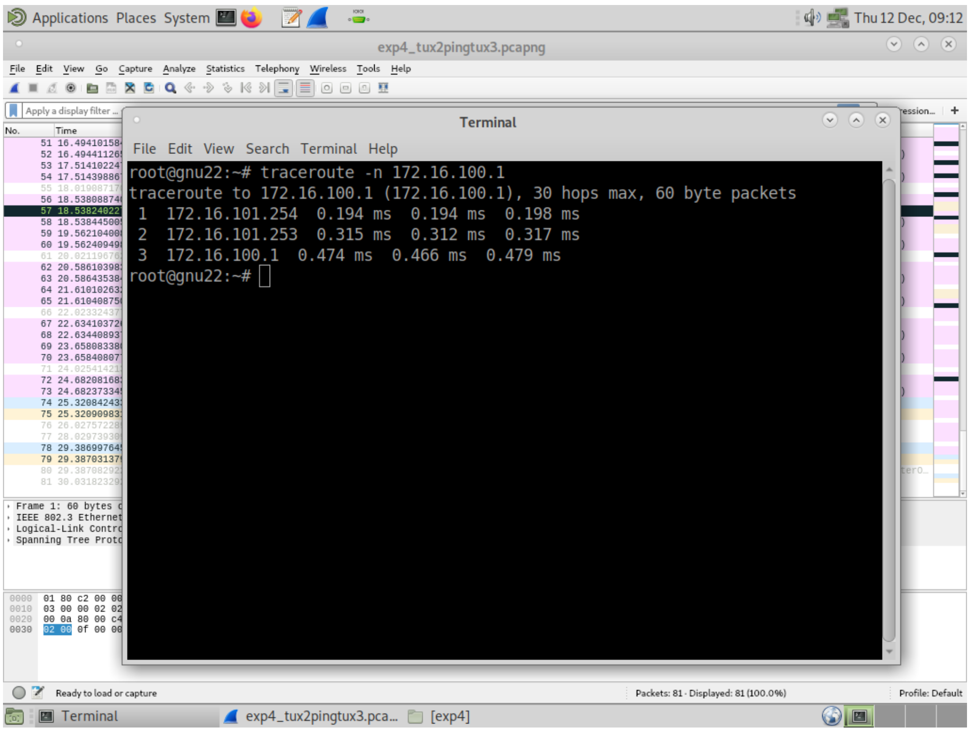


Figure 9: Traceroute on tux102 using RC as the gateway.

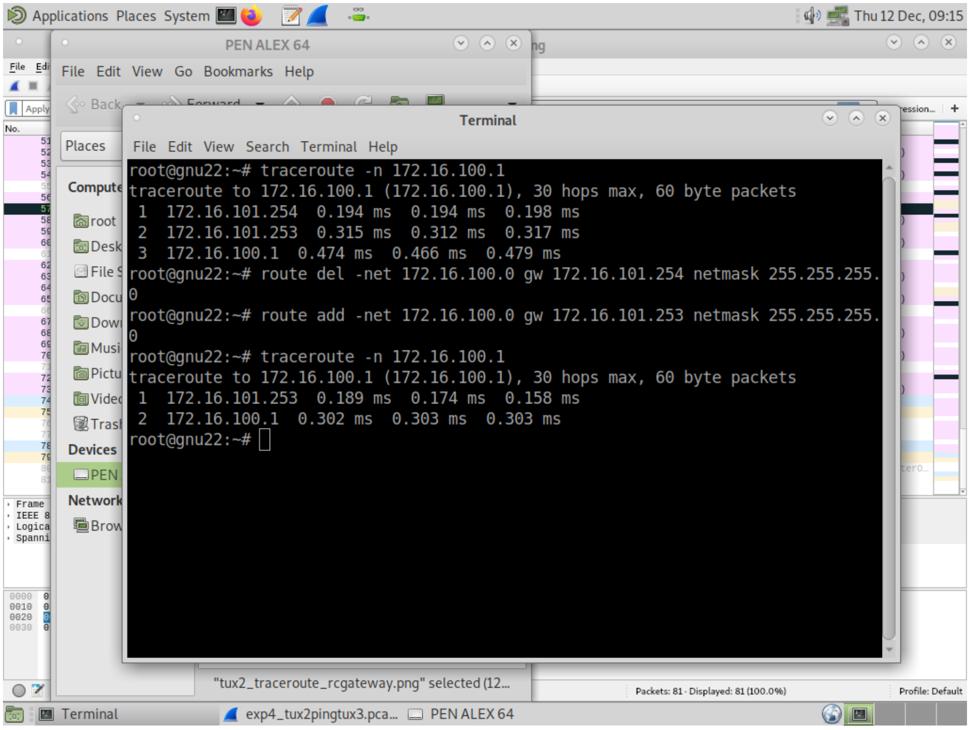
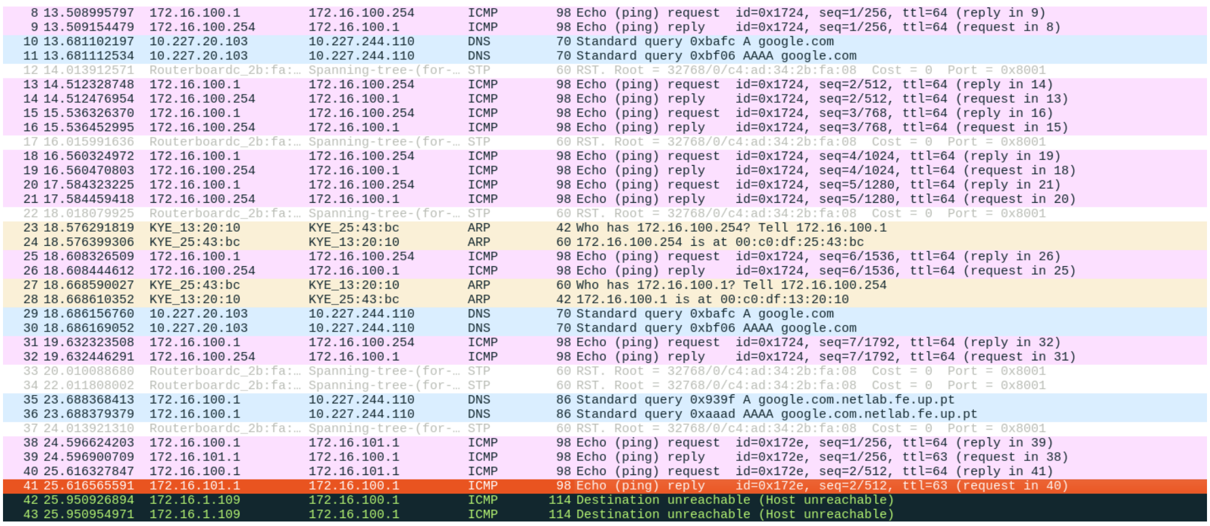


Figure 10: Traceroute on tux102 using tux104 as the gateway.

tux103 The capture from tux103 highlights its interactions with other devices and networks. Observations include:

- ICMP Echo Requests and Replies: Confirming connectivity within the network.
- ARP Requests and Replies: Validating the ARP table entries for communication.
- Destination Unreachable Messages: Indicating a lack of routes to certain destinations, which helps in understanding routing configurations.



8	13.508995797	172.16.100.1	172.16.100.254	ICMP	98	Echo (ping) request id=0x1724, seq=1/256, ttl=64 (reply in 9)
9	13.509154479	172.16.100.254	172.16.100.1	ICMP	98	Echo (ping) reply id=0x1724, seq=1/256, ttl=64 (request in 8)
10	13.681102197	10.227.20.103	10.227.244.110	DNS	70	Standard query 0xbafc A google.com
11	13.681112534	10.227.20.103	10.227.244.110	DNS	70	Standard query 0xbf06 AAAA google.com
12	14.013912571	Routerboardc_2b:fa:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:2b:fa:08 Cost = 0 Port = 0x8001
13	14.512328748	172.16.100.1	172.16.100.254	ICMP	98	Echo (ping) request id=0x1724, seq=2/512, ttl=64 (reply in 14)
14	14.512476954	172.16.100.254	172.16.100.1	ICMP	98	Echo (ping) reply id=0x1724, seq=2/512, ttl=64 (request in 13)
15	15.536326370	172.16.100.1	172.16.100.254	ICMP	98	Echo (ping) request id=0x1724, seq=3/768, ttl=64 (reply in 16)
16	15.536452995	172.16.100.254	172.16.100.1	ICMP	98	Echo (ping) reply id=0x1724, seq=3/768, ttl=64 (request in 15)
17	16.015991636	Routerboardc_2b:fa:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:2b:fa:08 Cost = 0 Port = 0x8001
18	16.560324972	172.16.100.1	172.16.100.254	ICMP	98	Echo (ping) request id=0x1724, seq=4/1024, ttl=64 (reply in 19)
19	16.560470803	172.16.100.254	172.16.100.1	ICMP	98	Echo (ping) reply id=0x1724, seq=4/1024, ttl=64 (request in 18)
20	17.584323225	172.16.100.1	172.16.100.254	ICMP	98	Echo (ping) request id=0x1724, seq=5/1280, ttl=64 (reply in 21)
21	17.584459418	172.16.100.254	172.16.100.1	ICMP	98	Echo (ping) reply id=0x1724, seq=5/1280, ttl=64 (request in 20)
22	18.018079925	Routerboardc_2b:fa:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:2b:fa:08 Cost = 0 Port = 0x8001
23	18.576291819	KYE_13:20:10	KYE_25:43:bc	ARP	42	Who has 172.16.100.254? Tell 172.16.100.1
24	18.576399306	KYE_25:43:bc	KYE_13:20:10	ARP	60	172.16.100.254 is at 00:c0:df:25:43:bc
25	18.608326509	172.16.100.1	172.16.100.254	ICMP	98	Echo (ping) request id=0x1724, seq=6/1536, ttl=64 (reply in 26)
26	18.608444612	172.16.100.254	172.16.100.1	ICMP	98	Echo (ping) reply id=0x1724, seq=6/1536, ttl=64 (request in 25)
27	18.668590027	KYE_25:43:bc	KYE_13:20:10	ARP	60	Who has 172.16.100.1? Tell 172.16.100.254
28	18.668610352	KYE_13:20:10	KYE_25:43:bc	ARP	42	172.16.100.1 is at 00:c0:df:13:20:10
29	18.686156760	10.227.20.103	10.227.244.110	DNS	70	Standard query 0xbafc A google.com
30	18.686169052	10.227.20.103	10.227.244.110	DNS	70	Standard query 0xbf06 AAAA google.com
31	19.632323508	172.16.100.1	172.16.100.254	ICMP	98	Echo (ping) request id=0x1724, seq=7/1792, ttl=64 (reply in 32)
32	19.632446291	172.16.100.254	172.16.100.1	ICMP	98	Echo (ping) reply id=0x1724, seq=7/1792, ttl=64 (request in 31)
33	20.010088680	Routerboardc_2b:fa:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:2b:fa:08 Cost = 0 Port = 0x8001
34	22.011808002	Routerboardc_2b:fa:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:2b:fa:08 Cost = 0 Port = 0x8001
35	23.688368413	172.16.100.1	10.227.244.110	DNS	86	Standard query 0x939f A google.com.netlab.fe.up.pt
36	23.688379379	172.16.100.1	10.227.244.110	DNS	86	Standard query 0xaaad AAAA google.com.netlab.fe.up.pt
37	24.013921310	Routerboardc_2b:fa:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:2b:fa:08 Cost = 0 Port = 0x8001
38	24.596624203	172.16.100.1	172.16.101.1	ICMP	98	Echo (ping) request id=0x172e, seq=1/256, ttl=64 (reply in 39)
39	24.596900709	172.16.101.1	172.16.100.1	ICMP	98	Echo (ping) reply id=0x172e, seq=1/256, ttl=63 (request in 38)
40	25.616327847	172.16.100.1	172.16.101.1	ICMP	98	Echo (ping) request id=0x172e, seq=2/512, ttl=64 (reply in 41)
41	25.616565591	172.16.101.1	172.16.100.1	ICMP	98	Echo (ping) reply id=0x172e, seq=2/512, ttl=63 (request in 40)
42	25.950926894	172.16.1.109	172.16.100.1	ICMP	114	Destination unreachable (Host unreachable)
43	25.950954971	172.16.1.109	172.16.100.1	ICMP	114	Destination unreachable (Host unreachable)

Figure 11: Wireshark Capture: ICMP packets, ARP resolution, and Destination Unreachable messages on tux103.

B.5 Experiment 5 - DNS

This appendix provides the supporting information for Experiment 5, including the screenshots and packet captures shared by colleagues Afonso Machado and Luis Arruda. These resources illustrate the DNS configuration and its verification through ICMP and reverse DNS lookup.

Executed Commands

The DNS server was configured on tux103, tux104, and tux102 using the following commands:

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

Listing 17: Commands executed to configure DNS

Wireshark Captures and Terminal Output

Below are the key outputs observed during the experiment:

Wireshark Capture The figure below shows the DNS queries and responses, including the ICMP packets and the reverse DNS lookup during the experiment.

17	0.683871384	10.227.20.43	142.250.184.14	ICMP	98	Echo (ping) request id=0x0e7a, seq=1/256, ttl=64 (reply in 18)
18	0.703433570	142.250.184.14	10.227.20.43	ICMP	98	Echo (ping) reply id=0x0e7a, seq=1/256, ttl=112 (request in 17)
19	0.703536728	10.227.20.43	10.227.20.3	DNS	87	Standard query 0xd8f4 PTR 14.184.250.142.in-addr.arpa
20	0.704033168	10.227.20.3	10.227.20.43	DNS	126	Standard query response 0xd8f4 PTR 14.184.250.142.in-addr.arpa PTR mad41s10-in-f14.1e100.net

Figure 12: Wireshark Capture: DNS queries, ICMP packets, and reverse DNS lookup.

Terminal Output The terminal screenshot below demonstrates the successful resolution of ‘google.com‘ and the resulting ICMP Echo Request and Reply messages.

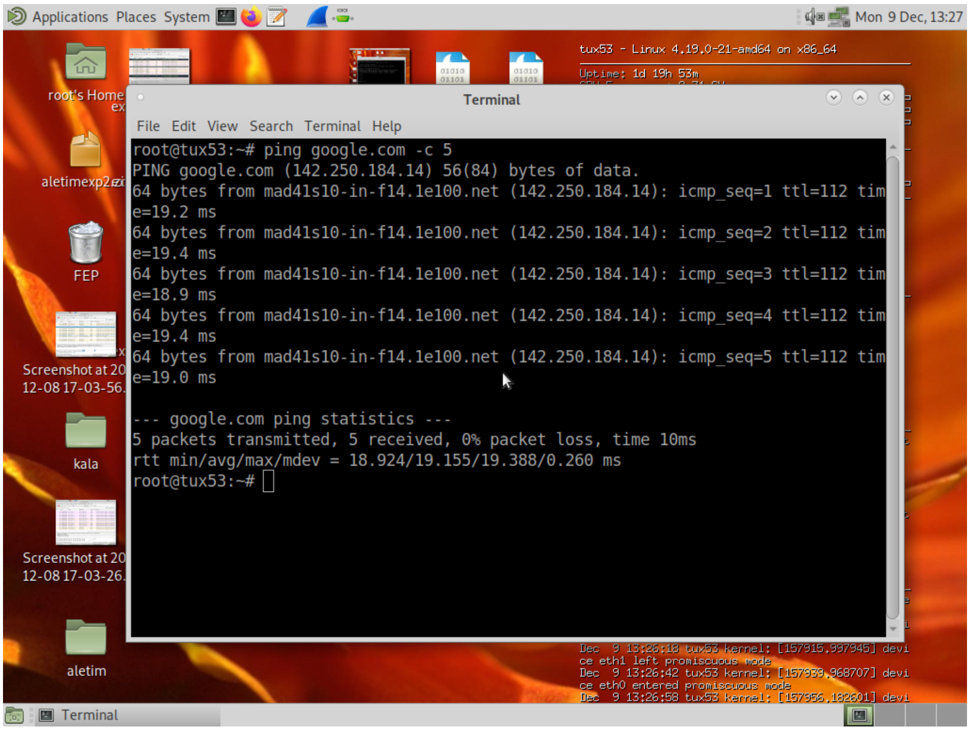


Figure 13: Terminal Output: Pinging ‘google.com‘ and the results.

B.6 Experiment 6 - TCP Connections

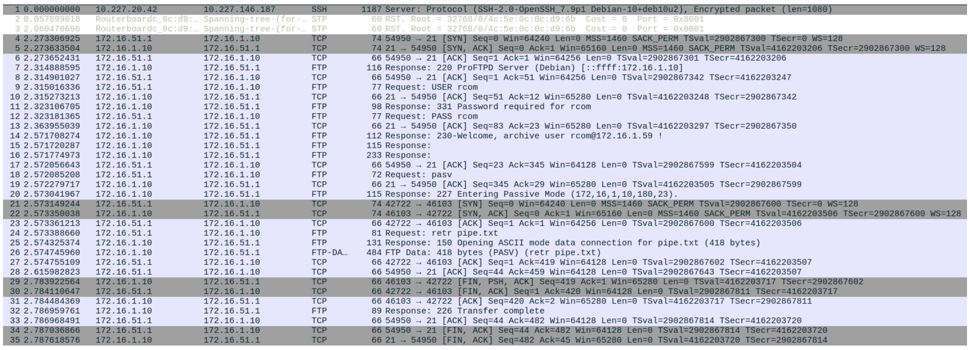
This appendix presents the captured logs and analysis of the network behavior during the download of two files, `ubuntu.iso` and `crab.mp4`, using the FTP application developed for this project. These observations are based on the Wireshark logs provided by Afonso Machado and Luis Arruda from group T07G05, as our experiment faced technical problems.

The logs provide evidence of TCP connection phases, throughput evolution, congestion control mechanisms, and error-handling techniques.

Captured Logs and Observations

The experiment began with the download of `ubuntu.iso` on `tux3`, followed by the initiation of a concurrent download of `crab.mp4` on `tux2`. This setup allowed for the observation of TCP behavior under simultaneous transfers and its response to network contention.

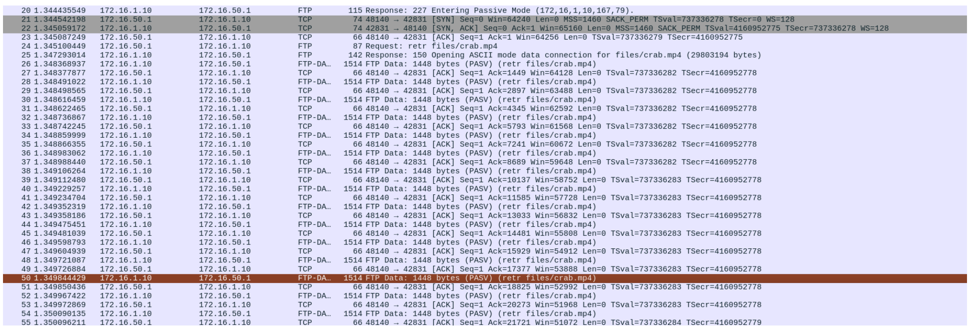
Download of `ubuntu.iso`: Figure **14** captures the logs for the `ubuntu.iso` download. The steady flow of FTP data packets and corresponding acknowledgments is highlighted, demonstrating effective TCP behavior during a single transfer. The logs also provide comparison when a second concurrent transfer is introduced.



1	0.000000000	10.227.20.42	10.227.146.187	SSH	1187	Server: Protocol (SSH-2.0-OpenSSH_7.9p1 Debian-10+deb10u2), Encrypted packet (len=1080)
2	0.057899018	Routerboardc_0c:d9:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/4c:5e:0c:0c:d9:6b Cost = 0 Port = 0x8001
3	2.060470696	Routerboardc_0c:d9:…	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/4c:5e:0c:0c:d9:6b Cost = 0 Port = 0x8001
4	2.273306925	172.16.51.1	172.16.1.10	TCP	74	54950 → 21 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=2902867300 TSecr=0 WS=128
5	2.273633504	172.16.1.10	172.16.51.1	TCP	74	21 → 54950 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=4162203206 TSecr=2902867300 WS=128
6	2.273652431	172.16.51.1	172.16.1.10	TCP	66	54950 → 21 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=2902867301 TSecr=4162203206
7	2.314888595	172.16.1.10	172.16.51.1	FTP	116	Response: 220 ProFTPD Server (Debian) [::ffff:172.16.1.10]
8	2.314901027	172.16.51.1	172.16.1.10	TCP	66	54950 → 21 [ACK] Seq=1 Ack=51 Win=64256 Len=0 TSval=2902867342 TSecr=4162203247
9	2.315016336	172.16.51.1	172.16.1.10	FTP	77	Request: USER rcom
10	2.315273213	172.16.1.10	172.16.51.1	TCP	66	21 → 54950 [ACK] Seq=51 Ack=12 Win=65280 Len=0 TSval=4162203248 TSecr=2902867342
11	2.323106705	172.16.1.10	172.16.51.1	FTP	98	Response: 331 Password required for rcom
12	2.323181365	172.16.51.1	172.16.1.10	FTP	77	Request: PASS rcom
13	2.363955039	172.16.1.10	172.16.51.1	TCP	66	21 → 54950 [ACK] Seq=83 Ack=23 Win=65280 Len=0 TSval=4162203297 TSecr=2902867350
14	2.571708274	172.16.1.10	172.16.51.1	FTP	112	Response: 230-Welcome, archive user rcom@172.16.1.59 !
15	2.571720287	172.16.1.10	172.16.51.1	FTP	115	Response:
16	2.571774973	172.16.1.10	172.16.51.1	FTP	233	Response:
17	2.572056643	172.16.51.1	172.16.1.10	TCP	66	54950 → 21 [ACK] Seq=23 Ack=345 Win=64128 Len=0 TSval=2902867599 TSecr=4162203504
18	2.572085208	172.16.51.1	172.16.1.10	FTP	72	Request: pasv
19	2.572279717	172.16.1.10	172.16.51.1	TCP	66	21 → 54950 [ACK] Seq=345 Ack=29 Win=65280 Len=0 TSval=4162203505 TSecr=2902867599
20	2.573041967	172.16.1.10	172.16.51.1	FTP	115	Response: 227 Entering Passive Mode (172,16,1,10,180,23).
21	2.573149244	172.16.51.1	172.16.1.10	TCP	74	42722 → 46103 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=2902867600 TSecr=0 WS=128
22	2.573350038	172.16.1.10	172.16.51.1	TCP	74	46103 → 42722 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=4162203506 TSecr=2902867600 WS=128
23	2.573361213	172.16.51.1	172.16.1.10	TCP	66	42722 → 46103 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=2902867600 TSecr=4162203506
24	2.573388660	172.16.51.1	172.16.1.10	FTP	81	Request: retr pipe.txt
25	2.574325374	172.16.1.10	172.16.51.1	FTP	131	Response: 150 Opening ASCII mode data connection for pipe.txt (418 bytes)
26	2.574745960	172.16.1.10	172.16.51.1	FTP-DA…	484	FTP Data: 418 bytes (PASV) (retr pipe.txt)
27	2.574755109	172.16.51.1	172.16.1.10	TCP	66	42722 → 46103 [ACK] Seq=1 Ack=419 Win=64128 Len=0 TSval=2902867602 TSecr=4162203507
28	2.615982823	172.16.51.1	172.16.1.10	TCP	66	54950 → 21 [ACK] Seq=44 Ack=459 Win=64128 Len=0 TSval=2902867643 TSecr=4162203507
29	2.783922564	172.16.1.10	172.16.51.1	TCP	66	46103 → 42722 [FIN, PSH, ACK] Seq=419 Ack=1 Win=65280 Len=0 TSval=4162203717 TSecr=2902867602
30	2.784110647	172.16.51.1	172.16.1.10	TCP	66	42722 → 46103 [FIN, ACK] Seq=1 Ack=420 Win=64128 Len=0 TSval=2902867811 TSecr=4162203717
31	2.784484369	172.16.1.10	172.16.51.1	TCP	66	46103 → 42722 [ACK] Seq=420 Ack=2 Win=65280 Len=0 TSval=4162203717 TSecr=2902867811
32	2.786959761	172.16.1.10	172.16.51.1	FTP	89	Response: 226 Transfer complete
33	2.786968491	172.16.51.1	172.16.1.10	TCP	66	54950 → 21 [ACK] Seq=44 Ack=482 Win=64128 Len=0 TSval=2902867814 TSecr=4162203720
34	2.787036866	172.16.51.1	172.16.1.10	TCP	66	54950 → 21 [FIN, ACK] Seq=44 Ack=482 Win=64128 Len=0 TSval=2902867814 TSecr=4162203720
35	2.787618576	172.16.1.10	172.16.51.1	TCP	66	21 → 54950 [FIN, ACK] Seq=482 Ack=45 Win=65280 Len=0 TSval=4162203720 TSecr=2902867814

Figure 14: Wireshark capture logs during `ubuntu.iso` download on `tux3`.

Download of `crab.mp4`: Figure **15** shows the detailed logs for the `crab.mp4` download on `tux2`. This capture highlights the dynamic adjustments in TCP's congestion control mechanisms as it manages simultaneous traffic, with clear evidence of resource sharing and contention between the two connections.



20	1.344435549	172.16.1.10	172.16.50.1	FTP	115	Response: 227 Entering Passive Mode (172,16,1,10,167,79).
21	1.344542198	172.16.50.1	172.16.1.10	TCP	74	48140 → 42831 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=737336278 TSecr=0 WS=128
22	1.345059172	172.16.1.10	172.16.50.1	TCP	74	42831 → 48140 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=4160952775 TSecr=737336278 WS=128
23	1.345087249	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=737336279 TSecr=4160952775
24	1.345100449	172.16.50.1	172.16.1.10	FTP	87	Request: retr files/crab.mp4
25	1.347293014	172.16.1.10	172.16.50.1	FTP	142	Response: 150 Opening ASCII mode data connection for files/crab.mp4 (29803194 bytes)
26	1.348368937	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
27	1.348377877	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=1449 Win=64128 Len=0 TSval=737336282 TSecr=4160952778
28	1.348491022	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
29	1.348498565	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=2897 Win=63488 Len=0 TSval=737336282 TSecr=4160952778
30	1.348616459	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
31	1.348622465	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=4345 Win=62592 Len=0 TSval=737336282 TSecr=4160952778
32	1.348736867	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
33	1.348742245	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=5793 Win=61568 Len=0 TSval=737336282 TSecr=4160952778
34	1.348859999	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
35	1.348866355	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=7241 Win=60672 Len=0 TSval=737336282 TSecr=4160952778
36	1.348983062	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
37	1.348988440	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=8689 Win=59648 Len=0 TSval=737336282 TSecr=4160952778
38	1.349106264	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
39	1.349112480	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=10137 Win=58752 Len=0 TSval=737336283 TSecr=4160952778
40	1.349229257	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
41	1.349234704	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=11585 Win=57728 Len=0 TSval=737336283 TSecr=4160952778
42	1.349352319	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
43	1.349358186	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=13033 Win=56832 Len=0 TSval=737336283 TSecr=4160952778
44	1.349475451	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
45	1.349481039	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=14481 Win=55808 Len=0 TSval=737336283 TSecr=4160952778
46	1.349598793	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
47	1.349604939	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=15929 Win=54912 Len=0 TSval=737336283 TSecr=4160952778
48	1.349721087	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
49	1.349726884	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=17377 Win=53888 Len=0 TSval=737336283 TSecr=4160952778
50	1.349844429	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
51	1.349850436	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=18825 Win=52992 Len=0 TSval=737336283 TSecr=4160952778
52	1.349967422	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
53	1.349972869	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=20273 Win=51968 Len=0 TSval=737336283 TSecr=4160952778
54	1.350090135	172.16.1.10	172.16.50.1	FTP-DA…	1514	FTP Data: 1448 bytes (PASV) (retr files/crab.mp4)
55	1.350096211	172.16.50.1	172.16.1.10	TCP	66	48140 → 42831 [ACK] Seq=1 Ack=21721 Win=51072 Len=0 TSval=737336284 TSecr=4160952779

Figure 15: Wireshark capture logs during `crab.mp4` download on `tux2`.

Throughput and Errors: Figure **16** visualizes the throughput and errors during the `crab.mp4` transfer. The graph shows the initial ramp-up in packet transmission, stabilization, and eventual tapering off. TCP retransmissions and duplicate acknowledgments, indicative of network congestion, are marked in the graph and correlate with the introduction of simultaneous downloads.

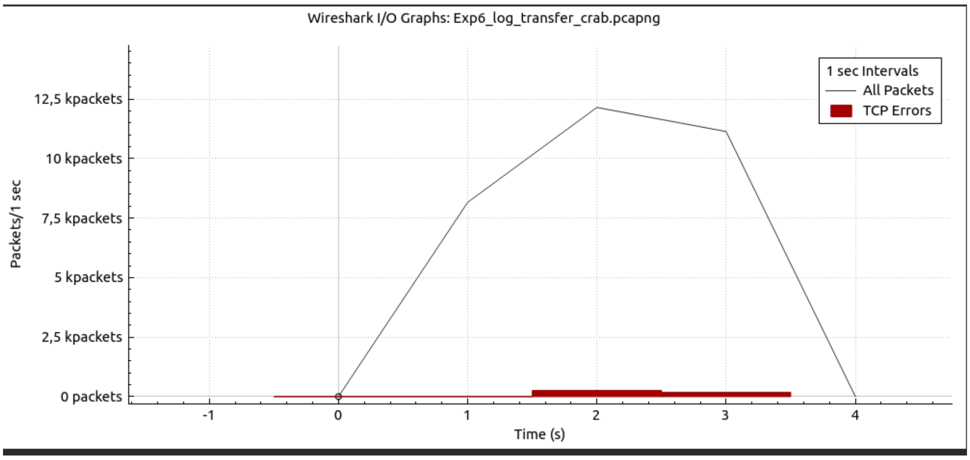


Figure 16: Throughput and TCP error visualization during `crab.mp4` download.

Detailed Packet Analysis: Figures **17** and **18** provide a closer look at the packet exchanges during the `crab.mp4` transfer. These logs highlight duplicate acknowledgments, retransmissions, and the steady flow of data packets.

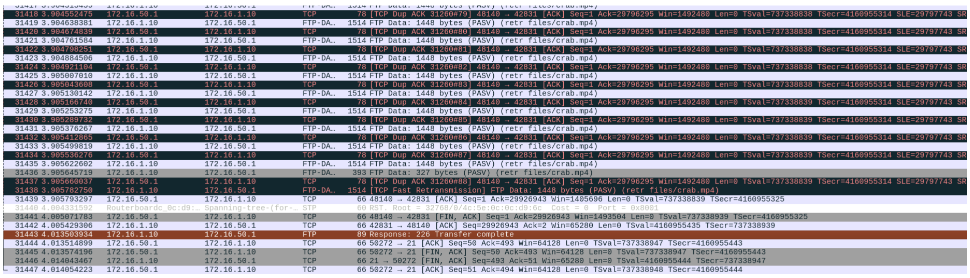


Figure 17: Detailed TCP and FTP interactions during `crab.mp4` download.

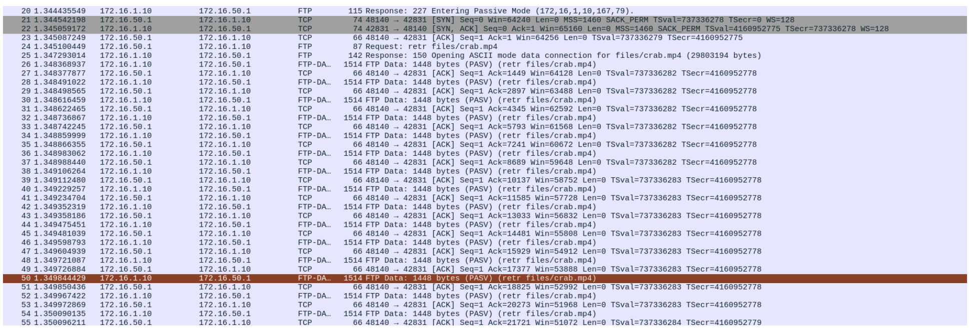


Figure 18: TCP duplicate acknowledgments and retransmissions during `crab.mp4` transfer.

Analysis and Key Observations

1. **TCP Connection Establishment:** Both downloads (`ubuntu.iso` and `crab.mp4`) exhibit proper three-way TCP handshakes (SYN, SYN-ACK, ACK), ensuring reliable communication channels.

2. **Simultaneous Transfers:** The initiation of the `crab.mp4` download on `tux2` during the ongoing `ubuntu.iso` transfer on `tux3` highlights TCP's congestion control mechanisms. Variations in throughput for both downloads underscore TCP's fair bandwidth allocation among concurrent connections.

3. **Throughput Variations:** As illustrated in Figure **16**, the throughput for `crab.mp4` experiences an initial ramp-up, stabilizes, and fluctuates due to network contention and retransmissions. These fluctuations align with TCP's congestion avoidance algorithms.

4. **Error Handling:** TCP's error recovery mechanisms are evident in the retransmissions and duplicate acknowledgments seen in Figures **17** and **18**. These ensure reliable delivery despite packet loss or network congestion.

5. **Connection Termination:** The logs confirm proper termination of both TCP connections with a FIN-ACK handshake, marking the end of each file transfer session.

Conclusion: These captured logs and observations comprehensively document TCP's behavior during simultaneous file transfers. The results highlight the protocol's robustness in managing multiple connections, ensuring reliability, and adapting to varying network conditions.

C Part 2 - Questions and Answers

This section addresses the questions posed for each experiment, providing detailed answers based on the observations, results, and knowledge acquired during the laboratory work.

C.1 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 in a local network. ARP requests are broadcasted to all devices in the subnet to find the MAC address corresponding to a specific IP address, and ARP replies are unicast responses containing the requested MAC address. These packets are essential for enabling communication within a local network. Examples of ARP packets can be seen in Figure **2** in the appendix.

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

   ARP request packets use the sender's MAC and IP addresses in the source fields, while the destination MAC address is set to the broadcast address (FF:FF:FF:FF:FF:FF). The ARP reply contains the MAC and IP addresses of both the sender (the device responding to the request) and the requester, enabling the requester to update its ARP table. These interactions are visible in Figure **2**.

3. **What packets does the ping command generate?**

   The ping command generates ICMP (Internet Control Message Protocol) Echo Request packets, which are sent to the target device. If the target device is reachable, it responds with ICMP Echo Reply packets. Both types of ICMP packets are highlighted in Figure **2**.

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

   For ICMP Echo Request packets, the source MAC and IP addresses correspond to the sender's network interface, and the destination MAC and IP addresses correspond to the target device. For ICMP Echo Reply packets, the source and destination addresses are reversed. This behavior is demonstrated in Figure **2**.

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

   The EtherType field in the Ethernet frame's header identifies the protocol. ARP packets have an EtherType of 0x0806, IPv4 packets use 0x0800, and ICMP is identified by the protocol field in the IPv4 header (protocol number 1). These distinctions are apparent in the captured packets shown in Figure **2**.

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

   The length of an Ethernet frame can be determined by examining the Frame Length field in Wireshark or by reading the size of the captured packet in the network interface buffer. Frame lengths are displayed in Figure **2**.

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

   The loopback interface is a virtual network interface with the reserved IP address 127.0.0.1. It is used for internal testing and communication within the same device, allowing applications to communicate without accessing the physical network.

C.2 Experiment 2 - Implement Two Bridges in a Switch

1. **How to configure bridgeY0?**

   To configure `bridgeY0`, use the MikroTik Switch commands as outlined in the appendix (Section **B.2**). Specifically:

   - Add a new bridge using the command:

   ```
   /interface bridge add name=bridgeY0
   ```

   - Remove the default bridge association for the relevant ports:

   ```
   /interface bridge port remove [find interface=ether5]
   /interface bridge port remove [find interface=ether6]
   ```

   - Associate the required ports with `bridgeY0`:

   ```
   /interface bridge port add bridge=bridgeY0 interface=ether5
   /interface bridge port add bridge=bridgeY0 interface=ether6
   ```

   This ensures that all devices connected to `ether5` and `ether6` communicate within `bridgeY0`, creating a separate broadcast domain.

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

   There are two broadcast domains: one for `bridgeY0` and another for `bridgeY1`. This conclusion is based on the Wireshark logs analyzed in the appendix, which show that broadcast packets (such as ARP requests) generated within one bridge are not propagated to the other bridge. For example:

   - ARP requests from devices in `bridgeY0` (e.g., `tux103`) do not reach devices in `bridgeY1` (e.g., `tux102`).
   - Similarly, ICMP broadcast pings are confined within their respective bridges, as seen in Figure **3**.

   These observations confirm that the network was successfully segmented into two isolated broadcast domains.

C.3 Experiment 3 - Configure a Router in Linux

1. **What routes are there in the tuxes? What are their meaning?**

   The configured routes in the tuxes are as follows:

   - `tux103`: A route to the subnet `172.16.101.0/24` via the gateway `172.16.100.254`.
   - `tux102`: A route to the subnet `172.16.100.0/24` via the gateway `172.16.101.253`.

   These routes indicate that packets destined for the specified subnets should be forwarded through their respective gateways, enabling communication across subnets. Details of these routes are shown in the appendix (Section **B.3**).

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

   Each entry in the forwarding table specifies:

   - The destination subnet or host.
   - The gateway through which packets should be forwarded.
   - The interface (e.g., `eth1`, `eth2`) to be used for forwarding.
   - The metric or cost of the route, if applicable.

   These entries are critical for directing packets to the correct next hop within the network.

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

   ARP requests and replies are observed when a device attempts to resolve the MAC address of the next-hop gateway for a given IP address. For example:

   - `tux103` sends an ARP request for `172.16.100.254`, resolved to the MAC address of `tux104`'s `eth1`.
   - Similarly, `tux102` resolves `172.16.101.253`, which corresponds to `tux104`'s `eth2`.

   These interactions can be seen in the Wireshark captures included in the appendix (Figures **6** and **7**).

4. **What ICMP packets are observed and why?**

   ICMP Echo Request and Echo Reply packets are observed during the ping operations between devices. These packets verify connectivity and measure round-trip times. For example:

   - `tux103` sends ICMP Echo Requests to `tux102`, with replies confirming successful communication through `tux104`.

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

   The ICMP packets observed have:

   - Source and destination IP addresses corresponding to the pinging and target devices (e.g., `172.16.100.1` and `172.16.101.1`).
   - MAC addresses of the source device and the next-hop gateway (e.g., `tux103`'s MAC as the source and `tux104`'s `eth1` as the destination).

   These address pairs ensure that packets are correctly forwarded between subnets, as illustrated in the packet captures.

C.4 Experiment 4 - Configure a Commercial Router and Implement NAT

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

   To configure a static route in a MikroTik router, the following command is used, as detailed in the appendix:

   ```
   /ip route add dst-address=<destination_subnet> gateway=<next_hop_ip>
   ```

   For instance, in this experiment, a static route for the subnet `172.16.100.0/24` via the gateway `172.16.101.253` was added. This ensures that packets destined for the specified subnet are forwarded through the correct gateway.

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:** Packets originating from `tuxY2` are first sent to `tuxY4`, which is not the optimal path. An ICMP redirect is then sent back to `tuxY2`, instructing it to use the optimal gateway (`RC`).
   - **Without ICMP redirect enabled:** `tuxY2` continues to send packets via `tuxY4`, even though this path is suboptimal.

   These behaviors are confirmed through traceroute outputs shown in the appendix (Figures **9** and **10**).

3. **How to configure NAT in a commercial router?**

   NAT (Network Address Translation) is configured using the following command in a MikroTik router:

   ```
   /ip firewall nat add chain=srcnat action=masquerade out-interface=<
       interface_to_internet>
   ```

   In this experiment, NAT was configured to allow packets from internal networks to reach external networks such as the FTP server.

4. **What does NAT do?**

   NAT modifies the source IP address of packets leaving the local network, replacing it with the router's public IP address. This allows multiple devices in a private network to share a single public IP and improves security by masking internal IPs from the outside world.

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

   When NAT is disabled, `tuxY3`'s ping packets reach the FTP server, but the replies fail to return. This occurs because the FTP server attempts to reply directly to `tuxY3`'s private IP address, which is not routable over the internet. This behavior is captured and documented in the appendix (Figure **11**).

C.5 Experiment 5 - DNS

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

   To configure DNS in a host like `tuxY3`, `tuxY4`, or `tuxY2`, you need to specify the DNS server's IP address in the `/etc/resolv.conf` file. For example:

   ```
   nameserver 10.227.20.3
   ```

This configuration points the host to the DNS server `services.netlab.fe.up.pt`, which resolves domain names to IP addresses. The appendix includes examples of successful DNS queries and pings (Figures **12** and **13**).

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

   DNS exchanges two main types of packets:

   - **DNS Query:** Sent from a host to a DNS server, requesting the resolution of a domain name (e.g., `google.com`) to an IP address.
   - **DNS Response:** Sent by the DNS server back to the host, providing the resolved IP address.

   Additionally, reverse DNS lookups allow querying an IP address to retrieve the associated domain name. In this experiment, DNS queries and reverse lookups are captured and shown in the appendix (Figure **12**). This figure highlights:

   - Queries for `google.com`, resulting in the resolution of its IP address.
   - Reverse lookups translating an IP address back to its hostname.

   These packets enable seamless communication between domain names and their corresponding IP addresses.

C.6 Experiment 6 - TCP Connections

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

   The FTP application opens two separate TCP connections:

   - **Control Connection:** Used for transmitting FTP commands and server responses.
   - **Data Connection:** Dynamically established (e.g., in passive mode) to transfer the requested file.

   These connections ensure a clear separation between control signals and data transfer, making the FTP protocol modular and efficient. This is visible in Figures **14** and **15**, which highlight distinct control and data connections during the transfers.

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

   The FTP control information is transmitted over the **control connection**, typically established on port 21. This connection handles commands such as `USER`, `PASS`, `PASV`, `RETR`, and `QUIT`, along with server responses. Figure **15** illustrates how control commands and responses are exchanged during the initialization of the file transfer.

3. **What are the phases of a TCP connection?**

   A TCP connection consists of three main phases:

   - **Connection Establishment:** Managed by the three-way handshake (SYN, SYN-ACK, ACK) to synchronize sequence numbers.
   - **Data Transfer:** Allows bidirectional data exchange with reliability ensured via acknowledgment and retransmission mechanisms.
   - **Connection Termination:** Gracefully closed with the four-way handshake (FIN and ACK exchanged between both ends).

   Figures **14** and **18** capture the handshake, data transfer, and termination phases, showcasing TCP's predictable and robust operations.

4. **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 ensures reliable data transmission:

   - Data packets are sent with a sequence number.
   - The receiver acknowledges received packets using acknowledgment numbers (ACK).
   - Lost packets are retransmitted after a timeout or upon receiving duplicate ACKs.

   Relevant TCP fields include:

   - **Sequence Number:** Identifies the byte position in the data stream.
   - **Acknowledgment Number:** Confirms receipt of data.
   - **Window Size:** Indicates the available buffer space at the receiver.

   Figures **17** and **18** highlight these fields, showing retransmissions and duplicate ACKs during congestion events.

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 congestion control solves congestion using three mechanisms:

   - **Slow Start:** Exponentially increases the congestion window (`cwnd`) until reaching a threshold (`ssthresh`).
   - **Congestion Avoidance:** Linearly increases `cwnd` beyond the threshold.
   - **Fast Recovery:** Retransmits lost packets and adjusts `cwnd` upon detecting congestion (e.g., duplicate ACKs).

   Relevant TCP fields include:

   - **Congestion Window (`cwnd`):** Determines the maximum number of packets that can be sent without acknowledgment.
   - **Acknowledgment (ACK):** Guides `cwnd` adjustments.

   The throughput evolution, shown in Figure **16**, aligns with these mechanisms. Initial exponential growth stabilizes during congestion avoidance, with drops and recovery evident during congestion events.

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

   Yes, the throughput of a TCP connection is affected by a second connection due to TCP fairness:

   - TCP allocates bandwidth equally among active connections, reducing the share of the original connection.
   - Network resources are redistributed, causing delays and reduced acknowledgment frequency for the first connection.

   Figures **14** and **15** demonstrate these effects, with throughput fluctuations and congestion recovery mechanisms responding to the competing connections.