Computer Networks

2nd Project - FTP Application and Network Configuration

Gabriela Rodrigues da Silva - up202206777
Karolina Jedraszek - up202402265

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Introduction

This project is divided into two main parts: the development of a File Transfer Protocol (FTP) application and the configuration of a network through a series of experiments.

- FTP Application **-** The first part of this project focuses on creating an application capable of downloading files from a specified source via the FTP protocol.
- Network Configuration - The second part of the project involves a series of experiments designed to configure a network. This includes tasks such as setting up an IP network, implementing bridges in switches, configuring routers (including a commercial router), implementing Network Address Translation (NAT), and managing topologies with various network connections.

Part 1 - Download Application

Overview

In the first part of the project, the objective was to develop an application that implements the File Transfer Protocol (FTP) with the aid of the Berkeley socket API to establish TCP connections.

File Transfer Protocol

The File Transfer Protocol is a standard network protocol used to transfer files between a client and a server over a network, such as the Internet. It operates on a client-server model and utilizes two separate connections: a **control connection** and a **data connection**.
Although it defines many commands, only a limited number were considered in the final implementation of the application, namely:

1) **USER <username>** - Indicates which user one will log in as. In the application's code, this is set to *anonymous* when not explicitly defined.
2) **PASS <password>** - Indicates the corresponding password. If correct, it finalizes the login process, enabling file transfer. In the application's code, this is set to *anonymous* when not explicitly defined.
3) **PASV** - Instructs the server to enter passive mode, establishing the data connection. The response contains the structure (IP1,IP2,IP3,IP4,PORT1,PORT2), from which the port of the data connection can be calculated as PORT1*255+PORT2.
4) **RETR <filename> -** Begins the file transfer, which will occur through the data connection.
5) **QUIT** - Terminates the control connection**.**

The commands were used in the order indicated. It is also of note that these commands must be terminated by a combination of a Carrier Return ('\r') and a Line Feed ('\n') character.

The FTP responses start with a response code - with codes starting with 4 and 5 indicating errors. If this code is followed by a dash ('-') character, there will be more lines contained in the response. It only terminates when the code is followed by a space and any string of characters.

Application Architecture

The FTP application is divided into three modules:

1) **main.c** - The entry point of the program. Contains the general sequence of FTP, as well as validation of command line parameters. If no command line parameters are present, they must be input during runtime.
2) **ftp.h/.c** - The FTP protocol implementation. Defines the following functions:
    a) **splitURL -** obtains the username, password, host name and file name from an FTP URL.
    b) **get_response** - Obtains bytes using TCP functions, gathering the message until its ending is detected.
    c) **interpret_pasv_response** - Obtains the number of the data connection port from a response to the PASV command.
    d) **check_is_error_code** - Checks if a given message starts with an error code.
    e) **ftp_init** - Sends the USER, PASS, and PASV commands, handling the responses accordingly and leaving the application in a download-ready state.
    f) **ftp_download_file** - Sends the RETR command and reads from the data connection until the transfer finishes, closing the download connection.
    g) **ftp_close** - Sends the QUIT command and closes the control connection.
3) **tcp.h/.c** - The TCP protocol implementation. Defines the following functions:
    a) **tcp_init** - Creates a socket and attempts to connect to the supplied host and port. Configures the socket to be non-blocking, in order to avoid infinite loops and to better handle error states. May be used multiple times to create different connections.
    b) **tcp_write -** Writes an array of bytes to the given socket.
    c) **tcp_read** - Reads up to a certain number of bytes from a given socket, with a timeout functionality in case there are none.
    d) **tcp_close** - Closes a connection and deallocates the corresponding memory.

Demonstration

To demonstrate a successful download, the application was downloaded, compiled and executed in Tux Y3 as presented in experiment 6. The results can be observed in Annex 2.

Part 2 - Network Configuration

Overview

The second part of the project involves progressively creating the network configuration shown in Annex 3. This includes configuring the IP addresses and routing tables for three computers, referred to as Tux 2, Tux 3, and Tux 4, as well as a MikroTik switch and router.
Note that, since these experiments were carried out throughout multiple weeks and on multiple computers, values that should have remained constant, such as MAC addresses or ethernet ports selected, may vary between examples.

Experiment 1 - Configuring an IP Network

This experiment consists only of connecting Tux 3 and Tux 4 through the switch and observing the network packets circulating between them.

In order to communicate the desired commands, cables were connected from the one of the Tux computers' serial port to the switch's serial port. On the connected Tux, GTKterm, set to a baud rate of 115200, was used to send the ***/system reset-configuration*** command, which resets the switch configuration, though requiring a username ("admin") and password (pressing enter). No further configuration was required at this point, since by default the two ethernet ports chosen for the computers belong to the same bridge and, therefore, packets from one are able to reach the other.

In the operating system console of both Tux3 and Tux4, the ***systemctl restart networking*** command was inserted to clear previous configurations. ***ifconfig eth1 up*** was used to activate the network interfaces on Tux3 and Tux4, alongside ***ifconfig eth1 172.16.Y0.1*** and ***ifconfig eth_ IP 172.16.Y0.254*** respectively, to assign IP addresses*.*

The loopback interface was used for internal testing without needing an external network or devices, through the IP address 127.0.0.1 - the loopback address.

The ***route -n*** command showed the routing table of the computer. Generally, no further configuration was necessary, as the entry associating the network 172.16.Y0.0/24 with the gateway 0.0.0.0 and the selected ethernet port was added automatically. However, it can be manually added with the command ***ip route add 172.16.Y0.0/24 dev <ethernet port>*** (here, the ethernet port is either eth1 or eth2, depending on configuration).

At this stage, two particular types of network packets, which could be observed through the Wireshark application, alongside with their contents, origins and destinations, lengths, etc, were considered of particular interest:

- ARP packets
- ICMP packets

The Address Resolution Protocol (ARP) is used to dynamically map variable IP addresses to fixed MAC addresses in the network. The ARP Request is broadcast using a destination MAC address of all 1's, while the ARP Reply, originating on the machine with the desired IP, uses the sender's MAC address as a destination for direct communication. The desired IP is included in the message.

The ***arp -a*** command displays the ARP table. ***arp -d <ip_address>*** was used to delete ARP mappings, triggering the transmission of an ARP Request during the experiment.

The Internet Control Message Protocol (ICMP) is a network layer protocol used mostly for diagnostic purposes. One of its commands, the ***ping*** command, generates ICMP Echo Request/Reply packets for testing connectivity.

Finally, a successful ping between Tux3 and Tux4 confirmed that the network was set up correctly. In this initial attempt, the Ping Request packet had Tux4's IP (172.16.50.254) and

MAC (00:c0:df:25:24:5b), and the Ping Reply had Tux3's IP (172.16.50.1) and MAC (00:08:54:71:74:10). However, eth2 was used on Tux 4.

Experiment 2 - Implementation of two bridges in a switch

This second experiment introduces the need to configure separate bridges: one containing Tux 3 and Tux 4, and another containing only Tux 2. A bridge is a device or functionality used to connect and manage traffic between two or more network segments and is responsible for forwarding data based on MAC addresses. Only computers in the same bridge are able to exchange messages because bridges provide isolation at a Data Link Layer level.

To configure them, the following commands were input in GTKTerm, after connecting one of Tux 2's ethernet ports to the switch:

- ***/interface bridge port remove [find interface=ether<port_number>]*** - removes the given port from any existing bridge. This command should be used for all ports connected to the computers.
- ***/interface bridge add name=bridgeY0*** and ***/interface bridge add name=bridgeY1*** - creates the new bridges.
- ***/interface bridge port add bridge=bridgeY0 interface=ether<port_number>*** - with port numbers corresponding to the ones Tux 3 and Tux 4 are connected to.
- ***/interface bridge port add bridge=bridgeY1 interface=ether<port_number>*** - with port numbers corresponding to the one Tux 2 is connected to. This assigns the ports to the bridges.

Additionally, Tux 2's IP address is configured with instructions similar to those used for Tux 3 and 4 above: ***systemctl restart networking, ifconfig eth1 up*** and ***ifconfig eth1 172.16.Y0.1***.
The experiment is successful if Tux 4 is able to ping Tux 3, but not Tux 2; Tux 3 is able to ping Tux 4 but not Tux 2 and Tux 2 is not able to ping anyone (except itself) - ***ping -b 172.16.Y0.255*** *and* ***ping -b 172.16.Y1.255*** were used to check connectivity. This happens because, at this point, there are 2 broadcast domains. This is especially apparent in the ARP Requests, which do not receive replies, as seen in the logs.

Experiment 3 - Configuration of a Router in Linux

The third experiment requires transforming Tux 4 into a router, allowing it - as well as Tux 3, after the addition of the proper route - to communicate with Tux 2.

To do this a new ethernet interface (eth2) is activated and connected to the switch, to a port added to bridge Y1:

- ***ifconfig eth2 up*** and ***ifconfig eth2 172.16.Y1.253*** is inserted into Tux 4's console.
- ***/interface bridge port remove [find interface=ether<port_number>]*** and ***/interface bridge port add bridge=bridgeY0 interface=ether<port_number>*** are used in the switch, through GTKTerm.

Additionally, IP forwarding was enabled (***sysctl net.ipv4.ip_forward=1***), so that Tux 4 automatically forwarded packets between its interfaces. The command ***sysctl***

net.ipv4.icmp_echo_ignore_broadcasts=0 allowed broadcast ping responses. The ***ifconfig*** command permits comparisons between **eth1** and **eth2** on Tux 4, as it prints the corresponding MAC and IP addresses. The MAC address is different because MAC addresses are associated with network interfaces, not computers themselves.

To allow Tux 3 to communicate with Tux 2, the command ***route add -net 172.16.Y1.0/24 gw 172.16.Y0.254*** on Tux 3 was used, alongside ***route add -net 172.16.Y0.0/24 gw 172.16.Y1.253*** on Tux 2.

This added entries to the respective routing tables, indicating that packets with a destination IP address in network ***172.16.Y1.0/24*** or ***172.16.Y0.0/24,*** respectively, should be sent to Tux 4. In this table, entries with destination 0.0.0.0 represent packets sent within the same subnet. Here, Tux 4 belongs to two subnets.

Each forwarding table entry specifies the destination network, gateway, subnet mask, and the interface to use. The **Genmask** (subnet mask) determines if an address belongs to the destination network using a bitwise AND.

To verify if the network was in accordance with the instructions, ICMP packets - Ping Requests and Ping Replies - were exchanged between Tux 3 and Tux2 via Tux 4. Moreover, after clearing the ARP table, ARP messages were exchanged to discover MAC addresses, as seen in Annex 4 - Experiment 3. Each ICMP packet carried a source and destination IP address and a source and destination MAC address:

1. Ping Requests:
   - From Tux3 (**172.16.Y0.1**) To Tux 2 (**172.16.Y0.1**)
2. Ping Replies:
   - From Tux2 (**172.16.Y1.1**) to Tux3 (**172.16.Y0.1**)

The IP addresses present in ICMP packets are those of the start and end points of the communication, while MAC addresses correspond to the devices sending and receiving the packet at a given moment, reflecting the communication path between the three computers. These MAC addresses are analysed in the Annex.

Experiment 4 - Configuration of a static router and NAT implementation

In the fourth experiment, a commercial router is added to subnetwork 172.16.Y1.0/24, allowing for communication with the netlab server. Its console is accessed similarly to that of the switch: through a serial port.

Also similarly to the switch, ***/system reset-configuration*** reset its configurations. Afterwards, ***/ip address add address=172.16.1.Y1/24 interface=ether1*** and ***/ip address add address=172.16.Y1.254/24 interface=ether2*** assigned IP addressed to the ports used. With this, port 1 of the router was connected to the network with the server, while port 2 was connected to the network 172.16.Y1.0/24 and the ping command could be used to get feedback from Tux 4 and Tux 2.

To reach Tux 3, the router's own routing table needed to be updated with ***/ip route add dst-address=172.16.Y0.0/24 gateway=172.16.Y1.253***, which allowed it to redirect packets headed for network **172.16.Y0.0/24** to Tux 4. All the Tux's routing tables were updated to include the 172.16.10.0/24 network:

- On Tux 3: ***route add -net 172.16.1.0/24 gw 172.16.Y0.254*** (172.16.Y0.254 is Tux 4)
- On Tux 4, ***route add -net 172.16.1.0/24 gw 172.16.Y1.254*** (172.16.Y1.254 is the commercial router)
- On Tux 2, ***route add -net 172.16.1.0/24 gw 172.16.Y1.254***

While this allowed all devices to contact any other, in order to better understand the flow the experimented indicated that the route to **172.16.Y0.0/24** (the network where Tux 3 is) via Tux 4 in Tux 2: ***ip route del 172.16.Y0/24 via 172.16.Y1.254 dev eth1*** (or whatever ethernet port was chosen for Tux 2). While the ***ping*** command now failed, the ***traceroute*** command was used.

traceroute is a network diagnostic tool used to track the path that packets take from one computer to a destination, providing information about each hop along the way. This is achieved by sending a sequence of packets with a progressively increasing Time-to-Live (TTL) value.

Without the correct route to Tux 3's network, the traceroute packets were directed to a second "hidden" internet router (**10.227.20.254**), which was automatically configured as the system's default gateway on eth0.

After re-adding the route and using the command ***sysctl net.ipv4.conf.all.accept_redirects=1***, which enables the system to accept ICMP Redirect messages for all interfaces, the traceroute packets were able to reach Tux 3.

The last part of this experiment demonstrated the Importance of Network Address Translation (NAT). NAT is a mechanism that enables the computers in a private network to share a single public IP address, through a router that matches their private addresses to its ports, allowing outside entities to reach these computers through this same corresponding port.

It can be disabled by writing ***/ip firewall nat disable 0*** on the router's console, through the serial port. In this state, a **ping** from Tux 3 to the server was performed. However, the replies from the server are unable to reach Tux 3, as they do not contain enough information in themselves for the router to redirect them to the right computer.

To re-enable NAT according to the previous configurations, the command ***/ip firewall nat enable 0*** can be used, while commands such as ***/ip firewall nat add chain=srcnat action=masquerade out-interface=<interface>*** or ***/ip firewall nat add chain=dstnat protocol=tcp dst-port=<port> action=dst-nat to-addresses=<private_ip> to-ports=<privated_port>*** can be used to define its behaviour in more detail, though they were not specifically contemplated in this experiment.

Experiment 5 - DNS

The Domain Name System is a way of translating human-readable domain names (like `www.example.com`) into IP addresses that computers can use to identify each other on a network. The DNS server used to consult this information can be configured in a specific system file - **/etc/resolv.conf** - by adding the line:

- ***nameserver: <IP address of the DNS server>***

During this experiment, in order to view DNS packets, the command ***ping google.com*** was executed in Tux 3. In this exchange, two kinds of messages were observed:

- Standard Queries - requests information about a domain name or record, by specifying its name and type. It can be recursive or iterative.
- Standard Query Responses - provides the requested information (or an error code), indicating which query it is answering, through a series of record entries.

Experiment 6 - TCP connections

The goal of the last experiment was to run the FTP application from Part 1 with the network configuration established thus far, seamlessly bringing together and showcasing everything accomplished throughout the project.

Starting in Tux 3, a simple download initiated through the FTP URL ***ftp:/rcom:rcom@ftp.netlab.fe.up.pt/pipe.txt*** *was* tested and its packets were captured. Later, the procedure was repeated; however, in the middle of the transfer a new capture was started on Tux 2.

As explained in Part 1, the application established two TCP connections. A TCP connection progresses through three main phases: connection establishment (via a three-way handshake), data transfer, and termination (via a four-way handshake).

The three-way handshake ensures that both client and server are ready to communicate by:

- The client sending a SYN packet to the server to indicate its intention to start a connection and provide an initial sequence number.
- The server responding with a SYN-ACK packet, acknowledging the client’s SYN and sending its own initial sequence number
- The client sending an ACK packet back to the server, acknowledging the server's SYN-ACK

The four-way handshake is used to terminate a TCP connection gracefully. It ensures that both the client and server agree to close the connection:

- The client sends a FIN packet to the server, indicating it has finished sending data.
- The server responds with an ACK packet, acknowledging the client’s FIN.
- The server then sends its own FIN packet to signal that it has also finished sending data.
- The client replies with an ACK packet, acknowledging the server’s FIN.

As the focus was on writing client code, the aim was to connect to an existing host rather than listen for incoming connections.

To ensure reliable data transmission over unreliable networks, TCP uses ARQ (Automatic Repeat reQuest) to detect and retransmit lost, corrupted, or out-of-order packets if no ACK is received within a set time. Frame headers help reconstruct the sequence and identify missing packets, while cyclic redundancy checks detect errors in received packets.

TCP congestion control dynamically adjusts the data transmission rate to balance efficiency and reliability. A congestion window determines how much data can be sent before requiring an acknowledgment. Throughput is influenced by round-trip time and the rate of ACKs received.

The slow-start phase begins by exponentially increasing the transmission rate until a timeout happens. Then, the congestion avoidance phase begins, where the rate grows linearly but halves upon a timeout. Mechanisms like fast retransmission and recovery triggered by three duplicate ACKs can retransmit lost packets without waiting for a timeout, improving efficiency.

Key TCP header fields for congestion control include:

- Sequence number: Tracks segment positions to detect lost packets.
- Acknowledgment number: Indicates the next expected byte.
- Window size: Adjusts data flow based on buffer availability.
- Flags: Signal phases like slow start or retransmission.
- Options: Convey extra information, enhancing flexibility.

Normally, two TCP connections can disrupt each other, as they share network resources and bandwidth, although without causing corruption of data noticeable to a higher layer, merely slowing the rate at which said data arrives.

Moreover, the congestion control mechanism adjusts to ensure fairly distributed bandwidth. When a second connection starts its slow-start phase, it increases network traffic, causing potential congestion and forcing the original connection to reduce its transmission rate.

However, in the setup of this experiment, while both downloads were completed successfully and simultaneously, there was no significant mutual interference, as the network might not have been operating in its maximum capacity.

Conclusions

During the development of this project, a simple remote download application using the File Transfer Protocol was successfully developed. Additionally, practical experience was gained in the basic tasks required to set up a small local network: resetting switch, router and operating system configurations; connecting ethernet cables; defining IP addresses and gateways; creating and assigning switch bridges and configuring a router in order to connect to a server.

Overall, we believe this process was relevant to our understanding of digital communication and how real-world systems are set up in order to provide services that are usually taken for granted.

References

To inform our development of this project, we used the slides provided on Moodle as the main reference.
Nevertheless, the following sources were also consulted:

RFC 959 - File Transfer Protocol
Address Resolution Protocol - Wikipedia
Internet Control Message Protocol - Wikipedia
File Transfer Protocol - Wikipedia

Annexes

Annex 1 - FTP Application Code

main.c

```
 #include <stdio.h>
#include <string.h>
#include "ftp.h"
#include <stdlib.h>
#define MIN_LENGTH 8
#define MAX_URL_LENGTH 600
char *createURL()
{
    char *url = (char *)malloc(MAX_URL_LENGTH * sizeof(char));
    if (url == NULL)
    {
        printf("URL creation: Memory allocation failed! \n");
        return NULL;
    }
    char bufferU[100];
    bufferU[99] = '\0';
    char bufferP[100];
    bufferP[99] = '\0';
    char bufferH[100];
    bufferH[99] = '\0';
    char bufferF[200];
    bufferF[199] = '\0';

    printf("Username: ");
    if (scanf("%99s", bufferU) != 1)
    {
        printf("URL creation: Error reading username!\n");
        return NULL;
    }
    printf("Password: ");
    if (scanf("%99s", bufferP) != 1)
    {
        printf("URL creation: Error reading password!\n");
        return NULL;
    }
    printf("Host: ");
    if (scanf("%99s", bufferH) != 1)
    {
```

```c
        printf("URL creation: Error reading host!\n");
        return NULL;
    }

    printf("File: ");
    if (scanf("%199s", bufferF) != 1)
    {
        printf("URL creation: Error reading file!\n");
        return NULL;
    }

    sprintf(url, "ftp://%s:%s@%s/%s", bufferU, bufferP, bufferH, bufferF);
    return url;
}

int main(int argc, char *argv[])
{
    char *url;
    if (argc < 2)
    {
        url = createURL();
        if(!url){
            printf("Error creating URL!\n");
            return 1;
        }
        printf("Created url: %s.\n",url);
    }
    else
    {
        printf("Supplied URL found!\n");
        url = argv[1];
    }
    if (strlen(url) < MIN_LENGTH)
    {
        printf("Usage: ./<executable name> <URL> [<local filename>] or ./<executable
name>\n");
        return -1;
    }

    printf("Argument verification successful!\n");

    if (ftp_init(url))
    {
        printf("Error: Couldn't init FTP layer!\n");
        return -1;
    }

    if (argc == 3)
    {
        printf("Local file name identified. Saving result to file %s",argv[2]);
        if (ftp_download_file(argv[2]))
        {
            printf("Error: couldn't download file!\n");
            return -1;
        }
    }
    else
    {
        printf("Saving result to file download");
        if (ftp_download_file("download"))
        {
            printf("Error: couldn't download file!\n");
            return -1;
        }
    }

    if (ftp_close())
    {
        printf("Error: couldn't close connection!\n");
        return -1;
    }
    if (argc < 2)
    {
        printf("Freeing URL memory!\n");
        free(url);
    }
    printf("Terminating successfully!\n");
    return 0;
}
```

tcp.c

```
#include "tcp.h"

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <netdb.h>
#include <fcntl.h>
#include <string.h>
#include <netinet/in.h>
#include <stdio.h>
#include <time.h>
#include <errno.h>
typedef struct Connection
{
    char *addressName;
    int port;
    int sockfd;
} Connection;

#define RANDOM_LONG_ENOUGH_NUMBER 200

// initializes a connection to a server by creating a socket
Connection *tcp_init(char *addressName, unsigned int port)
{
    Connection *connection = (Connection *)malloc(sizeof(Connection));
    if (!connection)
    {
        perror("TCP Error - malloc()! ");
        return 0;
    }

    char portStr[10];
    snprintf(portStr, sizeof(portStr), "%d", port);
```

```c
struct addrinfo hints, *a;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

if (getaddrinfo(addressName, portStr, &hints, &a) != 0)
{
    perror("TCP Error - getaddrinfo()! ");
    free(connection);
    return 0;
}

connection->sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (connection->sockfd < 0)
{
    perror("TCP Error: socket()! ");
    freeaddrinfo(a);
    free(connection);
    return 0;
}
printf("Socket opened!\n");

int flags = fcntl(connection->sockfd, F_GETFL, 0); // non blocking socket
if (flags == -1 || fcntl(connection->sockfd, F_SETFL, flags | O_NONBLOCK) == -1)
{
    perror("TCP Error: fcntl! ");
    close(connection->sockfd);
    freeaddrinfo(a);
    free(connection);
    return 0;
}

// start of connecting
int status = connect(connection->sockfd, a->ai_addr, a->ai_addrlen);
if (status < 0 && errno != EINPROGRESS)
{
    perror("TCP Error: connect()!");
    close(connection->sockfd);
    freeaddrinfo(a);
    free(connection);
    return 0;
}

if (status != 0) // connection in progress
{
    struct timeval timeout;
    timeout.tv_sec = 2;
    timeout.tv_usec = 0;

    fd_set writefds;
    FD_ZERO(&writefds);
    FD_SET(connection->sockfd, &writefds);

    status = select(connection->sockfd + 1, NULL, &writefds, NULL, &timeout);
    if (status <= 0)
    {
        if (status == 0)
        {
            fprintf(stderr, "TCP Error: connection timeout!\n");
            }
            else
            {
                perror("TCP Error - select() failed!");
            }
            close(connection->sockfd);
            freeaddrinfo(a);
            free(connection);
            return 0;
        }

        // checking if the socket is writable
        int optval;
        socklen_t optlen = sizeof(optval);
        if (getsockopt(connection->sockfd, SOL_SOCKET, SO_ERROR, &optval, &optlen) < 0 ||
optval != 0)
        {
            fprintf(stderr, "TCP Error: connect() failed with error %d\n", optval);
            close(connection->sockfd);
            freeaddrinfo(a);
            free(connection);
            return 0;
        }
    }

    printf("Connection established!\n");

    freeaddrinfo(a);

    connection->addressName = strdup(addressName); // returns NULL in case of an error,
    if (!connection->addressName)
    {
        perror("TCP Error - strdup()!");
        close(connection->sockfd);
        free(connection);
        return 0;
    }
    connection->port = port;

    return connection;
}

// sends data over the established TCP connection
int tcp_write(Connection *connection, char *message, unsigned int numBytes)
{
    int bytes = write(connection->sockfd, message, numBytes);
    if (bytes != numBytes)
    {
        if (bytes == 0)
        {
            perror("TCP Error - write()! ");
            return 1;
        }
        printf("TCP Error: Couldn't write all bytes!");
        return 2;
    }

    return 0;
}
```

```c
// reads data from the established TCP connection into a buffer
int tcp_read(Connection *connection, char *outBuffer, unsigned int outBufferCapacity)
{

    if (connection == NULL || outBuffer == NULL || outBufferCapacity <= 0)
    {
        printf("TCP Error: wrong read parameters!\n");
        return -1; // invalid input
    }

    fd_set readfds;
    FD_ZERO(&readfds);
    FD_SET(connection->sockfd, &readfds);

    struct timeval timeout;
    timeout.tv_sec = 2;
    timeout.tv_usec = 0;

    int status = select(connection->sockfd + 1, &readfds, NULL, NULL, &timeout);
    if (status > 0)
    {
        ssize_t bytes = read(connection->sockfd, outBuffer, outBufferCapacity);
        if (bytes < 0)
        {
            perror("TCP Error: read()! ");
        }
        else
        {
        }
        return bytes;
    }
    else if (status == 0)
    {
        printf("TCP Failure: Timeout waiting for data!\n");
    }
    else
    {
        perror("TCP Error: select()! ");
        return -1;
    }
    return 0;
}

int tcp_close(Connection *connection)
{
    if (close(connection->sockfd) < 0)
    {
        free(connection->addressName);
        free(connection);
        perror("TCP Error: close()! ");
        return 1;
    }
    printf("Socket closed!\n");
    free(connection->addressName);
    free(connection);
    return 0;
}
```

ftp.c

```c
#include "ftp.h"
#include "tcp.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#define MAX_FIELD_SIZE 200
#define MAX_COMMAND_SIZE 300
#define BATCH_SIZE 512
#define FTP_CONNECTION_PORT 21
#define FTP_END_OF_LINE "\r\n"

static char user[MAX_FIELD_SIZE];
static char pass[MAX_FIELD_SIZE];
static char host[MAX_FIELD_SIZE];
static char file[MAX_FIELD_SIZE];

int splitURL(char *url)
{

    if (
        url[0] != 'f' ||
        url[1] != 't' ||
        url[2] != 'p' ||
        url[3] != ':' ||
        url[4] != '/' ||
        url[5] != '/')
    {
        printf("URL Parse Error: Wrong URL starting characters!\n");
        return 1;
    }

    unsigned int currentIndex = 6;

    for (unsigned int targetIndex = 0; currentIndex < MAX_FIELD_SIZE; targetIndex++,
currentIndex++)
    {
        user[targetIndex] = url[currentIndex];
        if (url[currentIndex] == ':' || url[currentIndex] == '/' || url[currentIndex] == '\0')
        {
            user[targetIndex] = '\0';
            break;
        }
    }
    if (currentIndex >= MAX_FIELD_SIZE)
    {
        printf("URL Parse Error: Length exceeded!\n");
        return 1;
    }

    printf("URL Parse - read user: %s ?...\n", user);

    if (url[currentIndex] == '/')
    {
        strcpy(host, user);
        strcpy(user, "anonymous");
        strcpy(pass, "anonymous");
        printf("URL Parse: no user found, anonymous assumed, host set to: %s.\n", host);
    }
    else if (url[currentIndex] == ':')
    {
        printf("URL Parse: user interpreted successfully.\n");

        currentIndex++;
        for (unsigned int targetIndex = 0; currentIndex < MAX_FIELD_SIZE; targetIndex++,
currentIndex++)
        {
            pass[targetIndex] = url[currentIndex];
            if (url[currentIndex] == '@' || url[currentIndex] == '/' || url[currentIndex] ==
'\0')
            {
                pass[targetIndex] = '\0';
                break;
            }
        }

        printf("URL Parse: read password: %s.\n", pass);

        if (currentIndex >= MAX_FIELD_SIZE)
        {
            printf("URL Parse Error: Length exceeded!\n");
            return 1;
        }

        if (url[currentIndex] != '@')
        {
            printf("URL Parse Error: Wrong password termination code %c!\n",
url[currentIndex]);
            return 1;
        }

        currentIndex++;
        for (unsigned int targetIndex = 0; currentIndex < MAX_FIELD_SIZE; targetIndex++,
currentIndex++)
        {
            host[targetIndex] = url[currentIndex];
            if (url[currentIndex] == '/' || url[currentIndex] == '\0')
            {
                host[targetIndex] = '\0';
                break;
            }
        }
        printf("URL Parse: read host %s.\n", host);

        if (url[currentIndex] != '/')
        {
            printf("URL Parse Error: Wrong host termination code %c\n", url[currentIndex]);
            return 1;
        }
    }
    else
    {
        printf("URL Parse Error: Wrong user end code %c\n", url[currentIndex]);
        return 1;
    }

    currentIndex++;
    for (unsigned int targetIndex = 0; currentIndex < MAX_FIELD_SIZE; targetIndex++,
currentIndex++)
    {
        file[targetIndex] = url[currentIndex];
        if (url[currentIndex] == '\0')
        {
            file[targetIndex] = '\0';
            break;
        }
    }
    printf("URL Parse: read file %s.\n", file);

    return 0;
}

Connection *commandConnection;
Connection *downloadConnection;

enum MESSAGE_STATE
{
    READING_NUMBERS,
    LAST_LINE,
    NOT_LAST_LINE,
    NOT_LAST_LINE_FOUND_NEW_LINE
};

int get_response(Connection *connection, char **message)
{
    unsigned int mssgSize = 100, lastMessageIndex = 0;
    message[0] = (char *)malloc(mssgSize * sizeof(char));

    char t = 0;
    const char timeout = 3;

    printf("Reading FTP response:\n\n");
    char running = 1;
    enum MESSAGE_STATE state = READING_NUMBERS;
    while (running)
    {
        if (t >= timeout)
        {
            printf("Response Read Error: response timeout!\n");
            return 1;
        }
        t++;

        char buffer[BATCH_SIZE];
        int ret = tcp_read(connection, buffer, BATCH_SIZE);
        if (ret == -1)
        {
            printf("Response Read Error: TCP connection failed!\n");
            return 1;
        }
        if (ret)
        {
            t = 0;
            for (unsigned int i = 0; i < ret && running; i++)
            {
```

```
            message[0][lastMessageIndex] = buffer[i];
            lastMessageIndex++;
            if (lastMessageIndex >= mssgSize)
            {
                mssgSize *= 2;
                message[0] = realloc(message[0], mssgSize * sizeof(char));
                if (!message[0])
                {
                    fprintf(stderr, "Response Read Error: Memory allocation failed!\n");
                    return 1;
                }
            }

            printf("%c", buffer[i]);

            switch (state)
            {
            case READING_NUMBERS:
                if (buffer[i] == ' ')
                {
                    state = LAST_LINE;
                }
                else if (buffer[i] == '-')
                {
                    state = NOT_LAST_LINE;
                }
                else if (buffer[i] == '\n')
                {
                    running = 0;
                }
                else if (buffer[i] > '9' && buffer[i] < '0')
                {
                    return -1;
                }
                break;
            case LAST_LINE:
                if (buffer[i]=='\n')
                {
                    running=0;
                }

                break;
            case NOT_LAST_LINE:

                if (buffer[i]=='\n')
                {
                    state=NOT_LAST_LINE_FOUND_NEW_LINE;
                }
                break;
            case NOT_LAST_LINE_FOUND_NEW_LINE:
                if (buffer[i]>='0'&&buffer[i]<='9')
                {
                    state=READING_NUMBERS;
                }else{
                    state=NOT_LAST_LINE;
                }
                break;
            }
        }
    }
```

```c
    }
    printf("\n");
    return 0;
}

int interpret_pasv_response(char *message, unsigned int *port, char *serverCode)
{
    int h1, h2, h3, h4, p1, p2;

    char *start = strrchr(message, '(');
    char *end = strrchr(message, ')');

    if (start == NULL || end == NULL || start > end)
    {
        printf("PASV: response has a wrong format.\n");
        return 1;
    }

    if (sscanf(start + 1, "%d,%d,%d,%d,%d,%d", &h1, &h2, &h3, &h4, &p1, &p2) != 6)
    {
        printf("PASV Error: couldn't parse the response.\n");
        return 1;
    }

    sprintf(serverCode, "%u.%u.%u.%u", h1, h2, h3, h4);
    *port = (p1 * 256) + p2;

    return 0;
}

enum SpecialState
{
    STATE_NONE = 0,
    STATE_LOGIN = 1,
    STATE_RETR = 2,
};

int check_is_error_code(char *message, char specialstate)
{
    return message[0] != '2' && !(specialstate == STATE_LOGIN && message[0] == '3') &&
!(specialstate == STATE_RETR && message[0] == '1');
}

int ftp_init(char *url)
{

    if (splitURL(url))
    {
        printf("Init Error: invalid URL!\n");
        return 1;
    }
    printf("Init: URL parsed successfully!\n");

    char *message;

    if (!(commandConnection = tcp_init(host, FTP_CONNECTION_PORT)))
    {
        printf("Init Error: Couldn't connect to port!\n");
        return 1;
    }
    if (
        get_response(commandConnection, &message))
    {
        printf("Init Error: Couldn't get response!\n");
        return 1;
    }

    if (check_is_error_code(message, 0))
    {

        printf("Init Error: Wrong response code!\n");
        return 1;
    }
    free(message);
    printf("Int: Inserting user.\n");

    char command[MAX_FIELD_SIZE] = "USER ";
    strcat(command, user);
    strcat(command, FTP_END_OF_LINE);
    if (tcp_write(commandConnection, command, strlen(command)))
    {
        printf("Init Error: couldn't write USER.\n");
        return 1;
    }
    if (
        get_response(commandConnection, &message))
    {
        printf("Init Error: Couldn't get response!\n");
        return 1;
    }

    if (check_is_error_code(message, 1))
    {

        printf("Init Error: Wrong response code!\n");
        return 1;
    }
    free(message);
    printf("Init: User inserted.\nInit: Sending password.\n");

    strcpy(command, "PASS ");
    strcat(command, pass);
    strcat(command, FTP_END_OF_LINE);
    printf("%s", command);
    if (tcp_write(commandConnection, command, strlen(command)))
    {
        printf("Init Error: couldn't write PASS.\n");
        return 1;
    }

    if (
        get_response(commandConnection, &message))
    {
        printf("Init Error: Couldn't get response!\n");
        return 1;
    }

    if (check_is_error_code(message, 0))
    {
```

```c
        printf("Init Error: Wrong response code!\n");
        return 1;
    }
    free(message);

    printf("Init: Password inserted.\n Init: Sending PASV\n");

    strcpy(command, "PASV" FTP_END_OF_LINE);

    if (tcp_write(commandConnection, command, strlen(command)))
    {
        printf("Init Error: couldn't write PASV.\n");
        return 1;
    }

    unsigned int otherPort = 0;

    if (
        get_response(commandConnection, &message))
    {
        printf("Init Error: Couldn't get response!\n");
        return 1;
    }

    char secondSocketServerCode[MAX_FIELD_SIZE];
    if (interpret_pasv_response(message, &otherPort, secondSocketServerCode))
    {

        printf("Init Error: Couldn't parse PASSV response!\n");
        return 1;
    }
    free(message);
    printf("Init: Password inserted.\nInit: Opening download socket %d, on server %s!\n",
otherPort, secondSocketServerCode);

    if (!(downloadConnection = tcp_init(secondSocketServerCode, otherPort)))
    {
        printf("Init Error: invalid URL!\n");
        return 1;
    }

    printf("Init: Opened download connection!\n");
    return 0;
}

int ftp_download_file(char *filename)
{
    printf("Download: Writing RETR!\n");
    char command[MAX_FIELD_SIZE] = "RETR ";

    strcat(command, file);
    strcat(command, FTP_END_OF_LINE);
    printf("command: %s", command);
    if (tcp_write(commandConnection, command, strlen(command)))
    {
        printf("Download Error: couldn't write RETR.\n");
        return 1;
    }

    char *message;
    if (
        get_response(commandConnection, &message))
    {
        printf("Download Error: Couldn't get response!\n");
        return 1;
    }

    if (check_is_error_code(message, STATE_RETR))
    {

        printf("Download Error: Wrong response code!\n");
        return 1;
    }
    free(message);

    printf("Download: RETR Successful.\nOpening file.\n");
    FILE *f = fopen(filename, "w");

    if (f == NULL)
    {
        printf("Download Error: couldn't open file locally.\n");
        return 1;
    }

    printf("Download: Starting download loop\n");

    while (downloadConnection)
    {

        char buffer[BATCH_SIZE];
        int ret = !downloadConnection ? 0 : tcp_read(downloadConnection, buffer, BATCH_SIZE);
        if (ret < 0)
        {
            printf("Download Error: TCP connection failed!\n");
            return 1;
        }
        else if (ret)
        {
            printf(".");

            if (fwrite(buffer, 1 /*size in bytes of element, which is a byte, therefore it's
 1*/, ret, f) < ret)
            {
                printf("Download Error: failed to copy data to file!\n");
                return 1;
            }
        }
        else if (ret == 0)
        {
            printf("\nDownload: Data connection EOF reached!\n");
            printf("Download: Terminating file transfer!\n");
            tcp_close(downloadConnection);
            downloadConnection = 0;
        }
    }
    printf("Download: Finished download.\n");

    fclose(f);

    return 0;
```

```c
}

int ftp_close()
{
    char command[MAX_FIELD_SIZE];
    char *message;

    printf("Close: Reading end-of-transmission message.\n");
    if (
        get_response(commandConnection, &message))
    {
        printf("Close Error: Couldn't get response!\n");
        return 1;
    }

    if (check_is_error_code(message, 0))
    {

        printf("Close Error: Wrong response code!\n");
        return 1;
    }
    free(message);

    printf("Close: Sending QUIT command.\n");
    strcpy(command, "QUIT" FTP_END_OF_LINE);
    if (tcp_write(commandConnection, command, strlen(command)))
    {
        printf("Close Error: couldn't write RETR.\n");
        return 1;
    }

    if (
        get_response(commandConnection, &message))
    {
        printf("Close Error: Couldn't get response!\n");
        return 1;
    }

    if (check_is_error_code(message, 0))
    {

        printf("Close Error: Wrong response code!\n");
        return 1;
    }

    printf("Close: terminated command connection successfully.\n");
    return tcp_close(commandConnection);
}
```

tcp.h

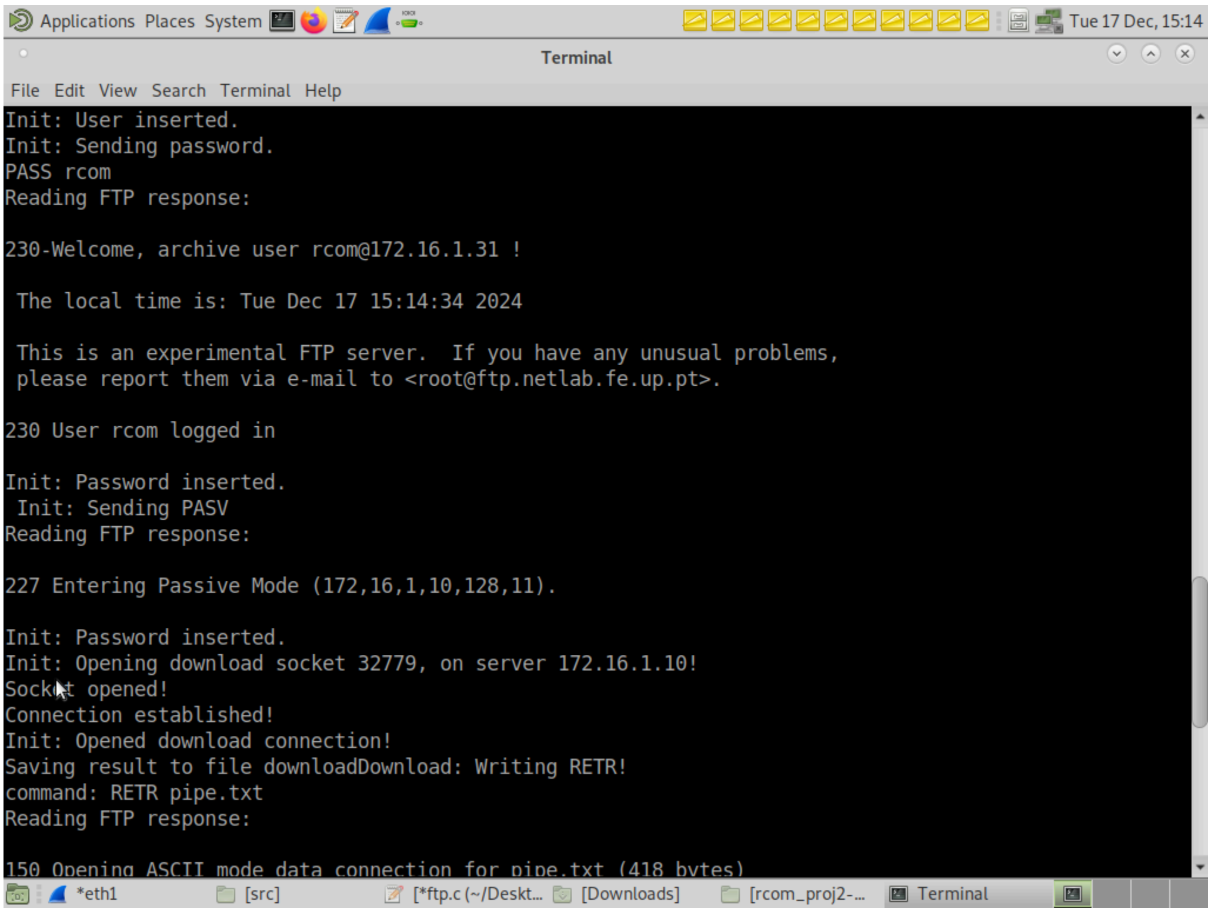
```c
#include <stdio.h>

/// Auxiliary data structure that serves as a wrapper to a socket file descriptor.
typedef struct Connection Connection;

/// @brief Sets up a TCP connection with a non-blocking socket.
/// @param addressName The address of the host to connect to
/// @param port The port of the host to connect to
/// @return The pointer to the connection on success, NULL/0 otherwise
Connection *tcp_init(char *addressName, unsigned int port);

/// @brief Reads the specified amount of bytes from the given connection. This function
allocates memory.
/// @param connection The pointer to the connection to read bytes from
/// @param outBuffer The pointer to the buffer where the bytes are stored
/// @param outBufferCapacity The capacity of the supplied buffer
/// @return The amount of bytes read, or -1 on failure.
int tcp_read(Connection *connection, char *outBuffer, unsigned int outBufferCapacity);
/// @brief Writes the specified message to a given connection.
/// @param connection The pointer to the connection to which to send the message
/// @param message The pointer to the message to be sent
/// @param numBytes The number of bytes in the message
/// @return 0 on success, 1 if no bytes were sent, or 2 if the wrong number of bytes were sent
int tcp_write(Connection *connection, char *message, unsigned int numBytes);
/// @brief Closes a TCP connection. This function frees the memory allocated for the
connection.
/// @param connection The pointer to the connection to be closed.
/// @return 0 on success, or 1 on failure
int tcp_close(Connection *connection);
```

ftp.h

```c
/// @brief Sets up the FTP state to be able to start a download. Opens the connections, logs
in and activates passive mode.
/// @param url a pointer to the FTP url, which contains the host to download from and name of
the file to download, and optional user and password information
/// @return 0 on success, 1 on failure
int ftp_init(char * url);
/// @brief Downloads the file: sends the RETR and reads from the data connection.
/// @param filename The name the file should be stored as
/// @return 0 on success, 1 on failure
int ftp_download_file(char *filename);
/// @brief Closes the FTP connection: sends the QUIT command and closes the socket.
/// @return 0 on success, 1 on failure
int ftp_close();
```

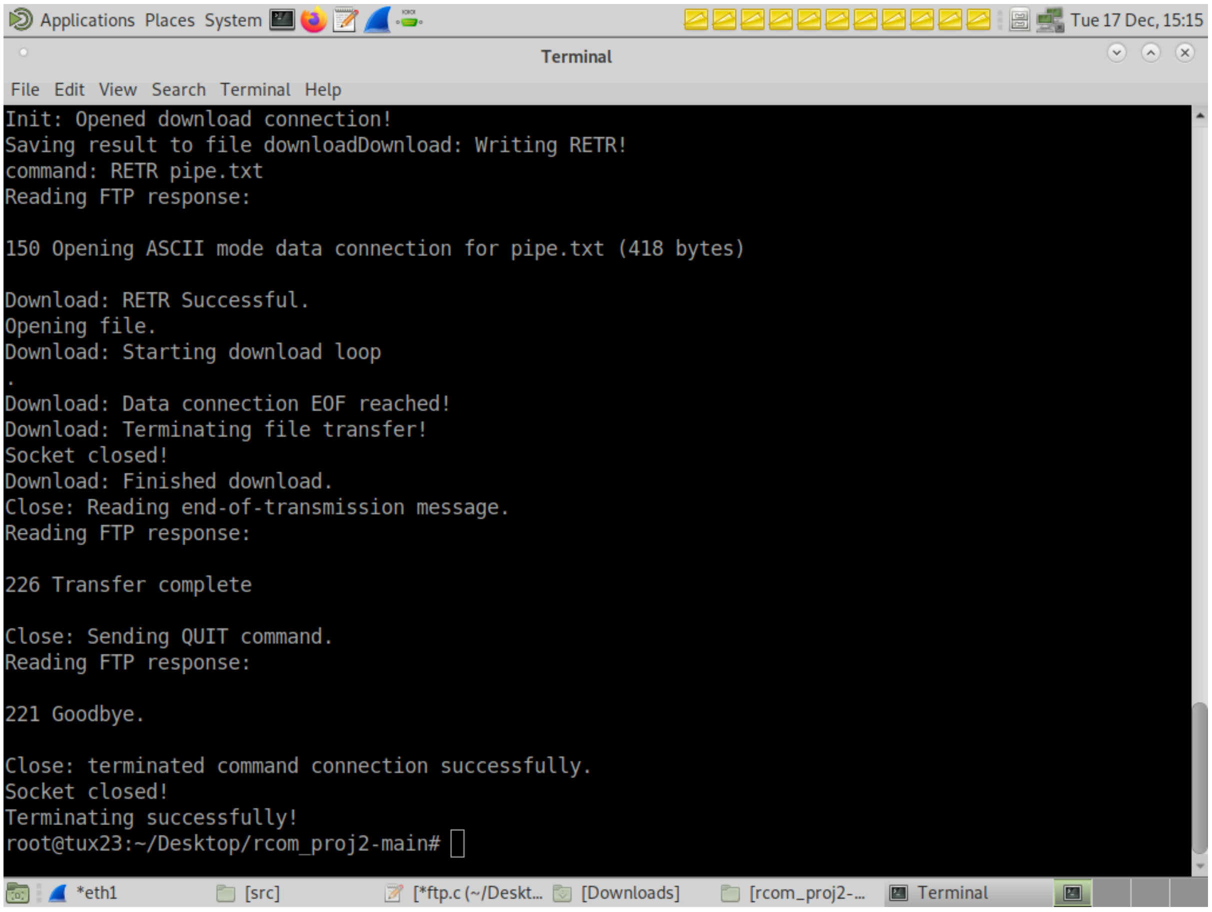
Annex 2 - FTP Application Screenshots

Screenshot 1 - Correct insertion of username, password and RETR command



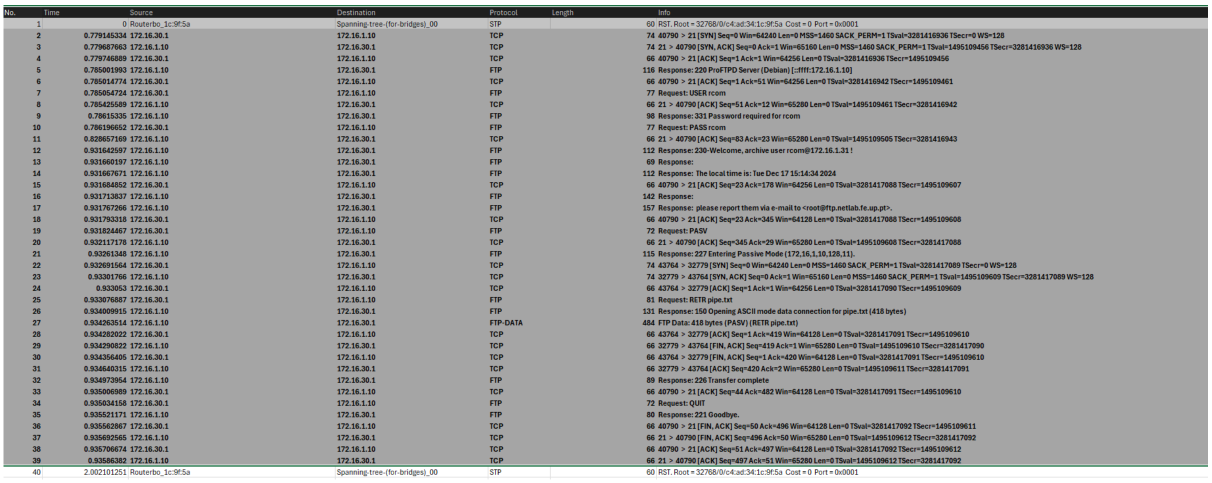
```
Init: User inserted.
Init: Sending password.
PASS rcom
Reading FTP response:

230-Welcome, archive user rcom@172.16.1.31 !

 The local time is: Tue Dec 17 15:14:34 2024

 This is an experimental FTP server.  If you have any unusual problems,
 please report them via e-mail to <root@ftp.netlab.fe.up.pt>.

230 User rcom logged in

Init: Password inserted.
 Init: Sending PASV
Reading FTP response:

227 Entering Passive Mode (172,16,1,10,128,11).

Init: Password inserted.
Init: Opening download socket 32779, on server 172.16.1.10!
Socket opened!
Connection established!
Init: Opened download connection!
Saving result to file downloadDownload: Writing RETR!
command: RETR pipe.txt
Reading FTP response:

150 Opening ASCII mode data connection for pipe.txt (418 bytes)
```

Screenshot 2 - Correct termination of the program



```
Init: Opened download connection!
Saving result to file downloadDownload: Writing RETR!
command: RETR pipe.txt
Reading FTP response:

150 Opening ASCII mode data connection for pipe.txt (418 bytes)

Download: RETR Successful.
Opening file.
Download: Starting download loop
.
Download: Data connection EOF reached!
Download: Terminating file transfer!
Socket closed!
Download: Finished download.
Close: Reading end-of-transmission message.
Reading FTP response:

226 Transfer complete

Close: Sending QUIT command.
Reading FTP response:

221 Goodbye.

Close: terminated command connection successfully.
Socket closed!
Terminating successfully!
root@tux23:~/Desktop/rcom_proj2-main#
```

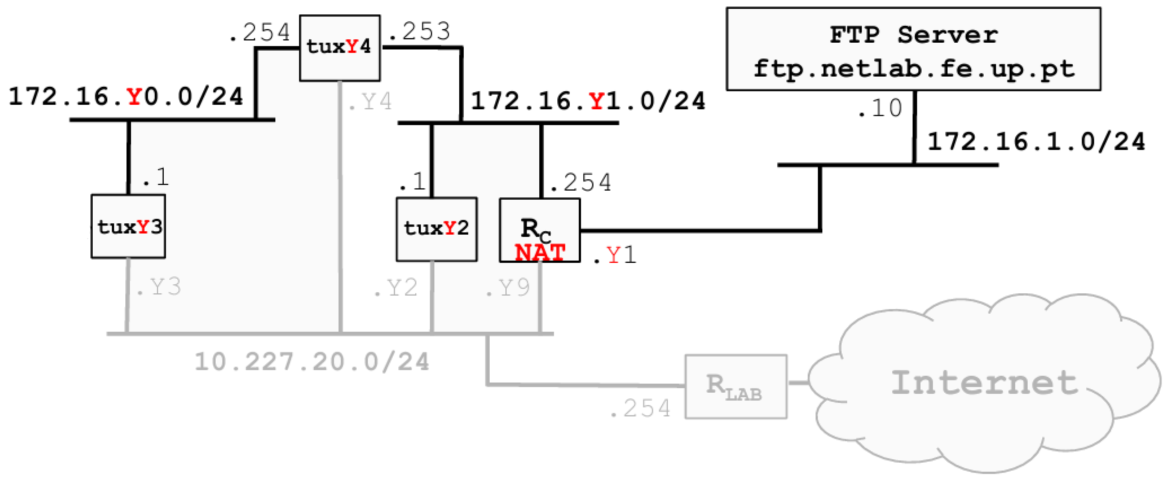
Screenshot 3 - Resulting packets (Exported as a .csv file)



No.	Time	Source	Destination	Protocol	Length	Info
1	0	Routerbo_1c:9f:5a	Spanning-tree-(for-bridges)_00	STP	60	RST. Root = 32768/0/c4:ad:34:1c:9f:5a Cost = 0 Port = 0x0001
2	0.779145334	172.16.30.1	172.16.1.10	TCP	74	40790 > 21 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=3281416936 TSecr=0 WS=128
3	0.779687663	172.16.1.10	172.16.30.1	TCP	74	21 > 40790 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=1495109456 TSecr=3281416936 WS=128
4	0.779746889	172.16.30.1	172.16.1.10	TCP	66	40790 > 21 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=3281416936 TSecr=1495109456
5	0.785001993	172.16.1.10	172.16.30.1	FTP	116	Response: 220 ProFTPD Server (Debian) [::ffff:172.16.1.10]
6	0.785014774	172.16.30.1	172.16.1.10	TCP	66	40790 > 21 [ACK] Seq=1 Ack=51 Win=64256 Len=0 TSval=3281416942 TSecr=1495109461
7	0.785054724	172.16.30.1	172.16.1.10	FTP	77	Request: USER rcom
8	0.785425589	172.16.1.10	172.16.30.1	TCP	66	21 > 40790 [ACK] Seq=51 Ack=12 Win=65280 Len=0 TSval=1495109461 TSecr=3281416942
9	0.78615335	172.16.1.10	172.16.30.1	FTP	98	Response: 331 Password required for rcom
10	0.786196652	172.16.30.1	172.16.1.10	FTP	77	Request: PASS rcom
11	0.828657169	172.16.1.10	172.16.30.1	TCP	66	21 > 40790 [ACK] Seq=83 Ack=23 Win=65280 Len=0 TSval=1495109505 TSecr=3281416943
12	0.931642597	172.16.1.10	172.16.30.1	FTP	112	Response: 230-Welcome, archive user rcom@172.16.1.31 !
13	0.931660197	172.16.1.10	172.16.30.1	FTP	69	Response:
14	0.931667671	172.16.1.10	172.16.30.1	FTP	112	Response: The local time is: Tue Dec 17 15:14:34 2024
15	0.931684852	172.16.30.1	172.16.1.10	TCP	66	40790 > 21 [ACK] Seq=23 Ack=178 Win=64256 Len=0 TSval=3281417088 TSecr=1495109607
16	0.931713837	172.16.1.10	172.16.30.1	FTP	142	Response:
17	0.931767266	172.16.1.10	172.16.30.1	FTP	157	Response: please report them via e-mail to <root@ftp.netlab.fe.up.pt>.
18	0.931793318	172.16.30.1	172.16.1.10	TCP	66	40790 > 21 [ACK] Seq=23 Ack=345 Win=64128 Len=0 TSval=3281417088 TSecr=1495109608
19	0.931824467	172.16.30.1	172.16.1.10	FTP	72	Request: PASV
20	0.932117178	172.16.1.10	172.16.30.1	TCP	66	21 > 40790 [ACK] Seq=345 Ack=29 Win=65280 Len=0 TSval=1495109608 TSecr=3281417088
21	0.93261348	172.16.1.10	172.16.30.1	FTP	115	Response: 227 Entering Passive Mode (172,16,1,10,128,11).
22	0.932691564	172.16.30.1	172.16.1.10	TCP	74	43764 > 32779 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=3281417089 TSecr=0 WS=128
23	0.93301766	172.16.1.10	172.16.30.1	TCP	74	32779 > 43764 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=1495109609 TSecr=3281417089 WS=128
24	0.933053	172.16.30.1	172.16.1.10	TCP	66	43764 > 32779 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=3281417090 TSecr=1495109609
25	0.933076887	172.16.30.1	172.16.1.10	FTP	81	Request: RETR pipe.txt
26	0.934009915	172.16.1.10	172.16.30.1	FTP	131	Response: 150 Opening ASCII mode data connection for pipe.txt (418 bytes)
27	0.934263514	172.16.1.10	172.16.30.1	FTP-DATA	484	FTP Data: 418 bytes (PASV) (RETR pipe.txt)
28	0.934282022	172.16.30.1	172.16.1.10	TCP	66	43764 > 32779 [ACK] Seq=1 Ack=419 Win=64128 Len=0 TSval=3281417091 TSecr=1495109610
29	0.934290822	172.16.1.10	172.16.30.1	TCP	66	32779 > 43764 [FIN, ACK] Seq=419 Ack=1 Win=65280 Len=0 TSval=1495109610 TSecr=3281417090
30	0.934356405	172.16.30.1	172.16.1.10	TCP	66	43764 > 32779 [FIN, ACK] Seq=1 Ack=420 Win=64128 Len=0 TSval=3281417091 TSecr=1495109610
31	0.934640315	172.16.1.10	172.16.30.1	TCP	66	32779 > 43764 [ACK] Seq=420 Ack=2 Win=65280 Len=0 TSval=1495109611 TSecr=3281417091
32	0.934973954	172.16.1.10	172.16.30.1	FTP	89	Response: 226 Transfer complete
33	0.935006989	172.16.30.1	172.16.1.10	TCP	66	40790 > 21 [ACK] Seq=44 Ack=482 Win=64128 Len=0 TSval=3281417091 TSecr=1495109610
34	0.935034158	172.16.30.1	172.16.1.10	FTP	72	Request: QUIT
35	0.935521171	172.16.1.10	172.16.30.1	FTP	80	Response: 221 Goodbye.
36	0.935562867	172.16.30.1	172.16.1.10	TCP	66	40790 > 21 [FIN, ACK] Seq=50 Ack=496 Win=64128 Len=0 TSval=3281417092 TSecr=1495109611
37	0.935692565	172.16.1.10	172.16.30.1	TCP	66	21 > 40790 [FIN, ACK] Seq=496 Ack=50 Win=65280 Len=0 TSval=1495109612 TSecr=3281417092
38	0.935706674	172.16.30.1	172.16.1.10	TCP	66	40790 > 21 [ACK] Seq=51 Ack=497 Win=64128 Len=0 TSval=3281417092 TSecr=1495109612
39	0.93586382	172.16.1.10	172.16.30.1	TCP	66	21 > 40790 [ACK] Seq=497 Ack=51 Win=65280 Len=0 TSval=1495109612 TSecr=3281417092
40	2.002101251	Routerbo_1c:9f:5a	Spanning-tree-(for-bridges)_00	STP	60	RST. Root = 32768/0/c4:ad:34:1c:9f:5a Cost = 0 Port = 0x0001

Annex 3 - Network Configuration

Figure 1 - Network Configuration



As it can be observed, there are four networks:

- **172.16.Y0.0/24**, containing one of the ports of Tux 3 and one of the ports of Tux 4.
- **172.16.Y0.1/24**, containing one of the ports of Tux 4, one of the ports of Tux 2, and the port ether2 of the router.
- **172.16.10.0/24**, containing the port ether1 of the router and the FTP Server.
- **10.227.20.0/24,** connecting all Tuxes and the router to the Internet, and which is only briefly relevant during the course of the experiments due to the router in it being configured as the default gateway.

Two of these, **172.16.Y0.0/24** and **172.16.Y1.0/24**, are configured during the course of the experiment.

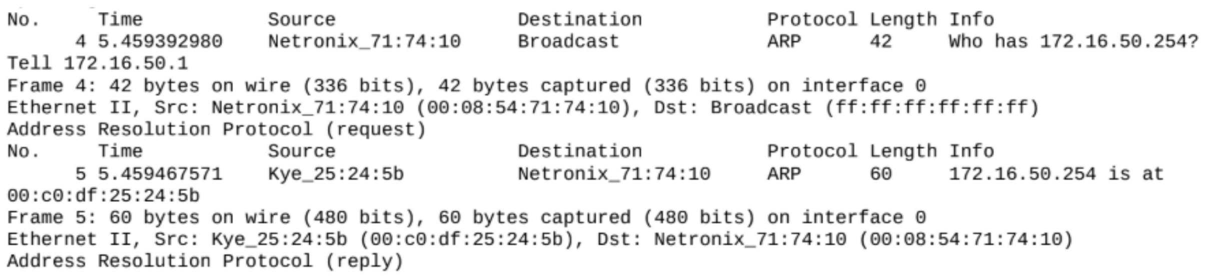
Annex 4 - Experiment Logs

Although screenshots of the relevant sections are presented, the complete logs can be accessed through links to Google Drive documents.

Experiment 1

Wireshark logs: tux3 ping tux4

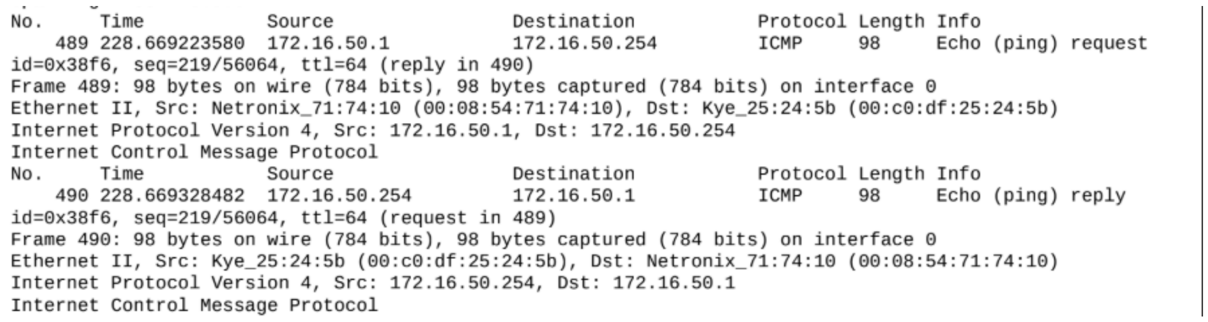
Figure 1 - ARP packets exchanged between Tux 3 and Tux 4 (captured in Tux 3).



```
No.     Time           Source                Destination           Protocol Length Info
      4 5.459392980    Netronix_71:74:10     Broadcast             ARP      42     Who has 172.16.50.254? Tell 172.16.50.1
Frame 4: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Netronix_71:74:10 (00:08:54:71:74:10), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
No.     Time           Source                Destination           Protocol Length Info
      5 5.459467571    Kye_25:24:5b          Netronix_71:74:10     ARP      60     172.16.50.254 is at 00:c0:df:25:24:5b
Frame 5: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
Ethernet II, Src: Kye_25:24:5b (00:c0:df:25:24:5b), Dst: Netronix_71:74:10 (00:08:54:71:74:10)
Address Resolution Protocol (reply)
```

As it can be seen, the Request packet is 42 bytes long, with a Broadcast address as destination and Tux 3’s MAC (**00:08:54:71:74:10**) as source. The Reply is 60 bytes long, with a destination address corresponding to Tux 3’s MAC and a source address corresponding to Tux4’s MAC (**00:c0:df:25:24:5b**). The time elapsed between the Request and Reply is roughly 0.3 milliseconds.

Figure 2 - ICMP packets exchanged between Tux 3 and Tux 4 (captured in Tux 3).

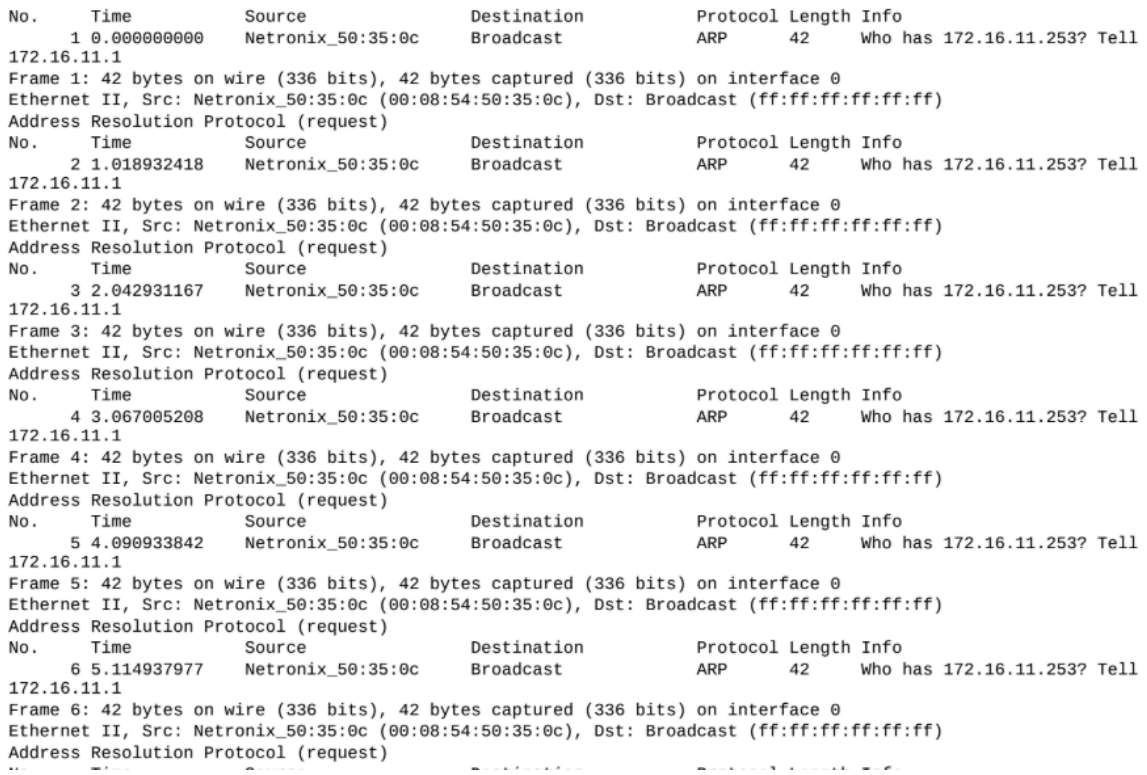


```
No.     Time           Source                Destination           Protocol Length Info
    489 228.669223580  172.16.50.1           172.16.50.254         ICMP     98     Echo (ping) request
id=0x38f6, seq=219/56064, ttl=64 (reply in 490)
Frame 489: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: Netronix_71:74:10 (00:08:54:71:74:10), Dst: Kye_25:24:5b (00:c0:df:25:24:5b)
Internet Protocol Version 4, Src: 172.16.50.1, Dst: 172.16.50.254
Internet Control Message Protocol
No.     Time           Source                Destination           Protocol Length Info
    490 228.669328482  172.16.50.254         172.16.50.1           ICMP     98     Echo (ping) reply
id=0x38f6, seq=219/56064, ttl=64 (request in 489)
Frame 490: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: Kye_25:24:5b (00:c0:df:25:24:5b), Dst: Netronix_71:74:10 (00:08:54:71:74:10)
Internet Protocol Version 4, Src: 172.16.50.254, Dst: 172.16.50.1
Internet Control Message Protocol
```

Both packets are 98 bytes long. The Request is sent from **172.16.50.1** (Tux 3) to **172.16.50.254** (Tux 4). The MAC addresses are the same as observed in the previous figure. In the Reply, these addresses are inverted.

Experiment 2

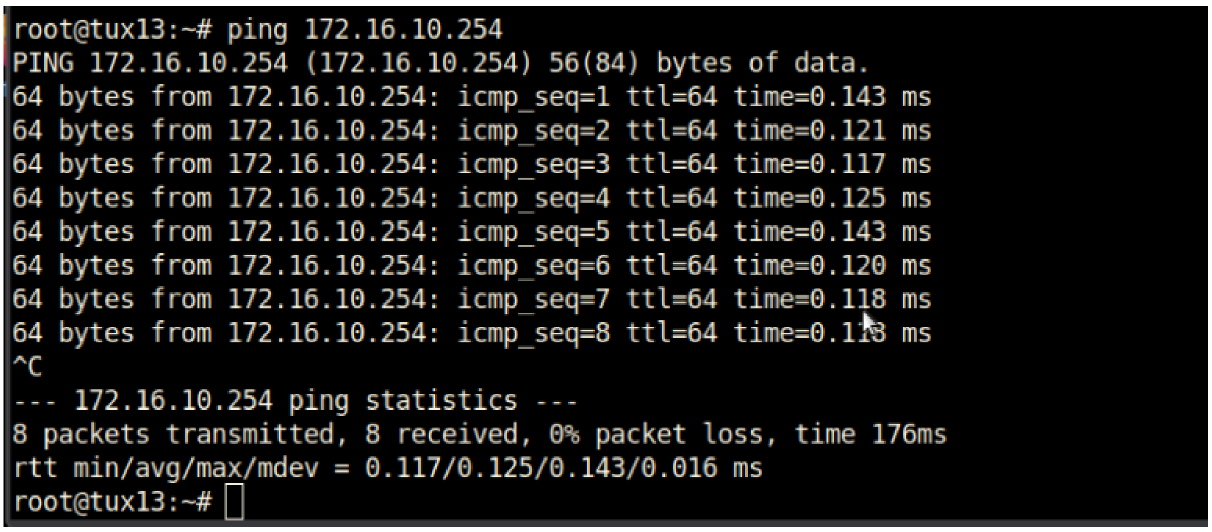
Figure 1 - ARP request packets (captured in Tux 2 while trying to ping Tux 4, in bench 1)



```
No.     Time           Source                Destination           Protocol Length Info
      1 0.000000000    Netronix_50:35:0c     Broadcast             ARP      42     Who has 172.16.11.253? Tell
172.16.11.1
Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Netronix_50:35:0c (00:08:54:50:35:0c), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
No.     Time           Source                Destination           Protocol Length Info
      2 1.018932418    Netronix_50:35:0c     Broadcast             ARP      42     Who has 172.16.11.253? Tell
172.16.11.1
Frame 2: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Netronix_50:35:0c (00:08:54:50:35:0c), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
No.     Time           Source                Destination           Protocol Length Info
      3 2.042931167    Netronix_50:35:0c     Broadcast             ARP      42     Who has 172.16.11.253? Tell
172.16.11.1
Frame 3: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Netronix_50:35:0c (00:08:54:50:35:0c), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
No.     Time           Source                Destination           Protocol Length Info
      4 3.067005208    Netronix_50:35:0c     Broadcast             ARP      42     Who has 172.16.11.253? Tell
172.16.11.1
Frame 4: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Netronix_50:35:0c (00:08:54:50:35:0c), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
No.     Time           Source                Destination           Protocol Length Info
      5 4.090933842    Netronix_50:35:0c     Broadcast             ARP      42     Who has 172.16.11.253? Tell
172.16.11.1
Frame 5: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Netronix_50:35:0c (00:08:54:50:35:0c), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
No.     Time           Source                Destination           Protocol Length Info
      6 5.114937977    Netronix_50:35:0c     Broadcast             ARP      42     Who has 172.16.11.253? Tell
172.16.11.1
Frame 6: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Netronix_50:35:0c (00:08:54:50:35:0c), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
```

These Requests are not being replied to.

Figure 2 - Successful ping between Tux 3 and Tux 4 (captured in Tux 3, in bench 1)

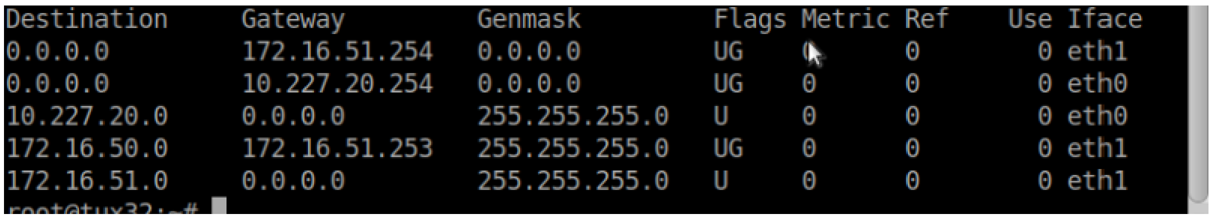


```
root@tux13:~# ping 172.16.10.254
PING 172.16.10.254 (172.16.10.254) 56(84) bytes of data.
64 bytes from 172.16.10.254: icmp_seq=1 ttl=64 time=0.143 ms
64 bytes from 172.16.10.254: icmp_seq=2 ttl=64 time=0.121 ms
64 bytes from 172.16.10.254: icmp_seq=3 ttl=64 time=0.117 ms
64 bytes from 172.16.10.254: icmp_seq=4 ttl=64 time=0.125 ms
64 bytes from 172.16.10.254: icmp_seq=5 ttl=64 time=0.143 ms
64 bytes from 172.16.10.254: icmp_seq=6 ttl=64 time=0.120 ms
64 bytes from 172.16.10.254: icmp_seq=7 ttl=64 time=0.118 ms
64 bytes from 172.16.10.254: icmp_seq=8 ttl=64 time=0.118 ms
^C
--- 172.16.10.254 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 176ms
rtt min/avg/max/mdev = 0.117/0.125/0.143/0.016 ms
root@tux13:~# 
```

Experiment 3

Wireshark logs: [tux3 ping other network interfaces]
Wireshark logs: [tux3 ping tux2 eth1(captured from tux4)]
Wireshark logs: [tux3 ping tux2 eth2 (captured from tux4)]

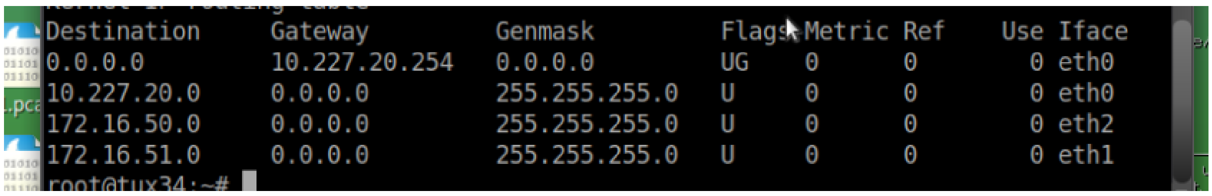
Figure 1 - Example of Tux 2's routing table



```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.16.51.254   0.0.0.0         UG    0      0        0 eth1
0.0.0.0         10.227.20.254   0.0.0.0         UG    0      0        0 eth0
10.227.20.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
172.16.50.0     172.16.51.253   255.255.255.0   UG    0      0        0 eth1
172.16.51.0     0.0.0.0         255.255.255.0   U     0      0        0 eth1
root@tux32:~#
```

In this instance of the experiment, an oversight in resetting the routing table made it so that there were two default routes, one for eth0 with the internet-connected router as a gateway; another with the lab's commercial router as a gateway. This did not affect the experiment, as the packets cannot reach the 172.16.Y0.0/24 network through there.

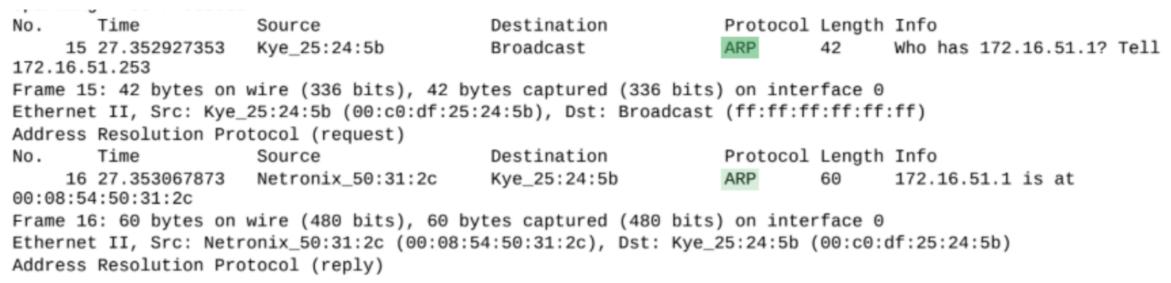
Figure 2 - Example of Tux 4's routing table



```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.227.20.254   0.0.0.0         UG    0      0        0 eth0
10.227.20.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
172.16.50.0     0.0.0.0         255.255.255.0   U     0      0        0 eth2
172.16.51.0     0.0.0.0         255.255.255.0   U     0      0        0 eth1
root@tux34:~#
```

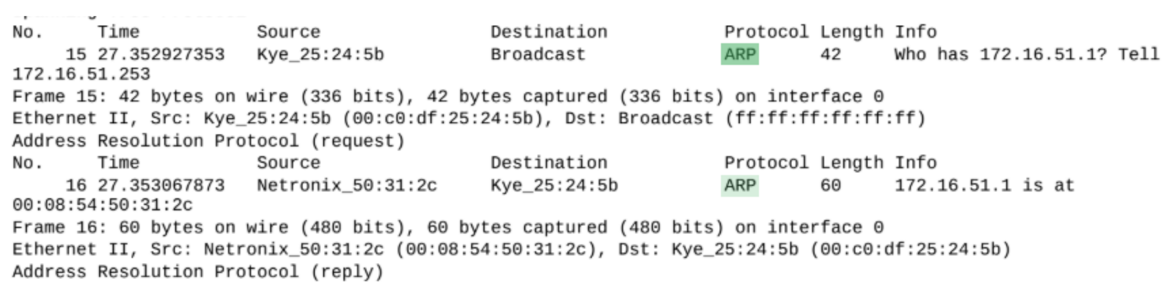
Notice that Tux 4 does not need to define specific gateways at this point, as it belongs to both relevant (ie. not related to the eth0 port, though it also belongs to that one) networks.

Figure 3, Tux 4 discovering Tux 2



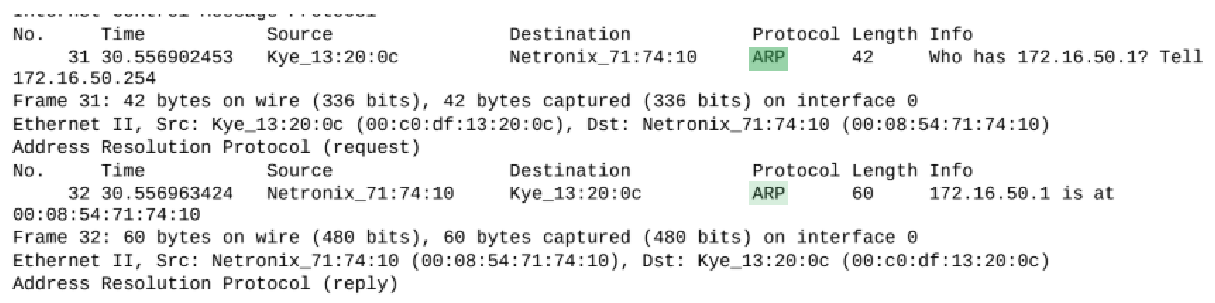
```
No.     Time           Source                Destination           Protocol Length Info
     15 27.352927353   Kye_25:24:5b          Broadcast             ARP      42     Who has 172.16.51.1? Tell
172.16.51.253
Frame 15: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Kye_25:24:5b (00:c0:df:25:24:5b), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
No.     Time           Source                Destination           Protocol Length Info
     16 27.353067873   Netronix_50:31:2c     Kye_25:24:5b          ARP      60     172.16.51.1 is at
00:08:54:50:31:2c
Frame 16: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
Ethernet II, Src: Netronix_50:31:2c (00:08:54:50:31:2c), Dst: Kye_25:24:5b (00:c0:df:25:24:5b)
Address Resolution Protocol (reply)
```

Figure 3 - ARP Request/Reply, Tux 4 discovering Tux 2



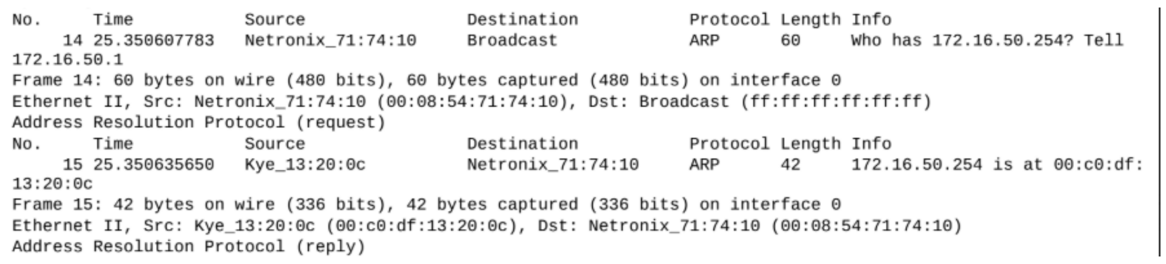
```
No.     Time           Source                Destination           Protocol Length Info
     15 27.352927353   Kye_25:24:5b          Broadcast             ARP      42     Who has 172.16.51.1? Tell 
172.16.51.253
Frame 15: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Kye_25:24:5b (00:c0:df:25:24:5b), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
No.     Time           Source                Destination           Protocol Length Info
     16 27.353067873   Netronix_50:31:2c     Kye_25:24:5b          ARP      60     172.16.51.1 is at 
00:08:54:50:31:2c
Frame 16: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
Ethernet II, Src: Netronix_50:31:2c (00:08:54:50:31:2c), Dst: Kye_25:24:5b (00:c0:df:25:24:5b)
Address Resolution Protocol (reply)
```

Figure 4 - ARP Request/Reply, Tux 4 discovering Tux 3



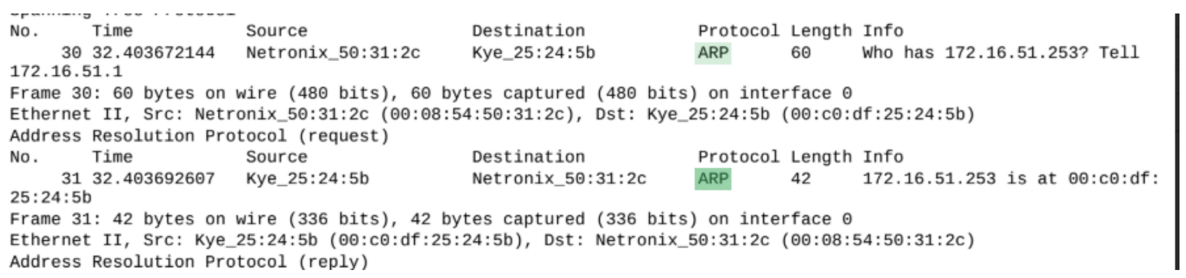
```
No.     Time           Source                Destination           Protocol Length Info
     31 30.556902453   Kye_13:20:0c          Netronix_71:74:10     ARP      42     Who has 172.16.50.1? Tell 
172.16.50.254
Frame 31: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Kye_13:20:0c (00:c0:df:13:20:0c), Dst: Netronix_71:74:10 (00:08:54:71:74:10)
Address Resolution Protocol (request)
No.     Time           Source                Destination           Protocol Length Info
     32 30.556963424   Netronix_71:74:10     Kye_13:20:0c          ARP      60     172.16.50.1 is at 
00:08:54:71:74:10
Frame 32: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
Ethernet II, Src: Netronix_71:74:10 (00:08:54:71:74:10), Dst: Kye_13:20:0c (00:c0:df:13:20:0c)
Address Resolution Protocol (reply)
```

Figure 5 - ARP Request/Reply, Tux 3 discovering Tux 4



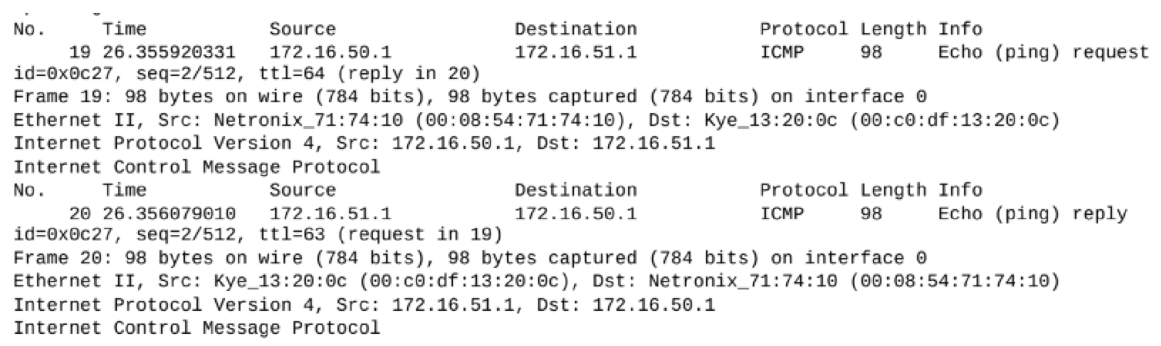
```
No.     Time           Source                Destination           Protocol Length Info
     14 25.350607783   Netronix_71:74:10     Broadcast             ARP      60     Who has 172.16.50.254? Tell 
172.16.50.1
Frame 14: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
Ethernet II, Src: Netronix_71:74:10 (00:08:54:71:74:10), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
No.     Time           Source                Destination           Protocol Length Info
     15 25.350635650   Kye_13:20:0c          Netronix_71:74:10     ARP      42     172.16.50.254 is at 00:c0:df:
13:20:0c
Frame 15: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Kye_13:20:0c (00:c0:df:13:20:0c), Dst: Netronix_71:74:10 (00:08:54:71:74:10)
Address Resolution Protocol (reply)
```

Figure 6 - ARP Request/Reply, Tux 2 discovering Tux 4



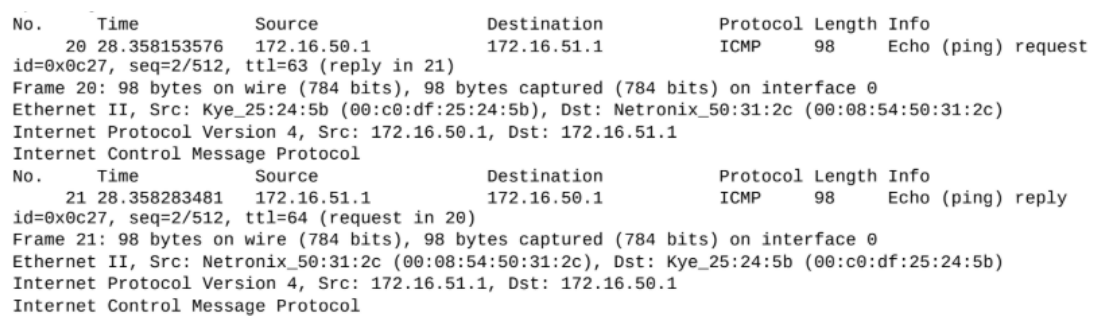
```
No.     Time           Source                Destination           Protocol Length Info
     30 32.403672144   Netronix_50:31:2c     Kye_25:24:5b          ARP      60     Who has 172.16.51.253? Tell 
172.16.51.1
Frame 30: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
Ethernet II, Src: Netronix_50:31:2c (00:08:54:50:31:2c), Dst: Kye_25:24:5b (00:c0:df:25:24:5b)
Address Resolution Protocol (request)
No.     Time           Source                Destination           Protocol Length Info
     31 32.403692607   Kye_25:24:5b          Netronix_50:31:2c     ARP      42     172.16.51.253 is at 00:c0:df:
25:24:5b
Frame 31: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Kye_25:24:5b (00:c0:df:25:24:5b), Dst: Netronix_50:31:2c (00:08:54:50:31:2c)
Address Resolution Protocol (reply)
```

Figure 7 - Ping Request/Reply, captured on Tux 3



```
No.     Time           Source                Destination           Protocol Length Info
     19 26.355920331   172.16.50.1           172.16.51.1           ICMP     98     Echo (ping) request
id=0x0c27, seq=2/512, ttl=64 (reply in 20)
Frame 19: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: Netronix_71:74:10 (00:08:54:71:74:10), Dst: Kye_13:20:0c (00:c0:df:13:20:0c)
Internet Protocol Version 4, Src: 172.16.50.1, Dst: 172.16.51.1
Internet Control Message Protocol
No.     Time           Source                Destination           Protocol Length Info
     20 26.356079010   172.16.51.1           172.16.50.1           ICMP     98     Echo (ping) reply
id=0x0c27, seq=2/512, ttl=63 (request in 19)
Frame 20: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: Kye_13:20:0c (00:c0:df:13:20:0c), Dst: Netronix_71:74:10 (00:08:54:71:74:10)
Internet Protocol Version 4, Src: 172.16.51.1, Dst: 172.16.50.1
Internet Control Message Protocol
```

Figure 8 - Ping Request/Reply, captured on Tux 2



```
No.     Time           Source                Destination           Protocol Length Info
     20 28.358153576   172.16.50.1           172.16.51.1           ICMP     98     Echo (ping) request
id=0x0c27, seq=2/512, ttl=63 (reply in 21)
Frame 20: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: Kye_25:24:5b (00:c0:df:25:24:5b), Dst: Netronix_50:31:2c (00:08:54:50:31:2c)
Internet Protocol Version 4, Src: 172.16.50.1, Dst: 172.16.51.1
Internet Control Message Protocol
No.     Time           Source                Destination           Protocol Length Info
     21 28.358283481   172.16.51.1           172.16.50.1           ICMP     98     Echo (ping) reply
id=0x0c27, seq=2/512, ttl=64 (request in 20)
Frame 21: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: Netronix_50:31:2c (00:08:54:50:31:2c), Dst: Kye_25:24:5b (00:c0:df:25:24:5b)
Internet Protocol Version 4, Src: 172.16.51.1, Dst: 172.16.50.1
Internet Control Message Protocol
```

Table 1 - IP and Mac addresses

	Between Tux 2 and Tux 4				**Between Tux 3 and Tux 4**			
	MAC - source	**MAC - destination**	**IP - source**	**IP - destination**	**MAC - source**	**MAC - destination**	**IP - source**	**IP - destination**
Ping Requests	00:c0:df:25:24:5b (Tux 4)	00:08:54:50:31:2c (Tux 2)	172.16.Y0.1 (Tux 3)	172.16.Y1.1 (Tux 2)	00:08:54:71:74:10 (Tux 3)	00:c0:df:13:20:0c (Tux 4)	172.16.Y0.1 (Tux 3)	172.16.Y1.1 (Tux 2)
Ping Replies	00:08:54:50:31:2c (Tux 2)	00:c0:df:25:24:5b (Tux 4)	172.16.Y1.1 (Tux 2)	172.16.Y0.1 (Tux 3)	00:c0:df:13:20:0c (Tux 4)	00:08:54:71:74:10 (Tux 3)	172.16.Y1.1 (Tux 2)	172.16.Y0.1 (Tux 3)

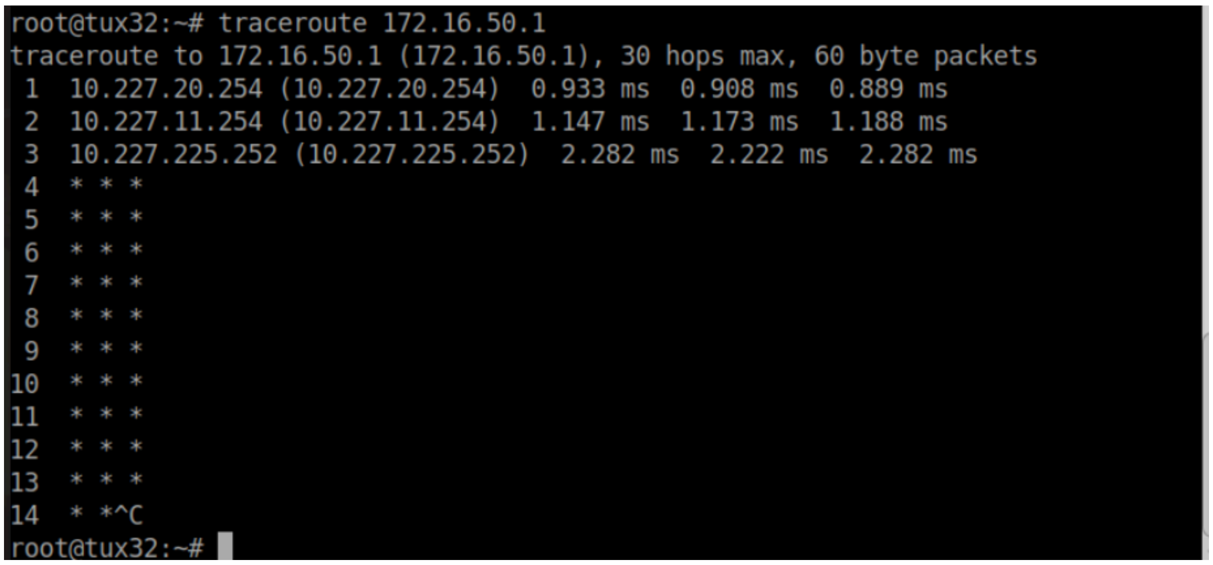
MAC addresses:

- Tux 2 - 00:08:54:50:31:2c
- Tux 3 - 00:08:54:71:74:10
- Tux 4 - 00:c0:df:25:24:5b (in the port used in network 172.16.Y1.0/24) and 00:c0:df:13:20:0c (in the port used in network 172.16.Y0.0/24)

Experiment 4

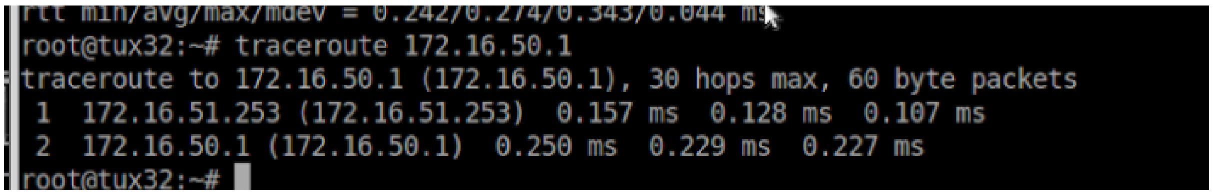
Wireshark logs: tux3 ping all network interfaces of tuxY2, tuxY4, Rc
Wireshark logs: tux2 ping tux3 (failed to establish connection)

Figure 1- Traceroute from Tux 2, after route deletion



```
root@tux32:~# traceroute 172.16.50.1
traceroute to 172.16.50.1 (172.16.50.1), 30 hops max, 60 byte packets
 1  10.227.20.254 (10.227.20.254)  0.933 ms  0.908 ms  0.889 ms
 2  10.227.11.254 (10.227.11.254)  1.147 ms  1.173 ms  1.188 ms
 3  10.227.225.252 (10.227.225.252)  2.282 ms  2.222 ms  2.282 ms
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * * *
10  * * *
11  * * *
12  * * *
13  * * *
14  * *^C
root@tux32:~# 
```

Figure 2 - Traceroute from Tux 2, after route re-insertion

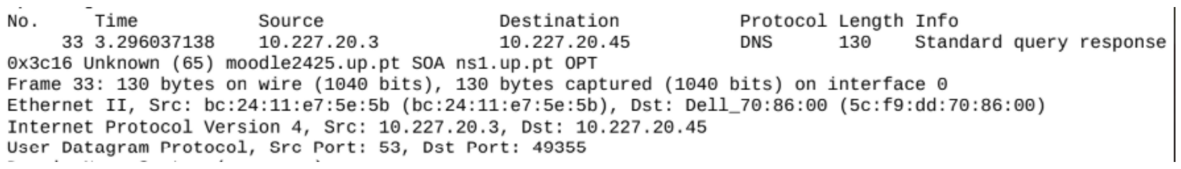


```
rtt min/avg/max/mdev = 0.242/0.274/0.343/0.044 ms
root@tux32:~# traceroute 172.16.50.1
traceroute to 172.16.50.1 (172.16.50.1), 30 hops max, 60 byte packets
 1  172.16.51.253 (172.16.51.253)  0.157 ms  0.128 ms  0.107 ms
 2  172.16.50.1 (172.16.50.1)  0.250 ms  0.229 ms  0.227 ms
root@tux32:~# 
```

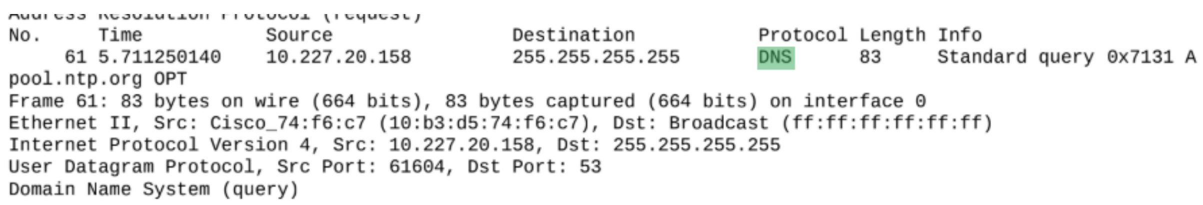
Experiment 5

Wireshark logs: tux3 ping Google
Wireshark logs: tux2 ping Facebook

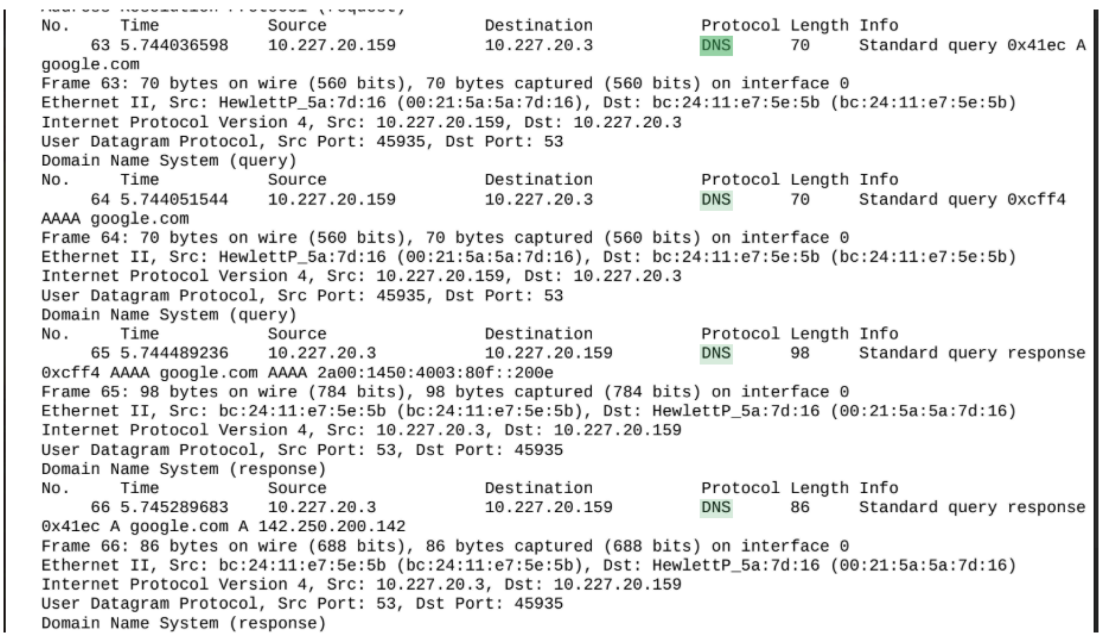
Figures 1, 2, 3, 4 - DNS packets detected when pinging google.com



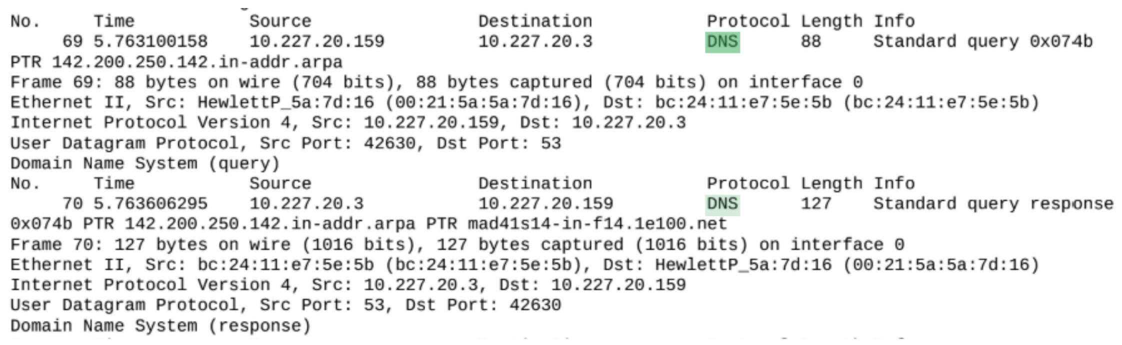
```
No.     Time           Source                Destination           Protocol Length Info
     33 3.296037138    10.227.20.3           10.227.20.45          DNS      130    Standard query response
0x3c16 Unknown (65) moodle2425.up.pt SOA ns1.up.pt OPT
Frame 33: 130 bytes on wire (1040 bits), 130 bytes captured (1040 bits) on interface 0
Ethernet II, Src: bc:24:11:e7:5e:5b (bc:24:11:e7:5e:5b), Dst: Dell_70:86:00 (5c:f9:dd:70:86:00)
Internet Protocol Version 4, Src: 10.227.20.3, Dst: 10.227.20.45
User Datagram Protocol, Src Port: 53, Dst Port: 49355
```



```
Address Resolution Protocol (request)
No.     Time           Source                Destination           Protocol Length Info
     61 5.711250140    10.227.20.158         255.255.255.255       DNS      83     Standard query 0x7131 A
pool.ntp.org OPT
Frame 61: 83 bytes on wire (664 bits), 83 bytes captured (664 bits) on interface 0
Ethernet II, Src: Cisco_74:f6:c7 (10:b3:d5:74:f6:c7), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Internet Protocol Version 4, Src: 10.227.20.158, Dst: 255.255.255.255
User Datagram Protocol, Src Port: 61604, Dst Port: 53
Domain Name System (query)
```



```
No.     Time           Source                Destination           Protocol Length Info
     63 5.744036598    10.227.20.159         10.227.20.3           DNS      70     Standard query 0x41ec A
google.com
Frame 63: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0
Ethernet II, Src: HewlettP_5a:7d:16 (00:21:5a:5a:7d:16), Dst: bc:24:11:e7:5e:5b (bc:24:11:e7:5e:5b)
Internet Protocol Version 4, Src: 10.227.20.159, Dst: 10.227.20.3
User Datagram Protocol, Src Port: 45935, Dst Port: 53
Domain Name System (query)
No.     Time           Source                Destination           Protocol Length Info
     64 5.744051544    10.227.20.159         10.227.20.3           DNS      70     Standard query 0xcff4
AAAA google.com
Frame 64: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0
Ethernet II, Src: HewlettP_5a:7d:16 (00:21:5a:5a:7d:16), Dst: bc:24:11:e7:5e:5b (bc:24:11:e7:5e:5b)
Internet Protocol Version 4, Src: 10.227.20.159, Dst: 10.227.20.3
User Datagram Protocol, Src Port: 45935, Dst Port: 53
Domain Name System (query)
No.     Time           Source                Destination           Protocol Length Info
     65 5.744489236    10.227.20.3           10.227.20.159         DNS      98     Standard query response
0xcff4 AAAA google.com AAAA 2a00:1450:4003:80f::200e
Frame 65: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: bc:24:11:e7:5e:5b (bc:24:11:e7:5e:5b), Dst: HewlettP_5a:7d:16 (00:21:5a:5a:7d:16)
Internet Protocol Version 4, Src: 10.227.20.3, Dst: 10.227.20.159
User Datagram Protocol, Src Port: 53, Dst Port: 45935
Domain Name System (response)
No.     Time           Source                Destination           Protocol Length Info
     66 5.745289683    10.227.20.3           10.227.20.159         DNS      86     Standard query response
0x41ec A google.com A 142.250.200.142
Frame 66: 86 bytes on wire (688 bits), 86 bytes captured (688 bits) on interface 0
Ethernet II, Src: bc:24:11:e7:5e:5b (bc:24:11:e7:5e:5b), Dst: HewlettP_5a:7d:16 (00:21:5a:5a:7d:16)
Internet Protocol Version 4, Src: 10.227.20.3, Dst: 10.227.20.159
User Datagram Protocol, Src Port: 53, Dst Port: 45935
Domain Name System (response)
```



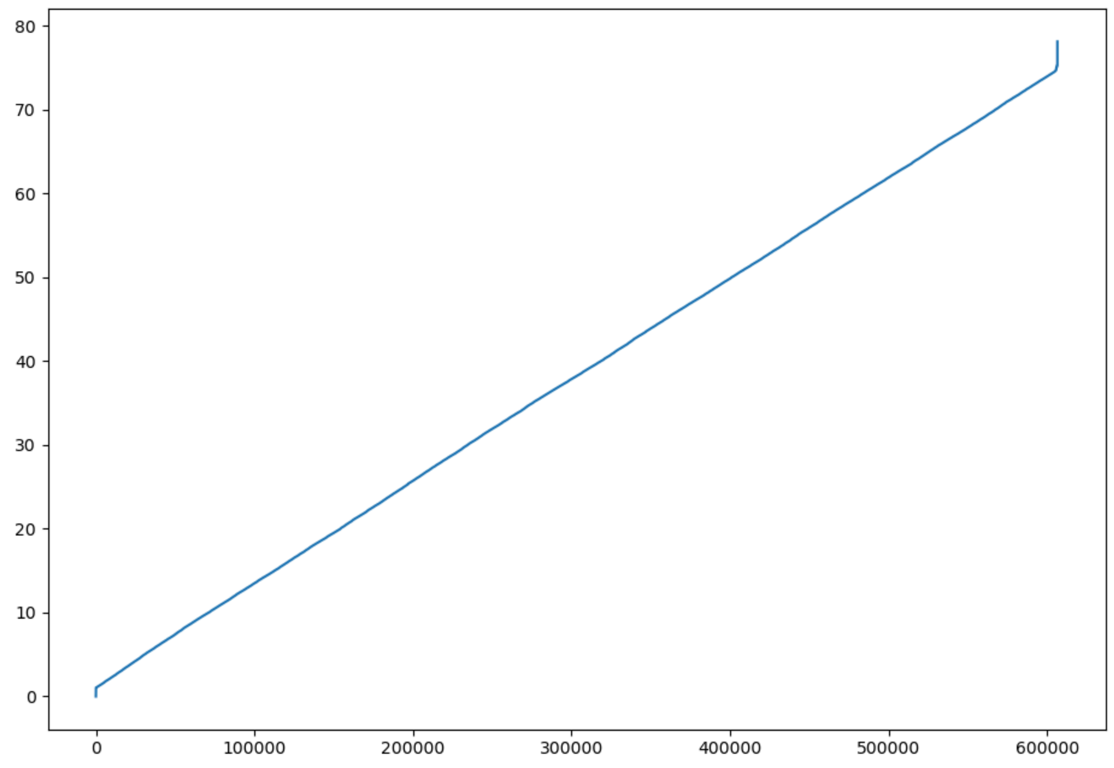
```
No.     Time           Source                Destination           Protocol Length Info
     69 5.763100158    10.227.20.159         10.227.20.3           DNS      88     Standard query 0x074b
PTR 142.200.250.142.in-addr.arpa
Frame 69: 88 bytes on wire (704 bits), 88 bytes captured (704 bits) on interface 0
Ethernet II, Src: HewlettP_5a:7d:16 (00:21:5a:5a:7d:16), Dst: bc:24:11:e7:5e:5b (bc:24:11:e7:5e:5b)
Internet Protocol Version 4, Src: 10.227.20.159, Dst: 10.227.20.3
User Datagram Protocol, Src Port: 42630, Dst Port: 53
Domain Name System (query)
No.     Time           Source                Destination           Protocol Length Info
     70 5.763606295    10.227.20.3           10.227.20.159         DNS      127    Standard query response
0x074b PTR 142.200.250.142.in-addr.arpa PTR mad41s14-in-f14.1e100.net
Frame 70: 127 bytes on wire (1016 bits), 127 bytes captured (1016 bits) on interface 0
Ethernet II, Src: bc:24:11:e7:5e:5b (bc:24:11:e7:5e:5b), Dst: HewlettP_5a:7d:16 (00:21:5a:5a:7d:16)
Internet Protocol Version 4, Src: 10.227.20.3, Dst: 10.227.20.159
User Datagram Protocol, Src Port: 53, Dst Port: 42630
Domain Name System (response)
```

Experiment 6

Wireshark logs: Download on tux3

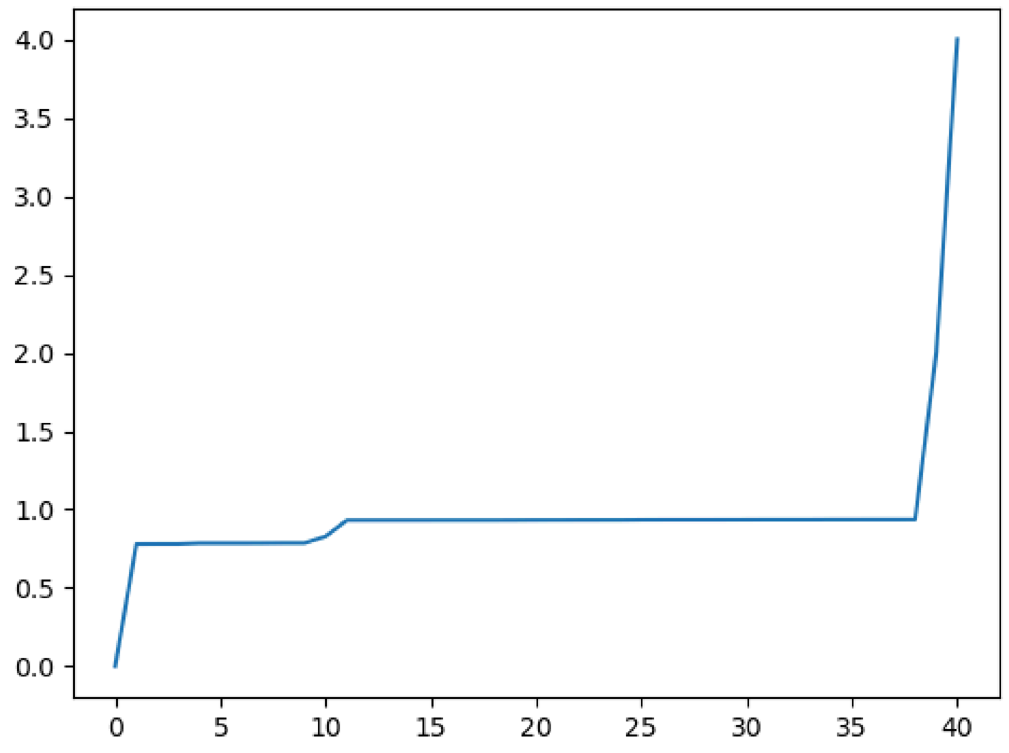
Wireshark logs: Download on tux3 with tux2 interruptions while downloading (captured by tux3)

Figure 1 - Graph of time (Y; in seconds) after X packets sent during the Tux 3 transmission with Tux 2 interruptions



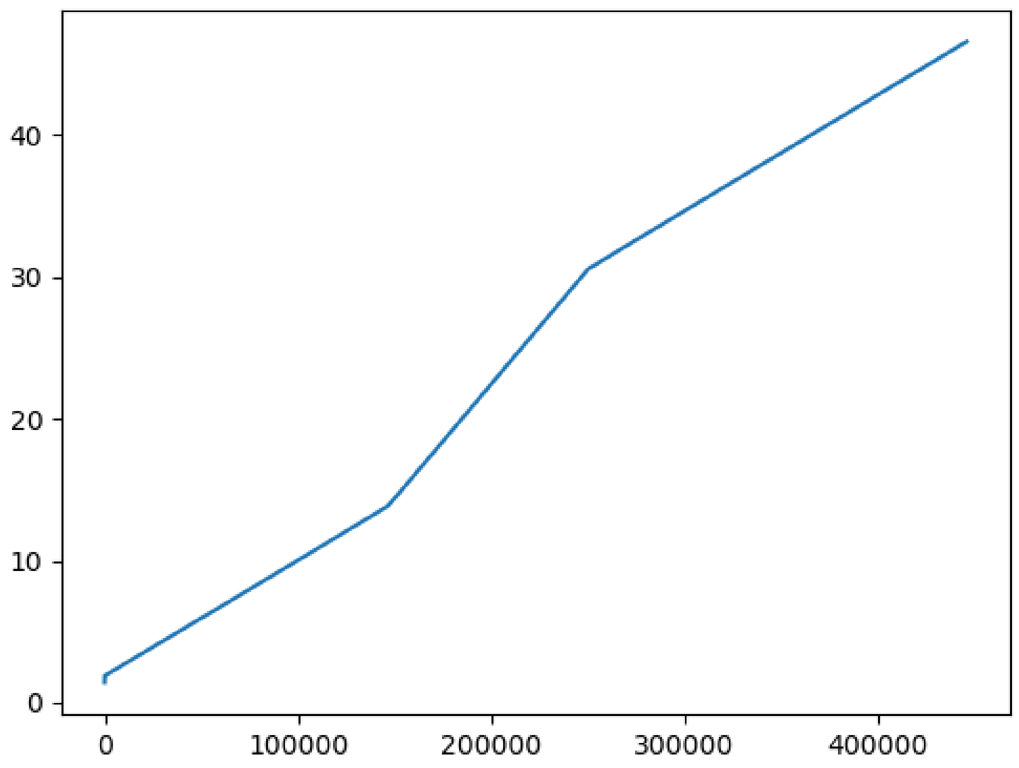
As the correlation is approximately linear, it can be concluded that the network was not operating in maximum capacity, and, therefore, the two transfers were able to coexist without disturbing each other. Another possible explanation is that the file downloaded by Tux 2 was much smaller than the one being downloaded by Tux 3. Moreover, the program prints to the console as it downloads, slowing down further (which could therefore slow down the rate at which it requests data).

Figure 2 - Graph of time (Y axis; in seconds) after X packets sent during the Tux 3 transmission *without* Tux 2 interruptions



The non-linear shape of this graph is due to the small size of the download chosen, which, alongside the smaller number of packets, makes the processing time of the application and other random variations more noticeable.

Figure 3 - Expected results, captured with an at-home setup, with Y = time, X = number of packets



The following graph was created based on the results of an at-home setup with two computers downloading from a third computer in a network with a bandwidth of 100Mbits/s, and ***scp*** commands. It corresponds to a scenario where interferences do occur. In the central area of the graph, there is a zone where the time until the next packet is higher than at the start and end of the transmission, corresponding to a decrease in bits received per second.
The corresponding log can be found here.