RC, L.EIC025 - Second Project
Computer Networks
Final Report

Isabel Silva - 201904925
Pedro Paixão - 202008467
Diogo Neves - 202108460

Introduction

This project was developed in the curricular unit Computer Networks, to develop a project containing two distinct parts. The first consists of developing an application that can download files through FTP, and the second is about six different experiences with the multiple steps to configure a network.

For the first part of the project, the application was developed using C programming language and can transfer files, while also being robust to errors.

In the second part of the project, the experiments were completed successfully, achieving all goals and having the main questions related to it to be answered correctly.

Download Application

Architecture

The FTP download application was developed according to the specifications in **RFC959** and **RFC1738**, focusing on implementing a simple file transfer system.

Its architecture is designed around the FTP protocol's client-server communication model, which involves two types of sockets: a **control** socket for command exchange and a **data** socket for file transfer.

The application starts by parsing the provided FTP URL, extracting essential components such as the username, password, host address, and file path. These elements are used to establish communication with the FTP server. Next, it resolves the server's hostname to an IP address using DNS services, followed by creating a control connection using TCP sockets.

After successfully connecting to the server, the application performs authentication by sending the fitting FTP commands, *USER* and *PASS*, with the parsed credentials.

To initiate the file transfer, the program switches to passive mode by issuing the *PASV* command. The server responds with an IP address and port, which the application uses to establish the data connection. Once the data channel is ready, the RETR command is sent to request the file. The application receives the file's data through the data connection and writes it to the local file system.

After completing the transfer, the application is also alert for a 226 response from the server, letting it know it can safely proceed to send a *QUIT* command and to receive the fitting server response to end the interaction with the server.

Results Analysis

In order to ensure that the application was correctly developed, several tests were conducted, using several files, ranging from different sizes, to ensure that the download program was functioning properly.

Along with successful transfer attempts, there were also instances where the intended result was the identification of errors, such as when attempting to transfer files that did not exist or when using incorrect or invalid authentication credentials. The application always behaved as expected.

The results of these tests were further corroborated using Wireshark, which allowed for detailed inspection of the FTP commands and responses exchanged during the file transfer process, as well as the data flow through the TCP connections. The logs demonstrated that the application adhered to the FTP protocol, properly handling both control and data connections.

Configuration and Network Analysis

Experiment 1 - Configure an IP Network

This experiment's goal was to configure two machines, *tux3,* and *tux4*, by assigning an IP address to each of them. The transmission of messages between these two machines was also observable, using the ping command.

Main commands:
- *ifconfig <network eth1 in this case, for both tuxes> up*
- *ifconfig <network eth1 in this case, for both tuxes> <IP address>/<mask>*

What are the ARP packets, and what are they used for?

The ARP (Address Resolution Protocol) protocol maps a machine's IP address to the machine's MAC address on the local network. When a computer tries to send a packet to another on the same local network (assuming that the ARP table has no entries for the IP of the receiving machine), it will send an ARP packet by broadcast to all the machines on the local network asking which machine has a MAC address that matches the IP address of the recipient. The recipient will send another ARP packet telling the sending machine its MAC address. In this way, the packet transfer can be carried out.

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

When the tux3 tries to send a packet to the tux4, since the entry in the ARP table referent to tux4 was deleted, the tux3 doesn't know the MAC address associated with the IP of tux4 172.16.10.254. For this reason, an ARP packet will be broadcast to the complete local network. This packet contains the IP address 172.16.10.1 and the MAC address 00:50:fc:ed:bb:37 of tux3. The MAC address of the destination is unknown at this point.

Afterward, the tux4 will send a packet to tux3 with its MAC 00:c0:df:04:20:99 and IP 172.16.10.254 addresses.

To conclude, each ARP packet contains fields for the origin and destination IP and MAC addresses.

What packets does the ping command generate?

As mentioned before, the ping command generates ARP packets to determine the destination's MAC address. Then, it generates ICMP packets (Internet Control Message Protocol).

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

When the ping command is done from tux3 to tux4, the following addresses are sent in the packet:

IP source (tux3): 172.16.10.1
MAC source (tux3): 00:50:fc:ed:bb:37
IP destination (tux4): 172.16.10.254

MAC destination (tux4): 00:c0:df:04:20:99

The reply packet has a similar structure but the source is the tux4 and the destination is the tux3:

IP source (tux4): 172.16.10.254
MAC source (tux4): 00:c0:df:04:20:99
IP destination (tux3): 172.16.10.1
MAC destination (tux3): 00:50:fc:ed:bb:37

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

We can determine the type of the receiving network by analyzing the Ethernet header of the network:

If this has the value 0x0800, the packet is of the IP type.
If it has the value 0x0806, then it's an ARP type.
If it is an IP type, we can analyze its IP header. If this header has a value of 1, then the protocol type is ICMP.

How to determine the length of a receiving frame?

The length of a receiving frame can be visualized through Wireshark.

What is the loopback interface and why is it important?

The loopback interface is a virtual network interface that allows the computer to receive responses from itself to test the correct configuration of the network.

Results:

The results can be seen in Appendix IIII: Experiments logs.

Experiment 2 - Implement Two Bridges in a Switch

This experiment's goal was to create two bridges in the switch.

Main commands:

- */interface bridge add name=bridgeY0*
- */interface bridge print*
- */interface bridge port remove [find interface =ether1]*
- */interface bridge port add bridge=bridgeY0 interface=ether1*
- */interface bridge port print*

How to configure bridgeY0?

Firstly, a terminal in GKTerm needs to be opened. Then, to create a new bridge, the command */interface bridge add name=bridgeY0* should be inserted in the terminal. In our specific case, since we were working at Table 1, the bridge name should be bridge10.

After that, it was necessary to remove the ports that we wanted to add to this bridge from the default bridge, using the command */interface bridge port remove [find interface =ether1]*.

Next, (and after checking that the ports we want to add to the new bridge were removed from the default one using the command */interface bridge port print*) the ports needed to be added to our newly created bridge, with the help of the command */interface* *bridge port add bridge=bridgeY0 interface=ether1*. After finishing this process, it is possible to check if the ports are correctly added to a bridge with the command */interface bridge port print*.

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

In this experience, there are two distinct broadcast domains. One is 172.16.10.0 and the other one is 172.16.11.0. The first one belongs to the virtual LAN connected to the bridge10 and the other one belongs to the bridge11's domain.

Results:

The results can be seen in Appendix IIII: Experiments logs.

Experiment 3 - Configure a Router in Linux

This experiment's goal was to configure a router.

Main commands:

- Commands already mentioned before.
- *sysctl net.ipv4.ip_forward=1*
- *sysctl net.ipv4.icmp_echo_ignore_broadcasts=0*

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

Some routes are generated automatically, "linking" the machines to the LAN's they belong to: tux3 has a route to eth1, tux2 has a route to eth2, and tux4 has a route to both. The gateway for these routes is 0.0.0.0.

In addition, tux3 has added the route *route add -net 172.16.10.0/24 gw 172.16.10.254*, i.e. when tux3 (172.16.10.1) wants to send a ping/message to eth2 (172.16.11.0) it will use the router that is tux4 (172.16.10.254) as its gateway.

Tux3 will have a forwarding table that makes this possible.

On tux4 we did something similar to what was done on tux3, only in the opposite direction. We created a route *route add -net 172.16.10.1/24 gw 172.16.11.253*, i.e. when we want to send a ping/message to eth0 we first send it to the router (172.16.11.253).

What information does an entry of the forwarding table contain?

In the forwarding table, each entry has information of the type [Destination - Gateway - Interface], where the destination is the IP address of the destination computer/network, the gateway is the IP address of the computer to which the message will be sent and this is the one that will give routing of the message to the destination. The interface is the network used to send the message, for example: eth1, eth2, etc.

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

An exchange of ARP messages exists when tux3 pings tux4 and tux3 doesn't know the MAC address of it. Therefore, tux3 sends an ARP message asking for the MAC address of tux4, through its IP address. When tux3 (that is the sender) sends an ARP message, it associates its own MAC address to the message, so the receiver knows to which tux has to respond. This message is sent in broadcast (receiver tux4 at this point has the default MAC address associated with it 00:00:00:00:00:00) since the MAC address of tux4 is still unknown. When this broadcast is received, tux4 sends an ARP message providing its MAC address. This message is not sent on broadcast, but only to the MAC address of tux3.

This exchange of ARP messages occurs every time that a message is sent and the MAC address destination is unknown.

What ICMP packets are observed and why?

ICMP packets of the types of request and reply are observed, because there are already routes that connect all the tuxes, so they recognize each others' presence. if they didn't know each other, the ICMP packets would be of the type "Host Unreachable".

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

The source and destination IP and MAC addresses associated with the ICMP packets are the IP and MAC addresses of the machines or interfaces that receive or send the packets.

For example, when a ICMP packet is sent from tux3 to the interface of tux4, connected to the same sub-network (both connected to eth1), the origin IP and MAC addresses will be the ones from tux3 (172.16.10.1 and 00:50:fc:ed:bb:37) and the destination IP and MAC addresses are from tux4 (172.16.10.254 and 00:c0:df:04:20:99).

If two machines are not connected to the same sub-network, similar to tux3 and tux2, it is not possible to send the information between the two machines with just one ICMP packet, without it being modified. In this case, tux3 sends an ICMP packet to the interface eth1 of tux4. Once tux4 is connected to both subnetworks (eth1 and eth2), it can interact directly with tux2. The IP addresses of the ICMP packet will be the ones of tux3 and tux2 (origin and destination, respectively), and the MAC addresses will be the ones of tux3 and the interface eth1 of tux4. When receiving this packet, tux4 will forward it to tux2, keeping the IP addresses of the packet, but changing the MAC addresses associated: the source MAC address will be of eth2 of tux4, and the destination will be the MAC address of tux2.

Results:

The results can be seen in Appendix IIII: Experiments logs.

Experiment 4 - Configure a Commercial Router and Implement NAT

This experiment's goal was to configure a commercial router and implement NAT.

Main commands:

- */ip address add address=<IP address>/<mask> interface=ether1*
- */ip address print*
- */ip route add dst-address=0.0.0.0/0 gateway=<IP address>*
- */ip route add dst-address=<IP address>/<mask> gateway=<IP address>*
- */ip route print*
- */ip firewall nat disable 0*

How to configure a static route in a commercial router?

Firstly, it is necessary to initiate a session in GTKTerm, and for that, it is necessary to connect a cable S1 of a tux to the router entry. To configure the routes, the command *ip route* needs to be executed inside GTKTerm. This command follows the structure *ip address add address=172.16.1.11/24 interface=ether1*, for example.

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

In the case where a route exists, the packets follow that same route. Otherwise, the packets go to the router, which is the default route.

When the redirects are disabled through the commands *sysctl net.ipv4.conf.eth0.accept_redirects=0* and *sysctl net.ipv4.conf.all.accept_redirects=0* and there are redirects within the same network, the tux doesn't store in its forwarding list an entry the redirect to the other tux.

In this experiment, tux2's default route was defined to be the router RC to the subnet 172.16.10.0/24, instead of tux4, and the redirects were disabled. In this scenario, only tux4 can communicate with tux3. On the other hand, tux4, tux2, and the router are all connected to the other network. Here, the router knows that to get to tux3, it needs to use tux4, and for that, a route was added to the router. When, from tux2 we try to ping tux3, it will send the ping to the router, since tux3 and tux2 are not in the same network. Taking into account that the router knows that tux4 can get to tux3, it sends the ping from the router to tux4, which then will forward it to tux3. In the case where the redirects are enabled, only the first time the ping is done, there will be redirects. After the first ping, tux2 will store in its forwarding table that to get to tux3, tux4 needs to be used as the default gateway. This way, tux2 sends the ping directly to tux4, instead of sending it to the router.

How to configure NAT in a commercial router?

In order to configure the router, some commands need to be inserted into GTKTerm:

- /ip address add address=172.16.1.11/24
- /ip address add address=172.16.11.254/24
- /ip route add dst-address=0.0.0.0/0 gateway=172.16.1.254
- /ip route add dst-address=172.16.10.0/24 gateway=172.16.11.254

What does NAT do?

NAT (Network Address Translation) is a protocol that associates and transforms an IP address into another IP address to mask the sender or receiver of sent packets. This could be used to assure the privacy and security of machines within a private network that is trying to communicate with external machines, by conserving its IP addresses, while also allowing this communication to exist. This allows private networks to use their IP addresses unregistered, but still connect and communicate with the Internet or public networks. The machines on that network, to the machines on the external network, are recognized through an unique IP, that represents all machines in the private network.

What happens when tux3 pings the FTP server with the NAT disabled? Why?

In the case where the NAT is disabled, when tux3 tries to ping the FTP server, here is what would happen:

> Tux3 will send an ICMP request to the IP address on the FTP network. Since the FTP server is not connected to the tux3's network, the packet will be forwarded to the default gateway, which is tux4. Therefore, tux4 would try to forward the packet to the FTP server via the router RC.
>
> Once NAT is disabled, tux3's IP address is not translated into a "routable" IP address. So, when the packet arrives at the router RC, it holds the private IP address that is not "routable" in the FTP network.

Since the FTP network will see the private IP of tux3, it will drop the packet, since this IP cannot be used in public or external networks. Given this, the ping request will fail.

This happens, because, again, the NAt translates the IP addresses of private networks to public or "routable" addresses. Without it, the FTP network can't reply, since it doesn't route packets back to private IP addresses, so the communication fails.

Results:

The results can be seen in Appendix IIII: Experiments logs.

Experiment 5 - DNS

For this experiment, we used the network configured during the previous experiments. The objective is to access the server ftp.netlab.fe.up.pt through the domain configured. To do that, each computer's DNS (Domain Name System) had to be established in order to finish this experiment.

For that, we had to modify the /etc/resolv.conf file on each machine and add the nameserver 10.227.20.3.

Main commands:

- The new line is inserted into the */etc/resolv.conf* file.

How to configure the DNS service in a host?

The DNS service is confirmed in resolv.conf file, present in the folder /etc/ of the host tux. The configuration is done through two commands, one representing the DNS server, and the other one with the respective IP address.

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

The host sends to the server a packet with the hostname, hoping that it will be returned with its IP address. The server then responds with a packet that contains the IP address of the hostname at stake here.

Results:

The results can be seen in Appendix IIII: Experiments logs.

Experiment 6 - TCP Connections

This experiment's goal was to observe the behavior and the functionality of the TCP protocol, by using the application developed previously.

Main commands:

- Compilation and execution of the application developed. There is a Makefile also available to help with this.

How many TCP connections are opened by your FTP application?

Two distinct TCP connections are opened by the application, one when the application connects to the server, through which commands can be sent and received, in order to prepare for the file transfer, and the other connections done when the file is actually transferred.

In what connection is transported the FTP control information?

The control information is transported in the first TCP connection, the one where commands are sent and received.

What are the phases of aTCP connection?

In a TCP connection firstly the connection is established, and then the data exchange occurs. After that, the connection is closed. Therefore, three distinct phases exist.

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

The mechanism ARQ TCP works through the congestion control window. This consists of the error control of data transmission. For this to work, acknowledgment numbers are used, which indicate the data, the window size that indicates the type of packets received, and finally, the sequence number, that corresponds to the packet number to be sent.

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

The purpose of TCP congestion management is to prevent congestion collapse and guarantee network stability and effective use of available bandwidth. It accomplishes this by modifying the data transmission rate in response to input regarding the network's condition.
A TCP state variable called the Congestion Window (CWND) sets a restriction on how much data a sender can transfer before getting an acknowledgment (ACK). It adapts dynamically to the observed state of the network. TCP starts with a modest CWND (often one or two segments) at the start of a connection or following a timeout, and it grows rapidly with each ACK received. This stage lasts until packet loss is detected or CWND hits the Slow Start Threshold (SSTHRESH). To prevent overloading the network, CWND growth turns linear after it surpasses SSTHRESH. If no losses are found, CWND rises by approximately one maximum segment size (MSS) for every round trip time (RTT). Fast Retransmission and Fast Recovery: TCP retransmits a packet without waiting for a timeout (Fast Retransmission) when it receives double ACKs, assuming it has been lost. During Fast Recovery, growth continues linearly once CWND is lowered to SSTHRESH, which is typically half of the CWND before the loss.

Is throughput of a TCP data connection distributed by the appearance of a second TPC connection? How?

Since both connections must share the available bandwidth, the throughput of the first connection is impacted when a second TCP connection shows up on a shared network. TCP's congestion management mechanisms, which seek to equitably allocate bandwidth among competing flows, regulate this change. The first connection uses up all of the bandwidth initially, but once the second connection starts sending, signs of increased network congestion, such as packet loss or longer round-trip times, appear. In response, both connections decrease their congestion windows (CWND), which causes the throughput to be redistributed. Assuming equivalent round-trip times and no other distinguishing characteristics, the two connections usually converge to an approximately equal portion of the bandwidth over time.

The additive increase and multiplicative decrease (AIMD) algorithm of TCP, which ensures stability and fairness, is responsible for this behavior. By dynamically modifying the transmitting rates of competing connections, the overall impact maximizes network use, even though some unpredictability may occur because of variations in RTTs or congestion control techniques.

Results:

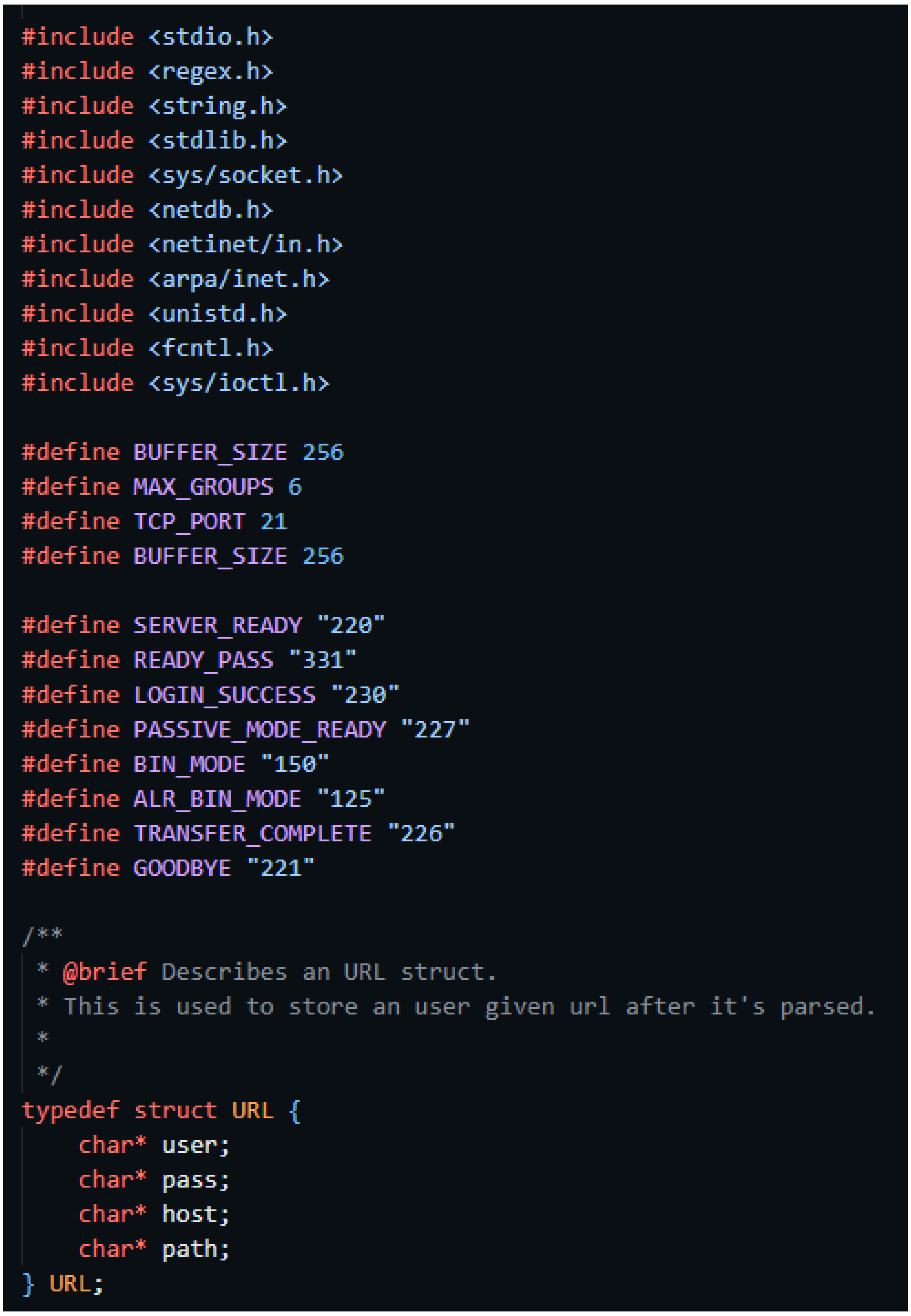
The results can be seen in Appendix IIII: Experiments logs.

Conclusions

Through this project, we consolidated the knowledge of each mechanism required to configure and analyze a computer network. In addition, we solidified our understanding of the data transfer protocols and other related concepts.

Appendices

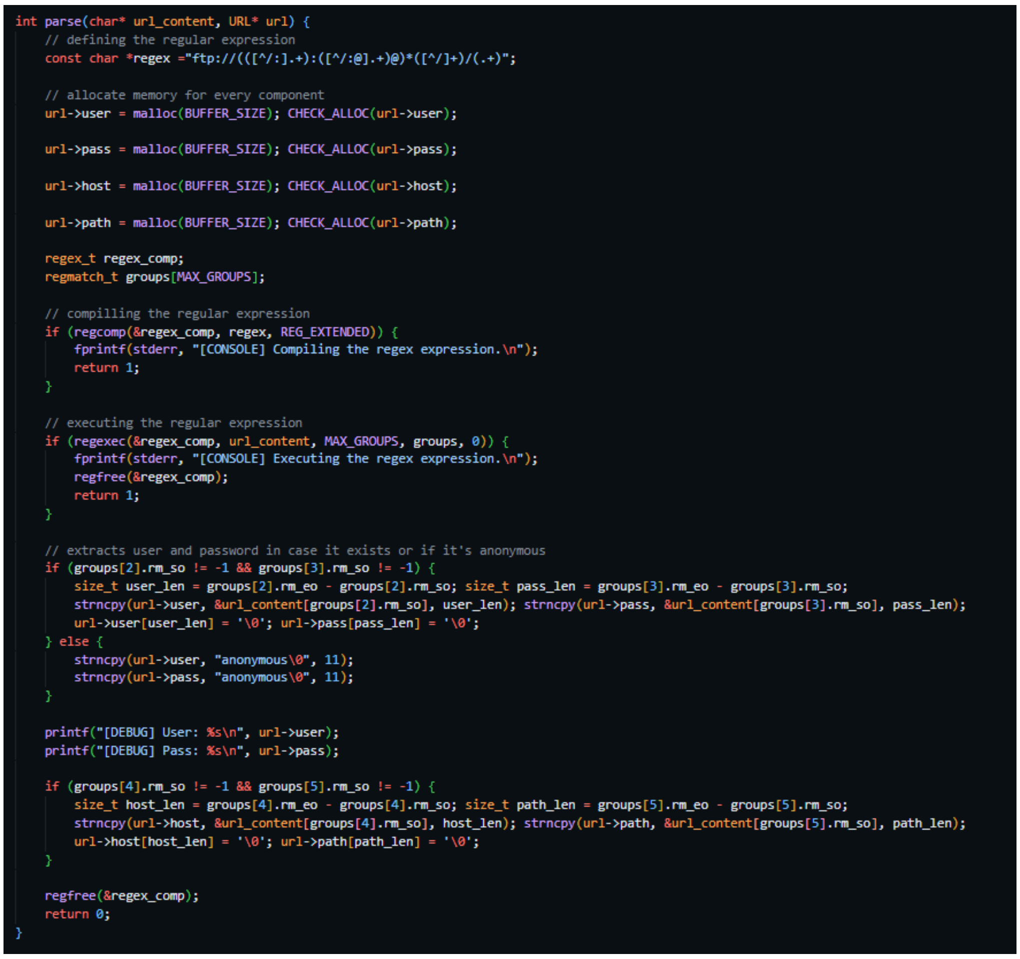
Appendix I: connection.h



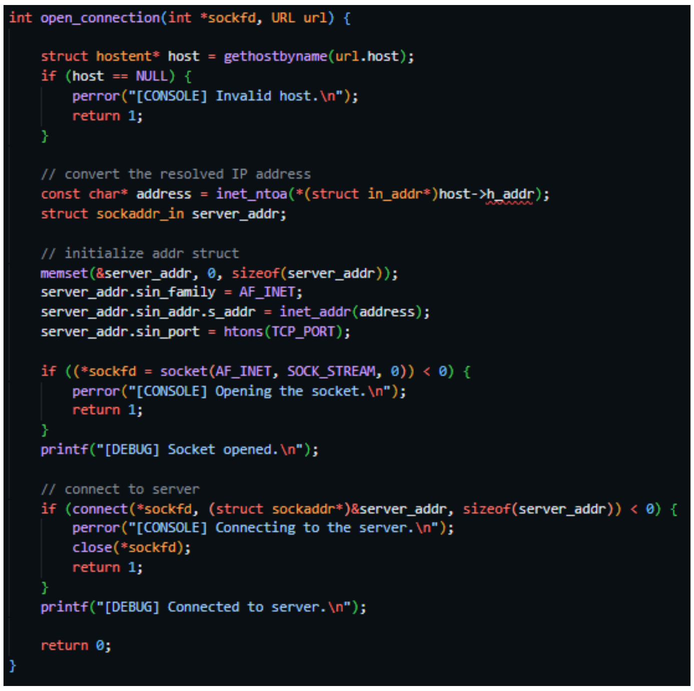
```c
#include <stdio.h>
#include <regex.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#define BUFFER_SIZE 256
#define MAX_GROUPS 6
#define TCP_PORT 21
#define BUFFER_SIZE 256

#define SERVER_READY "220"
#define READY_PASS "331"
#define LOGIN_SUCCESS "230"
#define PASSIVE_MODE_READY "227"
#define BIN_MODE "150"
#define ALR_BIN_MODE "125"
#define TRANSFER_COMPLETE "226"
#define GOODBYE "221"

/**
 * @brief Describes an URL struct.
 * This is used to store an user given url after it's parsed.
 *
 */
typedef struct URL {
    char* user;
    char* pass;
    char* host;
    char* path;
} URL;
```

Appendix II: connection.c

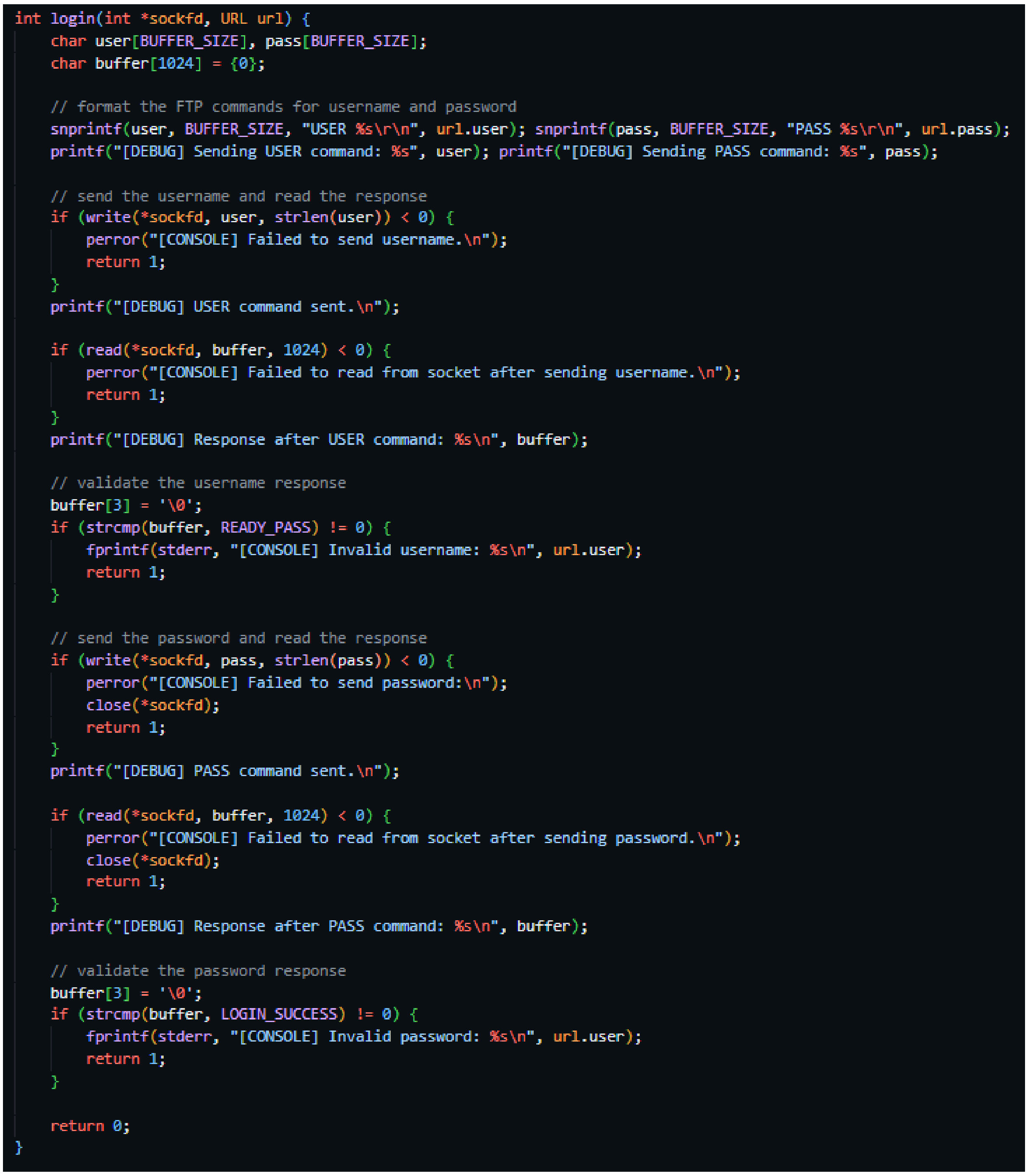
(some lines are combined into the same line to make the images more compact)



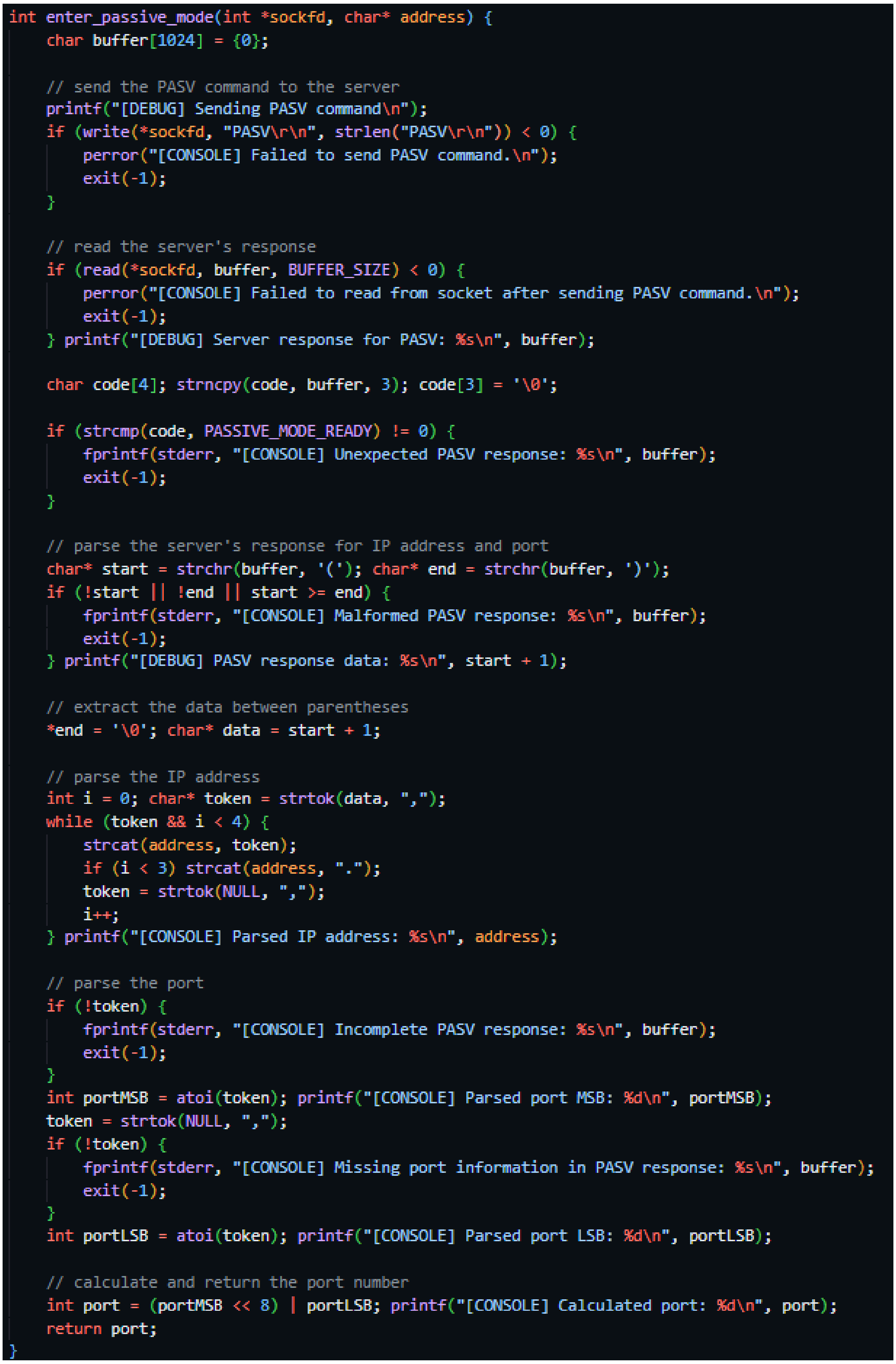
```c
int parse(char* url_content, URL* url) {
    // defining the regular expression
    const char *regex ="ftp://(([^/:].+):([^/:@].+)@)*([^/]+)/(.+)";

    // allocate memory for every component
    url->user = malloc(BUFFER_SIZE); CHECK_ALLOC(url->user);

    url->pass = malloc(BUFFER_SIZE); CHECK_ALLOC(url->pass);

    url->host = malloc(BUFFER_SIZE); CHECK_ALLOC(url->host);

    url->path = malloc(BUFFER_SIZE); CHECK_ALLOC(url->path);

    regex_t regex_comp;
    regmatch_t groups[MAX_GROUPS];

    // compilling the regular expression
    if (regcomp(&regex_comp, regex, REG_EXTENDED)) {
        fprintf(stderr, "[CONSOLE] Compiling the regex expression.\n");
        return 1;
    }

    // executing the regular expression
    if (regexec(&regex_comp, url_content, MAX_GROUPS, groups, 0)) {
        fprintf(stderr, "[CONSOLE] Executing the regex expression.\n");
        regfree(&regex_comp);
        return 1;
    }

    // extracts user and password in case it exists or if it's anonymous
    if (groups[2].rm_so != -1 && groups[3].rm_so != -1) {
        size_t user_len = groups[2].rm_eo - groups[2].rm_so; size_t pass_len = groups[3].rm_eo - groups[3].rm_so;
        strncpy(url->user, &url_content[groups[2].rm_so], user_len); strncpy(url->pass, &url_content[groups[3].rm_so], pass_len);
        url->user[user_len] = '\0'; url->pass[pass_len] = '\0';
    } else {
        strncpy(url->user, "anonymous\0", 11);
        strncpy(url->pass, "anonymous\0", 11);
    }

    printf("[DEBUG] User: %s\n", url->user);
    printf("[DEBUG] Pass: %s\n", url->pass);

    if (groups[4].rm_so != -1 && groups[5].rm_so != -1) {
        size_t host_len = groups[4].rm_eo - groups[4].rm_so; size_t path_len = groups[5].rm_eo - groups[5].rm_so;
        strncpy(url->host, &url_content[groups[4].rm_so], host_len); strncpy(url->path, &url_content[groups[5].rm_so], path_len);
        url->host[host_len] = '\0'; url->path[path_len] = '\0';
    }

    regfree(&regex_comp);
    return 0;
}
```



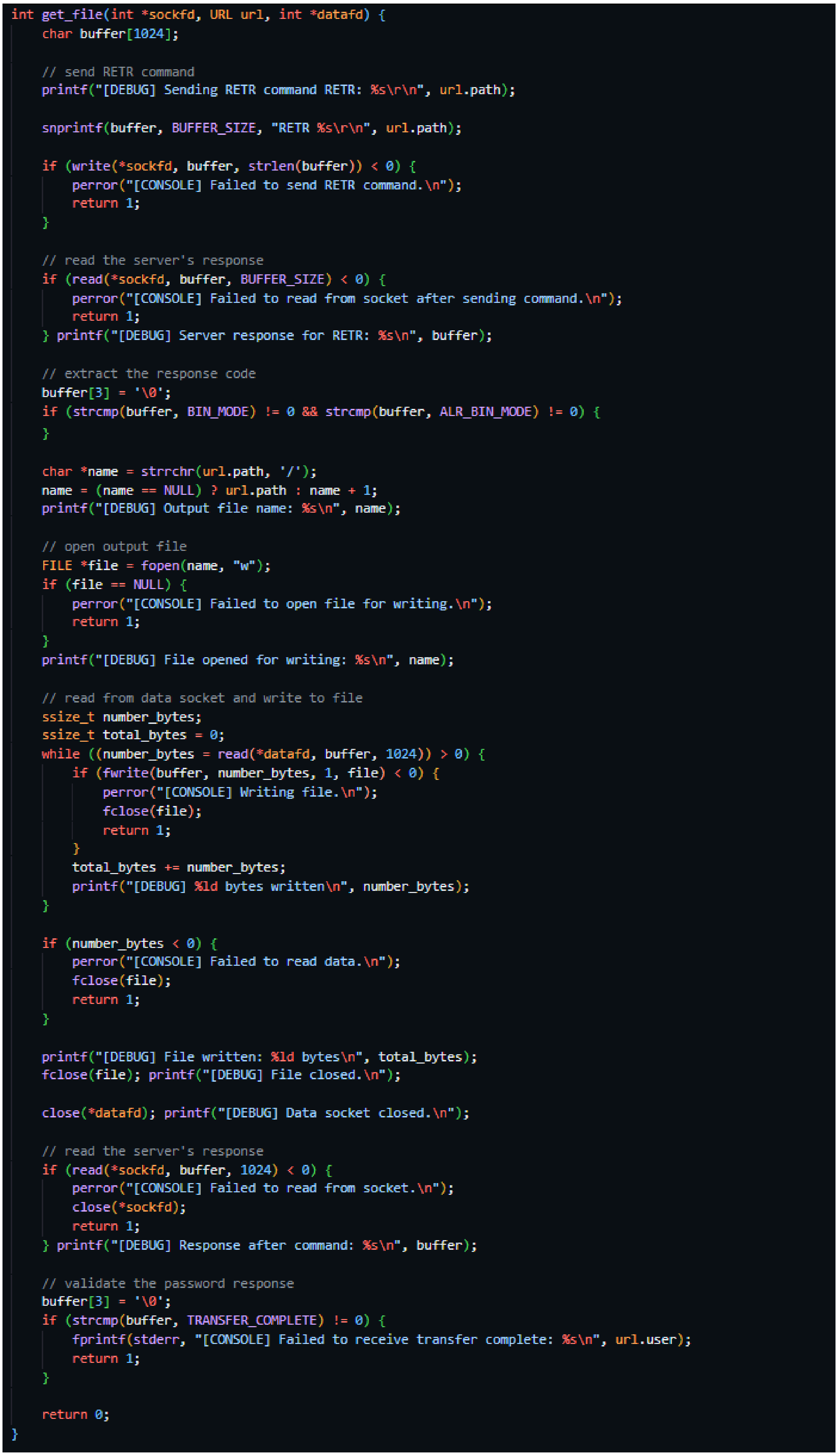
```c
int open_connection(int *sockfd, URL url) {

    struct hostent* host = gethostbyname(url.host);
    if (host == NULL) {
        perror("[CONSOLE] Invalid host.\n");
        return 1;
    }

    // convert the resolved IP address
    const char* address = inet_ntoa(*(struct in_addr*)host->h_addr);
    struct sockaddr_in server_addr;

    // initialize addr struct
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(address);
    server_addr.sin_port = htons(TCP_PORT);

    if ((*sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("[CONSOLE] Opening the socket.\n");
        return 1;
    }
    printf("[DEBUG] Socket opened.\n");

    // connect to server
    if (connect(*sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("[CONSOLE] Connecting to the server.\n");
        close(*sockfd);
        return 1;
    }
    printf("[DEBUG] Connected to server.\n");

    return 0;
}
```



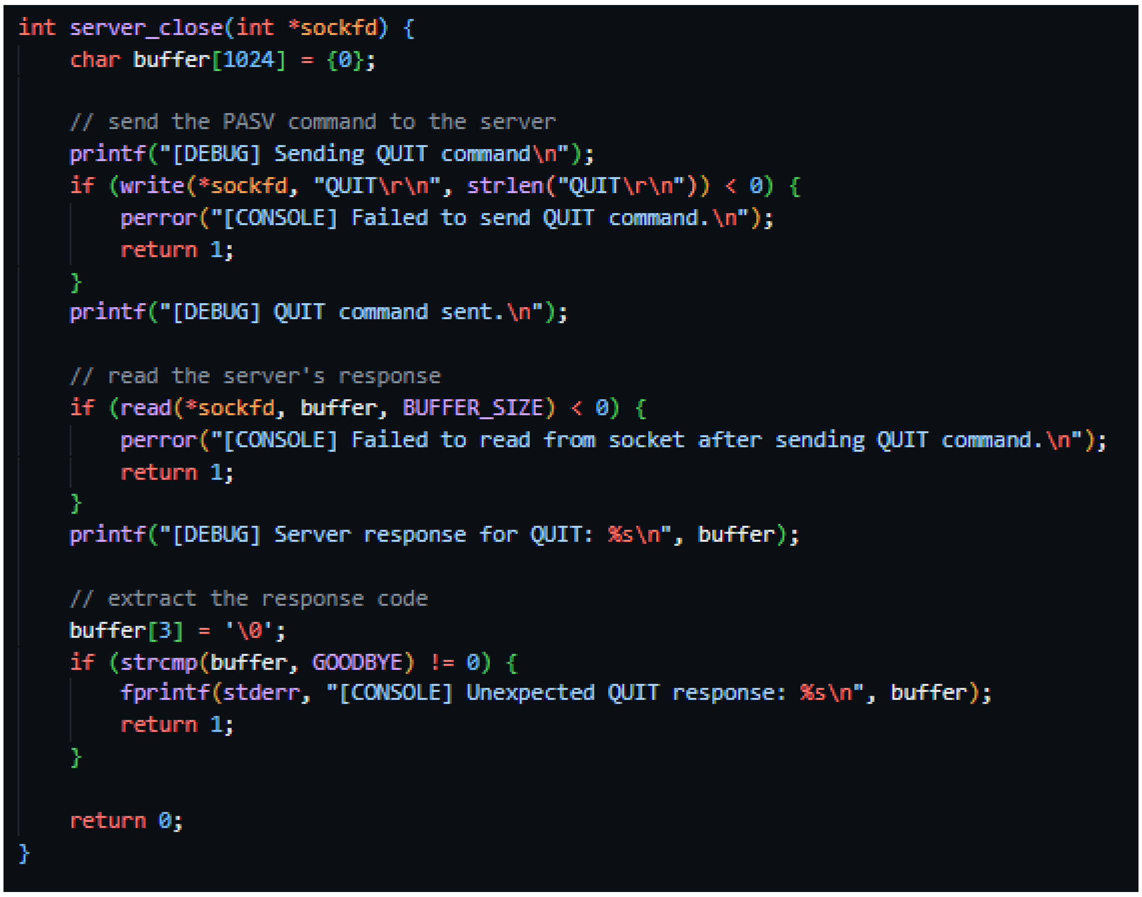
```c
int login(int *sockfd, URL url) {
    char user[BUFFER_SIZE], pass[BUFFER_SIZE];
    char buffer[1024] = {0};

    // format the FTP commands for username and password
    snprintf(user, BUFFER_SIZE, "USER %s\r\n", url.user); snprintf(pass, BUFFER_SIZE, "PASS %s\r\n", url.pass);
    printf("[DEBUG] Sending USER command: %s", user); printf("[DEBUG] Sending PASS command: %s", pass);

    // send the username and read the response
    if (write(*sockfd, user, strlen(user)) < 0) {
        perror("[CONSOLE] Failed to send username.\n");
        return 1;
    }
    printf("[DEBUG] USER command sent.\n");

    if (read(*sockfd, buffer, 1024) < 0) {
        perror("[CONSOLE] Failed to read from socket after sending username.\n");
        return 1;
    }
    printf("[DEBUG] Response after USER command: %s\n", buffer);

    // validate the username response
    buffer[3] = '\0';
    if (strcmp(buffer, READY_PASS) != 0) {
        fprintf(stderr, "[CONSOLE] Invalid username: %s\n", url.user);
        return 1;
    }

    // send the password and read the response
    if (write(*sockfd, pass, strlen(pass)) < 0) {
        perror("[CONSOLE] Failed to send password:\n");
        close(*sockfd);
        return 1;
    }
    printf("[DEBUG] PASS command sent.\n");

    if (read(*sockfd, buffer, 1024) < 0) {
        perror("[CONSOLE] Failed to read from socket after sending password.\n");
        close(*sockfd);
        return 1;
    }
    printf("[DEBUG] Response after PASS command: %s\n", buffer);

    // validate the password response
    buffer[3] = '\0';
    if (strcmp(buffer, LOGIN_SUCCESS) != 0) {
        fprintf(stderr, "[CONSOLE] Invalid password: %s\n", url.user);
        return 1;
    }

    return 0;
}
```



```c
int enter_passive_mode(int *sockfd, char* address) {
    char buffer[1024] = {0};

    // send the PASV command to the server
    printf("[DEBUG] Sending PASV command\n");
    if (write(*sockfd, "PASV\r\n", strlen("PASV\r\n")) < 0) {
        perror("[CONSOLE] Failed to send PASV command.\n");
        exit(-1);
    }

    // read the server's response
    if (read(*sockfd, buffer, BUFFER_SIZE) < 0) {
        perror("[CONSOLE] Failed to read from socket after sending PASV command.\n");
        exit(-1);
    } printf("[DEBUG] Server response for PASV: %s\n", buffer);

    char code[4]; strncpy(code, buffer, 3); code[3] = '\0';

    if (strcmp(code, PASSIVE_MODE_READY) != 0) {
        fprintf(stderr, "[CONSOLE] Unexpected PASV response: %s\n", buffer);
        exit(-1);
    }

    // parse the server's response for IP address and port
    char* start = strchr(buffer, '('); char* end = strchr(buffer, ')');
    if (!start || !end || start >= end) {
        fprintf(stderr, "[CONSOLE] Malformed PASV response: %s\n", buffer);
        exit(-1);
    } printf("[DEBUG] PASV response data: %s\n", start + 1);

    // extract the data between parentheses
    *end = '\0'; char* data = start + 1;

    // parse the IP address
    int i = 0; char* token = strtok(data, ",");
    while (token && i < 4) {
        strcat(address, token);
        if (i < 3) strcat(address, ".");
        token = strtok(NULL, ",");
        i++;
    } printf("[CONSOLE] Parsed IP address: %s\n", address);

    // parse the port
    if (!token) {
        fprintf(stderr, "[CONSOLE] Incomplete PASV response: %s\n", buffer);
        exit(-1);
    }
    int portMSB = atoi(token); printf("[CONSOLE] Parsed port MSB: %d\n", portMSB);
    token = strtok(NULL, ",");
    if (!token) {
        fprintf(stderr, "[CONSOLE] Missing port information in PASV response: %s\n", buffer);
        exit(-1);
    }
    int portLSB = atoi(token); printf("[CONSOLE] Parsed port LSB: %d\n", portLSB);

    // calculate and return the port number
    int port = (portMSB << 8) | portLSB; printf("[CONSOLE] Calculated port: %d\n", port);
    return port;
}
```

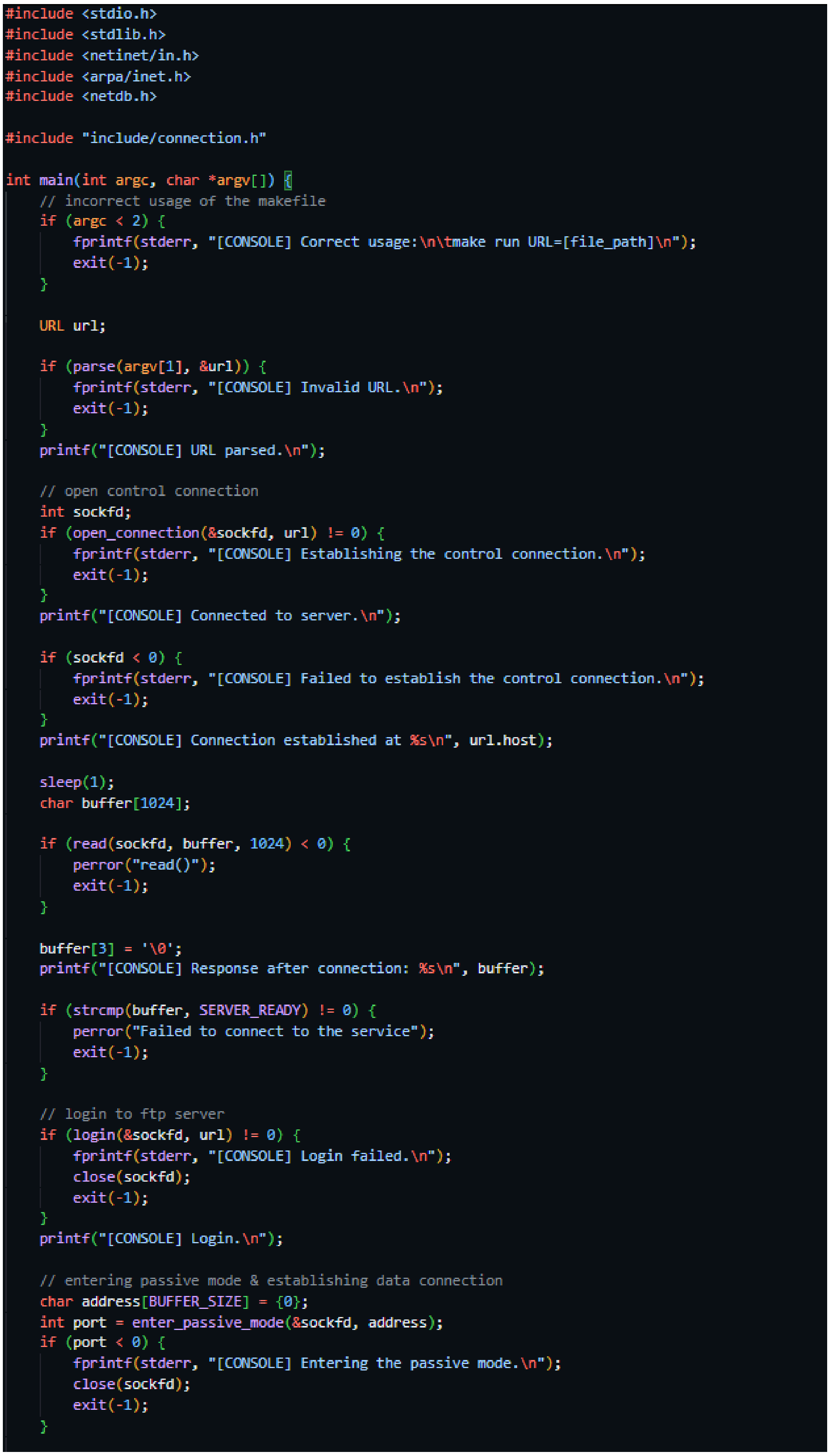


```c
int get_file(int *sockfd, URL url, int *datafd) {
    char buffer[1024];

    // send RETR command
    printf("[DEBUG] Sending RETR command RETR: %s\r\n", url.path);

    snprintf(buffer, BUFFER_SIZE, "RETR %s\r\n", url.path);

    if (write(*sockfd, buffer, strlen(buffer)) < 0) {
        perror("[CONSOLE] Failed to send RETR command.\n");
        return 1;
    }

    // read the server's response
    if (read(*sockfd, buffer, BUFFER_SIZE) < 0) {
        perror("[CONSOLE] Failed to read from socket after sending command.\n");
        return 1;
    } printf("[DEBUG] Server response for RETR: %s\n", buffer);

    // extract the response code
    buffer[3] = '\0';
    if (strcmp(buffer, BIN_MODE) != 0 && strcmp(buffer, ALR_BIN_MODE) != 0) {
    }

    char *name = strrchr(url.path, '/');
    name = (name == NULL) ? url.path : name + 1;
    printf("[DEBUG] Output file name: %s\n", name);

    // open output file
    FILE *file = fopen(name, "w");
    if (file == NULL) {
        perror("[CONSOLE] Failed to open file for writing.\n");
        return 1;
    }
    printf("[DEBUG] File opened for writing: %s\n", name);

    // read from data socket and write to file
    ssize_t number_bytes;
    ssize_t total_bytes = 0;
    while ((number_bytes = read(*datafd, buffer, 1024)) > 0) {
        if (fwrite(buffer, number_bytes, 1, file) < 0) {
            perror("[CONSOLE] Writing file.\n");
            fclose(file);
            return 1;
        }
        total_bytes += number_bytes;
        printf("[DEBUG] %ld bytes written\n", number_bytes);
    }

    if (number_bytes < 0) {
        perror("[CONSOLE] Failed to read data.\n");
        fclose(file);
        return 1;
    }

    printf("[DEBUG] File written: %ld bytes\n", total_bytes);
    fclose(file); printf("[DEBUG] File closed.\n");

    close(*datafd); printf("[DEBUG] Data socket closed.\n");

    // read the server's response
    if (read(*sockfd, buffer, 1024) < 0) {
        perror("[CONSOLE] Failed to read from socket.\n");
        close(*sockfd);
        return 1;
    } printf("[DEBUG] Response after command: %s\n", buffer);

    // validate the password response
    buffer[3] = '\0';
    if (strcmp(buffer, TRANSFER_COMPLETE) != 0) {
        fprintf(stderr, "[CONSOLE] Failed to receive transfer complete: %s\n", url.user);
        return 1;
    }

    return 0;
}
```

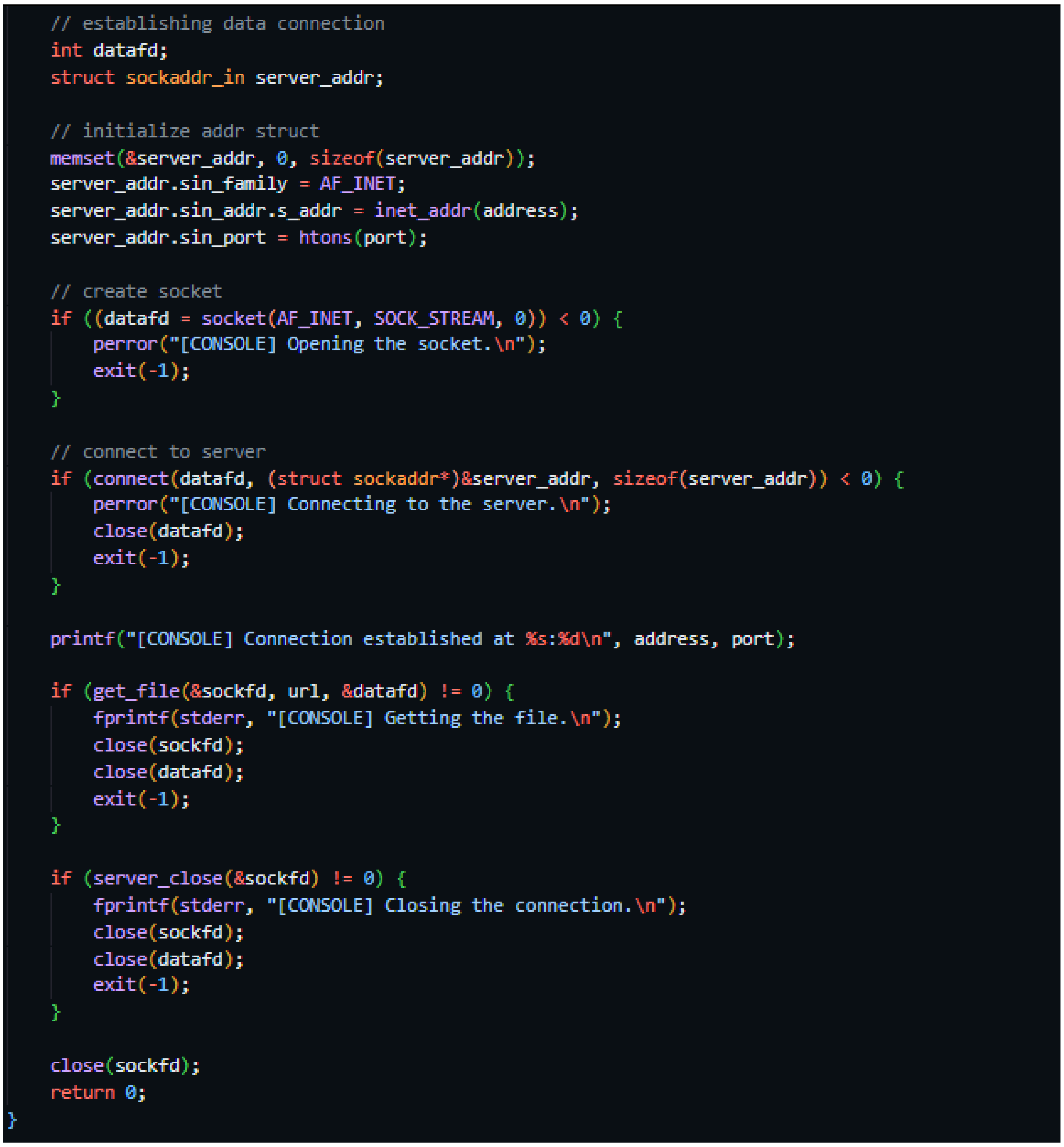


```c
int server_close(int *sockfd) {
    char buffer[1024] = {0};

    // send the PASV command to the server
    printf("[DEBUG] Sending QUIT command\n");
    if (write(*sockfd, "QUIT\r\n", strlen("QUIT\r\n")) < 0) {
        perror("[CONSOLE] Failed to send QUIT command.\n");
        return 1;
    }
    printf("[DEBUG] QUIT command sent.\n");

    // read the server's response
    if (read(*sockfd, buffer, BUFFER_SIZE) < 0) {
        perror("[CONSOLE] Failed to read from socket after sending QUIT command.\n");
        return 1;
    }
    printf("[DEBUG] Server response for QUIT: %s\n", buffer);

    // extract the response code
    buffer[3] = '\0';
    if (strcmp(buffer, GOODBYE) != 0) {
        fprintf(stderr, "[CONSOLE] Unexpected QUIT response: %s\n", buffer);
        return 1;
    }

    return 0;
}
```

Appendix III: main.c



```c
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include "include/connection.h"

int main(int argc, char *argv[]) {
    // incorrect usage of the makefile
    if (argc < 2) {
        fprintf(stderr, "[CONSOLE] Correct usage:\n\tmake run URL=[file_path]\n");
        exit(-1);
    }

    URL url;

    if (parse(argv[1], &url)) {
        fprintf(stderr, "[CONSOLE] Invalid URL.\n");
        exit(-1);
    }
    printf("[CONSOLE] URL parsed.\n");

    // open control connection
    int sockfd;
    if (open_connection(&sockfd, url) != 0) {
        fprintf(stderr, "[CONSOLE] Establishing the control connection.\n");
        exit(-1);
    }
    printf("[CONSOLE] Connected to server.\n");

    if (sockfd < 0) {
        fprintf(stderr, "[CONSOLE] Failed to establish the control connection.\n");
        exit(-1);
    }
    printf("[CONSOLE] Connection established at %s\n", url.host);

    sleep(1);
    char buffer[1024];

    if (read(sockfd, buffer, 1024) < 0) {
        perror("read()");
        exit(-1);
    }

    buffer[3] = '\0';
    printf("[CONSOLE] Response after connection: %s\n", buffer);

    if (strcmp(buffer, SERVER_READY) != 0) {
        perror("Failed to connect to the service");
        exit(-1);
    }

    // login to ftp server
    if (login(&sockfd, url) != 0) {
        fprintf(stderr, "[CONSOLE] Login failed.\n");
        close(sockfd);
        exit(-1);
    }
    printf("[CONSOLE] Login.\n");

    // entering passive mode & establishing data connection
    char address[BUFFER_SIZE] = {0};
    int port = enter_passive_mode(&sockfd, address);
    if (port < 0) {
        fprintf(stderr, "[CONSOLE] Entering the passive mode.\n");
        close(sockfd);
        exit(-1);
    }
```

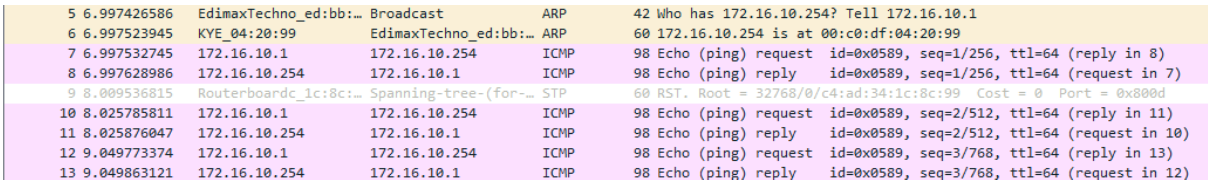


```c
    // establishing data connection
    int datafd;
    struct sockaddr_in server_addr;

    // initialize addr struct
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(address);
    server_addr.sin_port = htons(port);

    // create socket
    if ((datafd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("[CONSOLE] Opening the socket.\n");
        exit(-1);
    }

    // connect to server
    if (connect(datafd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("[CONSOLE] Connecting to the server.\n");
        close(datafd);
        exit(-1);
    }

    printf("[CONSOLE] Connection established at %s:%d\n", address, port);

    if (get_file(&sockfd, url, &datafd) != 0) {
        fprintf(stderr, "[CONSOLE] Getting the file.\n");
        close(sockfd);
        close(datafd);
        exit(-1);
    }

    if (server_close(&sockfd) != 0) {
        fprintf(stderr, "[CONSOLE] Closing the connection.\n");
        close(sockfd);
        close(datafd);
        exit(-1);
    }

    close(sockfd);
    return 0;
}
```

Appendix IV: Experiments logs

Experiment 1:

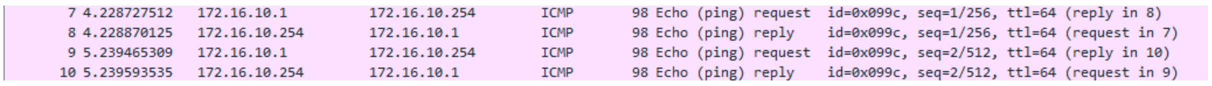
Tux3 to Tux4:



```
 5 6.997426586  EdimaxTechno_ed:bb:… Broadcast             ARP   42 Who has 172.16.10.254? Tell 172.16.10.1
 6 6.997523945  KYE_04:20:99         EdimaxTechno_ed:bb:… ARP   60 172.16.10.254 is at 00:c0:df:04:20:99
 7 6.997532745  172.16.10.1          172.16.10.254         ICMP  98 Echo (ping) request  id=0x0589, seq=1/256, ttl=64 (reply in 8)
 8 6.997628986  172.16.10.254        172.16.10.1           ICMP  98 Echo (ping) reply    id=0x0589, seq=1/256, ttl=64 (request in 7)
 9 8.009536815  Routerboardc_1c:8c:… Spanning-tree-(for-… STP   60 RST. Root = 32768/0/c4:ad:34:1c:8c:99  Cost = 0  Port = 0x800d
10 8.025785811  172.16.10.1          172.16.10.254         ICMP  98 Echo (ping) request  id=0x0589, seq=2/512, ttl=64 (reply in 11)
11 8.025876047  172.16.10.254        172.16.10.1           ICMP  98 Echo (ping) reply    id=0x0589, seq=2/512, ttl=64 (request in 10)
12 9.049773374  172.16.10.1          172.16.10.254         ICMP  98 Echo (ping) request  id=0x0589, seq=3/768, ttl=64 (reply in 13)
13 9.049863121  172.16.10.254        172.16.10.1           ICMP  98 Echo (ping) reply    id=0x0589, seq=3/768, ttl=64 (request in 12)
```

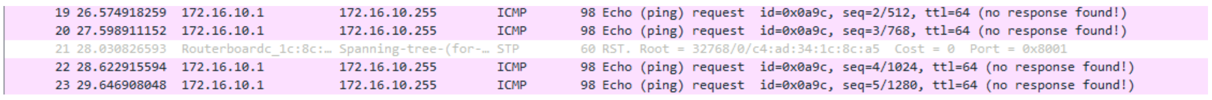
Experiment 2:

Tux3 to Tux4:



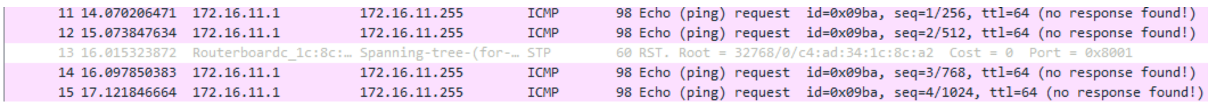
```
 7 4.228727512   172.16.10.1     172.16.10.254   ICMP   98 Echo (ping) request  id=0x099c, seq=1/256, ttl=64 (reply in 8)
 8 4.228870125   172.16.10.254   172.16.10.1     ICMP   98 Echo (ping) reply    id=0x099c, seq=1/256, ttl=64 (request in 7)
 9 5.239465309   172.16.10.1     172.16.10.254   ICMP   98 Echo (ping) request  id=0x099c, seq=2/512, ttl=64 (reply in 10)
10 5.239593535   172.16.10.254   172.16.10.1     ICMP   98 Echo (ping) reply    id=0x099c, seq=2/512, ttl=64 (request in 9)
```

Tux3 broadcast:



```
19 26.574918259  172.16.10.1            172.16.10.255          ICMP   98 Echo (ping) request  id=0x0a9c, seq=2/512, ttl=64 (no response found!)
20 27.598911152  172.16.10.1            172.16.10.255          ICMP   98 Echo (ping) request  id=0x0a9c, seq=3/768, ttl=64 (no response found!)
21 28.030826593  Routerboardc_1c:8c:…   Spanning-tree-(for-…   STP    60 RST. Root = 32768/0/c4:ad:34:1c:8c:a5  Cost = 0  Port = 0x8001
22 28.622915594  172.16.10.1            172.16.10.255          ICMP   98 Echo (ping) request  id=0x0a9c, seq=4/1024, ttl=64 (no response found!)
23 29.646908048  172.16.10.1            172.16.10.255          ICMP   98 Echo (ping) request  id=0x0a9c, seq=5/1280, ttl=64 (no response found!)
```

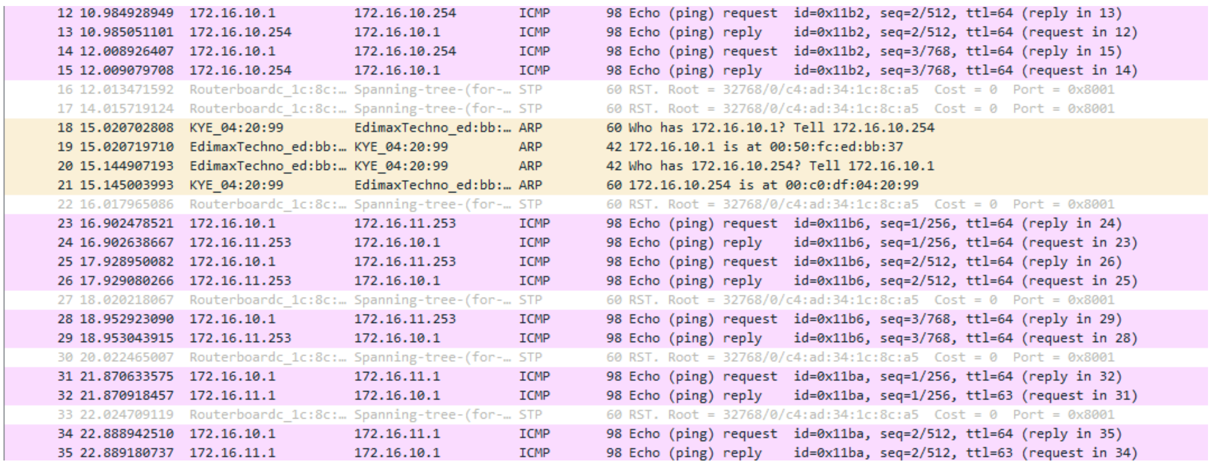
Tux2 broadcast:



```
11 14.070206471  172.16.11.1            172.16.11.255          ICMP   98 Echo (ping) request  id=0x09ba, seq=1/256, ttl=64 (no response found!)
12 15.073847634  172.16.11.1            172.16.11.255          ICMP   98 Echo (ping) request  id=0x09ba, seq=2/512, ttl=64 (no response found!)
13 16.015323872  Routerboardc_1c:8c:…   Spanning-tree-(for-…   STP    60 RST. Root = 32768/0/c4:ad:34:1c:8c:a2  Cost = 0  Port = 0x8001
14 16.097850383  172.16.11.1            172.16.11.255          ICMP   98 Echo (ping) request  id=0x09ba, seq=3/768, ttl=64 (no response found!)
15 17.121846664  172.16.11.1            172.16.11.255          ICMP   98 Echo (ping) request  id=0x09ba, seq=4/1024, ttl=64 (no response found!)
```

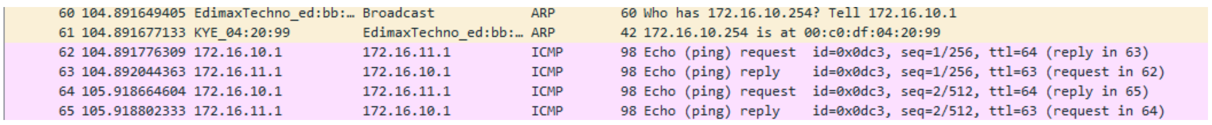
Experiment 3

Ping tux4 in eth1, tux4 in eth2 and tux2 from tux3::



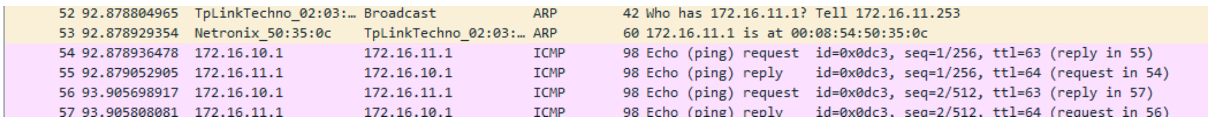
```
12 10.984928949  172.16.10.1            172.16.10.254          ICMP   98 Echo (ping) request  id=0x11b2, seq=2/512, ttl=64 (reply in 13)
13 10.985051101  172.16.10.254          172.16.10.1            ICMP   98 Echo (ping) reply    id=0x11b2, seq=2/512, ttl=64 (request in 12)
14 12.008926407  172.16.10.1            172.16.10.254          ICMP   98 Echo (ping) request  id=0x11b2, seq=3/768, ttl=64 (reply in 15)
15 12.009079708  172.16.10.254          172.16.10.1            ICMP   98 Echo (ping) reply    id=0x11b2, seq=3/768, ttl=64 (request in 14)
16 12.013471592  Routerboardc_1c:8c:…   Spanning-tree-(for-…   STP    60 RST. Root = 32768/0/c4:ad:34:1c:8c:a5  Cost = 0  Port = 0x8001
17 14.015719124  Routerboardc_1c:8c:…   Spanning-tree-(for-…   STP    60 RST. Root = 32768/0/c4:ad:34:1c:8c:a5  Cost = 0  Port = 0x8001
18 15.020702808  KYE_04:20:99           EdimaxTechno_ed:bb:…   ARP    60 Who has 172.16.10.1? Tell 172.16.10.254
19 15.020719710  EdimaxTechno_ed:bb:…   KYE_04:20:99           ARP    42 172.16.10.1 is at 00:50:fc:ed:bb:37
20 15.144907193  EdimaxTechno_ed:bb:…   KYE_04:20:99           ARP    42 Who has 172.16.10.254? Tell 172.16.10.1
21 15.145003993  KYE_04:20:99           EdimaxTechno_ed:bb:…   ARP    60 172.16.10.254 is at 00:c0:df:04:20:99
22 16.017965086  Routerboardc_1c:8c:…   Spanning-tree-(for-…   STP    60 RST. Root = 32768/0/c4:ad:34:1c:8c:a5  Cost = 0  Port = 0x8001
23 16.902478521  172.16.10.1            172.16.11.253          ICMP   98 Echo (ping) request  id=0x11b6, seq=1/256, ttl=64 (reply in 24)
24 16.902638667  172.16.11.253          172.16.10.1            ICMP   98 Echo (ping) reply    id=0x11b6, seq=1/256, ttl=64 (request in 23)
25 17.928950082  172.16.10.1            172.16.11.253          ICMP   98 Echo (ping) request  id=0x11b6, seq=2/512, ttl=64 (reply in 26)
26 17.929080266  172.16.11.253          172.16.10.1            ICMP   98 Echo (ping) reply    id=0x11b6, seq=2/512, ttl=64 (request in 25)
27 18.020218067  Routerboardc_1c:8c:…   Spanning-tree-(for-…   STP    60 RST. Root = 32768/0/c4:ad:34:1c:8c:a5  Cost = 0  Port = 0x8001
28 18.952923090  172.16.10.1            172.16.11.253          ICMP   98 Echo (ping) request  id=0x11b6, seq=3/768, ttl=64 (reply in 29)
29 18.953043915  172.16.11.253          172.16.10.1            ICMP   98 Echo (ping) reply    id=0x11b6, seq=3/768, ttl=64 (request in 28)
30 20.022465007  Routerboardc_1c:8c:…   Spanning-tree-(for-…   STP    60 RST. Root = 32768/0/c4:ad:34:1c:8c:a5  Cost = 0  Port = 0x8001
31 21.870633575  172.16.10.1            172.16.11.1            ICMP   98 Echo (ping) request  id=0x11ba, seq=1/256, ttl=64 (reply in 32)
32 21.870918457  172.16.11.1            172.16.10.1            ICMP   98 Echo (ping) reply    id=0x11ba, seq=1/256, ttl=63 (request in 31)
33 22.024709119  Routerboardc_1c:8c:…   Spanning-tree-(for-…   STP    60 RST. Root = 32768/0/c4:ad:34:1c:8c:a5  Cost = 0  Port = 0x8001
34 22.888942510  172.16.10.1            172.16.11.1            ICMP   98 Echo (ping) request  id=0x11ba, seq=2/512, ttl=64 (reply in 35)
35 22.889180737  172.16.11.1            172.16.10.1            ICMP   98 Echo (ping) reply    id=0x11ba, seq=2/512, ttl=63 (request in 34)
```

Tux3 to Tux2 (Tux4 eth1):



```
60 104.891649405 EdimaxTechno_ed:bb:…  Broadcast              ARP    60 Who has 172.16.10.254? Tell 172.16.10.1
61 104.891677133 KYE_04:20:99          EdimaxTechno_ed:bb:…   ARP    42 172.16.10.254 is at 00:c0:df:04:20:99
62 104.891776309 172.16.10.1           172.16.11.1            ICMP   98 Echo (ping) request  id=0x0dc3, seq=1/256, ttl=64 (reply in 63)
63 104.892044363 172.16.11.1           172.16.10.1            ICMP   98 Echo (ping) reply    id=0x0dc3, seq=1/256, ttl=63 (request in 62)
64 105.918664604 172.16.10.1           172.16.11.1            ICMP   98 Echo (ping) request  id=0x0dc3, seq=2/512, ttl=64 (reply in 65)
65 105.918802333 172.16.11.1           172.16.10.1            ICMP   98 Echo (ping) reply    id=0x0dc3, seq=2/512, ttl=63 (request in 64)
```

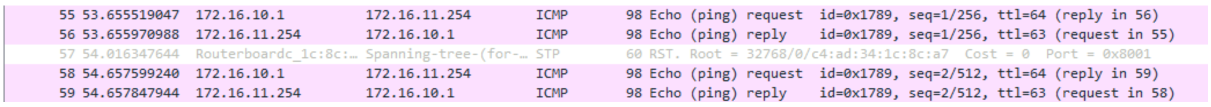
Tux3 to Tux2 (Tux4 eth2):



```
52 92.878804965  TpLinkTechno_02:03:…  Broadcast              ARP    42 Who has 172.16.11.1? Tell 172.16.11.253
53 92.878929354  Netronix_50:35:0c     TpLinkTechno_02:03:…   ARP    60 172.16.11.1 is at 00:08:54:50:35:0c
54 92.878936478  172.16.10.1           172.16.11.1            ICMP   98 Echo (ping) request  id=0x0dc3, seq=1/256, ttl=63 (reply in 55)
55 92.879052905  172.16.11.1           172.16.10.1            ICMP   98 Echo (ping) reply    id=0x0dc3, seq=1/256, ttl=64 (request in 54)
56 93.905698917  172.16.10.1           172.16.11.1            ICMP   98 Echo (ping) request  id=0x0dc3, seq=2/512, ttl=63 (reply in 57)
57 93.905808081  172.16.11.1           172.16.10.1            ICMP   98 Echo (ping) reply    id=0x0dc3, seq=2/512, ttl=64 (request in 56)
```

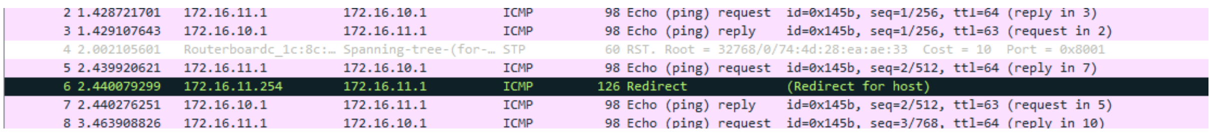
Experiment 4:

Tux3 to router RC:



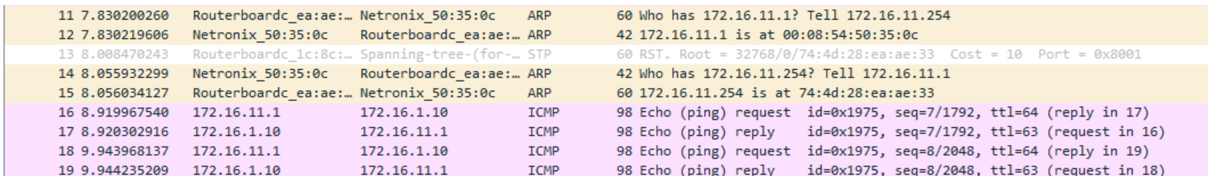
```
 55 53.655519047  172.16.10.1              172.16.11.254           ICMP      98 Echo (ping) request  id=0x1789, seq=1/256, ttl=64 (reply in 56)
 56 53.655970988  172.16.11.254            172.16.10.1             ICMP      98 Echo (ping) reply    id=0x1789, seq=1/256, ttl=63 (request in 55)
 57 54.016347644  Routerboardc_1c:8c:…     Spanning-tree-(for-…    STP       60 RST. Root = 32768/0/c4:ad:34:1c:8c:a7  Cost = 0  Port = 0x8001
 58 54.657599240  172.16.10.1              172.16.11.254           ICMP      98 Echo (ping) request  id=0x1789, seq=2/512, ttl=64 (reply in 59)
 59 54.657847944  172.16.11.254            172.16.10.1             ICMP      98 Echo (ping) reply    id=0x1789, seq=2/512, ttl=63 (request in 58)
```

Tux2 to tux3:



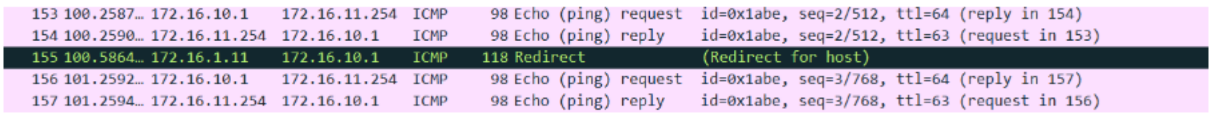
```
 2 1.428721701   172.16.11.1             172.16.10.1            ICMP      98 Echo (ping) request  id=0x145b, seq=1/256, ttl=64 (reply in 3)
 3 1.429107643   172.16.10.1             172.16.11.1            ICMP      98 Echo (ping) reply    id=0x145b, seq=1/256, ttl=63 (request in 2)
 4 2.002105601   Routerboardc_1c:8c:…    Spanning-tree-(for-…   STP       60 RST. Root = 32768/0/74:4d:28:ea:ae:33  Cost = 10  Port = 0x8001
 5 2.439920621   172.16.11.1             172.16.10.1            ICMP      98 Echo (ping) request  id=0x145b, seq=2/512, ttl=64 (reply in 7)
 6 2.440079299   172.16.11.254           172.16.11.1            ICMP     126 Redirect              (Redirect for host)
 7 2.440276251   172.16.10.1             172.16.11.1            ICMP      98 Echo (ping) reply    id=0x145b, seq=2/512, ttl=63 (request in 5)
 8 3.463908826   172.16.11.1             172.16.10.1            ICMP      98 Echo (ping) request  id=0x145b, seq=3/768, ttl=64 (reply in 10)
```

Tux2 to ???:



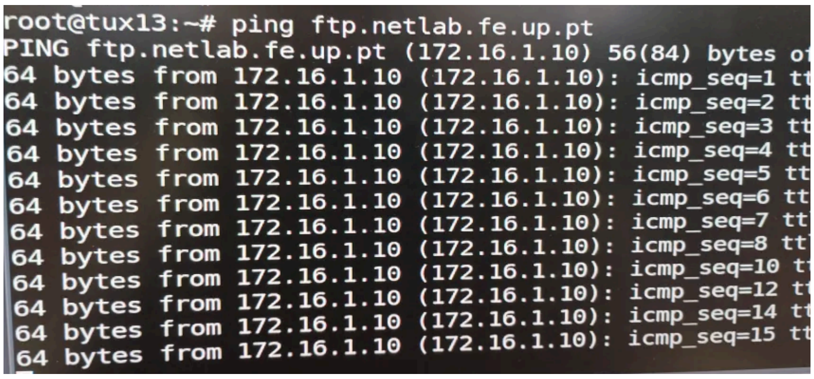
```
11 7.830200260   Routerboardc_ea:ae:…  Netronix_50:35:0c     ARP       60 Who has 172.16.11.1? Tell 172.16.11.254
12 7.830219606   Netronix_50:35:0c     Routerboardc_ea:ae:…  ARP       42 172.16.11.1 is at 00:08:54:50:35:0c
13 8.008470243   Routerboardc_1c:8c:…  Spanning-tree-(for-…  STP       60 RST. Root = 32768/0/74:4d:28:ea:ae:33  Cost = 10  Port = 0x8001
14 8.055932299   Netronix_50:35:0c     Routerboardc_ea:ae:…  ARP       42 Who has 172.16.11.254? Tell 172.16.11.1
15 8.056034127   Routerboardc_ea:ae:…  Netronix_50:35:0c     ARP       60 172.16.11.254 is at 74:4d:28:ea:ae:33
16 8.919967540   172.16.11.1           172.16.1.10           ICMP      98 Echo (ping) request  id=0x1975, seq=7/1792, ttl=64 (reply in 17)
17 8.920302916   172.16.1.10           172.16.11.1           ICMP      98 Echo (ping) reply    id=0x1975, seq=7/1792, ttl=63 (request in 16)
18 9.943968137   172.16.11.1           172.16.1.10           ICMP      98 Echo (ping) request  id=0x1975, seq=8/2048, ttl=64 (reply in 19)
19 9.944235209   172.16.1.10           172.16.11.1           ICMP      98 Echo (ping) reply    id=0x1975, seq=8/2048, ttl=63 (request in 18)
```

Tux3 to router RC with NAT disabled:



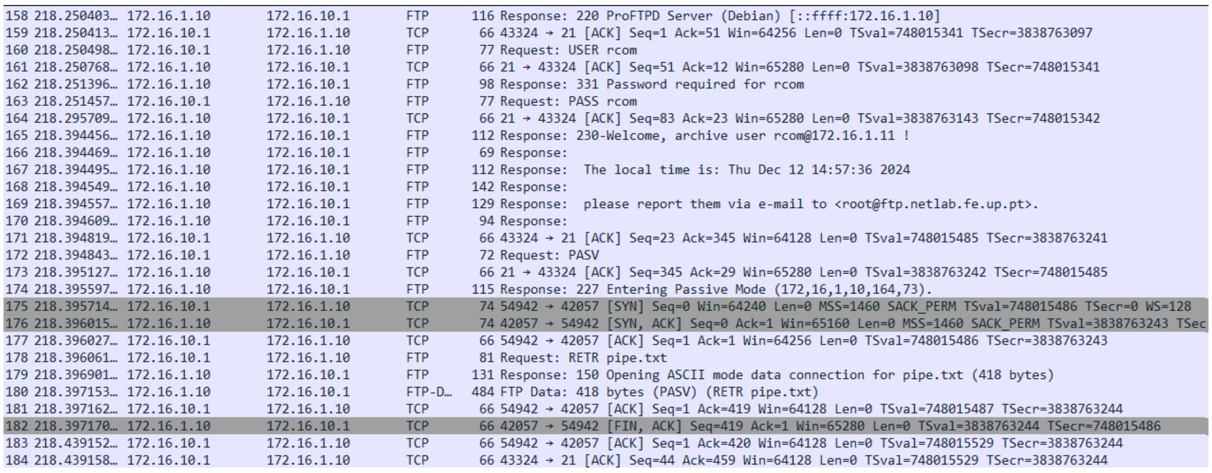
```
153 100.2587… 172.16.10.1   172.16.11.254  ICMP    98 Echo (ping) request  id=0x1abe, seq=2/512, ttl=64 (reply in 154)
154 100.2590… 172.16.11.254 172.16.10.1    ICMP    98 Echo (ping) reply    id=0x1abe, seq=2/512, ttl=63 (request in 153)
155 100.5864… 172.16.1.11   172.16.10.1    ICMP   118 Redirect             (Redirect for host)
156 101.2592… 172.16.10.1   172.16.11.254  ICMP    98 Echo (ping) request  id=0x1abe, seq=3/768, ttl=64 (reply in 157)
157 101.2594… 172.16.11.254 172.16.10.1    ICMP    98 Echo (ping) reply    id=0x1abe, seq=3/768, ttl=63 (request in 156)
```

Experiment 5:



```
root@tux13:~# ping ftp.netlab.fe.up.pt
PING ftp.netlab.fe.up.pt (172.16.1.10) 56(84) bytes o
64 bytes from 172.16.1.10 (172.16.1.10): icmp_seq=1 tt
64 bytes from 172.16.1.10 (172.16.1.10): icmp_seq=2 tt
64 bytes from 172.16.1.10 (172.16.1.10): icmp_seq=3 tt
64 bytes from 172.16.1.10 (172.16.1.10): icmp_seq=4 tt
64 bytes from 172.16.1.10 (172.16.1.10): icmp_seq=5 tt
64 bytes from 172.16.1.10 (172.16.1.10): icmp_seq=6 tt
64 bytes from 172.16.1.10 (172.16.1.10): icmp_seq=7 tt
64 bytes from 172.16.1.10 (172.16.1.10): icmp_seq=8 tt
64 bytes from 172.16.1.10 (172.16.1.10): icmp_seq=10 t
64 bytes from 172.16.1.10 (172.16.1.10): icmp_seq=12 t
64 bytes from 172.16.1.10 (172.16.1.10): icmp_seq=14 t
64 bytes from 172.16.1.10 (172.16.1.10): icmp_seq=15 t
64 bytes from
```

Experiment 6:



```
158 218.250403… 172.16.1.10     172.16.10.1     FTP     116 Response: 220 ProFTPD Server (Debian) [::ffff:172.16.1.10]
159 218.250413… 172.16.10.1     172.16.1.10     TCP      66 43324 → 21 [ACK] Seq=1 Ack=51 Win=64256 Len=0 TSval=748015341 TSecr=3838763097
160 218.250498… 172.16.10.1     172.16.1.10     FTP      77 Request: USER rcom
161 218.250768… 172.16.1.10     172.16.10.1     TCP      66 21 → 43324 [ACK] Seq=51 Ack=12 Win=65280 Len=0 TSval=3838763098 TSecr=748015341
162 218.251396… 172.16.1.10     172.16.10.1     FTP      98 Response: 331 Password required for rcom
163 218.251457… 172.16.10.1     172.16.1.10     FTP      77 Request: PASS rcom
164 218.295709… 172.16.1.10     172.16.10.1     TCP      66 21 → 43324 [ACK] Seq=83 Ack=23 Win=65280 Len=0 TSval=3838763143 TSecr=748015342
165 218.394456… 172.16.1.10     172.16.10.1     FTP     112 Response: 230-Welcome, archive user rcom@172.16.1.11 !
166 218.394469… 172.16.1.10     172.16.10.1     FTP      69 Response:
167 218.394495… 172.16.1.10     172.16.10.1     FTP     112 Response:  The local time is: Thu Dec 12 14:57:36 2024
168 218.394549… 172.16.1.10     172.16.10.1     FTP     142 Response:
169 218.394557… 172.16.1.10     172.16.10.1     FTP     129 Response:  please report them via e-mail to <root@ftp.netlab.fe.up.pt>.
170 218.394609… 172.16.1.10     172.16.10.1     FTP      94 Response:
171 218.394819… 172.16.10.1     172.16.1.10     TCP      66 43324 → 21 [ACK] Seq=23 Ack=345 Win=64128 Len=0 TSval=748015485 TSecr=3838763241
172 218.394843… 172.16.10.1     172.16.1.10     FTP      72 Request: PASV
173 218.395127… 172.16.1.10     172.16.10.1     TCP      66 21 → 43324 [ACK] Seq=345 Ack=29 Win=65280 Len=0 TSval=3838763242 TSecr=748015485
174 218.395597… 172.16.1.10     172.16.10.1     FTP     115 Response: 227 Entering Passive Mode (172,16,1,10,164,73).
175 218.395714… 172.16.10.1     172.16.1.10     TCP      74 54942 → 42057 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=748015486 TSecr=0 WS=128
176 218.396015… 172.16.1.10     172.16.10.1     TCP      74 42057 → 54942 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=3838763243 TSec
177 218.396027… 172.16.10.1     172.16.1.10     TCP      66 54942 → 42057 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=748015486 TSecr=3838763243
178 218.396061… 172.16.10.1     172.16.1.10     FTP      81 Request: RETR pipe.txt
179 218.396901… 172.16.1.10     172.16.10.1     FTP     131 Response: 150 Opening ASCII mode data connection for pipe.txt (418 bytes)
180 218.397153… 172.16.1.10     172.16.10.1     FTP-D…  484 FTP Data: 418 bytes (PASV) (RETR pipe.txt)
181 218.397162… 172.16.10.1     172.16.1.10     TCP      66 54942 → 42057 [ACK] Seq=1 Ack=419 Win=64128 Len=0 TSval=748015487 TSecr=3838763244
182 218.397170… 172.16.1.10     172.16.10.1     TCP      66 42057 → 54942 [FIN, ACK] Seq=419 Ack=1 Win=65280 Len=0 TSval=3838763244 TSecr=748015486
183 218.439152… 172.16.10.1     172.16.1.10     TCP      66 54942 → 42057 [ACK] Seq=1 Ack=420 Win=64128 Len=0 TSval=748015529 TSecr=3838763244
184 218.439158… 172.16.10.1     172.16.1.10     TCP      66 43324 → 21 [ACK] Seq=44 Ack=459 Win=64128 Len=0 TSval=748015529 TSecr=3838763244
```