File Transfer Protocol

Computer Networks
Bachelors in Informatics and Computing Engineering

3LEIC03_G6

Tiago Rodrigues up201907021@fe.up.pt
Mário Travassos up201905871@fe.up.pt

January 24, 2022

Summary

This report will cover the second project proposed for the Computer Networks Curricular Unit, which had the objective of developing an application that would download files using the FTP (File Transfer Protocol) standard, along with setting up a network between the labs computers.

The application supports both anonymous and authenticated downloads, and the arguments must be valid URL's, without the port.

Introduction

The second project had two main objectives: The development of the download application, and the configuration of the computer network. This report will examine both sections, detailing the architecture of the application and the setup used to launch the network. The application can be used to download files from any FTP server, since it adopts the standard URL syntax.

The report is structured as follows: First, the application is described in detail, breaking down the architecture and the more relevant components. Then, a report of a successful download is made. After that, an analysis of the network will take place. Finally, a set of relevant attachments will be included.

Download Application

Architecture

The application is simple by design. The way it operates internally resembles the way files were retrieved using the same protocol, in the first lab experiment that used **telnet**. It connects to the remote server using a socket, and after the connection is established, sends a set of FTP requests that will retrieve the desired file.

Program flow

The first two requests the program makes are authentication ones, which login the user. If the user does not specify a username and password in the URL, the default is "anonymous" and "pass", for the user and password respectively. If the login is successful, the program then requests the server to enter passive mode. The server's response to this command includes the IP address and the port for the file to be transferred from. The program determines this port, connects to it, and then asks the server to begin the transfer of the file specified in the URL. After the file is requested, the program reads the bytes from the socket, and writes them to the directory it was called from, with the same name as the transferred file. If at any point during execution an error occurs or the response from the server is not the expected one, the program terminates gracefully.

Main Modules

The program is logically divided in the following modules:

- parser: Responsible for decomposing the given URL and dividing it into the relevant tokens, saving them for later use.
- connection: Responsible for establishing a connection with the remote server.
- communication: Responsible for sending commands and reading responses, acting accordingly.
- file: Responsible for reading the incoming file and writing it onto the local machine.

There is also a shared file that includes common definitions like server response codes, and a main file that unites the modules described above.

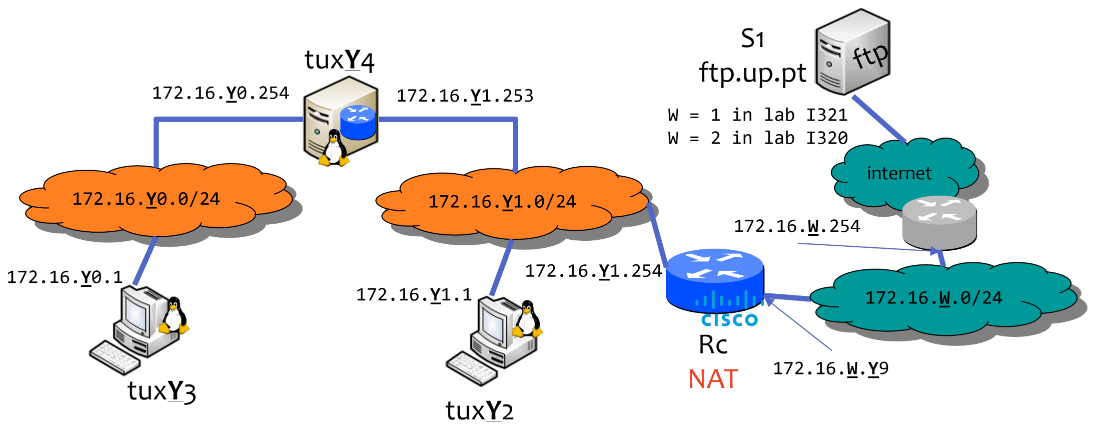
Case study

The majority of the anonymous download tests were done by downloading from the server at **ftp.up.pt**, but since it only supports anonymous login, the authenticated tests were done using the server at **netlab1.fe.up.pt**, using the username **rcom** and password **rcom**. All of them were successful.

The application allows the user to see the information he gave in the URL separated into the components, to check if it is correctly formatted. Also, in the event of a failure, the application shows the bad response code and message that led to the termination.

Network Configuration and Analysis

Part two of our assignment consisted in configuring a small local network as demonstrated in the picture below:

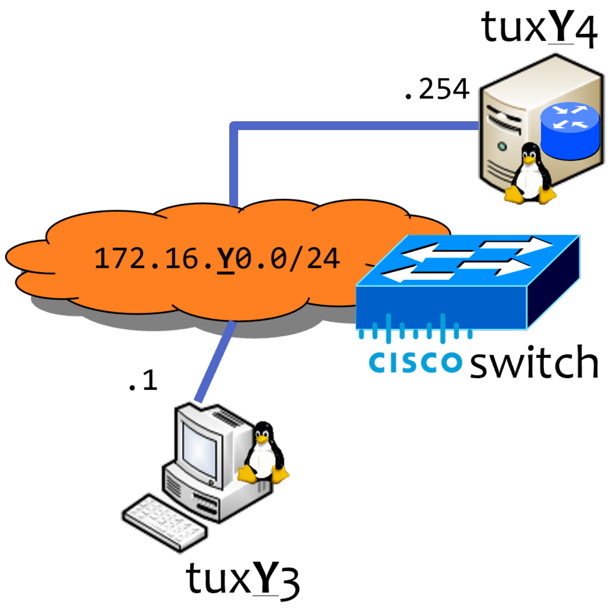


In this image, **Y** is the lab bench number, (which in our case was 4), and **W** has a value depending on the lab we were working in. In our case, since we were working in lab I320, said value was 2.

In order to correctly configure this network, we followed 4 experiment guides. Our experience will be described below.

Experiment 1:

Our goal in this experiment was to configure the ip address of tux43 and tux44, and connect those to a switch, which resulted in the following configuration:



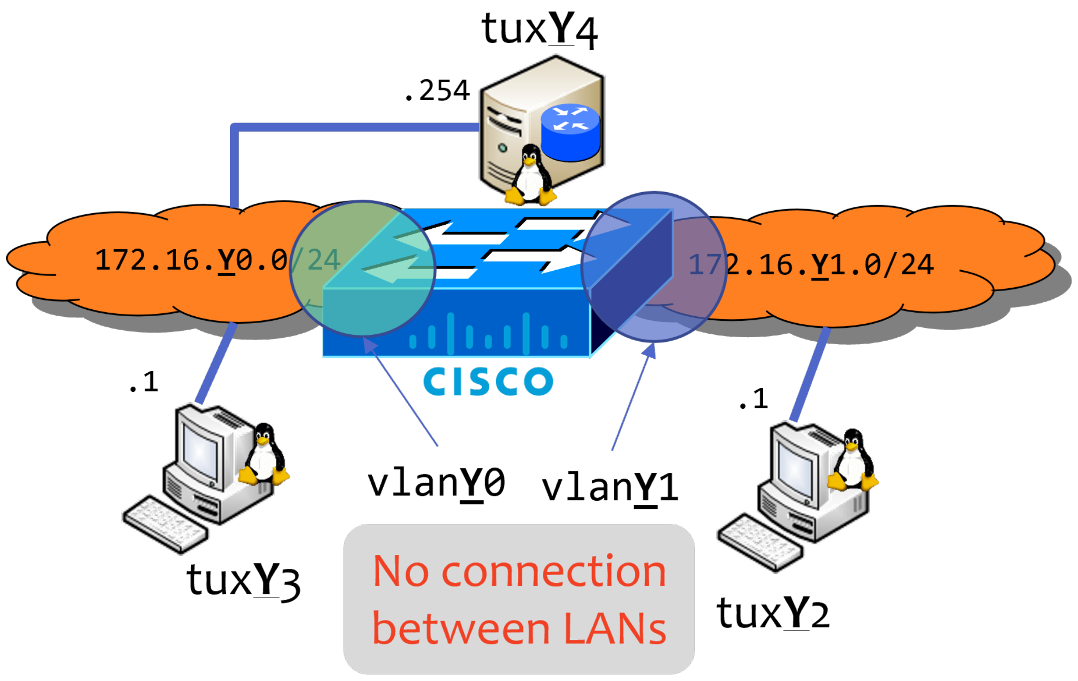
The steps we followed were as follows:

- We disconnected the switch from netlab, cleaned the router and switch configurations, restarted the computers and connected the eth0 interfaces of tux43 and tux44 to it:
  - port 1 - tux43;
  - port 2 - tux44.
- We configured tux43 and tux44 to have the IP addresses shown in the image above (replacing Y with 4), and defined the network 172.16.40.0/24, using ifconfig and the route commands;

- We noted down the IP and MAC addresses of the network interfaces on both tuxes, for later analysis;
- We pinged the computers from one another to verify the connectivity between them.;
- We inspected the forwarding table and the ARP table;
- We deleted the ARP table entries in tux43;
- After starting Wireshark in eth0 of tux43 and starting the packet capturing process, we pinged tux44 from tux43 for a few seconds;
- After saving the wireshark log, we analized it, in order to gain some insight into the inner workings of the network we just set up.

Experiment 2:

Our goal in this experiment was to configure the two virtual lands in our bench's switch, as demonstrated by the following diagram:



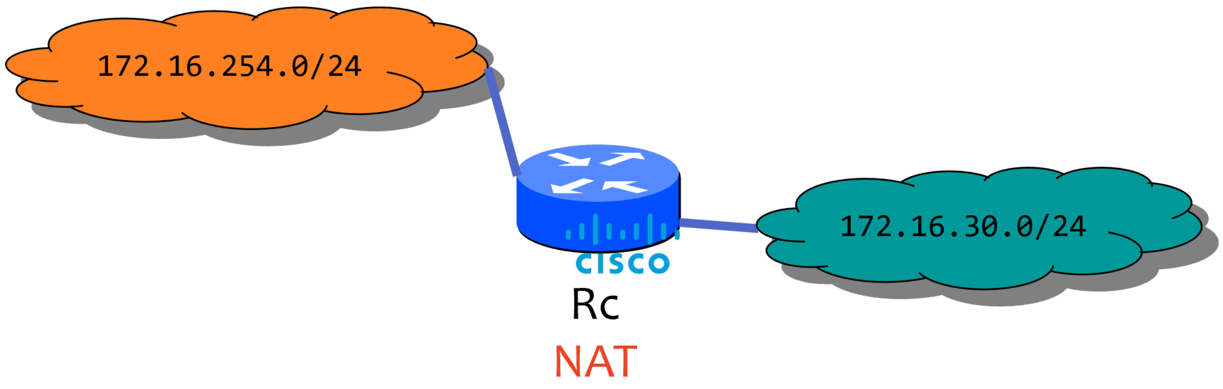
The steps we followed were as follows:

- Connected eth0 of tux42 to port 13 of the router and configured its network (taking note of its IP and MAC addresses, just as before);

- Created the vlans 40 and 41 in the switch and added their corresponding ports (1 and 2 to vlan40; 13 to vlan41);
- Started a Wireshark capture at eth0 of tux43, from where we pinged tux44 and tux42, saving the log after pinging;
- Started Wireshark captures in eth0 of tux43, eth0 of tux44 and eth0 of tux42;
- We made a ping with the broadcast flag to 172.16.40.255 and saved the resulting logs;
- We repeated the previous two steps, but this time, pinging 172.16.41.255.

Experiment 3:

During experience 3, we focused on analyzing and setting up a configuration file for a Cisco Router. Our goal was to better understand and to prepare to implement the following network diagram:



The steps we followed were as follows:

- Having been provided a configuration file for a Cisco router that is doing NAT and routing on its interfaces (as represented in the above figure), we analyzed its configuration file;
- With the help of the Cisco IOS documentation for the Basic Router Configuration, we checked, changing values where necessary, the following configurations:
  - Router name;
  - Ethernet Ports available and of what type (in our case, only fast-ethernet);
  - Configured IP addresses and netmask of ports;
  - Configured routes.

- We took a closer look into the router's NAT configuration, namely the fields related to the interfaces which are connected to the internet, the number of IP addresses available for NATing and router overloading.

After the previous steps, we ended up with the configuration file which we appended to this report in the Annexes section.

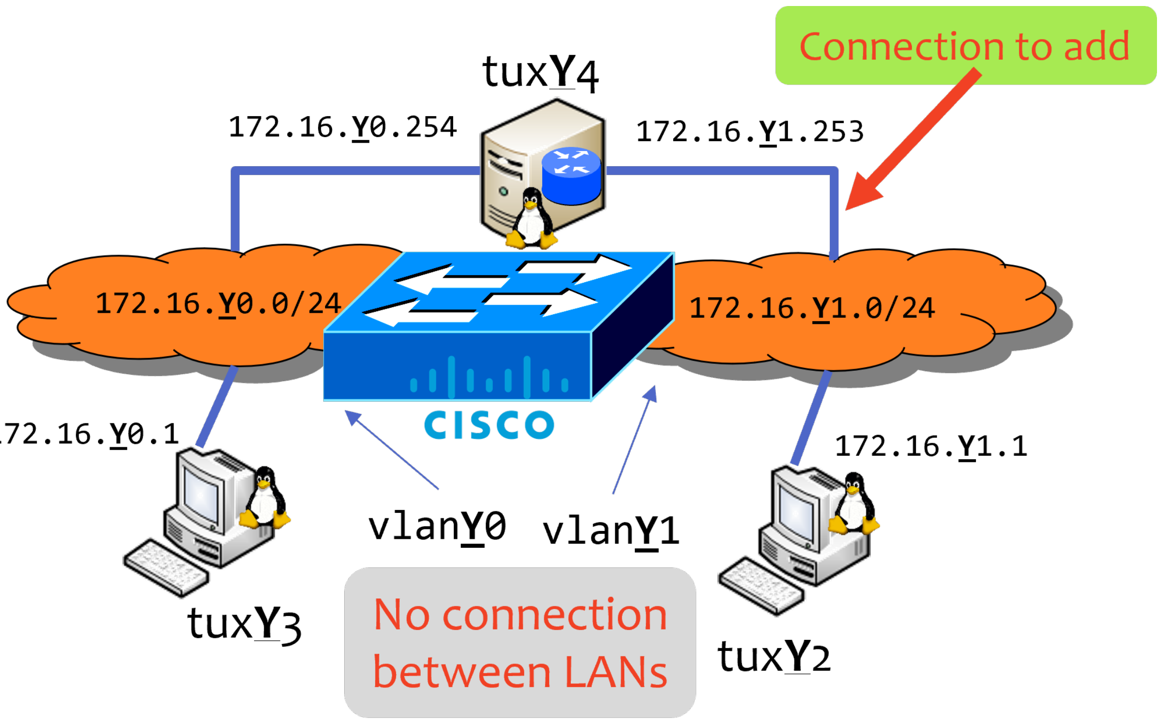
Experiment 4:

Experience 4 consisted of applying the configuration created during experience 3, on top of the work developed in labs 1 and 2.

This experience was devided into two smaller goals: **Configuring the Linux Router** and **Configuring the Cisco Router**.

Configuring the Linux Router

This procedure is geared towards correctly configuring the following network:



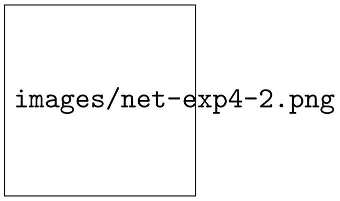
The steps we followed for this first procedure were as follows:

- We connected a cable from tux44's eth1 interface to port 14 of the switch, and added that port to vlan41, following the same procedure used in experiment 2;

- Configured tux44's eth1 interface's IP address to be the same as represented in the image above (172.16.41.253/24);
- Enabled IP forwarding and disabled ICMP echo ignore broadcast, on tux 44;
- Took note of the IP and MAC addresses in tux44, for both of its interface, for further study;
- Configured the routes in tux43 and tux42, in order for them to reach one another;
- Started a wireshark capture at tux43, from where we pinged the other network interfaces, 172.16.40.254, 172.16.41.253, 172.16.41.1. We saved the log for later reference;
- Started a capture in tux44, on both of its interfaces, eth0 and eth1;
- We cleared the ARP tables of all 3 tuxes;
- From tux43 we pinged tux42 for a few seconds, and stopped the capture in tux44, saving the logs.
- Inside Wireshark, we opened the last capture last capture log and viewed the packets from each interface using, in the display filter, the "test frame.interface_id == X" filter, where X is the number of the interface we want to study.

Configuring the Cisco Router

In this step of the experiment, we seek to connect our setup to the world wide internet, as illustrated by the image below:



The steps we followed for the second procedure were as follows:

- Connect the FE0 interface of the Cisco Router to the Lab Router, and the FE1 interface to the Switch;
- Configured the switch's port to be on the correct vlan;
- We connected to the router via serial port, and applied our configuration file on the router, thereby configuring it;

- To verify the connectivity, we did a ping from the Cisco Router to all the tuxes, to 172.16.2.254 and to 104.17.113.188 (on the internet);
- We set tux42 and tux44's default gateways to the Cisco Router (172.16.41.254);
- Pinged 172.16.2.254 from tux43;
- Pinged 104.17.113.188 (the internet) from tux43.

Testing our network

This marks the end of our experiments, thereby allowing us to successfully run our download application, in order to retrieve content from an FTP server on the internet, namely FEUP's FTP server.

In preparation for our final demonstration, we prepared a few bash scripts, and some configuration files which when executed would run crucial commands to set up the network just as it was set up after this configuration, which will be appended in the Annexes section of this document. These files are the following:

- tux42_conf.sh - to be run inside tux42;
- tux43_conf.sh - to be run inside tux43;
- tux44_conf.sh - to be run inside tux44;
- vlan.conf - vlan configuration commands to be run inside the switch console, accessible via serial port;
- router.conf - router configuration commands to be run inside the cisco router console, accessible via serial port.

Conclusions

The implemented download application achieved the proposed results, being able to download files from any FTP server using the FTP standard. More importantly, the standard was very well understood.

The 2nd part of the project allowed us to get some hands-on experience, giving us the chance to apply the theoretical knowledge we have acquired during our classes to a real-life example, albeit a simplified, albeit to some extent convoluted. This also allowed us to gain a better understanding of how networks operate, and some insight into how this process may be automated in order to provide the end user with a more streamlined experience, giving us a small taste of how the inner workings of technologies which abstract and automate some of this configuration steps, such as DHCP, might look like.

Furthermore, in order to simplify our reconfiguration process, we found ourselves dusting off some very rudimentary knowledge of bash scripting we had laying around. This was but a byproduct of the way in which the assignment was structured, but a well welcomed one. After getting the network up an running, we were able to run our download application from part 1, in order to download files from the faculty's ftp server, seeing the fruits of our hard labour.

References

- RFC959 - File Transfer Protocol: `https://www.rfc-editor.org/info/rfc959`
- RFC1738 - Uniform Resource Locator (URL): `https://www.rfc-editor.org/info/rfc1738`

Annexes

Annex I - Application Source Code

communication.c

```c
#include "communication.h"
#include "defines.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int send_cmd(int socket_fd, char *cmd, size_t cmd_size)
{
        if (write(socket_fd, cmd, cmd_size) != cmd_size) {
                fprintf(stderr, "Error writing command!\n");
                perror("write");
                return EXIT_FAILURE;
        }

        return EXIT_SUCCESS;
}

int read_response(int socket_fd, struct server_response *response)
{
        FILE *socket = fdopen(socket_fd, "r");

        if (socket == NULL) {
                close(socket_fd);
                fprintf(stderr, "Error opening file\n");
                return EXIT_FAILURE;
        }

        size_t response_size = 0;
        size_t size = 0;
        char *buf;

        while (getline(&buf, &size, socket) >= 0) {
                strncat(response->response, buf, size - 1);
                response_size += size;

                if (buf[RESPONSE_CODE_SIZE] == ' ') {
                        sscanf(buf, "%d", &response->response_code);
                        break;
                }
        }

        free(buf);

        return EXIT_SUCCESS;
}

int login(int socket_fd, char *user, char *password)
{
        if (user == NULL) {
                user = DEFAULT_USER;
                password = DEFAULT_PASSWORD;
        }

        size_t user_cmd_size = strlen(user) + USER_CMD_SIZE + CRLF_SIZE + 1;
        size_t password_cmd_size =
                strlen(password) + PASS_CMD_SIZE + CRLF_SIZE + 1;

        char *user_cmd = malloc(user_cmd_size);
        char *password_cmd = malloc(password_cmd_size);

        user_cmd[0] = '\0';
        strcat(user_cmd, USER);
        strcat(user_cmd, " ");
        strcat(user_cmd, user);
        strcat(user_cmd, CRLF);

        password_cmd[0] = '\0';
        strcat(password_cmd, PASS);
        strcat(password_cmd, " ");
        strcat(password_cmd, password);
        strcat(password_cmd, CRLF);

        if (send_cmd(socket_fd, user_cmd, user_cmd_size)) {
                fprintf(stderr, "Error sending user\n");
                free(user_cmd);
                free(password_cmd);
                return EXIT_FAILURE;
        }

        free(user_cmd);
        struct server_response response;
        if (read_response(socket_fd, &response)) {
                fprintf(stderr, "Error reading server response!\n");
                free(password_cmd);
                return EXIT_FAILURE;
        }

        if (response.response_code != USERNAME_OK) {
                fprintf(stderr, "Error with username.\nResponse: %d - %s\n",
                        response.response_code, response.response);
                free(password_cmd);
                return EXIT_FAILURE;
        }

        if (send_cmd(socket_fd, password_cmd, password_cmd_size)) {
                fprintf(stderr, "Error sending password\n");
                free(password_cmd);
                return EXIT_FAILURE;
        }

        free(password_cmd);
        memset(&response, 0, sizeof(struct server_response));

        if (read_response(socket_fd, &response)) {
                fprintf(stderr, "Error reading server response!\n");
                return EXIT_FAILURE;
        }

        if (response.response_code != LOGIN_SUCCESS) {
                fprintf(stderr, "Error with password.\nResponse: %d - %s\n",
                        response.response_code, response.response);
                return EXIT_FAILURE;
        }

        return EXIT_SUCCESS;
}

int enter_passive_mode(int socket_fd, struct server_response *response)
{
        int pasv_cmd_size = PASV_CMD_SIZE + CRLF_SIZE;
        char *pasv_cmd = malloc(pasv_cmd_size);

        pasv_cmd[0] = '\0';
        strcat(pasv_cmd, PASV);
        strcat(pasv_cmd, CRLF);
```

```
	if (send_cmd(socket_fd, pasv_cmd, pasv_cmd_size)) {
		fprintf(stderr, "Error sending PASV command\n");
		return EXIT_FAILURE;
	}

	if (read_response(socket_fd, response)) {
		fprintf(stderr, "Error reading server response\n");
		return EXIT_FAILURE;
	}

	if (response->response_code != PASV_SUCCESS) {
		fprintf(stderr,
			"Error entering passive mode.\n Response: %d - %s\n",
			response->response_code, response->response);
		return EXIT_FAILURE;
	}

	return EXIT_SUCCESS;
}

int convert_to_port(char *response, char *ip, int *port)
{
	char *cp = strdup(response);

	char *values = strtok(cp, "(");
	values = strtok(NULL, ")");

	int tmp[6];
	sscanf(values, "%d, %d, %d, %d, %d, %d", &tmp[0], &tmp[1], &tmp[2],
	       &tmp[3], &tmp[4], &tmp[5]);

	*port = tmp[4] * 256 + tmp[5];
	sprintf(ip, "%d.%d.%d.%d", tmp[0], tmp[1], tmp[2], tmp[3]);

	return EXIT_SUCCESS;
}

int request_file(int socket_fd, char *file)
{
	int retr_cmd_size = strlen(file) + RETR_CMD_SIZE + CRLF_SIZE + 1;
	char *retr_cmd = malloc(retr_cmd_size);

	retr_cmd[0] = '\0';
```

```
        strcat(retr_cmd, RETR);
        strcat(retr_cmd, " ");
        strcat(retr_cmd, file);
        strcat(retr_cmd, CRLF);

        if (send_cmd(socket_fd, retr_cmd, retr_cmd_size)) {
                fprintf(stderr, "Error sending retr command\n");
                return EXIT_FAILURE;
        }

        struct server_response response;
        memset(&response, 0, sizeof(struct server_response));

        if (read_response(socket_fd, &response)) {
                fprintf(stderr, "Error reading server response!\n");
                return EXIT_FAILURE;
        }

        if (response.response_code != RETR_SUCCESS) {
                fprintf(stderr, "Error with retr command.\nResponse: %d - %s\n",
                        response.response_code, response.response);
                return EXIT_FAILURE;
        }

        return EXIT_SUCCESS;
}

int logout(int socket_fd)
{
        struct server_response response;
        memset(&response, 0, sizeof(struct server_response));
        int logout_cmd_size = QUIT_CMD_SIZE + CRLF_SIZE;
        char *logout_cmd = malloc(logout_cmd_size);

        logout_cmd[0] = '\0';
        strcat(logout_cmd, QUIT);
        strcat(logout_cmd, CRLF);

        if (send_cmd(socket_fd, logout_cmd, logout_cmd_size)) {
                fprintf(stderr, "Error sending logout command\n");
                return EXIT_FAILURE;
        }

        if (read_response(socket_fd, &response)) {
                fprintf(stderr, "Error reading server response\n");
                return EXIT_FAILURE;
        }

        if (response.response_code != QUIT_SUCCESS) {
                fprintf(stderr, "Error quitting.\n Response: %d - %s\n",
                        response.response_code, response.response);
                return EXIT_FAILURE;
        }

        return EXIT_SUCCESS;
}
```

communication.h

```c
#ifndef RCOM_MESSAGES_H_
#define RCOM_MESSAGES_H_

#include <stddef.h>

struct server_response {
        char response[1024];
        int response_code;
};

int send_cmd(int socket_fd, char *cmd, size_t cmd_size);

int read_response(int socket_fd, struct server_response *response);

int login(int socket_fd, char *user, char *password);

int enter_passive_mode(int socket_fd, struct server_response *response);

int convert_to_port(char *response, char *ip, int *port);

int request_file(int socket_fd, char *file);

int logout(int socket_fd);

#endif // RCOM_MESSAGES_H_
```

connection.c

```c
#include "connection.h"
#include "defines.h"
#include <stdio.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>

int start_connection(char *ip, int port)
{
        int sockfd;
        struct sockaddr_in server_addr;

        memset((char *)&server_addr, 0, sizeof(server_addr));
        server_addr.sin_family = AF_INET;
        server_addr.sin_addr.s_addr = inet_addr(ip);
        server_addr.sin_port = htons(port);

        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                fprintf(stderr, "Error creating socket!\n");
                return EXIT_FAILURE;
        }

        if (connect(sockfd, (struct sockaddr *)&server_addr,
                    sizeof(server_addr)) < 0) {
                fprintf(stderr, "Error connecting to the given IP!\n");
                return EXIT_FAILURE;
        }

        return sockfd;
}
```

connection.h

```
#ifndef RCOM_CONNECTION_H_
#define RCOM_CONNECTION_H_

int start_connection(char *ip, int port);

#endif // RCOM_CONNECTION_H_
```

defines.h

```
#ifndef RCOM_DEFINES_H_
#define RCOM_DEFINES_H_

// Miscellaneous
#define TRUE 1
#define FALSE 0
#define MAX_BUFSIZE 1024
#define RESPONSE_CODE_SIZE 3
#define DEFAULT_FTP_PORT 21

// Server Codes
#define SERVER_READY 220
#define USERNAME_OK 331
#define LOGIN_SUCCESS 230
#define RETR_SUCCESS 150
#define PASV_SUCCESS 227
#define QUIT_SUCCESS 221
#define TRANSFER_COMPLETE 226

// FTP Commands
#define USER "USER"
#define PASS "PASS"
#define PASV "PASV"
#define RETR "RETR"
#define QUIT "QUIT"
#define USER_CMD_SIZE 4
#define PASS_CMD_SIZE 4
#define PASV_CMD_SIZE 4
#define RETR_CMD_SIZE 4
#define QUIT_CMD_SIZE 4

// Defaults
#define DEFAULT_USER "anonymous"
#define DEFAULT_PASSWORD "pass"

// End of line
#define CRLF "\r\n"
#define CRLF_SIZE 2

#endif // RCOM_DEFINES_H_
```

download.c

```c
#include "parser.h"
#include "file.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        if (argc != 2) {
                fprintf(stderr,
                        "Usage: ./download ftp://[<user>:<password>@]<host>/<path>");
                return EXIT_FAILURE;
        }

        struct url_parser url;
        if (parse_url(&url, argv[1])) {
                fprintf(stderr, "Error parsing url\n");
                return EXIT_FAILURE;
        }

        if (transfer_file(&url)) {
                fprintf(stderr, "Error transfering file\n");
                return EXIT_FAILURE;
        }

        return EXIT_SUCCESS;
}
```

file.c

```c
#include "file.h"
#include "connection.h"
#include "communication.h"
#include "defines.h"
#include <stdlib.h>
#include <libgen.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int transfer_file(struct url_parser *url)
{
        int socket_fd = -1;
        socket_fd = start_connection(url->ip, DEFAULT_FTP_PORT);
        if (socket_fd == EXIT_FAILURE) {
                fprintf(stderr, "Error starting connection\n");
                return EXIT_FAILURE;
        }

        struct server_response response;
        memset(&response, 0, sizeof(struct server_response));

        read_response(socket_fd, &response);

        if (login(socket_fd, url->user, url->password)) {
                fprintf(stderr, "Error logging in\n");
                return EXIT_FAILURE;
        }

        printf("Succesfully logged in!\n");
        memset(&response, 0, sizeof(struct server_response));

        if (enter_passive_mode(socket_fd, &response)) {
                fprintf(stderr, "Error entering passive mode\n");
                return EXIT_FAILURE;
        }

        char *ip = malloc(16);
        int *port = malloc(sizeof(int));
        if (convert_to_port(response.response, ip, port)) {
                fprintf(stderr,
                        "Error converting bytes received in response\n");
                free(ip);
                free(port);
                return EXIT_FAILURE;
        }

        int data_fd = -1;
        if ((data_fd = start_connection(ip, *port)) == EXIT_FAILURE) {
                fprintf(stderr, "Error connecting to other port\n");
                free(ip);
                free(port);
                return EXIT_FAILURE;
        }

        free(ip);
        free(port);

        if (request_file(socket_fd, url->path)) {
                fprintf(stderr, "Error retrieving file\n");
                return EXIT_FAILURE;
        }

        if (read_file(data_fd, url->path)) {
                fprintf(stderr, "Error reading file\n");
                return EXIT_FAILURE;
        }

        memset(&response, 0, sizeof(struct server_response));
        if (read_response(socket_fd, &response)) {
                fprintf(stderr, "Error reading response\n");
                return EXIT_FAILURE;
        }

        if (response.response_code != TRANSFER_COMPLETE) {
                fprintf(stderr, "File transfer not complete\n");
                return EXIT_FAILURE;
        }

        if (logout(socket_fd)) {
                fprintf(stderr, "Error quitting\n");
                return EXIT_FAILURE;
        }

        printf("Succesfully requested file %s from server\n", url->path);
```

```c
        close(socket_fd);
        close(data_fd);

        return EXIT_SUCCESS;
}

int read_file(int socket_fd, char *file_path)
{
        char *file_name = basename(file_path);
        int file_fd = 0;
        file_fd = open(file_name, O_WRONLY | O_CREAT,
                       S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

        printf("File fd is of %d\n", file_fd);

        if (file_fd == -1) {
                fprintf(stderr, "Error opening file\n");
                return EXIT_FAILURE;
        }

        char buf[MAX_BUFSIZE];
        int bytes_read = 0;

        while ((bytes_read = read(socket_fd, buf, MAX_BUFSIZE)) > 0) {
                if (write(file_fd, buf, bytes_read) == -1) {
                        fprintf(stderr, "Error writing to file\n");
                        close(file_fd);
                        return EXIT_FAILURE;
                }
        }

        if (close(file_fd)) {
                fprintf(stderr, "Error closing file\n");
                return EXIT_FAILURE;
        }

        return EXIT_SUCCESS;
}
```

file.h

```
#ifndef RCOM_FILE_H_
#define RCOM_FILE_H_

#include "parser.h"

int read_file(int socket_fd, char *file_path);

int transfer_file(struct url_parser *url);

#endif // RCOM_FILE_H_
```

parser.c

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

int parse_url(struct url_parser *parser, char *url)
{
        char *token;
        char *cp;

        cp = strdup(url);

        if (cp == NULL) {
                fprintf(stderr, "Error copying url");
                return EXIT_FAILURE;
        }

        //Protocl of the URL
        token = strtok(cp, "/");

        if (strcmp(token, "ftp:")) {
                fprintf(stderr,
                        "Invalid Protocl. The correct Protocol format is ftp://\n");
        }

        // Possibly the user and password, and the host
        token = strtok(NULL, "/");

        parser->path = strtok(NULL, "");

        if (url_has_user(token)) {
                parser->user = strtok(token, ":");
                parser->password = strtok(NULL, "@");
                parser->host = strtok(NULL, "");
        } else {
                parser->user = NULL;
                parser->password = NULL;
                parser->host = token;
        }

        struct hostent *h;

        if ((h = gethostbyname(parser->host)) == NULL) {
                herror("gethostbyname()");
                return EXIT_FAILURE;
        }

        parser->host_name = h->h_name;
        parser->ip = inet_ntoa(*((struct in_addr *)h->h_addr));

        printf("Info given:\n");
        printf("User: %s\n", parser->user);
        printf("Password: %s\n", parser->password);
        printf("Host: %s\n", parser->host);
        printf("Path: %s\n", parser->path);
        printf("Host name  : %s\n", parser->host_name);
        printf("IP Address : %s\n", parser->ip);

        return EXIT_SUCCESS;
}

int url_has_user(char *url)
{
        return strchr(url, ':') ? TRUE : FALSE;
}
```

parser.h

```c
#ifndef RCOM_PARSER_H_
#define RCOM_PARSER_H_

struct url_parser {
        char *user;
        char *password;
        char *host;
        char *host_name;
        char *ip;
        char *path;
};

/**
 * @brief Parses the URL supplied as argument and fills the struct with the
 *        appropriate fields
 *
 * @param parser Struct that holds the relevant fields of the URL
 *
 * @param url URL supplied as argument
 *
 * @return 0 if successfull, 1 otherwise
 *
 */
int parse_url(struct url_parser *parser, char *url);

/**
 * @brief Determines if the url has a user inside
 *
 * @param url URL to determine if it has a user or not
 *
 * @return 1 if TRUE, 0 if FALSE
 */
int url_has_user(char *url);

#endif // RCOM_PARSER_H_
```

Annex II - Configuration Commands

tux42conf.sh

```
#! /bin/bash

# Configure eth0 (port 13)
ifconfig eth0 down
ifconfig eth0 up
ifconfig eth0 172.16.41.1/24
# mac addr 00:21:5a:5a:7c:e7 (banc 5)

# configure route in tux2 to tux3
route add -net 172.16.40.0/24 gw 172.16.41.253

# set tux42's default gateway to the cisco router
route add default gw 172.16.41.254
```

tux43conf.sh

```
#! /bin/bash

# configure eth0 (port 1)
ifconfig eth0 down
ifconfig eth0 up
ifconfig eth0 172.16.40.1/24
# mac addr 00:21:5a:61:2c:54 (banc 5)

# configure route in tux3 to tux2
route add -net 172.16.41.0/24 gw 172.16.40.254
```

tux44conf.sh

```bash
#! /bin/bash

# Configure eth0 (port 2)
ifconfig eth0 down
ifconfig eth0 up
ifconfig eth0 172.16.40.254/24

# Configure eth1 (port 14)
ifconfig eth0 down
ifconfig eth0 up
ifconfig eth0 172.16.41.253/24
# mac addr 00:22:64:19:09:5c (banc 5)

# enable ip forwarding
echo 1 > /proc/sys/net/ipv4/ip_forward
# disable ICMP echo ignore broadcast
echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts

# set tux44's default gateway to the cisco router
route add default gw 172.16.41.254
```

vlan.conf

```
configure terminal

vlan 40
vlan 41

interface fastethernet 0/1
switchport mode access
switchport access vlan 40

interface fastethernet 0/2
switchport mode access
switchport access vlan 40

interface fastethernet 0/13
switchport mode access
switchport access vlan 41

interface fastethernet 0/14
switchport mode access
switchport access vlan 41

interface fastethernet 0/23
switchport mode access
switchport access vlan 41

end

show vlan brief
```

router.conf

```
!
interface fastEthernet0/0
 ip address 172.16.41.254 255.255.255.0
 ip nat inside
 ip virtual-reassembly
 duplex auto
 no shutdown
 speed auto
!
!
interface fastEthernet0/1
 ip address 172.16.2.49 255.255.255.0
 ip nat outside
 ip virtual-reassembly
 duplex auto
 no shutdown
 speed auto
!
! Change to correct bench
ip forward-protocol nd
ip route 0.0.0.0 0.0.0.0 172.16.2.254
ip route 172.16.40.0 255.255.255.0 172.16.41.253
ip http server
ip http access-class 23
ip http authentication local
ip http secure-server
ip http timeout-policy idle 60 life 86400 requests 10000
!
!
! Change to correct bench
ip nat pool ovrld 172.16.2.49 172.16.2.49 prefix-length 24
ip nat inside source list 1 pool ovrld overload
!
! Change to correct bench
access-list 1 permit 172.16.40.0 0.0.0.7
access-list 1 permit 172.16.41.0 0.0.0.7
```