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 14, 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

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.

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

```
#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

```c
#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);
```

```
        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

```c
#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

```
#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;
```

```c
        }

        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

Annex III - Data Logs