Relatório RCOM T2

João Miguel, Miguel Barraca, Pedro Fernandes

Dezembro 2018

Conteúdo

1 Introdução

2 Aplicação *download*

2.1 Arquitetura

A aplicação tem o objetivo de fazer o *download* de um ficheiro recorrendo ao protocolo FTP (*File Transfer Protocol*). Para tal, recebe como argumento um link FTP, que deve conter os seguintes campos:

- Nome de utilizador
- Palavra-passe
- Endereço do anfitrião
- Caminho do ficheiro

Caso os dados do utilizador não sejam fornecidos, o programa pedi-los-á posteriormente. Após a análise do argumento, o programa cria a ligação através de duas funções cujo código é já fornecido:

- `char *getServerIp(info_t info)`
- `int createSocketTCP(char *server_ip, int server_port)`

Em caso de conexão bem sucedida, o programa segue os seguintes passos:

1. Enviar `USER` e `PASS`
2. Entrar em modo passivo `pasv`
3. Criar novo socket TCP
4. Enviar comando `RETR`
5. Transferir o ficheiro
6. Terminar a conexão com `QUIT`

Esta estrutura é apoiada nas seguintes funções, cuja documentação pode ser encontrada nos anexos:

- `int sendCommand(int socketFd, char *command, char *argument)`
- `int readServerReply(int socketFd, char *reply)`
- `void createFile(int fd, char *filename)`

E nas seguintes estruturas:

- `enum state_t`

  Representa o estado da leitura da resposta aos comandos enviados.

- `enum reply_type_t`

  Representa o tipo da resposta do servidor aos comandos enviados.

- `struct info_t`

  Guarda informação fornecida no argumento do programa.

Ao longo do programa são impressas mensagens para informar o utilizador do avanço do processo.

2.2 Exemplo de um *download* com sucesso

Depois de apresentação colocar aqui

3 Configuração e análise de redes

3.1 Experiência 1 - Configurar uma rede IP

1. O que são os pacotes ARP e para que são usados?
2. O que são os endereços MAC e IP dos pacotes ARP e porquê?
3. Que pacotes gera o comando `ping`?
4. O que são os endereços MAC e IP dos pacotes `ping`?
5. Como determinar se um quadro *Ethernet* recebido é ARP, IP, ICMP?
6. Como determinar o tamanho de um quadro recebido?
7. Qual é a interface *loopback* e porque é importante?

3.2 Experiência 2 - Implementar duas VLANs num *switch*

1. Como configurar a *vlan10*?
2. Quantos domínios de transmissão existem? Como se pode concluir a partir dos registos?

3.3 Experiência 3 - Configurar um *router* em Linux

1. Que rotas existem nos *tuxes*? Qual o seu significado?
2. Que informação contém uma entrada da tabela de encaminhamento?
3. Que mensagens ARP e endereços MAC associados são observados, e porquê?
4. Que pacotes ICMP são observados e porquê?
5. O que são os endereços IP e MAC associados aos pacotes ICMP e porquê?

3.4 Experiência 4 - Configurar um *router* comercial e implementar NAT

1. Como configurar uma rota estática num *router* comercial?
2. Quais são os caminhos seguidos pelos pacotes nas experiências feitas e porquê?
3. Como configurar NAT num *router* comercial?
4. O que faz NAT?

3.5 Experiência 5 - DNS

1. Como configurar o servico DNS num anfitrião?
2. Que pacotes são trocados por DNS e que informação é transportada?

3.6 Experiência 6 - Conexões TCP

1. Quantas conexões TCP são abertas pela aplicação FTP?
2. Em que conexão é transportada a informação de controlo FTP?
3. Quais são as fases de uma conexão FTP?
4. Como funciona o mecanismo ARQ TCP? Quais são os campos TCP relevantes? Que informação relevante pode ser observada nos registos?
5. Como funciona o mecanismo de congestionamento de controlo TCP? Quais são os campos relevantes? Como é que a taxa de transferência da conexão de dados evoluiu ao longo do tempo? Está de acordo com o mecanismo de congestionamento de controlo TCP?
6. A taxa de transferência de uma conexão de dados TCP é perturbada pelo aparecimento de uma segunda conexão TCP? Como?

4 Conclusão

5 Referências

6 Anexos

6.1 Código da aplicação *download*

download.h

```
#pragma once

#include <stdbool.h>

#define MAX_BUF_SIZE 100
#define MAX_REPLY_SIZE 400
#define SOCKET_BUF_SIZE 1000
#define REPLY_CODE_SIZE 3
#define SERVER_PORT 21

typedef struct
{
    char serverName[MAX_BUF_SIZE];
```

```
    char filePath[MAX_BUF_SIZE];
    char fileName[MAX_BUF_SIZE];
    char user[MAX_BUF_SIZE];
    char pass[MAX_BUF_SIZE];
} info_t;

typedef enum
{
    READ_CODE,
    READ_LINE,
    READ_MULT_LINE,
    WAIT_FOR_PORT,
    FIRST_PORT_BYTE,
    SECOND_PORT_BYTE,
    END
} state_t;

typedef enum
{
    POSITIVE_PRE = 1,
    POSITIVE_INT,
    POSITIVE_COMP,
    TRANS_NEGATIVE_COMP,
    PERM_NEGATIVE_COMP
} reply_type_t;

/**
 * @brief Prints a message that shows how to run the program.
 *
 * @param argv array of arguments passed from the command line
 *
 * @return always return 1
 */
int usage(char *argv[]);

/**
 * @brief Parses the argument passed to the program, retrieving user information.
 *
 * @param argument argument from the command line, supposedly an FTP link
 * @param info structure that holds user and server info
 *
 * @return true if argument was successfully read, false otherwise
 */
bool parseArgument(char *argument, info_t *info);

/**
```

```
 * @brief Gets a server ip from a host name
 *
 * @param name host name
 *
 * @return server ip
 */
char *getServerIp(const char* name);

/**
 * @brief Creates a TCP socket, returning its respective file descriptor.
 *
 * @param server_ip
 * @param server_port
 *
 * @return socket's file descriptor
 */
int createSocketTCP(char *server_ip, int server_port);

/**
 * @brief Reads the server reply to an FTP command, returning it through the reply argument.
 *
 * @param socketFd socket's file descriptor
 * @param reply numeric descriptor of the server reply
 */
void readServerReply(int socketFd, char *reply);

/**
 * @brief Gets the server port after the program issues pasv command.
 *
 * @param socketFd socket's file descriptor
 *
 * @return the server port
 */
int getServerPort(int socketFd);

/**
 * @brief Sends and FTP command along with its argument (if applicable) and reads the server
 *
 *
 * @param socketFd socket's file descriptor
 * @param command FTP command
 * @param argument command's argument, if any
 *
 * @return 0 for POSITIVE_INT, 1 for POSITIVE_COMP and -1 for PERM_NEGATIVE_COMP
 */
int sendCommand(int socketFd, char *command, char *argument);
```

```c
/**
 * @brief Called after sending RETR command, reads data from the socket and creates a local
 *
 * @param fd second socket's file descriptor
 * @param filename name of the file to be retrieved
 */
void createFile(int fd, char *filename);
```

download.c

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

#include <sys/socket.h>
#include <sys/types.h>

#include <ctype.h>
#include <errno.h>
#include <netdb.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>

#include "download.h"

int usage(char *argv[])
{
    printf("Usage: %s ftp://[<user>:<password>@]<host>/<url-path>\n", argv[0]);
    return 1;
}

bool parseArgument(char *argument, info_t *info)
{
    char *sep;
    char* lastSep;
    int index1 = 6, index2;

    if (strncmp("ftp://", argument, 6) != 0)
        return false;

    if ((sep = strchr(argument + 6, ':')) != NULL)
    {
        int index;
```

```
        index = (int)(sep - argument);
        strncpy(info->user, argument + 6, index - 6);
        info->user[index - 6] = '\0';

        if ((sep = strchr(argument, '@')) == NULL)
            return false;

        int new_index = (int)(sep - argument);
        index++;
        strncpy(info->pass, argument + index, new_index - index);
        info->pass[new_index - index] = '\0';

        index1 = ++new_index;
    }
    else if ((sep = strchr(argument, '@')) != NULL)
        return false;
    else
        strncpy(info->user, "placeholder", 11);

    if ((sep = strchr(argument + 6, '/')) == NULL)
        return false;

    index2 = (int)(sep - argument);

    strncpy(info->serverName, argument + index1, index2 - index1);
    info->serverName[index2 - index1] = '\0';
    index2++;
    strncpy(info->filePath, argument + index2, strlen(argument) - index2);
    info->filePath[strlen(argument) - index2] = '\0';

    lastSep = strrchr(argument, '/');
    strcpy(info->fileName, lastSep + 1);
    info->fileName[strlen(lastSep)] = '\0';

    return true;
}

char *getServerIp(const char* name)
{
    struct hostent *h;

    if ((h = gethostbyname(name)) == NULL)
    {
        herror("gethostbyname");
        exit(1);
```

```
    }

    return inet_ntoa(*((struct in_addr *)h->h_addr_list[0]));
}

int createSocketTCP(char *server_ip, int server_port)
{
    int socketFd;
    struct sockaddr_in server_addr;

    /*server address handling*/
    bzero((char *)&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr =
        inet_addr(server_ip); /*32 bit Internet address network byte ordered*/
    server_addr.sin_port =
        htons(server_port); /*server TCP port must be network byte ordered */

    /*open an TCP socket*/
    if ((socketFd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket()");
        exit(0);
    }

    /*connect to the server*/
    if (connect(socketFd, (struct sockaddr *)&server_addr, sizeof(server_addr)) <
        0)
    {
        perror("connect()");
        exit(0);
    }

    return socketFd;
}

void readServerReply(int socketFd, char *reply)
{
    char c;
    int res = 0, i = 0;
    state_t state = READ_CODE;

    while (state != END)
    {
        if ((res = read(socketFd, &c, 1)) <= 0)
            continue;
```

```
        switch (state)
        {
        case READ_CODE:
            if (c == ' ')
            {
                state = READ_LINE;
                i = 0;
            }
            else if (c == '-')
            {
                state = READ_MULT_LINE;
                i = 0;
            }
            else if (isdigit(c))
            {
                reply[i++] = c;
            }
            break;
        case READ_LINE:
            if (c == '\n')
                state = END;
            break;
        case READ_MULT_LINE:
            if (c == reply[i])
            {
                i++;
            }
            else if (i == 3 && c == ' ')
            {
                state = READ_LINE;
            }
            break;
        case END:
            break;
        default:
            break;
        }
    }
}

int getServerPort(int socketFd)
{
    int res = 0;
    state_t state = WAIT_FOR_PORT;
    char c;
```

```
char first_byte[4], second_byte[4];
int numCommas = 0, i = 0;

while (state != END)
{
    if ((res = read(socketFd, &c, 1)) <= 0)
        continue;

    switch (state)
    {
    case READ_CODE:
        if (c == ' ')
        {
            if (i != 3)
            {
                printf(" > Error receiving response code\n");
                return -1;
            }
            i = 0;
            state = WAIT_FOR_PORT;
        }
        else
        {
            i++;
        }
        break;
        break;
    case WAIT_FOR_PORT:
        if (c == ',')
            numCommas++;

        if (numCommas == 4)
            state = FIRST_PORT_BYTE;
        break;
    case FIRST_PORT_BYTE:
        if (c == ',')
        {
            state = SECOND_PORT_BYTE;
            i = 0;
        }
        else
        {
            first_byte[i++] = c;
        }
        break;
    case SECOND_PORT_BYTE:
```

```
            if (c == ')')
                state = END;
            else
            {
                second_byte[i++] = c;
            }
            break;
        case END:
            break;
        default:
            break;
        }
    }

    return atoi(first_byte) * 256 + atoi(second_byte);
}

int sendCommand(int socketFd, char *command, char *argument)
{
    char reply[REPLY_CODE_SIZE];
    reply_type_t type;

    write(socketFd, command, strlen(command));
    if (argument != NULL)
        write(socketFd, argument, strlen(argument));
    write(socketFd, "\n", 1);

    while (true)
    {
        readServerReply(socketFd, reply);

        type = reply[0] - '0';

        switch (type)
        {
        case POSITIVE_PRE:
            break;
        case POSITIVE_INT:
            return 0;
        case POSITIVE_COMP:
            return 1;
        case TRANS_NEGATIVE_COMP:
            write(socketFd, command, strlen(command));
            if (argument != NULL)
                write(socketFd, argument, strlen(argument));
            write(socketFd, "\n", 1);
```

```
            break;
        case PERM_NEGATIVE_COMP:
            close(socketFd);
            return -1;
        default:
            break;
        }
    }
}

void createFile(int fd, char *filename)
{
    FILE *file = fopen(filename, "wb+");

    char fileData[SOCKET_BUF_SIZE];
    int nbytes;
    while ((nbytes = read(fd, fileData, SOCKET_BUF_SIZE)) > 0)
    {
        nbytes = fwrite(fileData, nbytes, 1, file);
    }

    fclose(file);
}

int main(int argc, char *argv[])
{
    info_t info;
    char *server_ip;
    char reply[MAX_REPLY_SIZE];
    int fd1, fd2, res, port;

    if (argc != 2 || !parseArgument(argv[1], &info))
        return usage(argv);

    server_ip = getServerIp(info.serverName);

    fd1 = createSocketTCP(server_ip, SERVER_PORT);
    readServerReply(fd1, reply);

    if (reply[0] == '2')
        printf(" > Connection established!\n");
    else
    {
        printf(" > Couldn't connect! Exiting.\n");
        exit(1);
    }
```

```
    if(strncmp(info.user, "placeholder", 11) == 0){
        printf(" > Please specify user and password.\n");
        printf(" > User: ");
        scanf("%s", info.user);
        printf(" > Password: ");
        scanf("%s", info.pass);
    }

    printf(" > Sending user\n");
    res = sendCommand(fd1, "user ", info.user);

    if (res == 0 || res == 1)
    {
        printf(" > Sending pass\n");
        res = sendCommand(fd1, "pass ", info.pass);
    }
    else
    {
        printf(" > Error sending username! Exiting.\n");
        exit(1);
    }

    write(fd1, "pasv\n", 5);
    port = getServerPort(fd1);

    fd2 = createSocketTCP(server_ip, port);

    printf(" > Sending retr\n");
    res = sendCommand(fd1, "retr ", info.filePath);

    if (res == 0)
    {
        printf(" > Downloading file...\n");
        createFile(fd2, info.fileName);
    }

    printf(" > Quiting connection\n");
    write(fd1, "quit\n", 5);

    close(fd1);
    close(fd2);

    printf(" > All done! Exiting now\n");
    return 0;
}
```

6.2 Comandos de configuração

6.3 *Logs* registados