

**Rede de Computadores**

***2º Trabalho Laboratorial***

1. **Aplicação de download**
2. **Configuração e análise de uma rede**

Mestrado Integrado em Engenharia Informática e Computação

André Esteves [up201606673@fe.up.pt](mailto:up201606673@fe.up.pt)

Francisco Friande [up201508213@fe.up.pt](mailto:up201508213@fe.up.pt)

Luís Diogo Silva [up201503730@fe.up.pt](mailto:up201503730@fe.up.pt)

**Índice**

[**Introdução** 3](#_Toc532910266)

[**Parte 1 – Aplicação de download** 4](#_Toc532910267)

[**Arquitetura da aplicação** 4](#_Toc532910268)

[**Exemplo de um download bem-sucedido** 5](#_Toc532910269)

[**Parte 2 – Configuração e análise de uma rede** 6](#_Toc532910270)

[**Experiência 1 – Configurar um IP de rede** 6](#_Toc532910271)

[**Experiência 2 – Implementar duas LAN’s virtuais num switch** 6](#_Toc532910272)

[**Experiência 3 – Configurar um Router em Linux** 6](#_Toc532910273)

[**Experiência 4 – Configurar um Router Comercial e Implementar NAT** 6](#_Toc532910274)

[**Experiência 5 – DNS** 7](#_Toc532910275)

[**Experiência 6 – Conexões TCP** 7](#_Toc532910276)

[**Conclusões** 8](#_Toc532910277)

[**Referências** 8](#_Toc532910278)

[**Anexos** 9](#_Toc532910279)

[**Imagens** 9](#_Toc532910280)

[**Código** 11](#_Toc532910281)

[info.h 11](#_Toc532910282)

[client.c 12](#_Toc532910283)

# **Introdução**

Este relatório foi elaborado no âmbito da unidade curricular de Rede de Computadores. O trabalho em causa consistia em duas partes. A primeira requer a elaboração de uma aplicação que faz o download de um ficheiro dado o seu URL e a segunda parte consiste na configuração e análise de uma rede no decorrer das aulas práticas sendo ao todo sete experiências.

O trabalho foi realizado no seu todo no ambiente disponibilizado, sendo concluído com sucesso em todos os aspetos, cumprindo os objetivos pedidos.

# **Parte 1 – Aplicação de download**

A primeira parte do trabalho consistia do desenvolvimento de uma aplicação de linha de comandos em C que fizesse um download de um ficheiro de um servidor FTP, como está descrito em RFC959. O programa recebe do utilizador um URL com o formato ***ftp://[<user>:<password>@]<host>/<url-path>***, como está descrito em RFC1738.

## **Arquitetura da aplicação**

A aplicação começa por fazer interpretar o URL inserido pelo utilizador, fazendo uso da função *parseInfo*, colocando esta informação numa instância da *struct* Info:

|  |
| --- |
| struct Info { |
|  | char hostname[100]; |
|  | char path[150]; |
|  | char\* filename; |
|  | char user[50]; |
|  | char password[50]; |
|  | }; |

Caso o utilizador tenha optado por não inserir o *username* e a *password* na linha de comandos, como está previsto em RFC1738, estes são obtidos através do *input* do utilizador, fazendo uso da função *getUserInfo*.

De seguida, é criada uma instância da *struct* hostent (definida em netdb.h), na qual é guardado o IP do servidor, fazendo uso da função *getHostInfo*. Esta informação é usada para criar o *socket* que vai ser utilizado para comunicar com o servidor. Isto é feito através da função *connectTCP*. Depois desta ligação, é lida a resposta do servidor (através da função *readResponseCode*), de forma a certificarmo-nos que esta foi feita com sucesso. De forma a concluir esta etapa inicial de conexão, é enviada ao servidor a informação de *login*, que está armazenada na *struct* Info, através da função *sendLoginInfo*.

Estando estabelecida a ligação, é pedido ao servidor para entrar em modo passivo. Isto é feito através da função *getServerPort*. Esta função retorna a porta que vai ser utilizada pelo servidor para o envio do ficheiro. Esta a informação é usada para criar um novo canal de comunicação com o servidor, na forma de um outro *socket*, fazendo uso, novamente, da função *connectTCP*.

Finalmente, o ficheiro é recebido por parte do utilizador após a chamada à função *retrieveFile*. Após esta chamada, os canais de comunicação são fechados e a aplicação termina.

## **Exemplo de um download bem-sucedido**

Na secção *Anexos*, estão documentados a compilação da aplicação (Figura 1) e dois downloads bem-sucedidos. No primeiro caso (Figura 2), o utilizador colocou o *username* e a *password* no argumento passado na linha de comandos. No segundo caso (Figuras 3 e 4), o utilizador optou por não inserir estes dados na linha de comandos. A figura 3 mostra o programa à espera que o utilizador insira o *username*. A figura 4 mostra o programa após a sua execução. Na figura 5, é possível ver o ficheiro transferido do diretório onde foi executada a aplicação.

# **Parte 2 – Configuração e análise de uma rede**

## **Experiência 1 – Configurar um IP de rede**

O objetivo desta experiência é ligar dois computadores (no caso, **tux1** e **tux4**) na mesma rede, fazendo uso de um *switch*. Isto foi realizado através dos seguintes comandos:

**Tux1** – ifconfig eth0 172.16.60.1/24

**Tux4** – ifconfig eth0 172.16.60.254/24

Isto fez com que ambos os computadores usassem como rota de comunicação a sua porta eth0 (que estavam ligadas ao *switch*), utilizando os endereços IP respetivos.

1. O que são os pacotes ARP? Para o que são usados?

Os pacotes ARP são pacotes que obedecem às definições do protocolo ARP (*Address Resolution Protocol*). Este protocolo serve para fazer o mapeamento de endereços de rede, como os endereços IP, com endereços físicos, como os endereços MAC.

1. O que são os endereços MAC e IP dos pacotes ARP e porquê?

Quando existe uma tentativa de comunicação entre dois computadores em rede, o computador recetor envia um pacote ao emissor no sentido de obter o endereço físico (endereço MAC) deste. Este pacote enviado pelo recetor é um pacote ARP, que contém simultaneamente o endereço IP e o endereço MAC deste, assim como o endereço IP do emissor (ou seja, o endereço IP que o recetor deseja corresponder com um endereço MAC). O emissor responde com um pacote ARP muito semelhante à pergunta enviada pelo recetor, exceto que contém também o seu endereço MAC (Figuras 6 e 7).

1. Quais os pacotes gerados pelo comando *ping*?

O comando *ping* gera pacotes ARP de forma a obter o endereço MAC do recetor (como foi explicado na pergunta anterior), e pacotes ICMP (*Internet Control Message Protocol*). Estes são usados para transferir mensagens de controlo entre endereços IP.

1. Quais são os endereços MAC e IP dos pacotes *ping*?

Os pacotes IP contêm os endereços MAC e IP tanto do emissor como do recetor (Figura 8).

1. Como determinar se a trama Ethernet recebida é ARP, IP, ICMP?

Todas as tramas Ethernet contêm um Ethernet Header que contém informação sobre o tipo de trama. Caso este valor seja 0x0800, estamos perante uma trama IP. Estas tramas, por sua vez, contêm um IP Header, que identifica o tipo de protocolo nelas utilizado. Caso este tome o valor 0x0806, a trama utiliza o protocolo ARP (Figura 9). Se, pelo contrário, o valor for 1, a trama utiliza o protocolo ICMP (Figura 10).

1. Como determinar o comprimento de uma trama recebida?

O tamanho de tramas IP está presente no IP Header (Figura 11).

1. O que é a interface *loopback* e porque é importante?

A interface *loopback* é uma interface de rede virtual que é utilizada pelo computador para testar o estado do sistema. Esta envia pacotes do tipo LOOP em intervalos de 10 segundos ao próprio computador (Figura 12), verificando se este ainda se encontra em estado de atividade

## **Experiência 2 – Implementar duas LAN’s virtuais num switch**

Nesta experiência

1. Como configurar vlany0?
2. Quantos domínios de transmissão existem? O que se pode concluir a partir dos registos?

## **Experiência 3 – Configurar um Router em Linux**

Nesta experiência.

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

## **Experiência 4 – Configurar um Router Comercial e Implementar NAT**

Nesta experiência.

1. Como configurar um router estático num router comercial?
2. Quais são as rotas seguidas pelos pacotes nas experiências anteriormente seguidas e porquê?
3. Como configurar o NAT num router comercial?
4. O que faz o NAT?

## **Experiência 5 – DNS**

Nesta experiência.

1. Como configurar o serviço DNS num *host*?
2. Que pacotes são trocados pelo DNS e que informações são transportadas?

## **Experiência 6 – Conexões TCP**

Nesta experiência.

1. Quantas conexões TCP são abertas pela aplicação FTP?
2. Em qual conexão é transportado a informação de controlo FTP?
3. Quais são as fases de uma conexão TCP?
4. Como é que o mecanismo ARQ TCP funciona? Quais são os campos TCP relevantes? Qual informação relevante pode ser observada nos *logs*?
5. Como é que o mecanismo de controlo de congestão TCP funciona? Quais são os campos relevantes? Como é que o fluxo da conexão de dados evoluiu ao longo do tempo? Está de acordo com o mecanismo de controlo de congestão TCP?
6. De que forma é afetada a conexão de dados TCP pelo aparecimento de uma segunda conexão TCP? Como?

# **Conclusões**

O grupo foi capaz de compreender e interiorizar os conceitos relacionados tanto com os protocolos FTP, assim como os que dizem respeito à configuração de uma rede.

O desenvolvimento da aplicação de download permitiu-nos ter uma maior compreensão sobre o funcionamento de protocolos de comunicação em geral, assim como por em perspetiva o protocolo de comunicação desenvolvido pelo grupo no âmbito do primeiro trabalho desta unidade curricular. De modo semelhante, a configuração da rede em ambiente de laboratório permitiu ao grupo ter um maior entendimento sobre conceitos de rede, assim como compreender como é que o que foi feito em ambiente controlado poderia ser escalado a uma implementação mais alargada.

Após o término no desenvolvimento deste projeto, podemos dizer que foram cumpridos os objetivos que nos foram propostos.

# **Referências**

<https://www.ietf.org/rfc/rfc959.txt> (consultado pela última vez em 2018/12/18)

<https://www.ietf.org/rfc/rfc1738.txt> (consultado pela última vez em 2018/12/18)

# **Anexos**

## **Imagens**

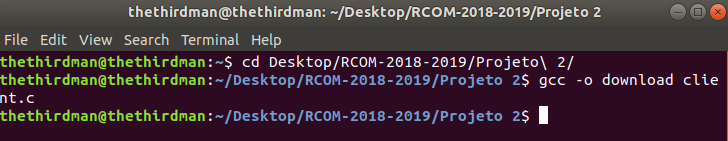
****

Figura 1 - Compilação da aplicação

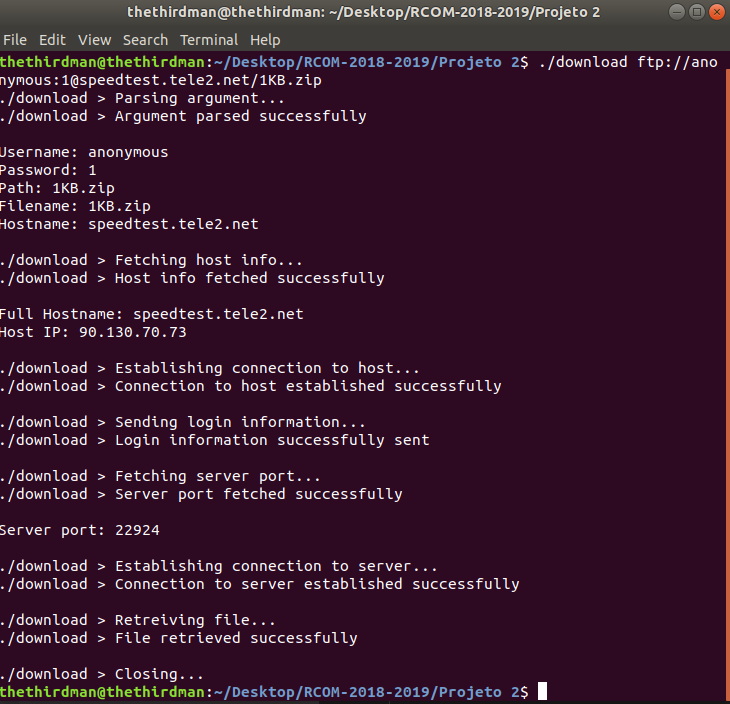
****

Figura 2 - Exemplo de execução com username

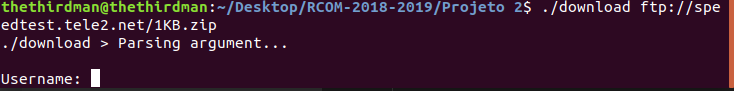
****

Figura 3 - Exemplo de execução sem username (1)

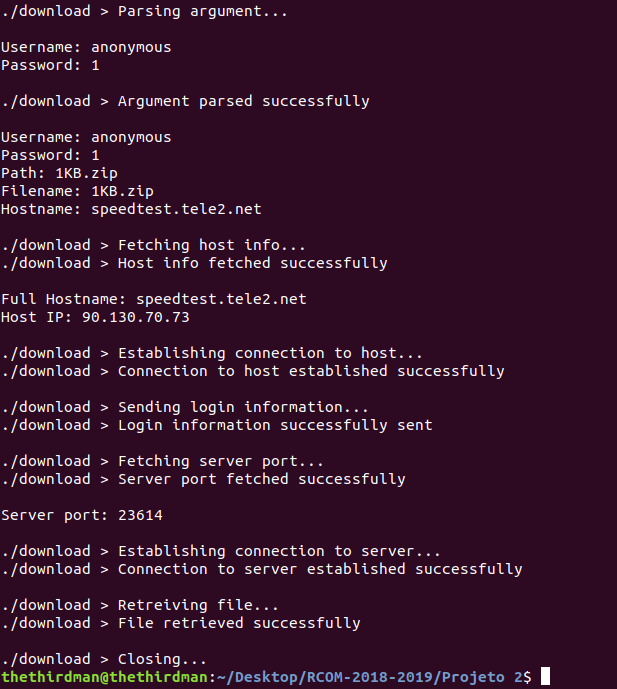
****

Figura 4 - Exemplo de execução sem username (2)

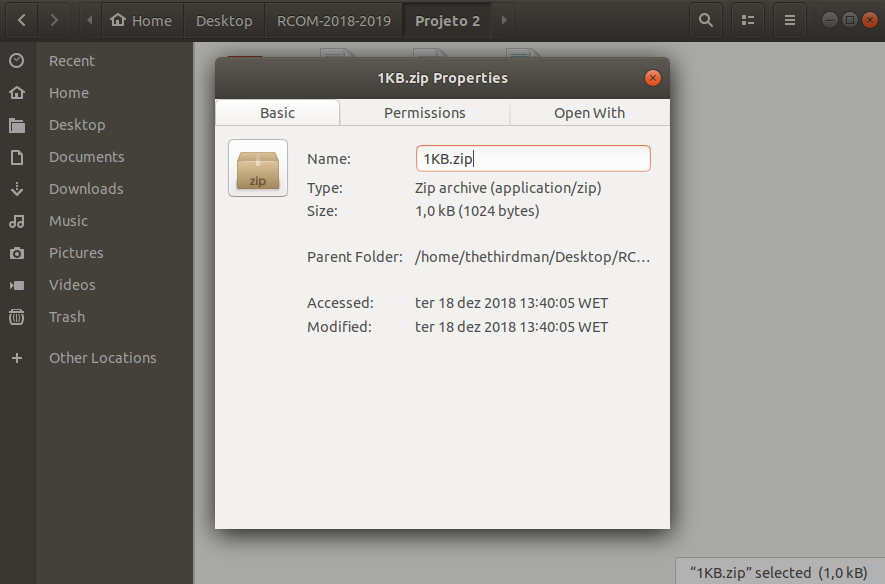
****

Figura 5 - Ficheiro recebido após execução da aplicação

## **Código**

### info.h

|  |
| --- |
| /\*\* |
|  | \* Compilation: gcc -o download client.c |
|  | \* Test: ./download ftp://anonymous:1@speedtest.tele2.net/1KB.zip |
|  | \*/ |
|  |  |
|  | #define BEGIN 0 |
|  | #define END 99 |
|  |  |
|  | #define USER 1 |
|  | #define PASSWORD 2 |
|  | #define HOST 3 |
|  | #define PATH 4 |
|  |  |
|  | #define CLEAR\_LINE 1 |
|  | #define MULTIPLE\_LINE 2 |
|  |  |
|  | #define PORT 21 |
|  |  |
|  | #define CMD\_USER "user " |
|  | #define CMD\_PASS "pass " |
|  | #define CMD\_PASSIVE "pasv" |
|  | #define CMD\_RETRIEVE "retr " |
|  |  |
|  | /\*\* @name Info Struct\*/ |
|  | /\*\*@{ |
|  | \* |
|  | \* Struct to store user information |
|  | \*/ |
|  | struct Info { |
|  | char hostname[100]; |
|  | char path[150]; |
|  | char\* filename; |
|  | char user[50]; |
|  | char password[50]; |
|  | }; |
|  | /\*\* @} end of Info Struct \*/ |

### client.c

|  |
| --- |
| #include <stdio.h> |
|  | #include <stdlib.h> |
|  | #include <netdb.h> |
|  | #include <sys/types.h> |
|  | #include <netinet/in.h> |
|  | #include <arpa/inet.h> |
|  | #include <sys/socket.h> |
|  | #include <unistd.h> |
|  | #include <signal.h> |
|  | #include <strings.h> |
|  | #include <string.h> |
|  | #include <ctype.h> |
|  | #include "info.h" |
|  |  |
|  | /\*\* |
|  | \* @brief Parses user information from argument input of user |
|  | \* |
|  | \* Implements a state machine to parse user information |
|  | \* |
|  | \* States: |
|  | \* - BEGIN - reads start section ftp:// |
|  | \* - USER - reads user username (optional field) |
|  | \* - PASSWORD - reads user password (optional field) |
|  | \* - HOST - reads host name |
|  | \* - PATH - reads file saving path |
|  | \* |
|  | \* @param cmd string containing user input argument |
|  | \* @param info struct where user info will be saved on |
|  | \* @return 0 on success, non-zero otherwise |
|  | \*/ |
|  | int parseInfo(char\* cmd, struct Info\* info) |
|  | { |
|  | int state = BEGIN; |
|  | int i = 0; |
|  | int aux = 0; |
|  | char c; |
|  |  |
|  | while(state != END) |
|  | { |
|  | c = cmd[i++]; |
|  | switch(state) |
|  | { |
|  | case BEGIN: |
|  | { |
|  | if(c == '\0') |
|  | return -1; |
|  |  |
|  | if(c != 'f') |
|  | return -2; |
|  | c = cmd[i++]; |
|  | if(c != 't') |
|  | return -2; |
|  | c = cmd[i++]; |
|  | if(c != 'p') |
|  | return -2; |
|  | c = cmd[i++]; |
|  | if(c != ':') |
|  | return -2; |
|  | c = cmd[i++]; |
|  | if(c != '/') |
|  | return -2; |
|  | c = cmd[i++]; |
|  | if(c != '/') |
|  | return -2; |
|  |  |
|  | if(strrchr(cmd, '@') != NULL) |
|  | state = USER; |
|  | else |
|  | state = HOST; |
|  | break; |
|  | } |
|  | case USER: |
|  | { |
|  | if(c == '\0') |
|  | return -1; |
|  |  |
|  | if(c == ':') |
|  | { |
|  | (\*info).user[aux] = 0; |
|  | aux = 0; |
|  | state = PASSWORD; |
|  | } |
|  | else |
|  | { |
|  | (\*info).user[aux++] = c; |
|  | } |
|  | break; |
|  | } |
|  | case PASSWORD: |
|  | { |
|  | if(c == '\0') |
|  | return -1; |
|  |  |
|  | if(c == '@') |
|  | { |
|  | (\*info).password[aux] = 0; |
|  | aux = 0; |
|  | state = HOST; |
|  | } |
|  | else |
|  | { |
|  | (\*info).password[aux++] = c; |
|  | } |
|  | break; |
|  | } |
|  | case HOST: |
|  | { |
|  | if(c == '\0') |
|  | return -1; |
|  |  |
|  | if(c == '/') |
|  | { |
|  | (\*info).hostname[aux] = 0; |
|  | aux = 0; |
|  | state = PATH; |
|  | } |
|  | else |
|  | { |
|  | (\*info).hostname[aux++] = c; |
|  | } |
|  | break; |
|  | } |
|  | case PATH: |
|  | { |
|  | if(c == '\0') |
|  | { |
|  | (\*info).path[aux] = 0; |
|  | aux = 0; |
|  | state = END; |
|  | } |
|  | else |
|  | { |
|  | (\*info).path[aux++] = c; |
|  | } |
|  | break; |
|  | } |
|  | case END: |
|  | { |
|  | return 0; |
|  | break; |
|  | } |
|  | default: return -1; |
|  | } |
|  | } |
|  | return 0; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* @brief Gets username and password from user |
|  | \* |
|  | \* This function is only called if user did not fill the optional |
|  | \* fields username and password in the command line argument |
|  | \* |
|  | \* @param info struct where user info will be saved on |
|  | \*/ |
|  | void getUserInfo(struct Info\* info) |
|  | { |
|  | char\* buf = malloc(50 \* sizeof(char)); |
|  | size\_t size = 50; |
|  | fflush(stdin); |
|  | printf("\nUsername: "); |
|  | getline(&buf, &size, stdin); |
|  | strncpy(info->user, buf, strcspn(buf, "\n")); |
|  |  |
|  | printf("Password: "); |
|  | getline(&buf, &size, stdin); |
|  | strncpy(info->password, buf, strcspn(buf, "\n")); |
|  | printf("\n"); |
|  | } |
|  |  |
|  | /\*\* |
|  | \* @brief Gets filename from the file saving path |
|  | \* |
|  | \* Isolates the filename from the path so that it can be sent to the server |
|  | \* |
|  | \* @param info struct where user info will be saved on |
|  | \*/ |
|  | void parseFilename(struct Info\* info) |
|  | { |
|  | char\* filename = strrchr(info->path, '/'); |
|  |  |
|  | if(filename == NULL) |
|  | { |
|  | info->filename = info->path; |
|  | } |
|  | else |
|  | { |
|  | info->filename = (filename + 1); |
|  | } |
|  | } |
|  |  |
|  | /\*\* |
|  | \* @brief Gets host informtaion using host name given by uer |
|  | \* |
|  | \* Gets host information (particullary the IP address), so that a connection |
|  | \* can be established with the server |
|  | \* |
|  | \* @param hostmane string containing host name |
|  | \* @param h struct where host info will be saved on |
|  | \* @return 0 on success, non-zero otherwise |
|  | \*/ |
|  | int getHostInfo(char\* hostname, struct hostent\*\* h) |
|  | { |
|  | if ((\*h=gethostbyname(hostname)) == NULL) |
|  | return -1; |
|  |  |
|  | return 0; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* @brief Establishes a connection with the server |
|  | \* |
|  | \* Establishes a socket connecting to the server using its IP address |
|  | \* |
|  | \* @param addr string containing server ip address |
|  | \* @param port port used by server to send and receive information |
|  | \* @return socket file descriptor on success, negative otherwise |
|  | \*/ |
|  | int connectTCP(char\* addr, int port) |
|  | { |
|  | int sockfd; |
|  | struct sockaddr\_in server\_addr; |
|  |  |
|  | bzero((char\*)&server\_addr,sizeof(server\_addr)); |
|  | server\_addr.sin\_family = AF\_INET; |
|  | server\_addr.sin\_addr.s\_addr = inet\_addr(addr); |
|  | server\_addr.sin\_port = htons(port); |
|  |  |
|  | if ((sockfd = socket(AF\_INET,SOCK\_STREAM,0)) < 0) |
|  | return -1; |
|  |  |
|  | if(connect(sockfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) |
|  | return -2; |
|  |  |
|  | return sockfd; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* @brief Reads response code from server |
|  | \* |
|  | \* Implements a state machine to reads response code followed by a user command |
|  | \* Must be called after each command is sent |
|  | \* |
|  | \* States: |
|  | \* - BEGIN - reads response code |
|  | \* - CLEAR\_LINE - clears line after code is read |
|  | \* - MULTIPLE\_LINES - reads additional lines if necessary |
|  | \* |
|  | \* @param socketfd socket file descriptor |
|  | \* @param responseCode string where the response code will be saved |
|  | \* @return 0 on success, non-zero otherwise |
|  | \*/ |
|  | int readResponseCode(int socketfd, char \*responseCode) |
|  | { |
|  | int state = BEGIN; |
|  | int index = 0; |
|  | char c; |
|  |  |
|  | while (state != END) |
|  | { |
|  | read(socketfd, &c, 1); |
|  | switch(state) |
|  | { |
|  | case BEGIN: |
|  | { |
|  | if(c == ' ') |
|  | { |
|  | if (index != 3) |
|  | { |
|  | return -1; |
|  | } |
|  | index = 0; |
|  | state = CLEAR\_LINE; |
|  | } |
|  | else if(c == '-') |
|  | { |
|  | state = MULTIPLE\_LINE; |
|  | index = 0; |
|  | } |
|  | else if(isdigit(c)) |
|  | { |
|  | responseCode[index++] = c; |
|  | } |
|  | break; |
|  | } |
|  | case CLEAR\_LINE: |
|  | { |
|  | if (c == '\n') |
|  | { |
|  | state = END; |
|  | } |
|  | break; |
|  | } |
|  | case MULTIPLE\_LINE: |
|  | { |
|  | if(c == responseCode[index]) |
|  | { |
|  | index++; |
|  | } |
|  | else if(index == 3 && c == ' ') |
|  | { |
|  | state = CLEAR\_LINE; |
|  | } |
|  | else if(index ==3 && c == '-') |
|  | { |
|  | index = 0; |
|  | } |
|  | break; |
|  | } |
|  | case END: |
|  | { |
|  | return 0; |
|  | break; |
|  | } |
|  | default: return -1; |
|  | } |
|  | } |
|  | return 0; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* @brief Writes command to server |
|  | \* |
|  | \* @param fd socket file descriptor |
|  | \* @param cmd command to be sent |
|  | \* @param info command arguments |
|  | \*/ |
|  | void writeCmd(int fd, char\* cmd, char\* info) |
|  | { |
|  | write(fd, cmd, strlen(cmd)); |
|  | write(fd, info, strlen(info)); |
|  | write(fd, "\n", 1); |
|  | } |
|  |  |
|  | /\*\* |
|  | \* @brief Sends command to user and awaits response |
|  | \* |
|  | \* Sends command to user and acts accordingly to server response |
|  | \* |
|  | \* Answer values: |
|  | \* - 1 - sending another response (except for retr command) |
|  | \* - 2 - success |
|  | \* - 3 - server needs additional information to proceed |
|  | \* - 4 - resend command |
|  | \* - 5 - error |
|  | \* |
|  | \* @param cmd string containing user input argument |
|  | \* @param info struct where user info will be saved on |
|  | \* @return Success values: |
|  | \* @return 0 - Command accepted |
|  | \* @return 1 - Command accepted - needs additional infromation |
|  | \* @return 2 - Command accepted - file ready for retrieval |
|  | \* @return Negative codes represent error |
|  | \*/ |
|  | int writeCommand(int fd, char\* cmd, char\* info) |
|  | { |
|  | char code[3]; |
|  | writeCmd(fd, cmd, info); |
|  | readResponseCode(fd, code); |
|  | int answer = code[0] - '0'; |
|  |  |
|  | while(1) |
|  | { |
|  | switch(answer) |
|  | { |
|  | case 1: |
|  | { |
|  | if(strcmp(cmd, "retr ")==0) |
|  | return 2; |
|  | readResponseCode(fd, code); |
|  | break; |
|  | } |
|  | case 2: |
|  | return 0; |
|  | case 3: |
|  | return 1; |
|  | case 4: |
|  | { |
|  | writeCmd(fd, cmd, info); |
|  | break; |
|  | } |
|  | case 5: |
|  | return -1; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | /\*\* |
|  | \* @brief Sends login info to server |
|  | \* |
|  | \* @param info struct containing user info |
|  | \* @param sockfd socket file descriptor |
|  | \* @return 0 on success, non-zero otherwise |
|  | \*/ |
|  | int sendLoginInfo(struct Info\* info, int sockfd) |
|  | { |
|  | if(writeCommand(sockfd, CMD\_USER, info->user) != 1) |
|  | return -1; |
|  |  |
|  | if(writeCommand(sockfd, CMD\_PASS, info->password) != 0) |
|  | return -1; |
|  |  |
|  | return 0; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* @brief Gets server port |
|  | \* |
|  | \* Gets port in which server will communicate using passive mode |
|  | \* Reads both bytes of the port and merges them |
|  | \* |
|  | \* @param sockfd socket file descriptor |
|  | \* @return server port on success, negative value otherwise |
|  | \*/ |
|  | int getServerPort(int sockfd) |
|  | { |
|  | writeCmd(sockfd, CMD\_PASSIVE, ""); |
|  |  |
|  | int state = 0; |
|  | int index = 0; |
|  | char msg1[4]; |
|  | memset(msg1, 0, 4); |
|  | char msg2[4]; |
|  | memset(msg2, 0, 4); |
|  |  |
|  | char c; |
|  |  |
|  | while (state != 7) |
|  | { |
|  | read(sockfd, &c, 1); |
|  | switch (state) |
|  | { |
|  | case 0: |
|  | { |
|  | if (c == ' ') |
|  | { |
|  | if (index != 3) |
|  | return -1; |
|  | index = 0; |
|  | state = 1; |
|  | } |
|  | else |
|  | { |
|  | index++; |
|  | } |
|  | break; |
|  | } |
|  | case 5: |
|  | { |
|  | if (c == ',') |
|  | { |
|  | index = 0; |
|  | state++; |
|  | } |
|  | else |
|  | { |
|  | msg1[index] = c; |
|  | index++; |
|  | } |
|  | break; |
|  | } |
|  | case 6: |
|  | { |
|  | if (c == ')') |
|  | { |
|  | state++; |
|  | } |
|  | else |
|  | { |
|  | msg2[index] = c; |
|  | index++; |
|  | } |
|  | break; |
|  | } |
|  | default: |
|  | { |
|  | if (c == ',') |
|  | { |
|  | state++; |
|  | } |
|  | break; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | int b1 = atoi(msg1); |
|  | int b2 = atoi(msg2); |
|  | return (b1 \* 256 + b2); |
|  | } |
|  |  |
|  | /\*\* |
|  | \* @brief Gets file from server |
|  | \* |
|  | \* Gets file information from server and saves it in the specified path |
|  | \* |
|  | \* @param info struct containing user info |
|  | \* @param sockfd socket file descriptor |
|  | \* @param serverfd passive server file descriptor |
|  | \* @return 0 on success, non-zero otherwise |
|  | \*/ |
|  | int retrieveFile(struct Info\* info, int sockfd, int serverfd) |
|  | { |
|  | if(writeCommand(sockfd, CMD\_RETRIEVE, info->path) != 2) |
|  | return -1; |
|  |  |
|  | FILE \*file = fopen(info->filename, "wb+"); |
|  |  |
|  | char buffer[1024]; |
|  | int bytes; |
|  |  |
|  | bytes = read(serverfd, buffer, 1024); |
|  |  |
|  | while (bytes > 0) { |
|  | bytes = fwrite(buffer, bytes, 1, file); |
|  | bytes = read(serverfd, buffer, 1024); |
|  | } |
|  |  |
|  | fclose(file); |
|  |  |
|  | return 0; |
|  | } |
|  |  |
|  | int main(int argc, char\*\* argv) |
|  | { |
|  | if(argc != 2) |
|  | { |
|  | printf("Usage: %s ftp://[<user>:<password>@]<host>/<url-path>\n", argv[0]); |
|  | return -1; |
|  | } |
|  |  |
|  | printf("%s > Parsing argument...\n", argv[0]); |
|  |  |
|  | struct Info info; |
|  |  |
|  | if(parseInfo(argv[1], &info) != 0) |
|  | { |
|  | printf("Error parsing client info: %s\n", argv[1]); |
|  | return -2; |
|  | } |
|  |  |
|  | parseFilename(&info); |
|  |  |
|  | if(info.user[0] == 0) |
|  | getUserInfo(&info); |
|  |  |
|  | printf("%s > Argument parsed successfully\n\n", argv[0]); |
|  | printf("Username: %s\n", info.user); |
|  | printf("Password: %s\n", info.password); |
|  | printf("Path: %s\n", info.path); |
|  | printf("Filename: %s\n", info.filename); |
|  | printf("Hostname: %s\n\n", info.hostname); |
|  |  |
|  | printf("%s > Fetching host info...\n", argv[0]); |
|  |  |
|  | struct hostent \*host; |
|  |  |
|  | if(getHostInfo(info.hostname, &host) != 0) |
|  | { |
|  | printf("Error getting host IP address: %s\n", info.hostname); |
|  | return -3; |
|  | } |
|  |  |
|  | char\* addr\_str = inet\_ntoa(\*((struct in\_addr \*)host->h\_addr)); |
|  |  |
|  | printf("%s > Host info fetched successfully\n\n", argv[0]); |
|  | printf("Full Hostname: %s\n", host->h\_name); |
|  | printf("Host IP: %s\n\n", addr\_str); |
|  |  |
|  | printf("%s > Establishing connection to host...\n", argv[0]); |
|  |  |
|  | int sockfd = connectTCP(addr\_str, PORT); |
|  |  |
|  | if(sockfd < 0) |
|  | { |
|  | printf("Error connecting to IP %s using port %d\n", addr\_str, PORT); |
|  | return -4; |
|  | } |
|  |  |
|  | char code[3]; |
|  |  |
|  | if(readResponseCode(sockfd, code) != 0) |
|  | { |
|  | printf("Error reading server response code\n"); |
|  | return -5; |
|  | } |
|  |  |
|  | if(code[0] != '2') |
|  | { |
|  | printf("Error: server responded with code %s\n", code); |
|  | return -6; |
|  | } |
|  |  |
|  | printf("%s > Connection to host established successfully\n\n", argv[0]); |
|  |  |
|  | printf("%s > Sending login information...\n", argv[0]); |
|  |  |
|  | if(sendLoginInfo(&info, sockfd) != 0) |
|  | { |
|  | printf("Error sending login information\n"); |
|  | return -7; |
|  | } |
|  |  |
|  | printf("%s > Login information successfully sent\n\n", argv[0]); |
|  |  |
|  | printf("%s > Fetching server port...\n", argv[0]); |
|  |  |
|  | int server\_port = getServerPort(sockfd); |
|  |  |
|  | if(server\_port < 0) |
|  | { |
|  | printf("Error fetching server port\n"); |
|  | return -8; |
|  | } |
|  |  |
|  | printf("%s > Server port fetched successfully\n\n", argv[0]); |
|  | printf("Server port: %d\n\n", server\_port); |
|  |  |
|  | printf("%s > Establishing connection to server...\n", argv[0]); |
|  |  |
|  | int serverfd = connectTCP(addr\_str, server\_port); |
|  |  |
|  | if(serverfd < 0) |
|  | { |
|  | printf("Error connecting to IP %s using port %d\n", addr\_str, PORT); |
|  | return -4; |
|  | } |
|  |  |
|  | printf("%s > Connection to server established successfully\n\n", argv[0]); |
|  |  |
|  | printf("%s > Retreiving file...\n", argv[0]); |
|  |  |
|  | if(retrieveFile(&info, sockfd, serverfd) < 0) |
|  | { |
|  | printf("Error retreiving file: %s\n", info.filename); |
|  | return -9; |
|  | } |
|  |  |
|  | printf("%s > File retrieved successfully\n\n", argv[0]); |
|  |  |
|  | printf("%s > Closing...\n", argv[0]); |
|  |  |
|  | close(sockfd); |
|  | close(serverfd); |
|  |  |
|  | return 0; |
|  | } |