Relatório

Trabalho Laboratorial 2

Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

3MIEIC06 - Bancada 2

Miguel Carvalho - **up201605757**
Sandro Campos - **up201605947**
Tiago Cardoso - **up201605762**

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

22 de Dezembro de 2018

Conteúdo

1 Sumário

O presente relatório, realizado no âmbito da unidade curricular de Redes de Computadores, aborda a realização de um trabalho laboratorial com a finalidade de configurar e estudar uma rede de computadores como suporte para o desenvolvimento de uma aplicação de *download*. Esta mesma aplicação permite realizar a transferência de um qualquer ficheiro a partir de um servidor FTP - *File Transfer Protocol*, conhecido o seu URL - *Uniform Resource Locator*.

2 Introdução

O objetivo deste trabalho é a implementação de uma aplicação que possa fazer uso de uma rede de computadores previamente configurada, e com este relatório pretende-se efetuar uma análise aprofundada de todo esse processo. Segue-se a sua estrutura:

- **Arquitetura** - apresentação dos blocos funcionais.
- **Estrutura do código** - exposição da *API*, principais funções, estruturas de dados e a sua relação com a arquitetura escolhida.
- **Casos de uso principais** - identificação e demonstração da invocação do programa.
- **Validação** - descrição dos testes efetuados à aplicação.
- **Experiências laboratoriais** - descrição e análise sumária de cada uma das experiências realizadas em contexto de laboratório.
- **Conclusões** - síntese da informação apresentada e reflexão acerca dos objetivos de aprendizagem alcançados.

3 Parte 1 - Aplicação de *download*

A primeira parte deste trabalho consiste no desenvolvimento de uma aplicação de *download* recorrendo à linguagem de programação C, fazendo uso de *sockets* a partir de ligações TCP - *Transmission Control Protocol*. Para tal houve necessidade de ter em consideração o funcionamento do protocolo FTP, e de estudar algumas das suas normas, como as descritas nos RFC959 e RFC1738, referentes ao processamento de endereços URL.

3.1 Arquitetura

Neste trabalho consideram-se existir dois blocos funcionais, o responsável pelo *parsing* do URL, argumento que se encontra no formato RFC1738 **ftp://<user>:<password>@<host>/<url-path>**, e o que interage com o servidor FTP, desencadeando a transferência do ficheiro pretendido.

3.2 Estrutura do Código

De acordo com o modelo arquitetural definido, seguem-se as principais funções e estruturas de dados referentes a cada bloco funcional.

Para processamento do URL as funções mais relevantes são a ***parseURL***, que efetua o *parsing* de cada um dos parâmetros passados através da linha de comandos, e a função auxiliar ***parseFilename***, responsável por extrair o nome do ficheiro através do seu caminho relativo no servidor.

O *parsing* encontra-se implementado por uma máquina de estados do tipo *URLstate*, com nomes correspondentes aos elementos relevantes do endereço.

```
/**
 * States for the URL parsing state machine
 */
typedef enum {INIT,FTP,USER,PWD,HOST,PATH} URLstate;
```

Estes elementos, após processados, são guardados em variáveis de uma *struct* do tipo **URLarguments**, cuja declaração se apresenta abaixo.

```
/**
 * Arguments obtained from the URL
 */
typedef struct URLarguments
{
  char user[256];
  char pwd[256];
  char hostname[256];
  char filepath[256];
  char filename[256];
  char hostIp[256];
} URLarguments;
```

No caso do bloco funcional relativo ao cliente TCP, as funções de maior importância são a ***openSocket***, ***sendCmd*** e ***receiveResponse***.

A primeira tem como objetivo criar um *socket* para comunicação com um servidor, dado o seu endereço IP e a porta à qual se conectar. Note-se que a porta para comunicação de controlo a um servidor com protocolo FTP é, por *default*, a nº 21.

A segunda, fazendo uso da ligação estabelecida ao servidor por meio de um *socket* de controlo, escreve comandos para serem recebidos e interpretados pelo mesmo.

A última recebe, por meio desse mesmo *socket*, informação relativa ao envio de um determinado comando para o servidor, um código de 3 dígitos e uma mensagem. Este código pode ser de vários tipos, destacando-se os de ação solicitada concluída com **êxito (2xx)**, os de ação solicitada a realizar após envio de **informação complementar**, como uma palavra-passe, **(3xx)**, e aqueles em que a ação solicitada foi **rejeitada (4xx e 5xx)**. Este

tipo de informação, também processada através de uma máquina de estados, é guardada em estruturas do tipo **ServerResponse** como a declarada a seguir.

```
/**
 * Server response code and message
 */
typedef struct ServerResponse
{
  char code[4];
  char msg[1024];
} ServerResponse;
```

Para se iniciar a transferência do ficheiro é ainda necessário autenticar o utilizador, mudar para o diretório correto, entrar em modo passivo, e requerer o ficheiro ao servidor. Este é posteriormente enviado através de um *socket* de dados criado pelo cliente, e que se encontra conectado a uma porta do servidor atribuída para esse efeito. As funções que intervêm neste processo são a ***login***, ***changeDir***, ***enterPasvMode***, ***openDataConnection*** e ***transferFile***, respetivamente.

3.3 Casos de uso principais

De forma a interagir com o programa o utilizador deve recorrer à interface existente da linha de comandos, caso de uso que é único.

Para dar início à aplicação deve então proceder-se à inserção de um só argumento, o URL de um ficheiro num servidor FTP *online*, que vai ser utilizado para transferência. Segue-se um exemplo com a sintaxe a usar:

```
./download ftp://<user>:<password>@<host>/<url-path>
```

3.4 Validação

A aplicação foi executada com ficheiros de diversas extensões e tamanhos, alojados em servidores distintos, e o comportamento foi o esperado. No terminal podem verificar-se as interações de comunicação existentes entre o cliente e o servidor, como observável na figura 1, em anexo I.

4 Parte 2 - Configuração e análise da rede

4.1 Experiência 1 - Configurar uma rede IP

Na primeira experiência iniciou-se a configuração de uma rede, conectando duas máquinas, tux1 e tux4, a um *switch*, de forma a possibilitar a comunicação entre elas. Para isso conectaram-se as interfaces eth0 de cada uma ao *switch*, e configuraram-se os seus endereços IP, através do uso do comando ***ifconfig***. A sua utilização pode ser observada nos dois comandos iniciais de cada script, em anexo II.

Ao executar o comando *ping* observa-se a troca de **pacotes ARP**, os de pedido, que são enviados de forma a requerer o **endereço MAC** da máquina com um determinado **endereço IP** a alcançar, e os correspondentes de resposta. Os endereços MAC e IP de origem de um pacote ARP correspondem ao sistema que o enviou e de destino os correspondentes no sistema a alcançar. Neste caso, o endereço MAC no pacote de destino é desconhecido, pelo que se questiona a rede acerca da máquina com o IP desejado e espera-se que essa mesma responda com o endereço MAC pretendido, que é posteriormente registado. Em anexo I, na figura 2, segue um exemplo da execução do comando *ping*, e na figura 6, um exemplo de troca de pacotes ARP.

Os diversos tipos de pacote que circulam na rede são distinguidos através de um processo de **desmultiplexagem**, que permite obter o cabeçalho de uma trama, e identificar o protocolo único a esta associado.

Existe ainda uma interface de rede virtual designada por ***loopback***, que cada máquina utiliza para comunicar consigo mesma, de forma a efetuar testes de software e de diagnóstico de conectividade. Observe-se a figura 3, em anexo I.

4.2 Experiência 2 - Implementar duas *LAN*s virtuais num *switch*

Nesta experiência implementaram-se duas redes locais virtuais no *switch*, uma englobando as máquinas configuradas na experiência anterior, a vlan20, e a outra contendo o tux22, a vlan21.

O processo de criação e configuração de portas do *switch* a associar a cada uma das *VLANs* encontra-se em anexo II, no registo 1. Para o realizar conectou-se a porta de série de um dos tux's disponíveis a uma das portas deste dispositivo, e invocaram-se os comandos suprareferidos através do **GTKTerm**.

Posto isto, e em concordância com os registos guardados, observa-se a existência de dois domínios distintos de *broadcast*, uma vez que a execução do comando *ping -b*, a partir de um qualquer tux, não origina respostas de todas as máquinas configuradas.

4.3 Experiência 3 - Configurar um router em *Linux*

O objetivo desta experiência esteve associada com a configuração do tux4 de forma a que este se comportasse como um router, estabelecendo uma conexão entre as *VLAN*s criadas anteriormente. Para isso houve necessidade de conectar a interface eth1 dessa mesma máquina à vlan21, sub-rede onde se encontrava já presente o tux2. De forma a concluir este processo adicionaram-se ainda duas rotas, uma ao tux1 e outra ao tux4, indicando os endereços IP para reencaminhamento dos pacotes provenientes de cada uma destas máquinas, ativou-se o reencaminhamento de IPs e desativou-se respostas ICMP a *pings broadcast*. No final já foi possível efetuar comuni-

cação entre os computadores, situação observável, por exemplo, através da execução do comando *ping*. Seguem em anexo I, nas figuras 4 e 5, as tabelas de *forwarding* para o tux21 e para o tux22. Nestas podemos observar, entre outros, parâmetros indicando o destino da rota, **Destination**, o IP do nó da rede por onde passa a rota, **Gateway**, e a interface de rede do tux responsável por essa mesma *gateway*, eth0 ou eth1.

Através do programa Wireshark consegue-se observar a troca de pacotes existentes entre os tux, como os ARP e os ICMP. Estes últimos possibilitam testar e efetuar um controlo da rede, enviando pacotes **Reply**, se o host é alcançável, **Host Unreachable** caso contrário. Em anexo I, na figura 6, existe um registo ilustrando a troca de pacotes ARP e ICMP.

4.4 Experiência 4 - Configurar um router comercial e implementar NAT

O objetivo desta experiência esteve relacionado com a configuração de um router comercial e com a implementação da funcionalidade NAT - *Network Address Translation* ao mesmo.

Para configurar o router comercial adicionaram-se, como segue em anexo II nos scripts individuais, as rotas *default* para cada um dos tux. O tux22 e o tux24 possuem como rota *default* o router comercial configurado na sua rede virtual local, enquanto que o tux21, possui como rota *default* o tux24. A partir deste momento o tux21 é capaz de alcançar qualquer interface da rede, inclusivé a do router. Em anexo I, na figura 7, observa-se um *ping* realizado ao router.

Na iteração seguinte, ao implementar a funcionalidade *NAT*, os endereços privados são convertidos em endereços públicos, recorrendo a tabelas *hash*, e a rede configurada passa a poder comunicar com redes externas. O processo de configuração apresenta-se em anexo I, na figura 8.

Após todo este processo, e para testar a correta configuração das rotas na rede, realizou-se um *ping* ao router a partir do tux21. Observou-se, como seria de esperar, que os pacotes de resposta enviados ao tux21 foram reencaminhados pelo tux24.

4.5 Experiência 5 - DNS

Na penúltima experiência, e para concluir a configuração de acesso à internet, configurou-se o serviço de DNS - *Domain Name System* - em cada uma das máquinas. Este tipo de servidor tem como função traduzir cada *hostname* para o seu respetivo endereço IP externo.

Esta configuração está em todas as máquinas presente no ficheiro '/etc/-resolv.conf', e deve conter informação acerca do servidor DNS a aceder cada vez que se pretende contactar com um *hostname* de IP desconhecido, presente na internet. Para isso executaram-se as dois últimos comandos presentes em cada um dos scripts, presentes em anexo II.

Em anexo I, na figura 9, observa-se a troca de **pacotes DNS** que ocorre

quando se efetua *ping* a 'google.com'. O pacote de pedido transporta parâmetros como o nome do *host* e o tempo de vida do pedido, enquanto que a resposta devolve o endereço IP do *host* requerido e o tempo de vida da resposta.

4.6 Experiência 6 - Ligações TCP

Para concluir, na última experiência procurou-se testar a aplicação desenvolvida sob a rede configurada, verificando-se o esperado comportamento do protocolo TCP. Um exemplo de execução da aplicação encontra-se na figura 1, em anexo I.

Durante a execução do programa observa-se o estabelecimento de duas ligações TCP ao servidor FTP. Enquanto uma destas possui a finalidade de trocar de **mensagens de controlo** entre os intervenientes, a outra permite o envio de **dados** para o cliente, por parte do servidor. Após terminada a sua função, cada uma destas conexões é devidamente encerrada. Em anexo I, nas figuras 10 e 11, observamos registos Wireshark ilustrando o referido anteriormente.

O protocolo TCP funciona de acordo com o mecanismo **Selective Repeat ARQ** - *Automatic Repeat Request*, semelhante ao Go-Back-N ARQ, efetuando no entanto o processamento de frames mesmo quando ocorre uma falha na receção. Neste caso, os frames perdidos identificados pelo recetor através do uso de um *acknowledgement number*, são requeridos ao emissor e por este posteriormente enviados. A gama de pacotes que o recetor pode receber a cada momento depende, por isso, da quantidade de pacotes com falha de receção ainda não repostos pelo emissor - **método da janela deslizante**.

Além disso, durante a transferência de um ficheiro o fluxo de dados depende, a cada momento, do número de ligações TCP que o servidor necessita de servir. Assim sendo, observa-se que quando a rede se encontra mais congestionada, menor é a taxa de transmissão, consequência do servidor distribuir de igual forma a taxa de transferência por cada uma das ligações.

5 Conclusões

A configuração e análise de rede proposta a propósito deste trabalho proporcionou uma maior compreensão dos conceitos abordados de forma teórica nas aulas da unidade curricular.

Posteriormente, com o desenvolvimento da aplicação de *download*, houve ainda a oportunidade de melhor explorar o funcionamento dos protocolos TCP/IP e FTP, beneficiando de todo o conhecimento técnico que daí advém.

De acordo com o guião fornecido consideramos que os objetivos deste trabalho foram alcançados, dando-o, portanto, por bem sucedido.

Referências

[1] Manuel P. Ricardo *Lab 2 - Computer Networks.* 2018.

6 Anexo I - Imagens

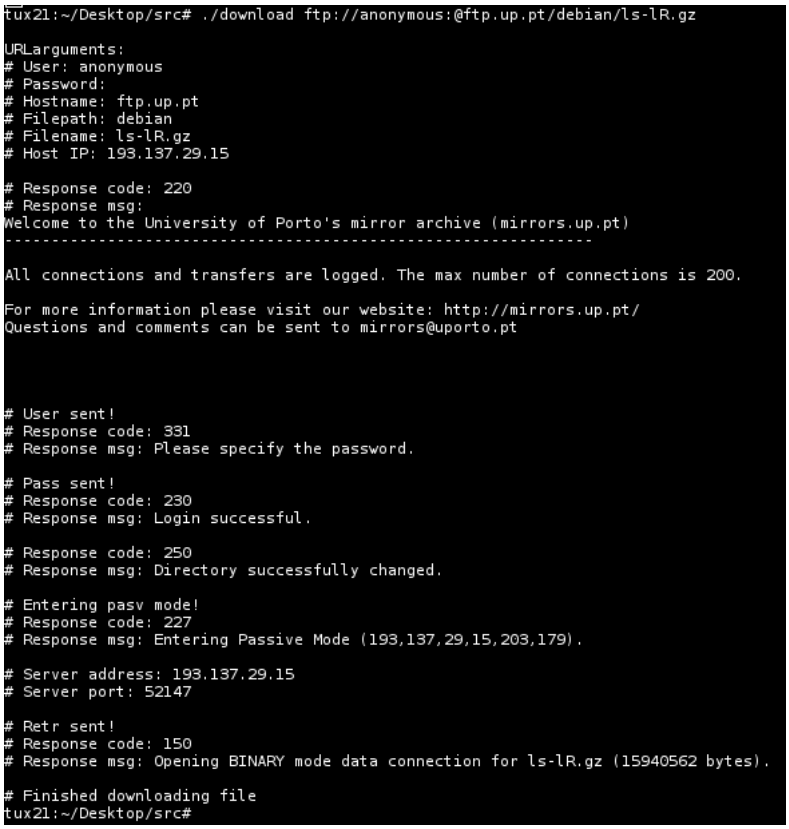

Figura 1: Exemplo de execução da aplicação.

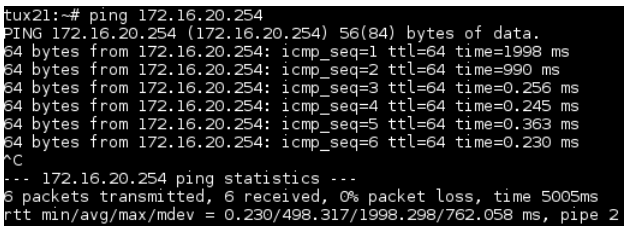

Figura 2: Exemplo de execução do comando ping.

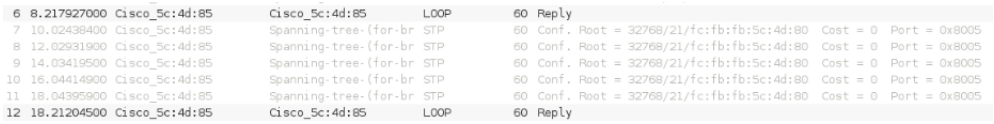

Figura 3: Rotina de *loopback* em execução.

```
tux21:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.20.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
172.16.21.0     172.16.20.254   255.255.255.0   UG    0      0        0 eth0
```

Figura 4: Rotas no tux21.

```
tux22:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.20.0     172.16.21.253   255.255.255.0   UG    0      0        0 eth0
172.16.21.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
```

Figura 5: Rotas no tux22.

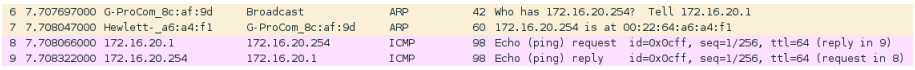

Figura 6: Troca de pacotes ARP e ICMP.

```
tux21:~# ping 172.16.1.254
PING 172.16.1.254 (172.16.1.254) 56(84) bytes of data.
64 bytes from 172.16.1.254: icmp_seq=1 ttl=62 time=1.03 ms
64 bytes from 172.16.1.254: icmp_seq=2 ttl=62 time=0.794 ms
64 bytes from 172.16.1.254: icmp_seq=3 ttl=62 time=0.802 ms
64 bytes from 172.16.1.254: icmp_seq=4 ttl=62 time=0.804 ms
^C
--- 172.16.1.254 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.794/0.857/1.030/0.103 ms
```

Figura 7: Execução de *ping* ao router.

```
conf t
interface gigabitethernet 0/0 *
ip address 172.16.y1.254 255.255.255.0
no shutdown
ip nat inside
exit

interface gigabitethernet 0/1*
ip address 172.16.1.y9 255.255.255.0
no shutdown
ip nat outside
exit

ip nat pool ovrld 172.16.1.y9 172.16.1.y9 prefix 24
ip nat inside source list 1 pool ovrld overload

access-list 1 permit 172.16.y0.0 0.0.0.7
access-list 1 permit 172.16.y1.0 0.0.0.7

ip route 0.0.0.0 0.0.0.0 172.16.1.254
ip route 172.16.y0.0 255.255.255.0 172.16.y1.253
end
```

Figura 8: Comandos para implementação de *NAT* num router *CISCO*.

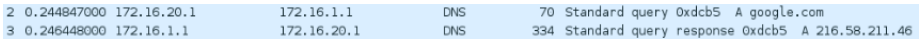

Figura 9: Troca de pacotes DNS.

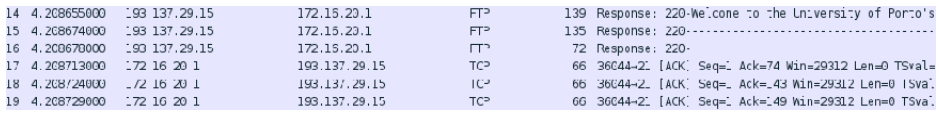

Figura 10: Exemplo de envio de pacotes de controlo.

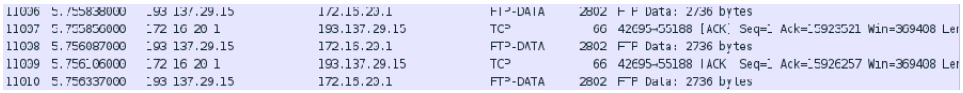

Figura 11: Exemplo de envio de pacotes de dados.

7 Anexo II - *Scripts* e comandos de configuração

Registo 1 - Criação e configuração de uma *VLAN*

```
# configure terminal
# vlan [num vlan]
# end

# configure terminal
# interface fastethernet 0/[num porta]
# switchport mode access
# switchport access vlan [num vlan]
# end
```

tux21.sh

```
#!/bin/bash

ifconfig eth0 up
ifconfig eth0 172.16.20.1
route add -net 172.16.21.0/24 gw 172.16.20.254
route add default gw 172.16.20.254
route -n
echo 1 > /proc/sys/net/ipv4/ip_forward
echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
echo 'search netlab.fe.up.pt' > /etc/resolv.conf
echo 'nameserver 172.16.1.1' >> /etc/resolv.conf
```

tux22.sh

```
#!/bin/bash

ifconfig eth0 up
ifconfig eth0 172.16.21.1
route add -net 172.16.20.0/24 gw 172.16.21.253
route add default gw 172.16.21.254
route -n
echo 1 > /proc/sys/net/ipv4/ip_forward
echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
echo 'search netlab.fe.up.pt' > /etc/resolv.conf
echo 'nameserver 172.16.1.1' >> /etc/resolv.conf
```

tux24.sh

```
#!/bin/bash

ifconfig eth0 up
ifconfig eth0 172.16.20.254
ifconfig eth1 up
ifconfig eth1 172.16.21.253
route add default gw 172.16.21.254
```

```
route -n
echo 1 > /proc/sys/net/ipv4/ip_forward
echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
echo 'search netlab.fe.up.pt' > /etc/resolv.conf
echo 'nameserver 172.16.1.1' >> /etc/resolv.conf
```

8 Anexo III - Código fonte

download.h

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>
#include <netdb.h>
#include <string.h>
#include <ctype.h>

#define h_addr h_addr_list[0]
#define FTP_PORT 21
#define SOCKET_BUF_SIZE 8192

/**
 * Arguments obtained from the URL
 */
typedef struct URLarguments
{
  char user[256];
  char pwd[256];
  char hostname[256];
  char filepath[256];
  char filename[256];
  char hostIp[256];
} URLarguments;

/**
 * Sockets file descriptors, server address and port
 */
typedef struct Sockets
{
  int controlSockFd;
  int dataSockFd;
  char serverAddr[15]; // maximum length for ipv4
  int serverPort;
} Sockets;

/**
 * States for the URL parsing state machine
 */
typedef enum URLstate
{
  INIT,
  FTP,
  USER,
```

```
  PWD,
  HOST,
  PATH
} URLstate;

/**
 * States for the server response state
 */
typedef enum ResponseState
{
  READ_CODE,
  READ_MSG,
  READ_MULTIPLE,
  READ_FINAL
} ResponseState;

/**
 * Server response code and message
 */
typedef struct ServerResponse
{
  char code[4];
  char msg[1024];
} ServerResponse;

/**
 * Parses the URL arguments (USER, PASS, HOST and PATH)
 */
void parseURL(char *url, struct URLarguments *arguments);

/**
 * Parses the file's name from its path
 */
void parseFilename(struct URLarguments *arguments);

/**
 * Gets the host's ip address according to its name
 */
char *getip(char *hostname);

/**
 * Handles the connection to the server through a control socket
 */
int initConnection(Sockets *sockets, char *ip);

/**
 * Performs login in to the ftp server
 */
int login(Sockets *sockets, char *user, char *pwd);

/**
 * Switches to file directory
```

```
 */
int changeDir(Sockets *sockets, char *filepath);

/**
 * Sets passive mode for file tranfer
 */
int enterPasvMode(Sockets *sockets);

/**
 * Handles the connection to the server through a data socket
 */
int openDataConnection(Sockets *sockets);

/**
 * Initializes the file transfer
 */
void transferFile(Sockets *sockets, char *filename);

/**
 * Frees the allocated memory
 */
void freeResources(Sockets *sockets);

/**
 * Handles the creation and the connection to a socket
 */
int openSocket(char *ip, int port);

/**
 * Sends a control command to the server
 */
int sendCmd(Sockets *sockets, char *msg);

/**
 * Retrieves the server response code - read in format [%d%d%d][ -]
 */
void receiveResponse(ServerResponse *response, int sockfd);

/**
 * Calculates server port to connect data socket to
 */
int calculatePort(Sockets *sockets, char *response);

/**
 * Reads data received from the data socket, saving it locally
 */
void saveFile(int fd, char *filename);

/**
 * Outputs the URL parsed arguments to the screen
 */
void printArguments(struct URLarguments *arguments);
```

download.c

```c
#include "download.h"

/*
:TEST FILES:
ftp://anonymous:anonymous@speedtest.tele2.net/1KB.zip
ftp://anonymous:anonymous@ftp.up.pt/parrot/README.html
*/

int main(int argc, char **argv)
{
  Sockets sockets;
  struct URLarguments arguments;

  if (argc != 2)
  {
    fprintf(stderr, "# Usage: ./download ftp://<user>:<password>@<host>/<url-path>\n");
    exit(1);
  }

  /*** URL PROCESSING ***/

  // parse url argument
  parseURL(argv[1], &arguments);

  /*** FTP CONNECTION HANDLING ***/

  // init ftp connection
  initConnection(&sockets, arguments.hostIp);

  // login to the server
  login(&sockets, arguments.user, arguments.pwd);

  // change to file directory
  changeDir(&sockets, arguments.filepath);

  // enter pasv
  enterPasvMode(&sockets);

  // open data socket
  openDataConnection(&sockets);

  // require file for transfer
  transferFile(&sockets, arguments.filename);

  // free the allocated mem
  freeResources(&sockets);

  return 0;
}
```

```
void parseURL(char *url, struct URLarguments *arguments)
{
  // initial state
  URLstate state = INIT;
  const char *ftp = "ftp://";

  int i = 0, j = 0;
  for (; i < strlen(url); i++)
  {
    // state machine for url matching
    switch (state)
    {
    case INIT:
      if (url[i] == ftp[i])
      {
        if (i == 5)
          state = FTP;
        continue;
      }
      else
      {
        printf("# Error parsing ftp://\n");
        exit(0);
      }
      break;
    case FTP:
      if (url[i] == ':')
      {
        state = USER;
        j = 0;
      }
      else
      {
        arguments->user[j++] = url[i];
      }
      break;
    case USER:
      if (url[i] == '@')
      {
        state = PWD;
        j = 0;
      }
      else
      {
        arguments->pwd[j++] = url[i];
      }
      break;
    case PWD:
      if (url[i] == '/')
      {
        state = HOST;
```

```
        j = 0;
      }
      else
      {
        arguments->hostname[j++] = url[i];
      }
      break;
    case HOST:
      arguments->filepath[j++] = url[i];
      break;
    default:
      printf("# Invalid URL state\n");
      exit(0);
    }
  }

  // parse filename from the file's path
  parseFilename(arguments);

  // remove filename in filepath
  char *path, *lastSlash = strrchr(arguments->filepath, '/');
  if (lastSlash != NULL)
  {
    int index = lastSlash - arguments->filepath;

    path = (char *)malloc(index * sizeof(char));
    memcpy(path, arguments->filepath, index);
  }
  else
  {
    // directory is the root
    path = (char *)malloc(sizeof(char));
    *path = '.';
  }
  memset(arguments->filepath,0,sizeof(arguments->filepath));
  memcpy(arguments->filepath,path,strlen(path));
  free(path);

  // get host ip address
  const char *ip = getip(arguments->hostname);
  memset(arguments->hostIp,0,sizeof(arguments->hostIp));
  memcpy(arguments->hostIp,ip,strlen(ip));

  printArguments(arguments);
}

void parseFilename(struct URLarguments *arguments)
{
  int i = 0, j = 0;
  for (; i < strlen(arguments->filepath); i++)
  {
    if (arguments->filepath[i] == '/')
```

```
    {
      // reset filename buffer
      memset(arguments->filename,0,sizeof(arguments->filename));
      j = 0;
    }
    else
    {
      arguments->filename[j++] = arguments->filepath[i];
    }
  }
}

char *getip(char *hostname)
{
  struct hostent *h;

  if ((h = gethostbyname(hostname)) == NULL)
  {
    herror("# gethostbyname");
    exit(1);
  }

  char *ip_addr = inet_ntoa(*((struct in_addr *)h->h_addr));

  return ip_addr;
}

int initConnection(Sockets *sockets, char *ip)
{
  ServerResponse response;

  // open login socket
  sockets->controlSockFd = openSocket(ip, FTP_PORT);

  // wait for server response
  receiveResponse(&response, sockets->controlSockFd);

  // if code is invalid exit
  if (response.code[0] != '2')
  {
    printf("# Error connecting to server!\n");
    exit(0);
  }

  return 0;
}

int login(Sockets *sockets, char *user, char *pwd)
{
  ServerResponse response;

  char *msg = (char *)malloc(strlen(user) + 5);
```

```
  // send user
  sprintf(msg, "user %s\n", user);
  sendCmd(sockets, msg);
  printf("# User sent!\n");

  receiveResponse(&response, sockets->controlSockFd);
  if (response.code[0] != '3')
  {
    printf("# Error receiving user response!\n");
    exit(0);
  }

  // send password
  sprintf(msg, "pass %s\n", pwd);
  sendCmd(sockets, msg);
  printf("# Pass sent!\n");

  receiveResponse(&response, sockets->controlSockFd);
  if (response.code[0] != '2')
  {
    printf("# Error receiving pass response!\n");
    exit(0);
  }

  free(msg);

  return 0;
}

int changeDir(Sockets *sockets, char *filepath)
{
  ServerResponse response;

  char *msg = (char *)malloc(strlen(filepath) + 4);

  sprintf(msg, "cwd %s\n", filepath);
  sendCmd(sockets, msg);

  receiveResponse(&response, sockets->controlSockFd);
  if (response.code[0] != '2')
  {
    printf("# Error receiving cd response!\n");
    exit(0);
  }

  free(msg);

  return 0;
}

int enterPasvMode(Sockets *sockets)
```

```
{
  ServerResponse response;

  sendCmd(sockets, "pasv\n");
  printf("# Entering pasv mode!\n");

  receiveResponse(&response, sockets->controlSockFd);
  if (response.code[0] != '2')
  {
    printf("# Error entering passive mode!\n");
    exit(0);
  }

  // get port to connect data socket
  calculatePort(sockets, response.msg);

  return 0;
}

int openDataConnection(Sockets *sockets)
{
  // open login socket
  sockets->dataSockFd = openSocket(sockets->serverAddr, sockets->serverPort);

  return 0;
}

void transferFile(Sockets *sockets, char *filename)
{
  ServerResponse response;

  char *msg = (char *)malloc(strlen(filename) + 5);

  // send retr
  sprintf(msg, "retr %s\n", filename);
  sendCmd(sockets, msg);
  printf("# Retr sent!\n");

  receiveResponse(&response, sockets->controlSockFd);
  if (response.code[0] != '1')
  {
    printf("# Error opening file!\n");
    exit(0);
  }

  free(msg);

  saveFile(sockets->dataSockFd, filename);
}

void freeResources(Sockets *sockets)
{
```

```
  // close the open sockets
  close(sockets->controlSockFd);
  close(sockets->dataSockFd);
}

int openSocket(char *ip, int port)
{
  // socket file descriptor
  int sockFd;
  // server struct
  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(ip); /* 32 bit Internet address network byte ordered
  server_addr.sin_port = htons(port);          /* server TCP port must be network byte ordered

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

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

  return sockFd;
}

int sendCmd(Sockets *sockets, char *msg)
{
  if (write(sockets->controlSockFd, msg, strlen(msg)) < 0)
  {
    perror("# Error sending control message\n");
    return 1;
  }

  return 0;
}

void receiveResponse(ServerResponse *response, int sockfd)
{
  char c;
  int i = 0, j = 0;

  memset(response->code,0,sizeof(response->code));
```

```
memset(response->msg,0,sizeof(response->msg));

ResponseState state = READ_CODE;

int multipleLineMsg = 0;

while (state != READ_FINAL)
{
  read(sockfd, &c, 1);

  switch (state)
  {
  case READ_CODE:
    // server 3-digit response code
    if (c == ' ')
    {
      if (i == 3)
      {
        state = READ_MSG;
        i = 0;
      }
      else
      {
        printf("# Error receiving server response code!\n");
        exit(0);
      }
    }
    else if (c == '-')
    {
      // multiple line message
      state = READ_MULTIPLE;
      multipleLineMsg = 1;
    }
    else if (isdigit(c))
    {
      response->code[i++] = c;
    }
    break;
  case READ_MSG:
    // reads message
    if (c == '\n')
      state = READ_FINAL;
    else
    {
      response->msg[j++] = c;
    }
    break;
  case READ_MULTIPLE:
    if (c == '\n')
    {
      state = READ_CODE;
      i = 0;
```

```
      }
      response->msg[j++] = c;
      break;
    default:
      fprintf(stderr, "# Invalid response state");
      break;
    }
  }

  printf("# Response code: %s\n", response->code);

  if (multipleLineMsg)
    printf("# Response msg: \n%s\n\n", response->msg);
  else
    printf("# Response msg: %s\n\n", response->msg);
}

int calculatePort(Sockets *sockets, char *response)
{
  int ipPart1, ipPart2, ipPart3, ipPart4;
  int port1, port2;

  if ((sscanf(response, "Entering Passive Mode (%d,%d,%d,%d,%d,%d)",
              &ipPart1, &ipPart2, &ipPart3, &ipPart4, &port1, &port2)) < 0)
  {
    printf("# Error retrieving pasv mode information!\n");
    return 1;
  }

  if ((sprintf(sockets->serverAddr, "%d.%d.%d.%d", ipPart1, ipPart2, ipPart3, ipPart4)) < 0)
  {
    printf("# Error forming server's ip address!\n");
    return 1;
  }

  sockets->serverPort = port1 * 256 + port2;

  printf("# Server address: %s\n", sockets->serverAddr);
  printf("# Server port: %d\n\n", sockets->serverPort);

  return 0;
}

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

  char bufSocket[SOCKET_BUF_SIZE];

  int bytes;
  while ((bytes = read(fd, bufSocket, SOCKET_BUF_SIZE)) > 0)
    bytes = fwrite(bufSocket, bytes, 1, file);
```

```
    fclose(file);

    printf("# Finished downloading file\n");
}

void printArguments(struct URLarguments *arguments)
{
    printf("\nURLarguments:\n");
    printf("# User: %s\n", arguments->user);
    printf("# Password: %s\n", arguments->pwd);
    printf("# Hostname: %s\n", arguments->hostname);
    printf("# Filepath: %s\n", arguments->filepath);
    printf("# Filename: %s\n", arguments->filename);
    printf("# Host IP: %s\n\n", arguments->hostIp);
}
```