Configuração e Estudo de uma Rede de Computadores

2º Trabalho Laboratorial

Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

3MIEIC07:
Diogo Dores - up201504614@fe.up.pt
Gaspar Pinheiro - up201704700@fe.up.pt
Gonçalo Marantes - up201706917@fe.up.pt

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

23 de Dezembro de 2019

Resumo

Este relatório foi elaborado no âmbito da unidade curricular de Redes de Computadores. Trata-se de uma complementação ao segundo trabalho laboratorial, que consiste na configuração e estudo de uma rede de computadores e desenvolvimento de uma aplicação de transferência de ficheiros de acordo com o protocolo FTP (*File Transfer Protocol*). Esta rede consiste em três computadores com o sistema operativo *Linux*, um *Router Cisco* e um *Switch Cisco*.

Posto isto, o trabalho foi concluído com sucesso, uma vez que todos os objectivos foram cumprimos e foi finalizada e testada a aplicação, tendo sido capaz de transferir um ficheiro, levando à conclusão que a rede foi configurada corretamente.

Conteúdo

1 Experiência 1 - Configurar uma rede IP

Questões

1.1 O que são os pacotes ARP e para que são usados?

ARP (*Address Resolution Protocol*) é um protocolo da camada de rede utilizado para converter um endereço IP num endereço físico, chamado endereço DLC (*Data Link Control*). Por sua vez, o endereço DLC é respetivo à camada de ligação de dados, imediatamente abaixo da camada de rede. Para redes que obedecem ao standard IEEE 802, como é o caso da Ethernet, o endereço DLC é normalmente chamado de endereço MAC (*Media Access Control*). Já o endereço MAC é um endereço único de *hardware* que identifica um nó numa rede.

Os pacotes ARP são utilizados quando um host deseja obter um endereço físico (endereço MAC) de outro host tendo apenas o seu endereço IP, fazendo *broadcast* de um pacote para uma rede TCP/IP. Já o host nessa rede que tenha o endereço IP respetivo envia uma resposta com o seu endereço físico.

1.2 O que são os endereços MAC e IP de pacotes ARP e porquê?

Os pacotes ARP contém o endereço MAC e IP tanto do transmissor como do recetor, embora que no caso do recetor o endereço MAC seja 0 no caso em que o transmissor queira descobrir o endereço MAC do recetor. A resposta do recetor será um pacote ARP semelhante com o seu endereço MAC.

1.3 Que pacotes gera o comando *ping*?

O comando ping gera inicialmente pacotes ARP caso o endereço físico do recetor não esteja na tabela ARP. Após ter conhecimento do seu endereço físico, este comando gera pacotes do tipo ICMP (*Internet Control Message Protocol*). Este protocolo é usado por dispositivos de rede para gerar mensagens de erro quando existe problemas de rede que não permitem a transferência de pacotes IP.

1.4 O que são os endereços MAC e IP dos pacotes de ping?

Os pacotes ICMP contém o endereço MAC e IP tanto do transmissor como do recetor, este são usados para efeitos de encaminhamento.

1.5 Como determinar se uma trama de *Ethernet* recebida é ARP, IP ou ICMP?

Recorrendo ao *Ethernet header* de um pacote conseguimos determinar o tipo de trama. Caso o *EtherType* (composto por 2 octetos) tenha o valor 0x0800 esta trama é do tipo IP. Sabendo que é do tipo IP, analisando o IP *header* conseguimos concluir que caso tenho o seu valor a 1, o tipo de protocolo é ICMP. Por outro lado se o valor do *EtherType* for 0x0806 a trama já é do tipo ARP.

1.6 Como determinar o tamanho de uma trama recebida?

Recorrendo ao *Ethernet header* de um pacote, terceiro e quarto octeto contém a informação relativa ao tamanho total da trama.

1.7 O que é a interface *loopback* e porque razão é importante?

Loopback refere-se encaminhamento de sinais digitais ou fluxos de dados para a fonte que os enviou inicialmente, sem qualquer tipo de processamento ou modificação. Isto é principalmente usado para testar a infraestrutura de comunicação.

Neste caso, a interface loopback é uma interface virtual da rede que permite ao computador receber respostas de si mesmo. A utilidade desta interface é testar se a carta da rede está configurada corretamente.

2 Experiência 2 - Implementar duas LANs virtuais num switch

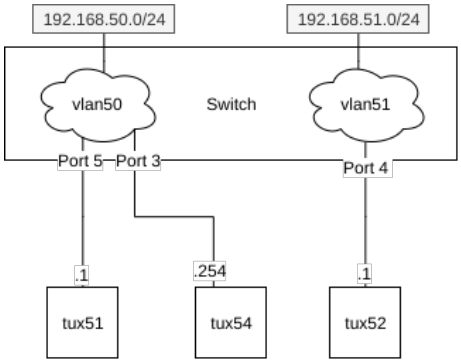

Figura 1: Esquema da rede no final da experiência 2

Nesta experiência criaram-se duas *LAN*'s virtuais (*vlan50* e *vlan51*), o *tux1* e *tux4* foram associados à *vlan50* enquanto o *tux2* foi associado à *vlan51*, como é possivel verificar na imagem.

Questões

2.1 Como configurar *vlany0*?

Na régua 1 da bancada, a porta T4 tem que estar ligada à porta *switch* console da régua 2. A porta T3, da régua 1, vai estar ligada à porta S0 do *tux* que se deseja estar ligado à consola do *switch*. Para criar uma *vlan* invocam-se os seguintes comandos usando o *GTKTerm* do tux escolhido.

```
> configure terminal
> vlan y0
> end
```

Depois deverá associar portas do switch à vlan criada.

```
> configure terminal
> interface fastethernet 0/(n mero da porta)
> switchport mode access
> switchport access vlan y0
> end
```

2.2 Quantos domínios de transmissão existem? Como pode concluí-lo a partir dos *logs*?

Existem dois domínios de transmissão, visto que o *tux* 1 recebe resposta do *tux* 4, mas não do *tux* 2, quando faz *ping broadcast*. O *tux* 2 não recebe resposta de ninguém quando faz *ping broadcast*. Assim existem dois domínios de *broadcast*: o que contém o *tux* 1 e *tux* 4, e o que contém o *tux* 2.

3 Experiência 3 - Configurar um Router em Linux

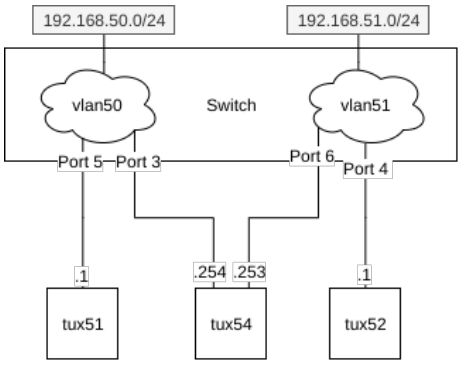

Figura 2: Esquema da rede no final da experiência 3

Nesta experiência foi configurado o *tux* 4 como *router* estabelecendo assim ligação entre as duas *VLAN*'s criadas anteriormente.

Questões

3.1 Que rotas existem nos *tuxes*? Quais os seus significados?

- Rotas do *Tux* 1
  - vlan50 (192.168.50.0) pela *gateway* 192.168.50.1
  - vlan51 (192.168.51.0) pela *gateway* 192.168.50.254
- Rotas do *Tux* 2
  - vlan50 (192.168.50.0) pela *gateway* 192.168.51.253
  - vlan51 (192.168.51.0) pela *gateway* 192.168.51.1

- Rotas do *Tux* 4
  - vlan50 (192.168.50.0) pela *gateway* 192.168.50.254
  - vlan51 (192.168.51.0) pela *gateway* 192.168.51.253

3.2 Que informação contém uma entrada da tabela de encaminhamento?

***Destination*:** o destino da rota.
***Gateway*:** o IP do próximo ponto por onde passará a rota
***Netmask*:** usado para determinar o ID da rede a partir do endereço IP do destino
***Flags*:** dá-nos informação sobre a rota
***Metric*:** o custo de cada rota
***Ref*:** número de referências para esta rota (não usado no *kernel* de *linux*)
***Use*:** contador de pesquisa pela rota, dependendo do uso de -F ou -C isto vai ser o número de falhas da cache (-F) ou o número de sucessos (-C)
***Interface*:** qual a placa de rede responsável pela *gateway* (*eth0*/*eth1*).

3.3 Que mensagens ARP, e endereços MAC associados, são observados e porquê?

Quando um *tux* dá ping a outro, o *tux* recetor que não conhece o MAC Address do que enviou o ping, "pergunta" qual o MAC *Address* do *tux* com aquele IP. E faz isso enviando uma mensagem ARP.

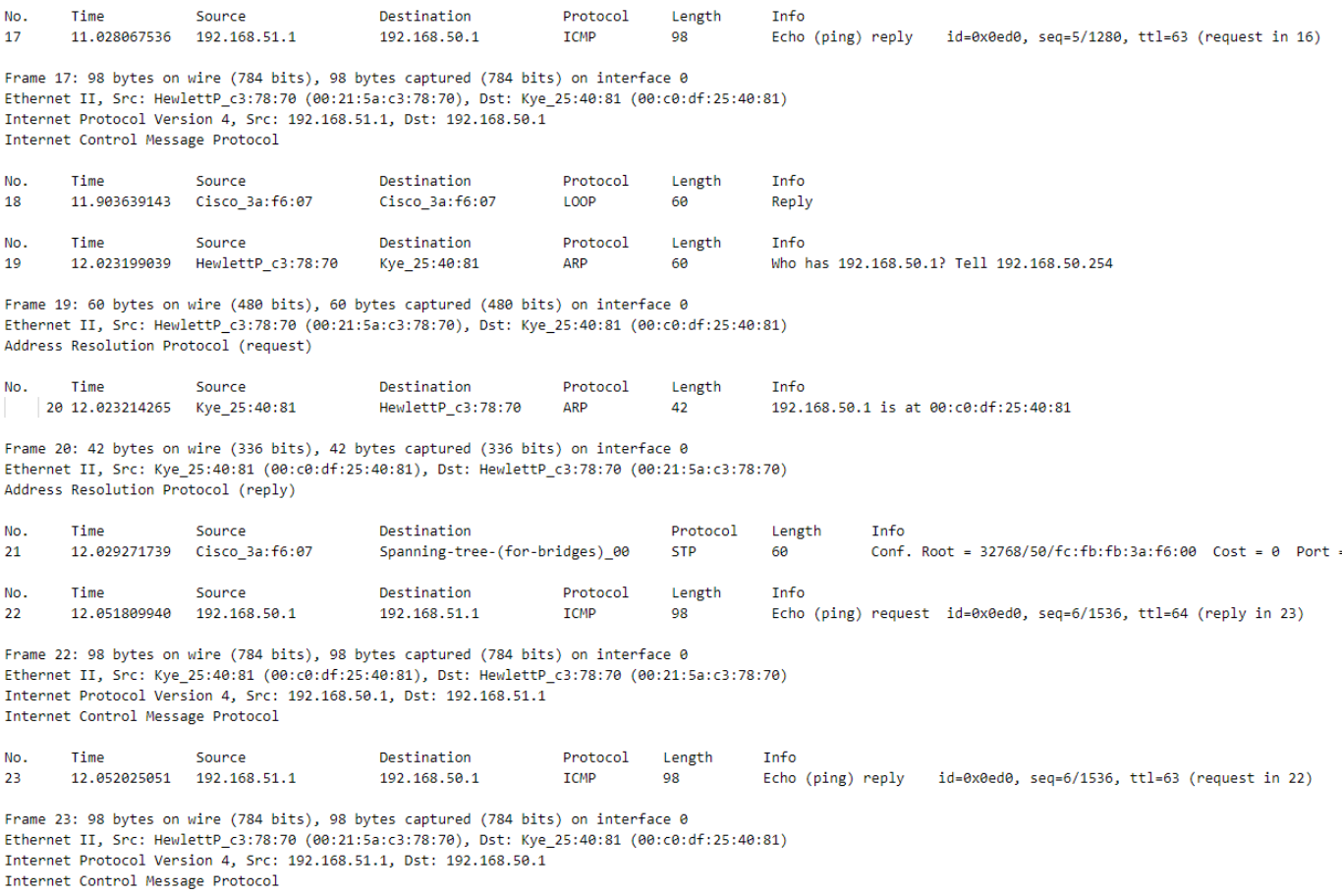

Figura 3: Exemplo de uma mensagem ARP na comunicação do tux1 para o tux2

Essa mensagem vai ter o MAC *Address* do *tux* de origem associado 00:00:00:00:00:00 (mensagem enviada em modo de *broadcast*) pois ainda não sabe qual o *tux* de destino. De seguida, o *tux* de destino responde uma mensagem ARP a dizer o seu MAC *Address*.

3.4 Que pacotes ICMP são observados e porquê?

São observados pacotes ICMP de *request* e *reply*, pois depois de serem adicionar as todas todos os *tux*'s se conseguem "ver" uns aos outros. Se não se conseguissem "ver", seriam enviados os pacotes ICMP de *Host Unreachable*.

3.5 O que são os endereços IP e MAC associados a pacotes ICMP e porquê?

Os endereços IP e MAC associados com os pacotes ICMP são os endereços IP e MAC dos *tux*'s de origem e destino. Por exemplo, quando se faz ping do *tux* 1 para o *tux* 4 os endereços de origem vão ser 192.168.50.1 (IP) e 00:21:5a:c3:78:70 (MAC) e de destino 192.168.51.253 (IP) e 00:c0:df:25:40:81 (MAC).

4 Experiência 4 - Configurar um *Router* comercial e implementar NAT

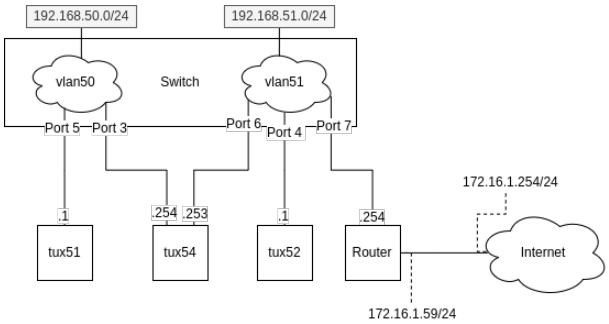

Figura 4: Esquema da rede no final da experiência 4

Questões

4.1 Como configurar uma rota estática num *router* comercial?

Para configurar o *router*, foi necessário ligar a porta T4, da régua 1, à porta do *router*, da régua 2. Relativamente à porta T3, da régua 1, esta vai estar ligada à porta S0 do *tux* que se pretende que esteja ligado ao *router*. No caso desta experiência, foi o *tux* 4.

De seguida invocam-se os seguintes comandos no *GTKTerm* do *Tux* ligado ao *router*.

```
> configure terminal
> ip route 0.0.0.0 0.0.0.0 172.16.1.254
> end
```

Nesta sequência de comandos estamos a dizer que qualquer pacote com destino a 192.168.50.0 com màscara 255.255.255.0, ou seja qualquer pacote com destino à sub-rede 50, deverá passar pela *gateway* 192.168.51.253.

4.2 Quais são os caminhos seguidos pelos pacotes nas experiências realizadas e porquê?

No caso de a rota existir, os pacotes usam essa mesma rota. Caso contrário, os pacotes vão ao *router* (rota *default*), o *router* informa o que o *tux* 4 existe, e deverá ser enviado pelo mesmo.

4.3 Como configurar NAT num *router* comercial?

De forma a configurar o *router*, foi necessário configurar a interface interna no processo de NAT, que foi feito seguindo o guião fornecido para a dada experiência. A partir do *GTKTerm* foram inseridos os seguintes comandos:

```
> configure terminal
```

```
> ip nat pool ovrld 172.16.1.59 172.16.1.59 prefix 24
> ip nat inside source list 1 pool ovrld overload
> access-list 1 permit 192.168.50.0 0.0.0.7
> access-list 1 permit 192.168.51.0 0.0.0.7
> end
```

Além destes comandos também é necessário reconfigurar as rotas ip:

```
> configure terminal
> interface gigabitethernet 0/1
> ip address 172.16.1.59 255.255.255.0
> no shutdown
> ip nat outside
> exit
```

```
> configure terminal
> interface gigabitethernet 0/0
> ip address 192.168.51.254 255.255.255.0
> no shutdown
> ip nat inside
> exit
```

4.4 O que é que faz NAT?

O NAT (Network Address Translation) tem como objetivo a conservação de endereços IP. Assim, permite que as redes IP privadas que usem endereços IP não registados se conectem à Internet ou a uma rede pública. O NAT opera num router, onde conecta duas redes e traduz os endereços privados, na rede interna, para endereços legais, antes que os pacotes sejam encaminhados para outra rede. Adicionalmente, o NAT oferece também funcçºões de segurança e é implementado em ambientes de acesso remoto.

Em suma, permite que os computadores de uma rede interna, como a que foi criada, tenham acesso ao exterior, sendo que, um único endereço IP é exigido para representar um grupo grande de computadores fora da sua própria rede.

5 Experiência 5 - DNS

Nesta experiência foi necessário configurar o DNS (*Domain Name System*) nos tux's 1, 2 e 4. Um servidor de DNS, neste caso, *services.netlab.fe.up.pt*, contém uma base de dados dos endereços IP públicos e dos seus respetivos *hostnames*. É usado para traduzir os *hostnames* para os seus respetivos endereços de IP.

Questões

5.1 Como configurar um serviço DNS num *host*?

De modo a configurar um serviço DNS num *host* é necessário aceder ao ficheiro *resolv.conf* presente na pasta *etc*, em *Linux*. Este ficheiro é responsável pela configuração de name servers de DNS.

De seguida, basta adicionar o comando *search*, seguido do domínio desejado (neste caso - *netlab.fe.up.pt*). Desta forma, o computador irá tentar utilizar o domínio inserido nos *websites* que o utilizador aceder posteriormente. Finalmente, basta utilizar a diretiva *nameserver ¡ip¿* de modo a apontar para o endereço IP das salas de redes (no nosso caso, o endereço utilizado foi 172.16.1.1).

5.2 Que pacotes são trocados pela DNS e que informação é transportada?

Numa primeira fase o *host* (*tux*) envia um pacote para o *server* que contém o *hostname* desejado, neste caso *google.com*, pedindo o seu endereço IP. De seguida o servidor responde com um pacote que contém o endereço IP do *hostname*.

No.	Time	Source	Destination	Protocol	Length	Info
4	6.014613150	Cisco_3a:f6:07	Spanning-tree-(for-…	STP	60	Conf. Root = 32768/50/fc:fb:fb:3a:f6:00 Cost = 0 Port = 0x8007
5	6.128919432	Cisco_3a:f6:07	Cisco_3a:f6:07	LOOP	60	Reply
6	8.019528669	Cisco_3a:f6:07	Spanning-tree-(for-…	STP	60	Conf. Root = 32768/50/fc:fb:fb:3a:f6:00 Cost = 0 Port = 0x8007
7	8.721426173	192.168.50.1	172.16.1.1	DNS	74	Standard query 0x4fe0 A www.google.com
8	8.721438116	192.168.50.1	172.16.1.1	DNS	74	Standard query 0xc6e9 AAAA www.google.com
9	8.722921559	172.16.1.1	192.168.50.1	DNS	338	Standard query response 0x4fe0 A www.google.com A 172.217.17.4 NS ns4.google.co
10	8.722988468	172.16.1.1	192.168.50.1	DNS	350	Standard query response 0xc6e9 AAAA www.google.com AAAA 2a00:1450:4003:802::200
11	8.723370503	192.168.50.1	172.217.17.4	ICMP	98	Echo (ping) request id=0x52d0, seq=1/256, ttl=64 (reply in 12)
12	8.738589555	172.217.17.4	192.168.50.1	ICMP	98	Echo (ping) reply id=0x52d0, seq=1/256, ttl=49 (request in 11)
13	8.738766395	192.168.50.1	172.16.1.1	DNS	85	Standard query 0x70db PTR 4.17.217.172.in-addr.arpa
14	8.740098141	172.16.1.1	192.168.50.1	DNS	381	Standard query response 0x70db PTR 4.17.217.172.in-addr.arpa PTR mad07s09-in-f4
15	9.725195501	192.168.50.1	172.217.17.4	ICMP	98	Echo (ping) request id=0x52d0, seq=2/512, ttl=64 (reply in 16)
16	9.740191688	172.217.17.4	192.168.50.1	ICMP	98	Echo (ping) reply id=0x52d0, seq=2/512, ttl=49 (request in 15)
17	10.024470251	Cisco_3a:f6:07	Spanning-tree-(for-…	STP	60	Conf. Root = 32768/50/fc:fb:fb:3a:f6:00 Cost = 0 Port = 0x8007
18	10.726310629	192.168.50.1	172.217.17.4	ICMP	98	Echo (ping) request id=0x52d0, seq=3/768, ttl=64 (reply in 19)

Figura 5: Exemplo de mensagens DNS oriundas do tux1

6 Experiência 6 - Conexões TCP

Por fim, nesta experiência foi testado e registado o comportamento do protocolo TCP utilizando para isso a aplicação desenvolvida na primeira parte do trabalho.

Questões

6.1 Quantas conexões TCP são/estão abertas pela sua aplicação FTP?

A aplicação FTP abre 2 conexões TCP, uma para controlo, isto é, para enviar comandos FTP ao servidor e receber as correspondentes respostas e outra para receber os dados enviado pelo servidor e enviar as respostas ao cliente.

6.2 Em que conexão é transportada a informação de controlo FTP?

O controlo de informação é transportado na conexão TCP responsável pela troca de comandos.

6.3 Quais são as fases de uma conexão TCP?

Uma conexão TCP tem três fases: o estabelecimento da conexão, troca de dados e encerramento da conexão.

6.4 Como funciona o mecanismo ARQ TCP? Quais são os campos TCP relevantes? Que informação relevante pode ser observada nos logs?

O TCP (*Transmission Control Protocol*) utiliza o mecanismo ARQ (*Automatic Repeat Request* ou *Automatic Repeat Query*) com o método da janela deslizante. Este mecanismo consiste no controlo de erros de transmissão de dados recorrendo a mensagens de *acknowledgments* e *timeouts* de forma a conseguir uma transmissão segura de dados através de um serviço não seguro. Para tal, utiliza *acknowledgment numbers*, que estão num campo das mensagens enviadas pelo recetor que indicam que a trama foi recebida corretamente, *window size* que indica a gama de pacotes que o emissor pode enviar e o *sequence number* o número de pacotes ser enviado.

6.5 Como funciona o mecanismo de controlo de congestionamento TCP? Quais são os campos relevantes? Como evoluiu o rendimento da conexão de dados ao longo do tempo? Está isto de acordo com o mecanismo de controlo de congestionamento TCP?

O mecanismo de controlo de congestão é feito quando o TCP mantém uma janela de congestão que consiste numa estimativa do número de octetos que a rede consegue encaminhar, não enviando mais octetos do que o mínimo da janela definida pelo recetor e pela janela de congestão.

6.6 O rendimento de uma conexão de dados é perturbado pela aparência de uma segunda conexão TCP? De que forma?

Com o aparecimento de uma segunda conexão TCP, a existência de uma transferência de dados em simultâneo pode levar a uma queda na taxa de transmissão, uma vez que a taxa de transferência é distribuída de igual forma para cada ligação.

7 Conclusões

O segundo trabalho da unidade curricular de Redes de Computadores teve como objetivo a configuração de uma rede de computadores e a implementação do cliente de *download*.

Efetivamente, foram descobertos, consolidados e interiorizados novos conceitos relacionados com funcionalidades que estão constantemente presentes no nosso quotidiano, assim como do protocolo tratado.

Concluindo, o trabalho foi finalizado com sucesso, tendo-se cumprido todos os objetivos, e a sua elaboração contribuiu positivamente para um aprofundamento do conhecimento, tanto teórico como prático, do tema em questão.

A Anexo

Este anexo contem todo o código utilizado para a utilização com sucesso da aplicação FTP.

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "url_handler.h"
#include "ftp_connecter.h"

int main(int argc, char** argv) {

    if(argc != 2) {
        perror("Usage: wrong number of arguments\n");
        exit(1);
    }

    url_t url;
    create_url_struct(&url);

    printf("\n\nPARSING URL...\n\n");
    if (get_url_info(&url, argv[1])) {
        perror("Error in URL syntax!\n");
        exit(1);
    }

    printf("\n\nGETTING IP ADDRESS BY HOST NAME...\n");
    if (get_ip_address(&url)) {
        perror("Getting IP address by host name\n");
        exit(1);
    }

    printf("\nURL INFO:\n\n");
    print_url(&url);

    ftp_t ftp;
    if (ftp_init_connection(&ftp, url.ip_address, url.port)) {
        perror("ftp_init_connection()\n");
        exit(1);
    }
    printf("Started ftp connection successfully\n");

    if (ftp_login(&ftp, url.user, url.password)) {
        perror("ftp_login()");
        exit(1);
    }
    printf("Loged in successfully\n");

    if (ftp_passive_mode(&ftp)) {
        perror("ftp_passive_mode()\n");
        exit(1);
    }
    printf("Entered passive mode successfully\n");

```

```
    char filepath[MAX_BUFFER_SIZE];
    sprintf(filepath, "%s%s", url.url_path, url.filename);
    if (ftp_retr_file(&ftp, filepath)) {
        perror("ftp_retr_file()");
        exit(1);
    }
    printf("Command retr 'file' executed successfully\n");

    if (ftp_download_file(&ftp, url.filename)) {
        perror("ftp_download_file()\n");
        exit(1);
    }
    printf("Downloaded file successfully\n");

    //disconnect

    return 0;
}
```

ftp_connecter.c

```
#include "ftp_connecter.h"

int ftp_init_connection(ftp_t* ftp, const char* ip_address, const
    int port) {

    int socket_fd;
  char buffer[MAX_BUFFER_SIZE];

  if ((socket_fd = open_socket(ip_address, port)) < 0) {
    perror("open_socket()\n");
    return 1;
  }

  ftp->control_socket_fd = socket_fd;
  ftp->data_socket_fd = 0;

  if (read_from_socket(ftp->control_socket_fd, buffer, sizeof(
    buffer))) {
    perror("ftp_read()");
    return 1;
  }

  return 0;
}

int ftp_command_with_response(const int socket_fd, const char*
    command, char* response) {

  // Buffer used to temporarily store comunication messages
  char buffer[MAX_BUFFER_SIZE];

  // User command
  sprintf(buffer, "%s\r\n", command);
  if (write_to_socket(socket_fd, buffer, strlen(buffer))) {
    perror("write_to_socket()\n");
    return 1;
  }

  // Reading response of user command
  if (read_from_socket(socket_fd, buffer, sizeof(buffer))) {
    perror("read_from_socket()\n");
```

```
39      return 1;
40    }
41
42    memcpy(response, buffer, sizeof(buffer));
43
44    return 0;
45  }
46
47  int ftp_command(const int socket_fd, const char* command) {
48
49    char response[MAX_BUFFER_SIZE];
50    ftp_command_with_response(socket_fd, command, response);
51
52    return 0;
53  }
54
55  int ftp_login(const ftp_t* ftp, const char* user, const char*
      password) {
56
57      char command[MAX_COMMAND_SIZE];
58
59    // User command
60    sprintf(command, "USER %s", user);
61    if (ftp_command(ftp->control_socket_fd, command)) {
62      perror("ftp_command()\n");
63      return 1;
64    }
65
66    // Cleaning command buffer
67    memset(command, 0, sizeof(command));
68
69    // Password command
70    sprintf(command, "PASS %s", password);
71    if (ftp_command(ftp->control_socket_fd, command)) {
72      perror("ftp_command()\n");
73      return 1;
74    }
75
76    return 0;
77  }
78
79  int ftp_passive_mode(ftp_t* ftp) {
80
81    char command[MAX_BUFFER_SIZE];
82    char response[MAX_BUFFER_SIZE];
83
84    sprintf(command, "PASV");
85    if (ftp_command_with_response(ftp->control_socket_fd, command,
      response)) {
86      perror("ftp_command()\n");
87      return 1;
88    }
89
90    char ip_address[MAX_BUFFER_SIZE];
91    int port_num;
92
93    if (ftp_process_pasv_response(response, ip_address, &port_num)) {
94      perror("ftp_process_pasv_response()\n");
95      return 1;
96    }
97
98    printf("IP: %s\n", ip_address);
99    printf("PORT: %d\n", port_num);
100
```

```
101   if ((ftp->data_socket_fd = open_socket(ip_address, port_num)) <
        0) {
102     perror("open_socket()\n");
103     return 1;
104   }
105
106   return 0;
107 }
108
109 int ftp_process_pasv_response(const char* response, char*
        ip_address, int* port_num) {
110
111   int ip1, ip2, ip3, ip4;
112   int port1, port2;
113
114   if (sscanf(response, "227 Entering Passive Mode (%d,%d,%d,%d,%
        d)", &ip1, &ip2, &ip3, &ip4, &port1, &port2) < 0) {
115     perror("sscanf()\n");
116     return 1;
117   }
118
119   // Creating ip address
120   sprintf(ip_address, "%d.%d.%d.%d", ip1, ip2, ip3, ip4);
121
122   // Calculating new port number
123   *port_num = port1 * 256 + port2;
124
125   return 0;
126 }
127
128 int ftp_retr_file(const ftp_t* ftp, const char* filename) {
129   char command[MAX_BUFFER_SIZE];
130
131   sprintf(command, "RETR %s", filename);
132   if (ftp_command(ftp->control_socket_fd, command)) {
133     perror("ftp_command()\n");
134     return 1;
135   }
136
137   return 0;
138 }
139
140 int ftp_download_file(ftp_t* ftp, const char* filename) {
141
142   char buffer[MAX_BUFFER_SIZE];
143   int file_fd;
144   int bytes_read;
145
146   if((file_fd = open(filename, O_WRONLY | O_CREAT, 0666)) < 0) {
147     perror("open()\n");
148     return 1;
149   }
150
151   while ((bytes_read = read(ftp->data_socket_fd, buffer, sizeof(
        buffer)))) {
152
153     if (bytes_read < 0) {
154       perror("read()\n");
155       return 1;
156     }
157
158     if (write(file_fd, buffer, bytes_read) < 0) {
159       perror("write()\n");
160       return 1;
161     }
```

```
162
163     //sleep(1);
164   }
165
166   close(file_fd);
167   close(ftp->data_socket_fd);
168
169   return 0;
170 }
```

ftp_connecter.h

```
1  #pragma once
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <errno.h>
7  #include <netdb.h>
8  #include <sys/types.h>
9  #include <netinet/in.h>
10 #include <arpa/inet.h>
11 #include <sys/stat.h>
12 #include <fcntl.h>
13 #include <unistd.h>
14
15 #include "socket_handler.h"
16
17 typedef struct ftp_t {
18     int control_socket_fd;
19     int data_socket_fd;
20 } ftp_t;
21
22 #define MAX_BUFFER_SIZE 1024
23 #define MAX_COMMAND_SIZE 50
24
25 int ftp_init_connection(ftp_t* ftp, const char* ip_addres, const
      int port);
26 int ftp_command(const int socket_fd, const char* command);
27 int ftp_command_with_response(const int socket_fd, const char*
      command, char* response);
28 int ftp_login(const ftp_t* ftp, const char* user, const char*
      password);
29 int ftp_passive_mode(ftp_t* ftp);
30 int ftp_retr_file(const ftp_t* ftp, const char* filename);
31 int ftp_download_file(ftp_t* ftp, const char* filename);
32
33 // Auxiliay functions
34 int ftp_process_pasv_response(const char* response, char*
      ip_address, int* port_num);
```

socket_handler.c

```
1  #include "socket_handler.h"
2
3  int open_socket(const char* ip_address, const int port) {
4      int socket_fd;
5    struct sockaddr_in server_addr;
6
```

```
  // 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_address); /*32 bit
    Internet address network byte ordered*/
  server_addr.sin_port = htons(port); /*server TCP port must be
    network byte ordered */

  // open a TCP socket
  if ((socket_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket()\n");
    return -1;
  }

  // connect to the server
  if (connect(socket_fd, (struct sockaddr *) &server_addr, sizeof(
    server_addr)) < 0) {
    perror("connect()\n");
    return -1;
  }

  return socket_fd;
}

int write_to_socket(const int socket_fd, const char* str, const
    size_t str_size) {
    int bytes;

    // Write a string to the server
  if ((bytes = write(socket_fd, str, str_size)) <= 0) {
    perror("Write to socket\n");
    return 1;
  }

  printf("Bytes written to server: %d\nInfo: %s\n", bytes, str);

  return 0;
}

int read_from_socket(const int socket_fd, char* str, size_t
    str_size) {
    FILE* fp = fdopen(socket_fd, "r");

  do {
    memset(str, 0, str_size);
    str = fgets(str, str_size, fp);
    printf("%s", str);

  } while (!('1' <= str[0] && str[0] <= '5') || str[3] != ' ');

  return 0;
}
```

socket_handler.h

```
#pragma once

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <netdb.h>
#include <strings.h>
#include <string.h>

int open_socket(const char* ip_address, const int port);
int write_to_socket(const int socket_fd, const char* str, const
    size_t str_size);
int read_from_socket(const int socket_fd, char* str, size_t
    str_size);
```

url_handler.c

```c
#include "url_handler.h"

void create_url_struct(url_t* url) {
    memset(url->user, 0, MAX_STRING_SIZE);
  memset(url->password, 0, MAX_STRING_SIZE);
  memset(url->host_name, 0, MAX_STRING_SIZE);
    memset(url->ip_address, 0, MAX_STRING_SIZE);
  memset(url->url_path, 0, MAX_STRING_SIZE);
  memset(url->filename, 0, MAX_STRING_SIZE);
  url->port = 21;
}

int get_url_info(url_t* url, const char* str) {

    // str = ftp://[<user>:<password>@]<host>/<url-path>
    if (!check_ftp(str))
        return 1;

    char* temp_url = (char*) malloc(strlen(str));
    char* url_path = (char*) malloc(strlen(str));
  memcpy(temp_url, str, strlen(str));

    // removing ftp:// from string
  strcpy(temp_url, temp_url + 6); // temp_url = [<user>:<password>@
    ]<host>/<url-path>

    char* url_rest = strchr(temp_url, '@');

    if (url_rest == NULL) {
        printf("User not defined\n");

        strcpy(url_path, temp_url);  // url_path = <host>/<url-path
    >

        strcpy(url->user, "anonymous");
        strcpy(url->password, "any");
    }
    else {
        printf("User defined\n");

        strcpy(url_path, url_rest + 1); // url_path = <host>/<url-
    path>

        get_username(temp_url, url->user);
        printf("Username obtained: %s\n", url->user);

        strcpy(temp_url, temp_url + strlen(url->user) + 1); //
    temp_url = <password>@<host>/<url-path>
```

```
45
46          get_password(temp_url, url->password);
47          printf("Password obtained: %s\n", url->password);
48      }
49
50      get_host_name(url_path, url->host_name);
51      printf("Host name obtained: %s\n", url->host_name);
52
53      strcpy(url_path, url_path + strlen(url->host_name) + 1); //
        url_rest = /<url-path>
54
55      get_url_path(url_path, url->url_path, url->filename);
56      printf("URL path obtained: %s\n", url->url_path);
57
58      return 0;
59  }
60
61  int get_ip_address(url_t* url) {
62      struct hostent* h;
63
64      if ((h = gethostbyname(url->host_name)) == NULL) {
65          perror("gethostbyname()\n");
66          return 1;
67      }
68
69      char* ip = inet_ntoa(*( (struct in_addr *) h->h_addr) );
70    strcpy(url->ip_address, ip);
71
72    return 0;
73  }
74
75  int check_ftp(const char* str) {
76
77      char* ftp_str = "ftp://";
78      char substring[7];
79
80      memcpy(substring, str, 6);
81
82      return !strcmp(ftp_str, substring);
83  }
84
85  int get_username(const char* str, char* username) {
86      strcpy(username, get_str_before_char(str, ':'));
87
88      return 0;
89  }
90
91  int get_password(const char* str, char* password) {
92      strcpy(password, get_str_before_char(str, '@'));
93
94      return 0;
95  }
96
97  int get_host_name(const char* str, char* host_name) {
98      strcpy(host_name, get_str_before_char(str, '/'));
99
100     return 0;
101 }
102
103 int get_url_path(const char* str, char* url_path, char* filename) {
104
105     char* path = (char*) malloc(strlen(str));
106
107     char* working_str = (char*) malloc(strlen(str));
108     memcpy(working_str, str, strlen(str));
```

```
    char* temp_str = (char*) malloc(strlen(str));

  while (strchr(working_str, '/')) {

        temp_str = get_str_before_char(working_str, '/');
        strcpy(working_str, working_str + strlen(temp_str) + 1);

    strcat(path, temp_str);
    strcat(path, "/");
  }

  strcpy(url_path, path);
  strcpy(filename, working_str);

  free(path);
    free(temp_str);

    return 0;
}

char* get_str_before_char(const char* str, const char chr) {

  char* temp = (char*) malloc(strlen(str));
  int index = strlen(str) - strlen(strcpy(temp, strchr(str, chr)));

  temp[index] = '\0';
  strncpy(temp, str, index);

  return temp;
}

void print_url(url_t* url) {
    printf("USER: %s\n", url->user);
    printf("PASSWORD: %s\n", url->password);
    printf("HOST: %s\n", url->host_name);
    printf("IP: %s\n", url->ip_address);
    printf("PATH: %s\n", url->url_path);
    printf("FILE: %s\n\n\n", url->filename);
}
```

url_handler.h

```
#pragma once

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

#define MAX_STRING_SIZE 256
#define h_addr h_addr_list[0] //The first address in h_addr_list.

typedef struct url_t {
  char user[MAX_STRING_SIZE];
  char password[MAX_STRING_SIZE];
  char host_name[MAX_STRING_SIZE];
  char ip_address[MAX_STRING_SIZE];
```

```
  char url_path[MAX_STRING_SIZE];
  char filename[MAX_STRING_SIZE];
  int port;
} url_t;

void create_url_struct(url_t* url);
int get_url_info(url_t* url, const char* str);
int get_ip_address(url_t* url);

// Auxiliary functions
int check_ftp(const char* str);
int check_username(const char* str);
int get_username(const char* str, char* username);
int get_password(const char* str, char* password);
int get_host_name(const char* url_rest, char* host_name);
int get_url_path(const char* str, char* url_path, char* filename);
char* get_str_before_char(const char* str, const char chr);
void print_url(url_t* url);
```