



Redes de Computadores - Trabalho 2

Ligação de Dados

Professores:

Manuel Pereira Ricardo (Regente)

Rui Pedro de Magalhães Claro Prior (Regente)

Rui Lopes Campos (Práticas)

Estudantes:

Dinis Ribeiro dos Santos Bessa de Sousa

up202006303

Francisca Oliveira e Silva

up202005140

Resumo

Trabalho realizado no âmbito da unidade curricular Redes de Computadores, presente no plano curricular da Licenciatura em Engenharia Informática e Computação (L.EIC) no 1º semestre do ano letivo 2022/2023.

O objetivo deste trabalho consiste no desenvolvimento de experiências ao longo da segunda metade do semestre, para melhor entender o funcionamento de uma rede de computadores e diversos protocolos de comunicação.

O trabalho divide-se em duas partes, a primeira o desenvolvimento de uma aplicação de *download* através de um protocolo FTP e a segunda a configuração e estudo de uma rede de computadores.

Índice

A. Siglas e acrónimos	3
1. Introdução	3
2. Parte 1 - Aplicação de download	4
2.1 Arquitetura da Aplicação	4
2.1.1 Processamento do URL	5
2.1.1 Comunicação FTP	6
2.2 Funcionamento da aplicação	7
3. Parte 2 - Configuração e estudo de uma rede de computadores	8
3.1 Experiência 1 - Configuração de um IP de rede	8
3.2 Experiência 2 - Implementação de duas bridges num switch	9
3.3 Experiência 3 - Configuração de um router em Linux	10
3.4 Experiência 4 - Configuração de um router comercial e implementação NAT	13
3.5 Experiência 5 - DNS	15
3.6 Experiência 6 - TCP connections	16
4. Conclusão	19
5. Referências	19
6. Anexos	20
6.1 Fotos e Logs	20
6.2 Script para preparação	20
6.3 Código fonte parte 1	20
main.c	20
FTP.h	22
URL.h	23
FTP.c	23
URL.c	31

A. Siglas e acrónimos

API - Application Programming Interface

TCP - Transmission Control Protocol

IP - Internet Protocol

DNS - Domain Name System

mDNS - Multicast DNS

NAT - Network Address Translation

FTP - File Transfer Protocol

LAN - Local Area Network

URL - Uniform Resource Locator

RFC - Request for Comments

ARP - Address Resolution Protocol

ARQ - Automatic repeat request/automatic repeat query

MAC - Media Access Control

ICMP - Internet Control Message Protocol

ACK - Acknowledgment

MNDP- MikroTik Neighbor Discovery Protocol

CDP - Cisco Discovery Protocol

1. Introdução

O objetivo deste trabalho é por em prática os conhecimentos adquiridos ao longo da segunda metade do semestre de uma forma prática, que culminam na criação de uma aplicação de download de ficheiros usando protocolo FTP e a implementação de uma rede de computadores, implementação essa que é realizada ao longo de 6 experiências diferentes que não só servem como passo para a implementação da rede mas também de aprendizagem sobre a forma como a comunicação dentro da rede funciona, com perguntas que nos desafiaram a pensar com mais profundidade sobre o porquê de estarmos a tomar os passos que tomamos enquanto construímos essa rede.

Este relatório encontra-se dividido em:

- Introdução, onde são descritos os objetivos do trabalho.
- Parte 1 - Aplicação de download, onde são descritos os protocolos TCP/IP por

alto, com especial atenção ao protocolo FTP (aquele que foi implementado), para além de uma discussão sobre a elaboração do trabalho e a arquitetura usada no programa.

- Parte 2 - Configuração e estudo de uma rede de computadores, onde são descritos os passos que tomamos ao longo das 6 experiências, bem como aquilo que aprendemos/pusemos em prática na elaboração de cada experiência e as conclusões que retiramos de cada experiência.
- Conclusão, com uma conclusão geral sobre a elaboração do trabalho.
- Anexos, onde se encontra o código da aplicação e os logs gravados.

2. Parte 1 - Aplicação de download

Nesta primeira parte do segundo projeto de Redes de Computadores, tínhamos como objetivo o desenvolvimento de uma aplicação de download de ficheiros, a desenvolver na linguagem de programação C.

A aplicação tem por base um modelo de ligação cliente-servidor, em que a máquina que corre a aplicação (cliente) irá pedir a uma outra máquina (servidor) um ficheiro.

Esta aplicação usa o protocolo de comunicação FTP, que é um protocolo usado para a transferência de arquivos entre computadores numa rede TCP/IP.

```
dinis@DESKTOP-RTCT13E:/mnt/c/Users/User/Documents/FEUP/3ano/1Semestre/RCOM/proj2$ gcc -Wall -o main main.c src/FTP.c src/URL.c
dinis@DESKTOP-RTCT13E:/mnt/c/Users/User/Documents/FEUP/3ano/1Semestre/RCOM/proj2$ ./main
Usage: ./main ftp://[<user>:<password>@]host[:port]/path/to/filename
dinis@DESKTOP-RTCT13E:/mnt/c/Users/User/Documents/FEUP/3ano/1Semestre/RCOM/proj2$ ./main ftp://ftp.up.pt/pub/gnu/GNUinfo/Audio/index.txt
```

Imagem 1 - Exemplo de input na aplicação

2.1 Arquitetura da Aplicação

Na implementação da aplicação dividimos a mesma em duas camadas: uma responsável pelo processamento do URL dado pelo user e outra responsável por estabelecer ligação com o servidor, enviar comandos FTP e transferir o ficheiro. Estas camadas servem como APIs para ser usada na função principal, em main.c

2.1.1 Processamento do URL

Código correspondente à camada presente no ficheiro URL.C.

Como pudemos ver na Imagem 1, acima, de forma a iniciar a transferência do ficheiro o utilizador tem de fornecer ao programa um URL, que contem as seguintes informações: utilizador (opcional), password (opcional), host, *path*/caminho para o ficheiro e nome do ficheiro.

Para processar o URL que é dado por input pelo utilizador, utilizamos expressões regulares, tendo por base na criação da expressão regular usado o documento RFC1738 e os exemplos dados, para teste.

Após algum aprimorar chegamos à seguinte expressão:

```
"^ftp:/( ([a-zA-Z0-9]+) : ([a-zA-Z0-9]+) @ )? ([a-zA-Z0-9$_ .+!*' () , -]+) ( : ([0-9]+) )? ([a-zA-Z0-9 / () _ , . - ]*?) / ([a-zA-Z0-9 () _ , . - ]+)$"
```

Como se pode verificar a presença de utilizador e password são opcionais, o que permite o uso de utilizadores anónimos/não autenticados.

Quando um URL dá match com a expressão, as partes do URL dão também match com os diferentes grupos da expressão regular, sendo que cada grupo representa uma parte diferente do URL.

Para guardar a informação sobre um URL, é usada a seguinte estrutura de dados:

```
typedef char url_content[MAX_STRING_LENGTH];

typedef struct URL {
    url_content user;
    url_content password;
    url_content host;
    url_content ip;
    url_content path;
    url_content filename;
    int port;
} url;
```

Imagem 2 - Estrutura de dados responsável por guardar a informação do URL

Esta estrutura, quando inicializada, utiliza a porta 21 por defeito, pois essa é a porta por onde é feita a comunicação através do protocolo FTP.

A camada de processamento do URL é ainda responsável por traduzir o *host* para um endereço IP. Este procedimento acontece na função `getIPByHostName(url *u)`. Esta função, faz uso da função `gethostbyname` que retorna uma estrutura do tipo *hostnet* que contem o endereço de IP que corresponde ao *host* dado.

2.1.1 Comunicação FTP

Código correspondente à camada presente no ficheiro **FTP.C**.

Esta camada é responsável pela abertura da comunicação com o servidor, através do uso de *sockets*, a troca de várias mensagens e comandos FTP e a transferência do ficheiro propriamente dito.

Para guardar as *sockets* usadas na comunicação FTP é usada a seguinte estrutura de dados:

```
typedef struct FTP
{
    int control_socket_fd; // file descriptor to control socket
    int data_socket_fd; // file descriptor to data socket
} ftp;
```

Imagem 3 - Estrutura de dados responsável por guardar os *file descriptors* dos *sockets* usados

Esta camada contem diversas funções entre elas:

- Função que realiza a abertura de uma conexão FTP - `ftpConnect()` - com um dado endereço *host* (dado o IP), criando uma *socket* que permite a comunicação entre cliente e servidor e função que fecha a ligação - `ftpClose()`.
- Funções que através de comandos FTP efetuam diversas funções como o login de um dado *user* com a respetiva *password* - `ftpLogin()`, passar a comunicação para o modo passivo (abertura da ligação TCP para os dados e transferência do ficheiro fica a cargo do cliente) - `ftpPassv()`, mudar de diretório - `ftpCWD()`, mostrar diretório atual - `ftpPWD()`, colocar o ficheiro disponível na *socket* de transferência - `ftpRetr()`.

- Quando é pedido ao servidor para passar a comunicação para o modo passivo, este devolve como resposta um tuplo com seis números: os primeiros quatro representam o IP da máquina onde vai estar disponível o ficheiro, os últimos dois a porta na qual irá estar disponível o ficheiro para transferência. Para calcular a porta, é apenas fazer o seguinte cálculo: $5^{\circ}\text{número} * 256 + 6^{\circ}\text{número}$.
- Função que realiza o download do ficheiro, após ele estar disponível na porta de transferência, e que o lê em *chunks* e guarda no cliente num ficheiro - *ftpDownload()*.

2.2 Funcionamento da aplicação

A aplicação está implementada no ficheiro *main.c*, e usa as funções anteriormente descritas.

Aquando da chamada da aplicação com um número de argumentos diferente de um, o user verá uma mensagem de erro, como já vimos na Imagem 1.

Validado o número de argumentos, é inicializada a estrutura que guarda o URL, é confirmada a validade do URL com recurso a expressões regulares e os diferentes grupos do URL guardados na estrutura. É guardado ainda o IP do *host*.

Após isso passamos para a parte da conexão FTP. Primeiro inicializa-se a conexão FTP, em que se cria uma *socket* que vai servir para a comunicação que tem que ver com envio de comando FTP e receção de códigos de resposta (*control socket*).

Depois, utilizando a *socket* criada, é feito o *login* do utilizador usando comandos FTP, primeiro com o utilizador e depois especificada a *password*.

Se o *login* estiver correto, é chamada a função que permite a passagem da comunicação para o modo passivo e é criada a *socket* responsável pela transferência de dados.

Após estar no modo passivo, o diretório de trabalho é alterado para o diretório onde se situa o ficheiro e é usado o comando *RETR* para colocar o ficheiro disponível na porta de transferência de dados.

Assim, apenas resta ao cliente fazer o Download do ficheiro, para a sua máquina, e por fim fechar as ligações criadas.

3. Parte 2 - Configuração e estudo de uma rede de computadores

3.1 Experiência 1 - Configuração de um IP de rede

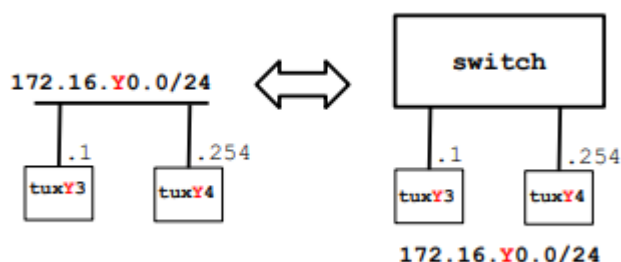


Imagem 4 - Experiência 1 (diagrama retirado do guião)

O objetivo desta experiência foi compreender a configuração de IP's em máquinas diferentes, de uma mesma rede local, de modo a que estas consigam comunicar entre si. Para isso primeiro foram configurados os IPs das portas *eth0* das duas máquinas, com o comando **ifconfig** e depois adicionar rotas de comunicação com o comando **route**.

Depois foi enviado um sinal, através do comando **ping**, de forma a perceber se de facto as duas máquinas estão ligadas e se há comunicação entre elas. Durante o envio dos pacotes, foram capturados os pacotes enviados e recebidos com o *Wireshark*.

No *Wireshark* é possível ver várias informações sobre os pacotes enviados e recebidos, tais como o **tempo** após o início de salvar os logs em que a mensagem foi enviada, o **remetente**, o **destinatário**, o **protocolo**, o **tamanho** do pacote e **informações** várias sobre o pacote.

A comunicação começa com o envio e receção de pacotes ARP, em que é enviado um pacote com o IP da máquina para qual vão ser enviados os pacotes, em *broadcast* para toda a rede, e a resposta é enviada pela máquina correspondente ao IP, com o seu endereço MAC, necessário para o envio dos pacotes.

8	13.751845260	HewlettP_5a:7d:b7	Broadcast	ARP	42 Who has 172.16.30.254? Tell 172.16.30.1
9	13.751966364	HewlettP_5a:74:3e	HewlettP_5a:7d:b7	ARP	60 172.16.30.254 is at 00:21:5a:5a:74:3e

Imagem 5 - Envio e receção de pacotes ARP

Depois é iniciada a comunicação e enviados pacotes ICMP, um protocolo que é parte

integrante do protocolo IP e que transmite informação sobre o estado do envio e receção de pacotes.

10	13.751974466	172.16.30.1	172.16.30.254	ICMP	98 Echo (ping) request	id=0x1414, seq=1/256, ttl=64 (reply in 11)
11	13.752061279	172.16.30.254	172.16.30.1	ICMP	98 Echo (ping) reply	id=0x1414, seq=1/256, ttl=64 (request in 10)

Imagem 6 - Envio e receção de pacotes ICMP

Uma interface de loopback é uma interface de rede virtual lógica de um router Cisco. O computador utiliza esta interface para comunicar com ele próprio, para fazer diferentes ações, tais como: realizar testes de diagnóstico, aceder a servidores da própria máquina, como se fosse um cliente, entre outras. Uma interface de loopback está sempre ativa e a correr, mesmo que outras interfaces físicas no router não estejam disponíveis.

Podemos ainda ver o envio de pacotes MNDP e CDP, que servem para encontrar outras máquinas que comuniquem nestes protocolos dentro de uma mesma rede de *broadcast*.

35	22.793297808	192.168.88.1	255.255.255.255	MNDP	157 5678 → 5678 Len=115
36	22.793323370	Routerbo_1c:a3:2a	CDP/VTP/DTP/PagP/UD...	CDP	108 Device ID: MikroTik Port ID: bridge

Imagem 7 - Pacotes MNDP e CDP

3.2 Experiência 2 - Implementação de duas bridges num switch

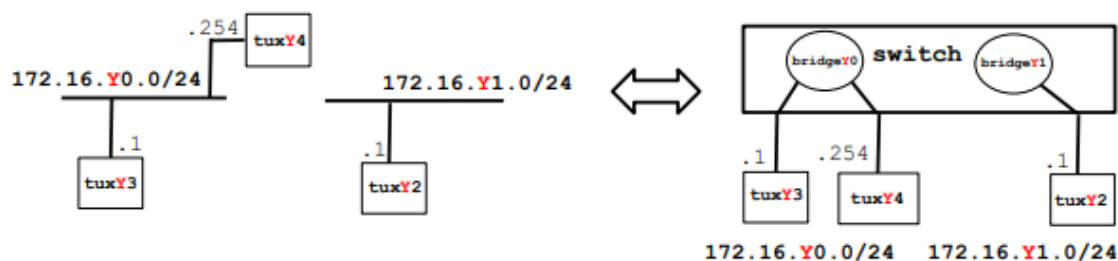


Imagem 8 - Experiência 2 (diagrama retirado do guião)

Antes de começar, é de especial interesse definir bem aquilo que é uma **bridge** e aquilo que é um **switch**. Um **switch** é sistema físico (*hardware*) que é responsável por orientar a informação que entra por uma dada porta e tem de ir para outra.



Imagem 9 - Switch usado no laboratório de Redes de Computadores

Por seu lado, uma **bridge** é uma ligação virtual entre duas portas, responsável por dividir uma mesma rede em vários segmentos.

Para configurar uma nova **bridge**, começamos por criar a bridge - `/interface bridge add name=bridge30`, temos de libertar da *default bridge* as portas que vão ser adicionadas a essa bridge - `/interface bridge port remove [find interface=etherX]`, onde X é o número da porta do switch e depois adicionar as portas respectivas à **bridge** já criada - `/interface bridge port add bridge=bridge30 interface=etherX` (no nosso caso usamos as portas 10 e 14 para as ligações das portas *eth0* dos tuxs 33 e 34 (em todas as aulas trabalhamos na bancada 3), respetivamente, à **bridge30**, e a porta 2 para ligar a porta *eth0* do tux 32 à **bridge31**).

Depois de realizar os comandos de *ping broadcast* nos tuxs 33 e 32, para as suas **bridges** respectivas, conseguimos perceber que existem dois domínios de *broadcast* diferentes, um que contém os tuxs 33 e 34 e outro que contém o tux32, o que é bastante visível nos logs, pois no primeiro *broadcast* apenas os tuxs 33 e 34 recebem e enviam pacotes e no segundo apenas o tux32.

3.3 Experiência 3 - Configuração de um router em Linux

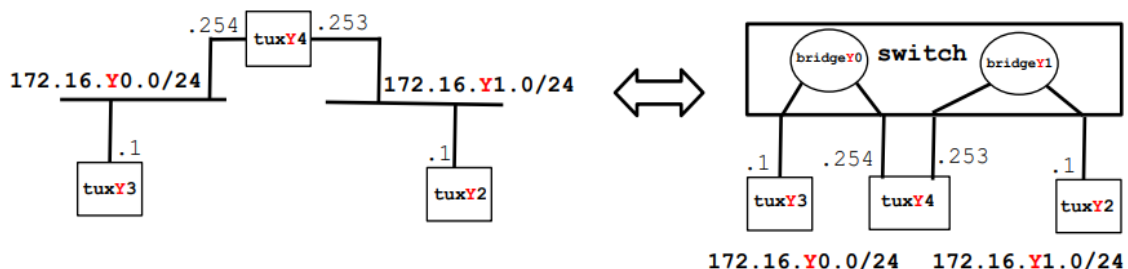


Imagem 10 - Experiência 3 (diagrama retirado do guião)

O primeiro passo desta experiência é transformar o tux34 num *router*. Para isso, é

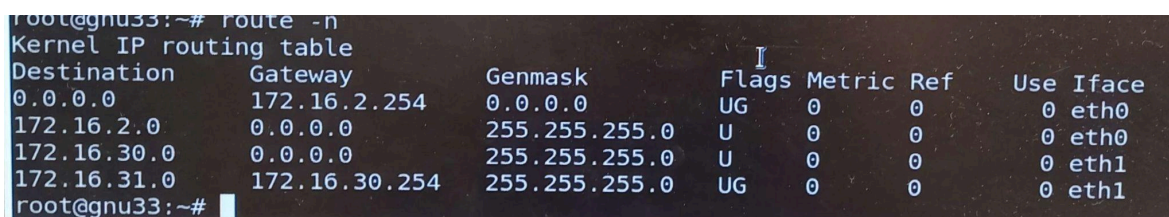
necessário ligá-lo à *bridge31*, configurando a porta *eth1* do *tux34*, com um novo IP(172.16.31.253), que tem ligação por cabo a uma porta no *switch*(6 para nós) que deve ser adicionada à *bridge31*, como é descrito anteriormente. Posteriormente, é preciso ativar o encaminhamento de IPs (`echo 1 > /proc/sys/net/ipv4/ip_forward`), e desabilitar ICMP *echo-ignore-broadcast*

```
(echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts).
```

Com tudo devidamente configurado, o passo seguinte é adicionar as rotas necessárias para que os *tuxs* 32 e 33 consigam comunicar entre si, ou seja:

- no *tux32* - `route add -net 172.16.30.0/24 gw 172.16.31.253` - para chegar à rede 172.16.30.0/24, não conhecida pelo *tux32*) é necessário utilizar o IP 172.16.31.253 como *gateway*.
- no *tux33* - `route add -net 172.16.31.0/24 gw 172.16.30.254` - para chegar à rede 172.16.31.0/24, não conhecida pelo *tux32*) é necessário utilizar o IP 172.16.30.254 como *gateway*.

Em ambos os casos, a *gateway* é um IP definido para o *tux34*, que agora é capaz de servir de *router*. Ao *pingar* o *tux32* de IP 172.16.31.1(pertencente à sub-rede 172.16.31.0/24) a partir do *tux33*, o pedido é reencaminhado para o *tux34* de IP 172.16.30.254. Como o *tux34* está ligado à sub-rede do *tux32* através da sua porta *eth1*, é capaz de reencaminhar o pacote para o *tux32*. O mesmo acontece para a resposta, mas no sentido contrário.



```
root@gnu33:~# route -n
Kernel IP routing table
Destination    Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0        172.16.2.254   0.0.0.0         UG    0      0      0 eth0
172.16.2.0     0.0.0.0        255.255.255.0   U     0      0      0 eth0
172.16.30.0    0.0.0.0        255.255.255.0   U     0      0      0 eth1
172.16.31.0    172.16.30.254 255.255.255.0   UG    0      0      0 eth1
```

Imagem 11 - Tabela de encaminhamento do *tux33*, obtida a partir do comando `route -n`

Cada linha da tabela a cima contém informação sobre o caminho(*gateway*) que deve ser tomado para mandar um pedido para uma *destination*(máquina específica ou rede), tal como:

- *Genmask* - a máscara de rede para a rede de destino
- *Flags* - cada letra tem o seu significado sendo que 'U' significa que está em

funcionamento(route is up), e 'G' que utiliza uma gateway

- *Metric* - custo da rota
- *Ref* - número de referências para esta rota (não usado no kernel do linux).
- *Use* - contador de pesquisas pela rota, dependendo do uso de -F ou -C (número de falhas da cache e número de sucessos, respetivamente).
- *Iface* - o interface para onde serão enviados os pacotes desta rota.

Quando executamos o comando ping no tux33 para o tux32, cujo MAC address é desconhecido, o tux33 envia uma mensagem ARP a pedir o MAC address do tux32 utilizando o IP que conhece. Nesta, o tux33 associa o seu próprio MAC address para que o recetor (tux32) saiba a quem responder("Who has [IP]? Tell [IP próprio]"). Esta mensagem é enviada em modo *broadcast* para que, quando o recetor com o IP pretendido, a receber, responda com outra mensagem ARP, privada, na qual diga o seu próprio MAC address("[IP] is at [MAC address]").

Esta troca ocorre sempre que uma mensagem é enviada entre máquinas que não conheçam os MAC address' uma da outra.

36	40.251431416	Netronix_50:31:bc	HewlettP_5a:74:3e	ARP	42	Who has 172.16.30.254? Tell 172.16.30.1
37	40.251549308	HewlettP_5a:74:3e	Netronix_50:31:bc	ARP	60	172.16.30.254 is at 00:21:5a:5a:74:3e
38	40.315463371	172.16.30.1	172.16.30.254	ICMP	98	Echo (ping) request id=0x3f69, seq=6/1536, ttl=64 (reply in 39)
39	40.315585384	172.16.30.254	172.16.30.1	ICMP	98	Echo (ping) reply id=0x3f69, seq=6/1536, ttl=64 (request in 38)
40	40.412989181	HewlettP_5a:74:3e	Netronix_50:31:bc	ARP	60	Who has 172.16.30.1? Tell 172.16.30.254
41	40.413000007	Netronix_50:31:bc	HewlettP_5a:74:3e	ARP	42	172.16.30.1 is at 00:08:54:50:31:bc
42	41.316437797	172.16.30.1	172.16.30.254	ICMP	98	Echo (ping) request id=0x3f69, seq=7/1792, ttl=64 (reply in 43)
43	41.316565327	172.16.30.254	172.16.30.1	ICMP	98	Echo (ping) reply id=0x3f69, seq=7/1792, ttl=64 (request in 42)

Imagem 12 - Troca de pacotes ARP e ICMP do tux33 para o tux34-eth0

Podemos observar também os pacotes ICMP do tipo *request* e *reply*, o que significa que os tuxs se reconhecem mutuamente, visto que as rotas tinham sido todas adicionadas. Se este não fosse o caso, os pacotes ICMP enviados seriam do tipo *Host Unreachable*.

Os endereços IPs e MAC address' associados a estes pacotes sao os endereços dos hosts que recebem ou enviam os pacotes (é possível ver que são iguais aos divulgados nas mensagens ARP).

170	107.067426943	Netronix_50:31:bc	HewlettP_5a:74:3e	ARP	42	Who has 172.16.30.254? Tell 172.16.30.1
171	107.067480093	172.16.30.1	172.16.31.1	ICMP	98	Echo (ping) request id=0x3f93, seq=7/1792, ttl=64 (reply in 173)
172	107.067554264	HewlettP_5a:74:3e	Netronix_50:31:bc	ARP	60	172.16.30.254 is at 00:21:5a:5a:74:3e
173	107.067722512	172.16.31.1	172.16.30.1	ICMP	98	Echo (ping) reply id=0x3f93, seq=7/1792, ttl=63 (request in 171)
174	108.091466243	172.16.30.1	172.16.31.1	ICMP	98	Echo (ping) request id=0x3f93, seq=8/2048, ttl=64 (reply in 175)
175	108.091725215	172.16.31.1	172.16.30.1	ICMP	98	Echo (ping) reply id=0x3f93, seq=8/2048, ttl=63 (request in 174)
176	108.120784636	Routerbo_1c:a3:33	Spanning-tree-(for-...	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:2f Cost = 0 Port = 0x8002
177	109.115463918	172.16.30.1	172.16.31.1	ICMP	98	Echo (ping) request id=0x3f93, seq=9/2304, ttl=64 (reply in 178)
178	109.115706477	172.16.31.1	172.16.30.1	ICMP	98	Echo (ping) reply id=0x3f93, seq=9/2304, ttl=63 (request in 177)
179	110.123025215	Routerbo_1c:a3:33	Spanning-tree-(for-...	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:2f Cost = 0 Port = 0x8002
180	110.130465783	172.16.30.1	172.16.31.1	ICMP	98	Echo (ping) request id=0x3f93, seq=10/2560, ttl=64 (reply in 181)

Imagem 13 - Ping do tux33 para o tux32

Quando as duas máquinas não estão conectadas à mesma sub-rede virtual, é necessário alterar o pacote ICMP, de forma a que o MAC address do destinatário seja trocado pelo do tux34-eth0, que por estar ligado a ambas as sub-redes, irá reencaminhar a mensagem para o endereço IP, que não foi trocado. Assim, na segunda troca de requests e replies, o MAC address de origem é o de tux34-eth1.

3.4 Experiência 4 - Configuração de um router comercial e implementação NAT

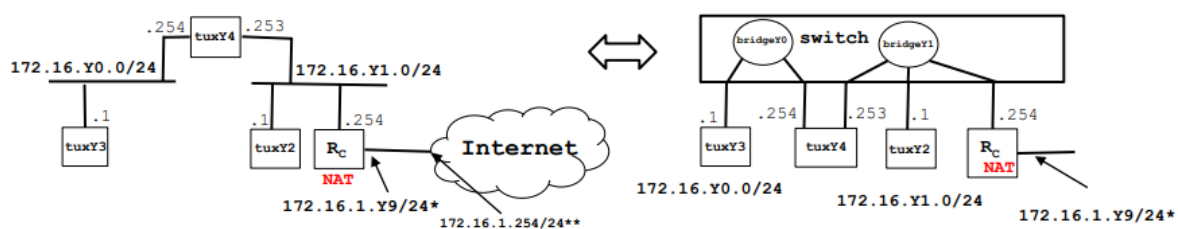


Imagem 14 - Experiência 4 (diagrama retirado do guião)

Começamos por conectar a porta *eth1* do Rc à rede do laboratório (o NAT está ativado por defeito) e a porta *eth2* do Rc à *bridge31*.

Depois adicionamos rotas *default* para diversas máquinas, de modo a todas conseguirem comunicar umas com as outras diretamente:

- tux34 como *default* do tux33.
- Rc como *default* do tux32 e 34.
- tux32 e Rc com rota para 172.16.30/24 (para conseguirem comunicar com o tux3).

No passo 3 comprovaos que de facto o tux33 tem ligação às outras máquinas da rede.

Na passo 4, vamos abrimos o tux32 e começamos por desativar a aceitação de redirecionamentos (`echo 0 > /proc/sys/net/ipv4/conf/eth0/accept_redirects` e `echo 0 > /proc/sys/net/ipv4/conf/all/accept_redirects`) e remover a rota para 172.16.30.0/24 via tux34.

```

> Destination: Routerbo_eb:24:12 (74:4d:28:eb:24:12)
> Source: HewlettP_61:24:01 (00:21:5a:61:24:01)
Type: IPv4 (0x0800)

> Destination: HewlettP_61:24:01 (00:21:5a:61:24:01)
> Source: EncoreNe_b4:b8:94 (00:e0:7d:b4:b8:94)
Type: IPv4 (0x0800)

```

Imagem 15 - Endereços MAC de *request* e de *reply* sem rota para o tux33 via tux34 e com a aceitação de redirects desativada

Depois de analisar os endereços MAC:

```

00:21:5a:61:24:01 - tux32 eth0
00:08:54:50:31:bc - tux33 eth0
00:21:5a:5a:74:3e - tux34 eth0
00:e0:7d:b4:b8:94 - tux34 eth1
74:4d:28:eb:24:12 - Rc

```

O tux32 envia o pacote para o Rc, pois ele tem uma rota que comunica com o tux33 via o tux34, o que torna possível a comunicação entre as duas máquinas.

De seguida, voltou-se a adicionar a rota para 172.16.30.0/24 via tux34 e activou-se a aceitação de redirecionamentos.

```

> Destination: EncoreNe_b4:b8:94 (00:e0:7d:b4:b8:94)
> Source: HewlettP_61:24:01 (00:21:5a:61:24:01)
Type: IPv4 (0x0800)

> Destination: HewlettP_61:24:01 (00:21:5a:61:24:01)
> Source: EncoreNe_b4:b8:94 (00:e0:7d:b4:b8:94)
Type: IPv4 (0x0800)

```

Imagem 16 - Endereços MAC de *request* e de *reply* com rota para o tux33 via tux34 e com a aceitação de redirects ativada

A principal conclusão a tirar desta imagem é a de que agora a comunicação entre o tux32 e o tux33 é feita diretamente, pois o tux32 já conhece o caminho (tem uma rota) que permite essa comunicação, via tux34.

Os últimos passos da experiência 4 servem para perceber o funcionamento do NAT. No passo 5, no tux33, fazemos *ping* à rede do laboratório e verificamos que existe comunicação. Depois de desativar o NAT no Rc e voltar a repetir a experiência, verificamos que já não existe conectividade. Porquê? O NAT faz o mapeamento de um endereço privado de uma rede local para um endereço público e vice-versa. Com o NAT desativado, a rede do laboratório não vai saber para quem enviar a resposta, e daí os resultados obtidos!

3.5 Experiência 5 - DNS

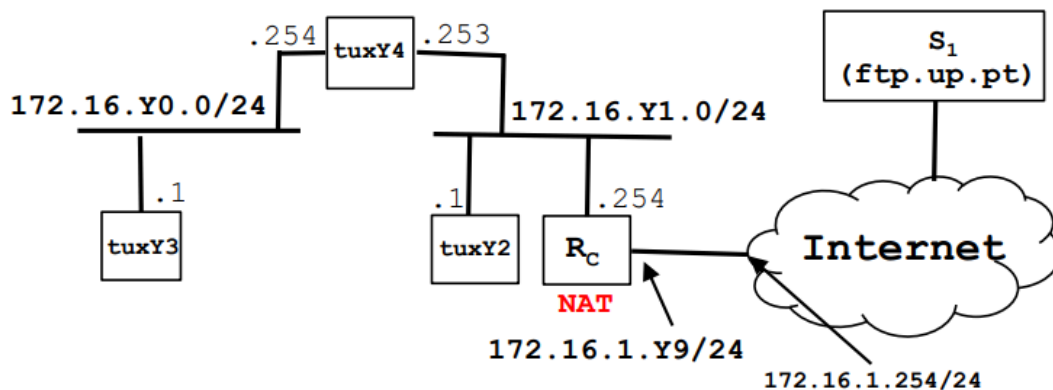


Imagem 17 - Experiência 5 (diagrama retirado do guião)

Nesta experiência o objetivo era perceber qual o propósito do DNS.

Primeiro configuramos o DNS nas três máquinas tux, editando o ficheiro `/etc/resolv.conf` e adicionando `nameserver 172.16.2.1`.

Podemos verificar que nos era possível fazer coisas tais como: abrir o *browser*, pingar sites como *google.com* ou *youtube.com*, entre outras.

7	11.845703095	172.16.31.253	172.16.2.1	DNS	75 Standard query 0xe0b4 A www.youtube.com
8	11.845714339	172.16.31.253	172.16.2.1	DNS	75 Standard query 0x1abe AAAA www.youtube.com
9	11.846520247	172.16.2.1	172.16.31.253	DNS	304 Standard query response 0xe0b4 A www.youtube.com CNAME youtube-ui.1
10	11.846536940	172.16.2.1	172.16.31.253	DNS	224 Standard query response 0x1abe AAAA www.youtube.com CNAME youtube-u
11	11.846922398	172.16.31.253	172.217.17.14	ICMP	98 Echo (ping) request id=0x24ec, seq=1/256, ttl=64 (reply in 12)
12	11.862734025	172.217.17.14	172.16.31.253	ICMP	98 Echo (ping) reply id=0x24ec, seq=1/256, ttl=113 (request in 11)

Imagem 18 - Pacotes DNS na receção e envio de pacotes para um *hostname* na internet (youtube.com)

O DNS é um sistema hierárquico e distribuído de gestão de nomes para computadores, serviços ou qualquer outra máquina conectada à internet ou a uma rede privada. O DNS permite a resolução de nomes de domínios em endereços IP, o que permite a comunicação com domínios externos como vimos anteriormente. A troca de pacotes consiste no envio do *hostname* para o servidor, e a recepção do endereço IP correspondente.

3.6 Experiência 6 - TCP connections

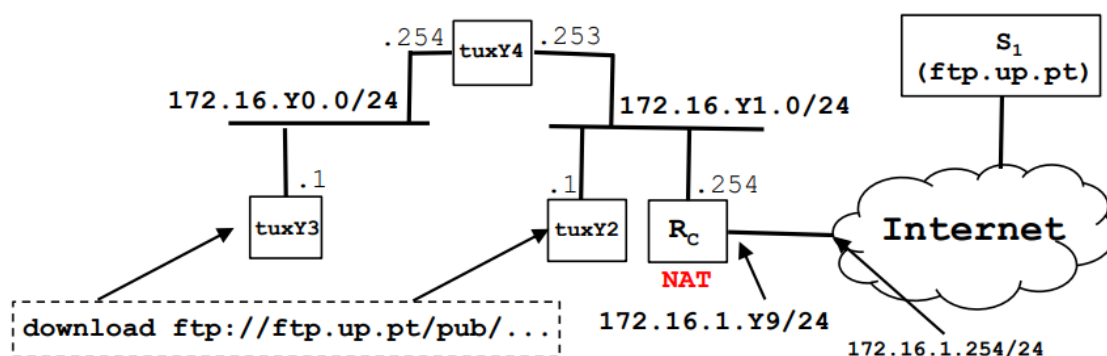


Imagem 19 - Experiência 6 (diagrama retirado do guião)

Na última experiência, juntamos a aplicação da parte 1 com a rede que foi sendo montada ao longo da parte 2 do trabalho. Começamos então por, no tux33, compilar a aplicação e transferir um ficheiro, capturando a transferência de pacotes. Durante o funcionamento da aplicação, são abertas duas conexões TCP, que correspondem às duas *sockets* criadas. Uma delas serve para transportar a informação de controlo de funcionamento da aplicação, que prepara a transferência de dados, e a outra para a transferência de dados em si.

A conexão TCP é dividida em três partes - abertura da conexão, transferência de dados e fecho da conexão.

11	9.624167924	172.16.30.1	192.168.109.136	TCP	74	39488	→ 21	[SYN]	Seq=0	Win=64240	Len=0	MSS=1460	SACK_PERM	TSval=1482344970	TSecr=
12	9.625204435	192.168.109.136	172.16.30.1	TCP	74	21	→ 39488	[SYN, ACK]	Seq=0	Ack=1	Win=65160	Len=0	MSS=1460	SACK_PERM	TSval=9663
13	9.625236771	172.16.30.1	192.168.109.136	TCP	66	39488	→ 21	[ACK]	Seq=1	Ack=1	Win=64256	Len=0	TSval=1482344971	TSecr=966339640	
14	9.627434249	192.168.109.136	172.16.30.1	FTP	100	Response: 220 Welcome to netlab-FTP server									
15	9.627446122	172.16.30.1	192.168.109.136	TCP	66	39488	→ 21	[ACK]	Seq=1	Ack=35	Win=64256	Len=0	TSval=1482344974	TSecr=966339642	
16	9.627498713	172.16.30.1	192.168.109.136	FTP	77	Request: USER rcom									
17	9.628353218	192.168.109.136	172.16.30.1	TCP	66	21	→ 39488	[ACK]	Seq=35	Ack=12	Win=65280	Len=0	TSval=966339643	TSecr=1482344974	
18	9.628464335	192.168.109.136	172.16.30.1	FTP	100	Response: 331 Please specify the password.									
19	9.628488220	172.16.30.1	192.168.109.136	FTP	77	Request: PASS rcom									
20	9.629413125	192.168.109.136	172.16.30.1	TCP	66	21	→ 39488	[ACK]	Seq=69	Ack=23	Win=65280	Len=0	TSval=966339644	TSecr=1482344975	
21	9.638363044	192.168.109.136	172.16.30.1	FTP	89	Response: 230 Login successful.									

Imagem 20 - Abertura da ligação e envio de pacotes de controlo (login)

Depois de uma transferência normal no tux33, repetimos o *download* no tux33 mas ao mesmo tempo começamos um novo *download* no tux32.

Podemos ver que mesmo assim o *download* é efetuado com sucesso, uma vez que o FTP usa um sistema de retransmissão *Selective Repeat ARQ*, semelhante ao *Go-Back-N ARQ*, que não deixa de processar os frames recebidos mesmo quando deteta um erro.

ARQ (*Automatic Repeat reQuest*) é um método de controlo de erros e recuperação de pacotes para transmissão de dados, que consiste num receptor enviar um alerta ao remetente caso algum pacote falhe, para que o remetente possa reenviar o pacote em falta.

Os campos de TCP mais relevantes são:

- as portas dos dispositivos de origem e destino 48010 → 41737
41737 → 48010
- o número de sequência, que seja incrementado de acordo com o número de bytes transmitidos e indica o número do pacote a ser enviado Seq=144
- o número de confirmação, que começa no 0 e incrementa com o número de bytes recebido e indica o correto recebimento da trama Ack=23
- o *window size*, que indica a quantidade de bytes que o servidor pode enviar antes de ter de esperar por um ACK(*acknowledgement*) e por um *update* de *window size* por parte do host que recebe Win=65280

O TCP *congestion control mechanism* baseia-se na análise dos ACK recebidos, isto é, calcula um valor por conexão - *Congestion Window* - que regula os *window sizes* de pacotes com base na congestão da conexão. Este valor incrementa por 1 a cada RTT (“*round trip time*”). Quando um pacote é perdido, ou seja, depois de um intervalo de tempo estipulado sem receber um ACK (*timeout*), o valor da *Congestion Window* passa para metade. O *bitrate* da conexão será aproximadamente igual a *CongestionWindow/RTT*.

No início da conexão, pode também haver uma fase de *slow start*, de modo a delimitar um *threshold* que é utilizado numa fase posterior de *congestion avoidance*.

No nosso caso, como na altura em que realizamos os testes não sabíamos qual resultado esperado, escolhemos fazer download de um ficheiro demasiado pequeno que não alterou quase nada a taxa de transferência do que já estava a decorrer.

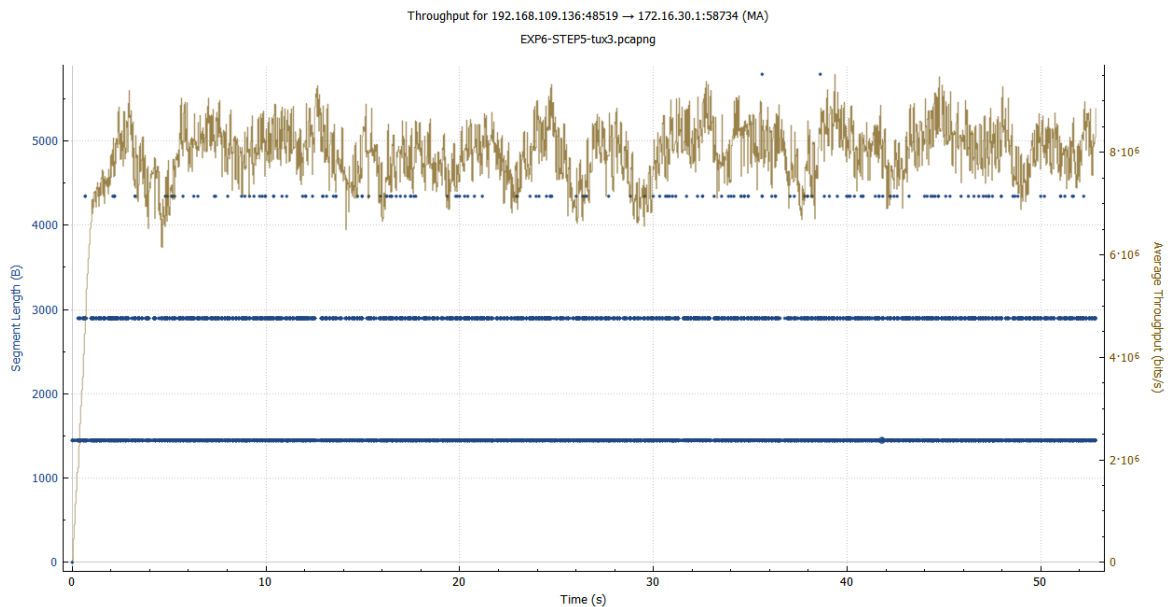


Imagem 21 - Gráfico da evolução do *throughput* da conexão TCP, para o tux33, ao longo do tempo, para a transferência simultânea de ficheiros em tux33 e tux32

Acreditamos que a transferência do segundo ficheiro tenha acontecido na primeira queda logo após a subida.

Podemos apenas explicar aquilo que deveria ter acontecido. Aquando o começo do primeiro *download*, a taxa de transmissão do tux33 aumentaria rapidamente, estabilizando em poucos segundos. Após o começo do segundo *download*, no tux32, a taxa de transmissão do tux33, deveria diminuir rapidamente até estabilizar num nível mais baixo, até ao fim do segundo *download*.

Estas mudanças iriam de acordo com o mecanismo de controlo de congestão do TCP, pois diminuem o *bitrate* das conexões já em curso, quando aumenta o congestionamento da rede.

Como deveria ter sido observado, o *throughput* da conexão de dados TCP seria perturbado pela segunda transmissão, visto que a transmissão de pacotes da primeira diminui, para que a taxa de transferência fosse distribuída igualmente para cada ligação.

4. Conclusão

Com a elaboração deste trabalho, foi nos possível por em prática os conhecimentos apresentados na aula teórica, de uma forma prática e assim melhor perceber como funciona e como pode ser implementada uma pequena rede de computadores, quais os protocolos existentes para comunicação e como funciona um protocolo de comunicação a um nível mais profundo, como foi o caso do FTP.

Foi um trabalho que gostamos de fazer, e que consideramos ter feito num ritmo saudável, com as experiências a ser feitas ao longo das aulas práticas e a parte 1 depois de já ter feito as experiências e também ter experimentado usar ftp na primeira aula desta segunda metade do semestre foi algo que não nos causou grandes problemas.

O acompanhamento nas aulas práticas foi muito bom, e consideramos que as apresentações sobre o trabalho para aula no início de cada aula foram muito importantes não só na elaboração do trabalho mas também para melhor perceber o porquê de se fazer as coisas.

No final podemos estar orgulhosos do trabalho realizado, e acrescentar também que chegar à experiência 6 e por tudo junto a funcionar é algo que para nós foi bastante prazeroso.

5. Referências

Aulas teóricas e guião das aulas práticas da UC RC.

https://en.wikipedia.org/wiki/List_of_FTP_commands

https://pt.wikipedia.org/wiki/Protocolo_de_Transfer%C3%A2ncia_de_Arquivos

https://pt.wikipedia.org/wiki/Internet_Control_Message_Protocol

<https://www.ibm.com/docs/en/zos-basic-skills?topic=layer-address-resolution-protocol-arp>

<https://ipwithease.com/difference-between-a-switch-and-a-bridge/>

https://en.wikipedia.org/wiki/Automatic_repeat_request

https://en.wikipedia.org/wiki/Domain_Name_System

<https://www.comptia.org/content/guides/what-is-network-address-translation>

6. Anexos

6.1 Fotos e Logs

[Pasta com fotos e logs](#)

6.2 Script para preparação

[Guião com passos para preparar as experiências a partir da nº 4](#)

6.3 Código fonte parte 1

[Link para o repositório no github](#)

main.c

```
#include "include/URL.h"
#include "include/FTP.h"

//unique use case: connect, login host, passive, get path, success
(file saved in CWD) or un-success (indicating failing phase)
//parseUrl
//getIpByHostName
//connectSocket
//ftpLogin
//ftpCWD
//ftpPasv
//ftpRetr
//ftpDownload
//ftpClose

int main(int argc, char *argv[]) {
```

```

    if (argc != 2) {
        fprintf(stderr, "Usage: %s\n", argv[0]);
        exit(-1);
    }

    //URL PARSE no match
    url u;
    initUrl(&u);
    parseUrl(&u, argv[1]);
    getIpByHostName(&u);

    printf("host: %s\n with ip: %s", u.host, u.ip);

    //FTP CONNECT
    ftp ftp;
    ftpConnect(&ftp, u.ip, u.port);

    //FTP LOGIN
    //Verify if it is a user or if it is anonymous
    if(strlen(u.user) > 0) {
        ftpLogin(&ftp, u.user, u.password);
    } else {
        ftpLogin(&ftp, "anonymous", "anonymous");
    }

    ftpPasv(&ftp);

    ftpCWD(&ftp, u.path);

    ftpPWD(&ftp);

    ftpRetr(&ftp, u.filename);

    ftpDownload(&ftp, u.filename);

    ftpClose(ftp.control_socket_fd);

    return 0;

```

```
}
```

FTP.h

```
#pragma once

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

#include <string.h>

typedef struct FTP
{
    int control_socket_fd; // file descriptor to control socket
    int data_socket_fd; // file descriptor to data socket
} ftp;

int connectSocket(const char* IPAddress, int port);
int ftpConnect(ftp* ftp, const char* ip, int port);
int ftpLogin(ftp* ftp, const char* username, const char* password);
int ftpPasv(ftp* ftp);
int ftpCWD(ftp* ftp, const char* path);
int ftpPWD(ftp* ftp);
int ftpRetr(ftp* ftp, const char* filename);
int ftpDownload(ftp* ftp, const char* filename);
int ftpWrite(int sockfd, char* buf, size_t buf_size);
int ftpRead(int sockfd, char* buf, size_t size);
int ftpReadAndPrint(int sockfd);
int ftpReadToFile(int sockfd, const char* filename);
int ftpClose(int sockfd);
```

URL.h

```
#pragma once

#include <stdio.h>
#include <stdlib.h>

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

#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MAX_STRING_LENGTH 256
typedef char url_content[MAX_STRING_LENGTH];

typedef struct URL {
    url_content user;
    url_content password;
    url_content host;
    url_content ip;
    url_content path;
    url_content filename;
    int port;
} url;

void initUrl(url *u);
int parseUrl(url *u, const char *urlString);
int getIpByHostName(url *u);
```

FTP.c

```
#include "../include/FTP.h"

//given a idAddress and a port, connect a socket
```

```

int connectSocket(const char* ipAddress, int port) {
    int sockfd;
    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(ipAddress);    /*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 ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket()");
        return -1;
    }
    /*connect to the server*/
    if (connect(sockfd, (struct sockaddr *) &server_addr,
sizeof(server_addr)) < 0) {
        perror("connect()");
        return -1;
    }

    return sockfd;
}

//Given an and a port, connect to a socket and save its file
descriptor in the ftp struct
int ftpConnect(ftp* ftp, const char* ip, int port) {
    int sockfd;

    if ((sockfd = connectSocket(ip, port)) < 0) {
        printf("ERROR: Cannot connect socket.\n");
        return 1;
    }

    ftp->control_socket_fd = sockfd;
    ftp->data_socket_fd = 0;
}

```



```

        if (ftpReadAndPrint(ftp->control_socket_fd)) {
            printf("ERROR: ftpReadAndPrint failure.\n");
            return 1;
        }

        return 0;
    }

    //Given a socket file descriptor, a user and a password, login to
the ftp server
    int ftpLogin(ftp* ftp, const char* username, const char* password){
        char buf[1024];

        //USER
        sprintf(buf, "USER %s\r\n", username);

        if (ftpWrite(ftp->control_socket_fd, buf, strlen(buf))) {
            printf("ERROR: ftpWrite failure on login username.\n");
            return 1;
        }

        if (ftpReadAndPrint(ftp->control_socket_fd)) {
            printf("ERROR: ftpReadAndPrint failure on login
username.\n");
            return 1;
        }

        memset(buf, 0, sizeof(buf));

        //PASS
        sprintf(buf, "PASS %s\r\n", username);

        if (ftpWrite(ftp->control_socket_fd, buf, strlen(buf))) {
            printf("ERROR: ftpWrite failure on login password.\n");
            return 1;
        }

        if (ftpReadAndPrint(ftp->control_socket_fd)) {

```

```

        printf("ERROR: ftpReadAndPrint failure on login
password.\n");
        return 1;
    }

    return 0;
}

//Given a socket file descriptor, enter passive mode
int ftpPasv(ftp* ftp) {
    char buf[1024];
    char* ip;
    int port;

    if (ftpWrite(ftp->control_socket_fd, "PASV\r\n",
strlen("PASV\r\n"))) {
        printf("ERROR: ftpWrite failure on PASV.\n");
        return 1;
    }

    if (ftpRead(ftp->control_socket_fd, buf, sizeof(buf))) {
        printf("ERROR: ftpRead failure on PASV.\n");
        return 1;
    }

    ip = strtok(buf, "(");
    ip = strtok(NULL, "(");
    ip = strtok(ip, ")");
    char ip2[16];
    bzero(ip2, 16);
    strcpy(ip2, strtok(ip, ","));
    for (int i = 0; i < 3; i++) {
        strcat(ip2, ".");
        strcat(ip2, strtok(NULL, ","));
    }

    port = atoi(strtok(NULL, ",")) * 256;
    port += atoi(strtok(NULL, ","));
}

```

```

printf("IP: %s\n", ip2);
printf("PORT: %d\n", port);

if ((ftp->data_socket_fd = connectSocket(ip2, port)) < 0) {
    printf("ERROR: Cannot connect socket.\n");
    return 1;
}

printf("Data socket connected.\n");
return 0;
}

//Given a socket file descriptor, and a path, change the current
directory
int ftpCWD(ftp* ftp, const char* path) {
    char buf[1024];

    sprintf(buf, "CWD %s\r\n", path);

    if (ftpWrite(ftp->control_socket_fd, buf, strlen(buf))) {
        printf("ERROR: ftpWrite failure on CWD.\n");
        return 1;
    }

    if (ftpReadAndPrint(ftp->control_socket_fd)) {
        printf("ERROR: ftpReadAndPrint failure on CWD.\n");
        return 1;
    }

    return 0;
}

//Given a socket file descriptor, get the current directory
int ftpPWD(ftp* ftp) {
    char buf[1024];

    sprintf(buf, "PWD\r\n");

    if (ftpWrite(ftp->control_socket_fd, buf, strlen(buf))) {

```

```

        printf("ERROR: ftpWrite failure on PWD.\n");
        return 1;
    }

    if (ftpReadAndPrint(ftp->control_socket_fd)) {
        printf("ERROR: ftpReadAndPrint failure on PWD.\n");
        return 1;
    }

    return 0;
}

//Given a socket file descriptor and a filename, retrieve a file
int ftpRetr(ftp* ftp, const char* filename) {
    char buf[1024];

    sprintf(buf, "RETR %s\r\n", filename);

    if (ftpWrite(ftp->control_socket_fd, buf, strlen(buf))) {
        printf("ERROR: ftpWrite failure on Retr.\n");
        return 1;
    }

    if (ftpReadAndPrint(ftp->control_socket_fd)) {
        printf("ERROR: ftpReadAndPrint failure on Retr.\n");
        return 1;
    }

    return 0;
}

//Given a socket file descriptor and a filename, download a file
int ftpDownload(ftp* ftp, const char* filename) {

    if (ftpReadToFile(ftp->data_socket_fd, filename)) {
        printf("ERROR: ftpReadToFile failure on Download.\n");
        return 1;
    }
}

```

```

        return 0;
    }

    //Given a socket file descriptor, write a string to it
    int ftpWrite(int sockfd, char* buf, size_t size) {
        /*send a string to the server*/
        size_t bytes = write(sockfd, buf, size);

        if (bytes > 0){
            printf("Bytes escritos %ld\n", bytes);
        }
        else {
            perror("write()");
            exit(-1);
        }

        return 0;
    }

    //Given a socket file descriptor, read from it
    int ftpRead(int sockfd, char* buf, size_t size) {
        FILE* fp = fdopen(sockfd, "r");

        do {
            memset(buf, 0, size);
            fgets(buf, size, fp);
            printf("%s", buf);
        } while (!('1' <= buf[0] && buf[0] <= '5') || buf[3] != ' ');

        return 0;
    }

    int ftpReadAndPrint(int sockfd) {
        FILE* fp = fdopen(sockfd, "r");
        size_t size = 1024;
        char* buf = malloc(size);

        do {
            memset(buf, 0, size);

```

```

        fgets(buf, size, fp);
        printf("%s", buf);
    } while (!('1' <= buf[0] && buf[0] <= '5') || buf[3] != ' ');

    free(buf);
    return 0;
}

//Given a socket file descriptor, read from it and save it to a file
with the given name

int ftpReadToFile(int sockfd, const char* filename) {
    size_t size = 1024;
    char* buf = malloc(size);
    FILE* file = fopen(filename, "w");
    int bytes = 0;

    while ((bytes = read(sockfd, buf, size)) != 0) {

        if (bytes < 0){
            printf("Error reading from file.\n");
        }

        printf("%s", buf);

        if ((bytes = fwrite(buf, bytes, 1, file)) == 0){
            printf("Error writing to file from file.\n");
        }
        memset(buf, 0, size);
    }

    free(buf);

    fclose(file);
    close(sockfd);

    return 0;
}

```

```

//Given a socket file descriptor, close it
int ftpClose(int sockfd) {
    if (close(sockfd)<0) {
        perror("close()");
        exit(-1);
    }
    return 0;
}

```

URL.c

```

#include "../include/URL.h"

/**
 * The struct hostent (host entry) with its terms documented
 *
 * struct hostent {
 *     char *h_name;    // Official name of the host.
 *     char **h_aliases; // A NULL-terminated array of alternate
names for the host.
 *     int h_addrtype;    // The type of address being returned;
usually AF_INET.
 *     int h_length;    // The length of the address in bytes.
 *     char **h_addr_list; // A zero-terminated array of network
addresses for the host.
 *     // Host addresses are in Network Byte Order.
 * };
 *
 * #define h_addr h_addr_list[0]    The first address in
h_addr_list.
 */

void initUrl(url *u) {
    memset(u->user, 0, MAX_STRING_LENGTH);
    memset(u->password, 0, MAX_STRING_LENGTH);
    memset(u->host, 0, MAX_STRING_LENGTH);
    memset(u->path, 0, MAX_STRING_LENGTH);
    memset(u->filename, 0, MAX_STRING_LENGTH);
    u->port = 21; //FTP is on port 21
}

```

```

}

//Function able to parse the ulr string and return the different
parts of the url in the url struct
//Return 0 if the url is valid, -1 if not

int parseUrl(url *u, const char *urlString){
    regex_t re;

    int res = 0;

    res = regcomp(&re,
"^ftp://([a-zA-Z0-9]+):([a-zA-Z0-9]+)@?([a-zA-Z0-9$_+!*'(),-]+)(:([0-9]+))?(([a-zA-Z0-9 /()_.,-]*?)/([a-zA-Z0-9 ()_.,-]+)$", REG_EXTENDED);
    if (res != 0) {
        char buffer[256];
        regerror(res, &re, buffer, sizeof(buffer));
        printf("regcomp() failed with '%s'\n", buffer);
        exit(EXIT_FAILURE);
    }

    regmatch_t m[re.re_nsub + 1];
    if (regexexec(&re, urlString, re.re_nsub + 1, m, 0) == 0) {

        int start = m[2].rm_so, end = m[2].rm_eo;
        if(start > 0 && end > 0){
            strncpy(u->user, urlString + start, end - start);
            printf("user: %s\n", u->user);
        }

        start = m[3].rm_so, end = m[3].rm_eo;
        if(start > 0 && end > 0){
            strncpy(u->password, urlString + start, end - start);
            printf("password: %s\n", u->password);
        }

        start = m[4].rm_so, end = m[4].rm_eo;
        strncpy(u->host, urlString + start, end - start);
        printf("host: %s\n", u->host);
    }
}

```



```

        start = m[7].rm_so, end = m[7].rm_eo;
        strncpy(u->path, urlString + start, end - start);
        printf("path: %s\n", u->path);

        start = m[8].rm_so, end = m[8].rm_eo;
        strncpy(u->filename, urlString + start, end - start);
        printf("filename: %s\n", u->filename);
    }
    else
        printf("No match! Correct usage:
ftp://[<user>:<password>@]host[:port]/path/to/filename\n");

    regfree(&re);
    return 0;
}

int getIpByHostName(url *u) {
    struct hostent *h;

    if ((h = gethostbyname(u->host)) == NULL) {
        perror("gethostbyname()");
        exit(-1);
    }

    printf("Host name : %s\n", h->h_name);
    printf("IP Address : %s\n", inet_ntoa(*(struct in_addr *)
h->h_addr));

    char *ip = inet_ntoa(*(struct in_addr *) h->h_addr);
    strcpy(u->ip, ip);

    return 0;
}

```