

**RELATÓRIO**

**Redes de Computadores**

**Trabalho Prático nº2**

*Turma 1*

Luís Afonso Sampaio Oliveira - up201707229@fe.up.pt

Pedro Miguel Ribeiro Alves – up201707234@fe.up.pt

Ricardo França Domingues Cardoso – up201604686@fe.up.pt

*Dezembro de 2019*

Índice

[Sumário 4](#_Toc28019909)

[Introdução 4](#_Toc28019910)

[Parte 1 – Aplicação de Download 4](#_Toc28019911)

[1.1 – Arquitetura 4](#_Toc28019912)

[1.2 – Resultados 5](#_Toc28019913)

[Parte 2 – Configuração e Análise de Rede 5](#_Toc28019914)

[2.1 – Configuração de uma rede IP 5](#_Toc28019915)

[2.1.1 – O que são pacotes ARP e qual a sua utilidade? 5](#_Toc28019916)

[2.1.2 – Quais são os endereços MAC e IP dos pacotes ARP e porquê? 5](#_Toc28019917)

[2.1.3 – Quais são os pacotes gerados pelo comando ping? 6](#_Toc28019918)

[2.1.4 – Quais são os endereços MAC e IP dos pacotes ping? 6](#_Toc28019919)

[2.1.5 – Como determinar se uma trama Ethernet recebida é do tipo ARP, IP, ICMP? 6](#_Toc28019920)

[2.1.6 – Como determinar o tamanho de uma trama recebida? 6](#_Toc28019921)

[2.1.7 – O que é a interface de loopback e porque é que esta é tão importante? 6](#_Toc28019922)

[2.2 – Implementação de LAN’s virtuais num switch 7](#_Toc28019923)

[2.2.1 – Como configurar a vlany0? 7](#_Toc28019924)

[2.2.2 – Quantos domínios de broadcast existem? 7](#_Toc28019925)

[2.3 – Configuração de um router em Linux 7](#_Toc28019926)

[2.3.1 – Que rotas existem nos tuxes e quais os seus significados? 7](#_Toc28019927)

[2.3.2 – Que informação está contida numa entrada da tabela de forwarding? 8](#_Toc28019928)

[2.3.3 – Que mensagens ARP e endereços MAC associados são observados e porquê? 8](#_Toc28019929)

[2.3.4 – Que pacotes ICMP são observados e porquê? Quais são os endereços IP e MAC associados aos pacotes ICMP e porquê? 8](#_Toc28019930)

[2.4 – Configuração de um router comercial e implementação de NAT 9](#_Toc28019931)

[2.4.1 – Como configurar uma rota estática num router comercial? 9](#_Toc28019932)

[2.4.2 – Quais os caminhos percorridos pelos pacotes e porquê? 9](#_Toc28019933)

[2.4.3 – Como configurar NAT num router comercial? 9](#_Toc28019934)

[2.4.4 – O que faz o NAT? 9](#_Toc28019935)

[2.5 – DNS 9](#_Toc28019936)

[2.5.1 – Como configurar o serviço de DNS num host? 9](#_Toc28019937)

[2.5.2 – Que pacotes são trocados pelo DNS e que informação contêm? 10](#_Toc28019938)

[2.6 – Ligações TCP 10](#_Toc28019939)

[2.6.1 – Quantas ligações TCP são abertas pela aplicação FTP? Em que ligação é transportada a informação de controlo FTP? 10](#_Toc28019940)

[2.6.3 – Quais as fases de uma ligação FTP? 10](#_Toc28019941)

[2.6.4 – Como é que funciona o mecanismo ARQ TCP? Quais são os campos TCP relevantes? 10](#_Toc28019942)

[2.6.5 – Como é que funciona o mecanismo de controlo de congestão TCP? Quais os campos relevantes? 10](#_Toc28019943)

[2.6.6 – O aparecimento de uma segunda ligação TCP afeta o fluxo de dados de uma ligação TCP já existente? De que forma? 10](#_Toc28019944)

[Conclusão 11](#_Toc28019945)

[Anexo I – Código fonte 11](#_Toc28019946)

[clientTCP.h 11](#_Toc28019947)

[clientTCP.c 11](#_Toc28019948)

[getip.h 12](#_Toc28019949)

[getip.c 13](#_Toc28019950)

[downloadApp.c 13](#_Toc28019951)

[Anexo II – Execução da aplicação de download 28](#_Toc28020003)

[Anexo III – DNS 28](#_Toc28020004)

[Anexo IV – TCP Connection Phases 29](#_Toc28020005)

[a) Connection established 29](#_Toc28020006)

[b) Connection terminated 29](#_Toc28020007)

[Anexo V – Graphs 30](#_Toc28020008)

[a) Tux 2 30](#_Toc28020009)

[b) Tux 1 30](#_Toc28020010)

# Sumário

O presente relatório foi efetuado no âmbito da unidade curricular de Redes de Computadores, de modo a complementar o segundo trabalho laboratorial cujo tema é a configuração e o estudo de uma rede de computadores e a sua integração com uma aplicação de transferência de ficheiros.

# Introdução

O trabalho prático divide-se em duas componentes: a configuração de uma rede de computadores e a criação de uma aplicação de transferência de ficheiros através do protocolo FTP (*File Transfer Protocol*).

A rede é constituída por três computadores (tuxy1, tuxy2, tuxy4), duas VLAN’s (*Virtual Local Area Network*) num switch e um router comercial com a funcionalidade de NAT (*Network Address Translation*). A vlany0 é constituída pelo tuxy1 e tuxy4 e a vlany1 é constituída pelo tuxy4, tuxy2 e pelo router comercial. O tuxy4 é utilizado como router, de forma a ligar as duas VLAN’s.

Após a configuração da rede e do desenvolvimento da aplicação foi efetuada a configuração do servidor DNS (*Domain Name System*).

# Parte 1 – Aplicação de Download

A primeira parte do trabalho consistiu no desenvolvimento de uma aplicação de download na linguagem de programação C, que aceita, como argumento, um link no formato **ftp://<user>:<password>@<host>/<url-path>**. Esta aplicação descarrega um ficheiro, de qualquer tipo, de um servidor FTP.

Durante o seu desenvolvimento foram estudados o RFC959, que aborda o FTP, e o RFC1738, que fala sobre o tratamento de informação proveniente de URL’s.

## – Arquitetura

Em primeiro lugar processa-se o URL, reservando-se espaço para as variáveis **user**, **pass**, **host**, e **path** e chamando a função **parseArguments** para obtê-las a partir do URL. O endereço de ip é obtido através da função **getIPByName** (adaptação do código fornecido). A porta usada é sempre a 21, visto que, de acordo com o RFC1738, a porta padrão é a 21 e deve ser utilizada se a porta não for especificada no URL.

Em seguida, abre-se a socket onde se realizará a conexão entre o cliente e o servidor. Utiliza-se então a função **ReadReplyNoInfo**, que lê um código de resposta enviado pelo servidor, para receber o código de “connection greetings”, que indica que o servidor aguarda input do utilizador. Caso o código recebido indique que o servidor não pode, de momento, aceitar input, o programa termina.

A comunicação entre o cliente e o servidor assenta no usa das funções **sendCommandProcessReply.** Esta função envia um comando para o servidor FTP e processa a resposta respetiva. O primeiro dígito indica se a resposta é positiva (1,2,3) ou negativa (4,5). 1 significa que se deve esperar outra resposta antes de enviar um novo comando; 2 que a ação pedida foi realizada com sucesso e se pode enviar um novo comando; 3 que o comando foi aceite, mas são esperados novos comandos com informação adicional; 4 que a ação não foi realizada com sucesso, mas devido a um erro temporário, podendo ser reenviada a mesma sequência de comandos; 5 que o comando não foi aceite e o processo do utilizador não deve reenviar a mesma sequência. Nem todos os comandos podem receber todos estes códigos, e por isso a função **sendCommandProcessReply** tem dois argumentos adicionais, **canWaitForExtraReplies** e **canRetryCommand**, que indicam se a receção dos códigos 1 e 4, respetivamente, pode ser tratada imediatamente dentro da função, recebendo a próxima resposta ou reenviando o comando,respetivamente, Se a sua receção não for esperada ou, no caso do código 4, o comando enviado seja parte de uma sequência que tenha de ser enviada do início, a função termina, de modo a que o processamento do erro ou o reenvio da sequência possa ser tratado fora da função.

De modo a estabelecer a conexão entre o cliente e o servidor, é enviado o comando USER user e PASS pass, para realizar o login. Depois, envia-se o comando PASV, para que o servidor entre no modo passivo, sendo enviado na resposta a porta necessária à abertura de outra socket que servirá para a transferência de dados.

Em seguida, envia-se o comando RETR filename, sendo o nome do ficheiro obtido através da função **getFileNameFromPath**, para pedir o ficheiro ao servidor. É então realizada a transferência do ficheiro, utilizando a função **createFile**. No final, é enviado o comando QUIT, para fazer o logout do utilizador, e são fechadas ambas as sockets (de comunicação cliente-servidor e da transferência de dados).

## – Resultados

O programa foi testado em vários contextos: modo anónimo, modo não anónimo, diferentes tipos de ficheiro e diversos tamanhos de ficheiro. O programa termina em caso de erro ou se o ficheiro não for encontrado.

# Parte 2 – Configuração e Análise de Rede

## 2.1 – Configuração de uma rede IP

### 2.1.1 – O que são pacotes ARP e qual a sua utilidade?

Os pacotes ARP (*Address Resolution Protocol*) são pacotes utilizados para fazer o mapeamento de um endereço de rede a um endereço físico.

### 2.1.2 – Quais são os endereços MAC e IP dos pacotes ARP e porquê?

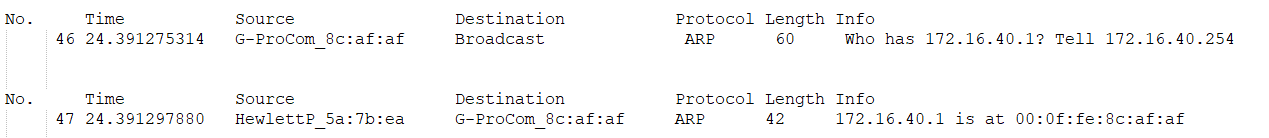


Figura 1- Pacotes ARP de 'request' e 'reply'

Na tentativa de comunicar com um determinado endereço de rede, é enviado um pacote ARP (request) com endereços de origem, tanto MAC como IP, do tux de origem e com ‘broadcast’ como destino, visto que este request é feito à toda a rede local em busca do endereço físico que corresponde ao tal endereço de rede. Finalmente, o tux de destino envia um pacote ARP (reply) ao tux de origem contendo o seu endereço físico.

### 2.1.3 – Quais são os pacotes gerados pelo comando ping?

O comando ping gera pacotes ICMP (*Internet Control Message Protocol*) de pedido e resposta.

### 2.1.4 – Quais são os endereços MAC e IP dos pacotes ping?

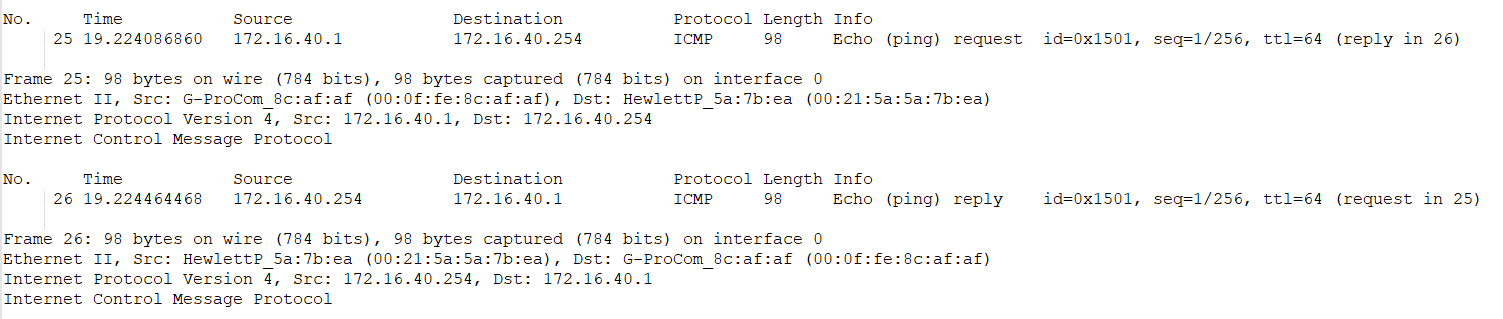
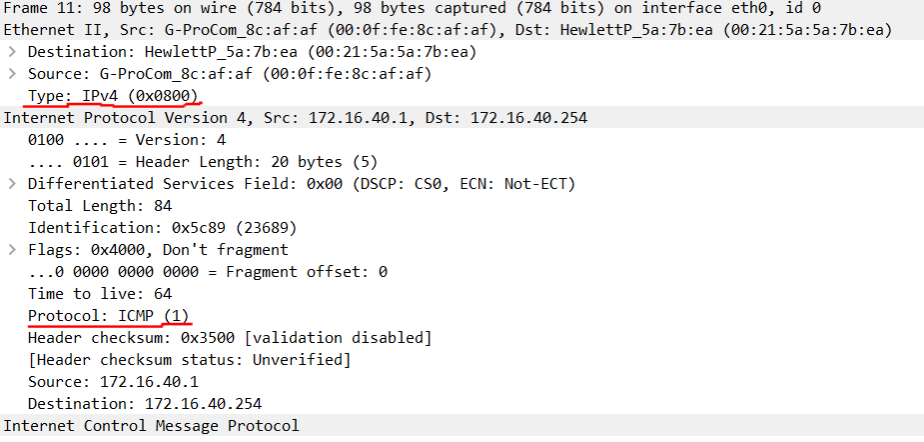


Figura 2- Pacotes ICMP de 'request' e 'reply'

O pacote ICMP de ‘request’ tem como endereços de origem os do tux que executou o comando ping e endereços de destino os do tux que se quer contactar. O pacote ICMP de ‘reply’ tem os mesmos endereços que o pacote de ‘request’, sendo que os endereços de origem e de destino estão trocados.

### 2.1.5 – Como determinar se uma trama Ethernet recebida é do tipo ARP, IP, ICMP?

Para determinar o tipo de trama Ethernet é necessário analisar o seu cabeçalho, que contém campo ‘EtherType’. Um valor de 0x0806 neste campo representa uma trama do tipo ARP e um valor de 0x0800 representa uma trama do tipo IP. Pode-se averiguar se se trata de uma trama do tipo ICMP, caso se trate de uma trama IP e o seu cabeçalho contenha o valor 0x01 no campo respetivo ao protocolo.

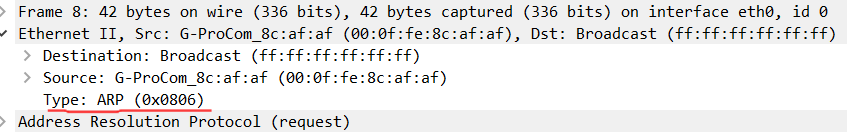


Figura 3- Trama Ethernet do tipo ARP

Figura 4- Trama Ethernet do tipo ICMP

### 2.1.6 – Como determinar o tamanho de uma trama recebida?

Visto que as tramas Ethernet não possuem nenhum campo com o tamanho da trama, para que o tamanho da mesma seja conhecido é necessário lê-la por completo. No entanto, se se tratar de uma trama IPv4, o seu cabeçalho possui um campo que armazena o tamanho total do pacote.

### 2.1.7 – O que é a interface de loopback e porque é que esta é tão importante?

A interface de loopback é uma interface de rede virtual que permite que um cliente e um servidor comuniquem entre si através de protocolos TCP/IP, sendo que ambos se localizam no mesmo computador. O endereço IP comummente utilizado para esta interface é **127.0.0.1**.

Esta interface é relevante no contexto da execução de testes de software e conectividade.

## 2.2 – Implementação de LAN’s virtuais num switch

### 2.2.1 – Como configurar a vlany0?

Para criar a vlany0 é necessário inserir os seguintes comandos no *GTKTerm:*

* configure terminal
* vlan y0
* end

Para associar as várias portas dos tuxes à vlany0:

* configure terminal
* interface fastethernet 0/[número da porta]
* switchport mode access
* switchport access vlan y0
* end

### 2.2.2 – Quantos domínios de broadcast existem?

Existem dois domínios de broadcast, um por cada vlan, visto que o tuxy2 se encontra isolado na vlany1 que, aquando desta experiência, não possui qualquer conexão com a vlany0.

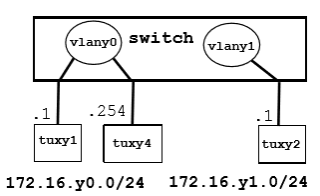


Figura 5- Configuração da Rede

## 2.3 – Configuração de um router em Linux

### 2.3.1 – Que rotas existem nos tuxes e quais os seus significados?

Tuxy1:

* ***172.16.y1.0/24 via 172.16.y0.254 dev eth0*** - esta rota significa que qualquer comunicação com a vlany1 será efetuada utilizando o tuxy4 como intermediário.
* ***172*.16.y0.0/24 dev eth0 –** tuxy1 está conectado à rede 172.16.y0.0/24 através da interface eth0.

Tuxy2:

* ***172.16.y0./24 via 172.16.y1.253 dev eth0*** - esta rota significa que qualquer comunicação com a vlany0 será efetuada utilizando o tuxy4 como intermediário.
* ***172*.16.y1.0/24 dev eth0 –** tuxy1 está conectado à rede 172.16.y1.0/24 através da interface eth0.

Tuxy4:

* ***172*.16.y0.0/24 dev eth0 –** tuxy4 está conectado à rede 172.16.y0.0/24 através da interface eth0.
* ***172*.16.y1.0/24 dev eth1 –** tuxy4 está conectado à rede 172.16.y1.0/24 através da interface eth1.

### 2.3.2 – Que informação está contida numa entrada da tabela de forwarding?

A informação principal contida numa entrada da tabela de forwarding é a seguinte:

* **Destino:** representa o endereço do destino final do pacote;
* **Gateway:** endereço para onde o pacote será redirecionado de modo a atingir o seu destino final;
* **Netmask:** representa a mascara de rede da rede de destino, que, associando-a ao endereço de destino, permite determinar o ID da rede de destino;
* **Metric:** representa o custo que está associado à rota. Esta informação é útil para calcular a rota mais eficiente entre dois pontos da rede;
* **Interface:** representa a interface responsável por atingir o gateway.

### 2.3.3 – Que mensagens ARP e endereços MAC associados são observados e porquê?

Na tentativa de efetuar um comando de ping do tuxy1 para o tuxy2, este é redirecionado através do tuxy4. O tuxy1 envia um pacote ARP de modo a conhecer o endereço físico do tuxy4 e de seguida, o tuxy4 envia também um pacote ARP para conhecer o endereço físico do tuxy2.

### 2.3.4 – Que pacotes ICMP são observados e porquê? Quais são os endereços IP e MAC associados aos pacotes ICMP e porquê?

Os pacotes ICMP enviados quando se efetua o comando ping do tuxy1 para o tuxy2 comportam-se como se não existisse o tuxy4 a servir de intermediário entre os dois tuxes: tratam-se simplesmente de pacotes ICMP com origem no endereço 172.16.y0.1 e com destino no endereço 172.16.y1.1.

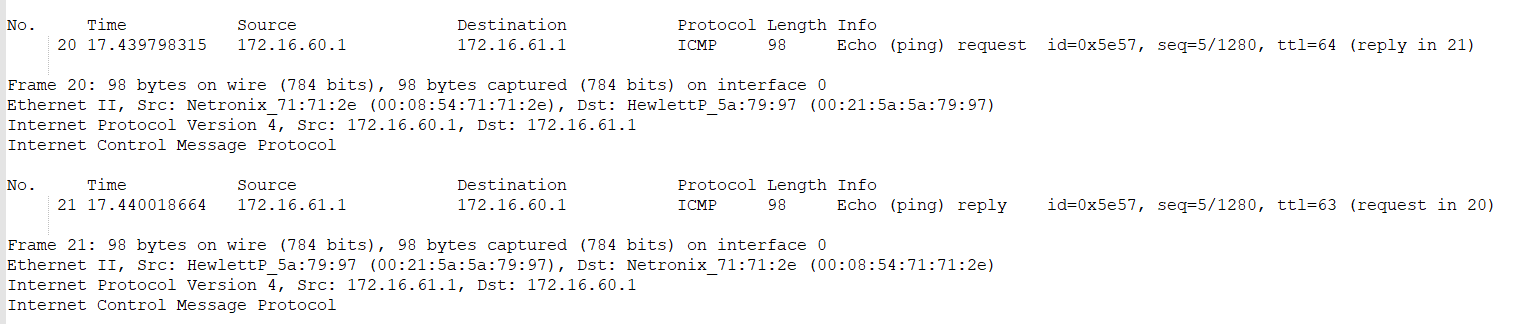


Figura 6- Pacotes ICMP do tuxy1 para o tuxy2

## 2.4 – Configuração de um router comercial e implementação de NAT

### 2.4.1 – Como configurar uma rota estática num router comercial?

Para configurar uma rota estática num router comercial basta usar o seguinte comando: **ip route [destino] [mascara] [gateway]**

### 2.4.2 – Quais os caminhos percorridos pelos pacotes e porquê?

O tuxy4 é o router padrão do tuxy1 e o router comercial é o router padrão do tuxy2 e do tuxy4. Deste modo, qualquer ligação entre a vlany0 e vlany1 será feita utilizando o tuxy4 como intermediário, como acontecia previamente. Qualquer tentativa de ligação à rede do laboratório (172.16.1.0/24) é sempre redirecionada para o router comercial, visto que este está definido como router padrão de ambos os tuxes da vlany1.

Após a remoção da rota 172.16.y0.0/24 via tuxy4, qualquer ligação da vlany1 com a vlany0 não será efetuada diretamente pelo tuxy4, mas sim pelo router e só depois pelo tuxy4.

### 2.4.3 – Como configurar NAT num router comercial?

O NAT foi configurado conforme as instruções fornecidas no guião:

***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***

### 2.4.4 – O que faz o NAT?

O NAT (*Network Address Translation*) tem como função traduzir endereços privados de uma rede interna para endereços públicos, de forma a que seja possível existir uma ligação à internet.

## 2.5 – DNS

Nesta experiência foi configurado o DNS (Domain Name System) nos tuxes 1, 2 e 4. Um servidor DNS armazena os endereços IP públicos e respetivos hostnames. É utilizado para traduzir os hostnames para os respetivos endereços IP. O servidor DNS foi, nesta experiência, **netlab.fe.up.pt**.

### 2.5.1 – Como configurar o serviço de DNS num host?

A configuração do serviço DNS consiste em alterar o ficheiro **resolv.conf**, localizado em **/etc**. Esse ficheiro passa a conter a seguinte informação:

*search netlab.fe.up.pt*

*nameserver 172.16.2.1*

netlab.fe.up.pt corresponde ao nome do servidor DNS e 172.16.2.1 ao seu endereço de IP. Realizada esta experiência, é possível aceder à internet nos tuxes.

### 2.5.2 – Que pacotes são trocados pelo DNS e que informação contêm?

Em primeiro lugar, o Host envia dois pacotes para o Server (No. 6 e 7) com o hostname desejado, pedindo os seus endereços de IP, IPV4 e IPV6, respetivamente **(Anexo III).**

O servidor envia, como resposta (no. 8 e 9), dois pacotes com a informação pedida pelo Host.

## 2.6 – Ligações TCP

Nesta experiência observou-se o comportamento do protocolo TCP, recorrendo à aplicação desenvolvida na primeira parte do trabalho.

### 2.6.1 – Quantas ligações TCP são abertas pela aplicação FTP? Em que ligação é transportada a informação de controlo FTP?

A aplicação FTP abre 2 conexões TCP, a conexão de controlo, para enviar os comandos FTP ao servidor e receber as respostas, e a conexão de dados, para receber os dados enviados pelo servidor.

### 2.6.3 – Quais as fases de uma ligação FTP?

Uma conexão TCP tem três fases: o estabelecimento da conexão, troca de dados e encerramento da conexão. As fases de estabelecimento e encerramento foram monitorizadas no Wireshark (**Anexo IV**).

### 2.6.4 – Como é que funciona o mecanismo ARQ TCP? Quais são os campos TCP relevantes?

O TCP utiliza um mecanismo de ARQ do tipo Go-Back-N com recurso a janela deslizante, utilizando mensagens de confirmação (ACKs) para indicar que as tramas foram enviadas corretamente e que o número de sequência é o esperado.

### 2.6.5 – Como é que funciona o mecanismo de controlo de congestão TCP? Quais os campos relevantes?

O mecanismo de congestão TCP utiliza uma janela de congestão que limita o número de bytes que podem ser enviados a cada instante. O tamanho desta janela é determinado através de uma estimativa resultante da análise da congestão da ligação.

Este mecanismo implementa também o algoritmo conhecido como Additive Increase/Multiplicative Decrease (*AIMD*) que faz com que a janela aumente linearmente conforme a congestão diminui e com que ela diminuía exponencialmente quando é detetada congestão

### 2.6.6 – O aparecimento de uma segunda ligação TCP afeta o fluxo de dados de uma ligação TCP já existente? De que forma?

Ao realizar uma segunda conexão TCP, a existência de uma transferência de dados em simultâneo pode provocar uma queda na taxa de transmissão, visto que a taxa de transferência é distribuída irmãmente para cada ligação.

Nas imagens do Anexo V , que representam a monitorização do Wireshark, inicializada em momentos separados, no tuxy1 e no tuxy2, ilustram as transferências do mesmo ficheiro, utilizando a aplicação desenvolvida, tendo a transferência do tuxy2 começado, aproximadamente, 8 segundos após a do tux1. Como evidenciam os gráficos, o número de pacotes transferidos por unidade de tempo no tuxy1 diminuem com o início da transferência do tuxy2 e, assim que o tuxy1 termina a transferência, o número de pacotes transferidos por unidade de tempo no tuxy2 aumenta.

# Conclusão

Este projeto foi útil de forma a consolidar a aprendizagem dos conceitos lecionados de modo a que promoveu o contacto direto com os conteúdos da mesma.

Em conclusão, todos os objetivos do projeto foram concluídos com sucesso, quer no que corresponde à configuração da rede, quer no que corresponde ao desenvolvimento da aplicação de transferência de ficheiros.

# Anexo I – Código fonte

## clientTCP.h

*#ifndef* CLIENT\_TCP\_H

*#define* CLIENT\_TCP\_H

int initSocket(int server\_port, char \*server\_address);

*#endif //CLIENT\_TCP\_H*

## clientTCP.c

*/\*      (C)2000 FEUP  \*/*

*#include* <stdio.h>

*#include* <sys/types.h>

*#include* <sys/socket.h>

*#include* <netinet/in.h>

*#include* <arpa/inet.h>

*#include* <stdlib.h>

*#include* <unistd.h>

*#include* <signal.h>

*#include* <netdb.h>

*#include* <string.h>

int initSocket(int server\_port, char \*server\_address){

    int sockfd;

    struct  sockaddr\_in server\_addr;

*// char buf[] = "Mensagem de teste na travessia da pilha TCP/IP\n";*

*// int  bytes;*

*/\*server address handling\*/*

    bzero((char\*)&server\_addr,sizeof(server\_addr));

    server\_addr.sin\_family = AF\_INET;

    server\_addr.sin\_addr.s\_addr = inet\_addr(server\_address);    */\*32 bit Internet address network byte ordered\*/*

    server\_addr.sin\_port = htons(server\_port);              */\*server TCP port must be network byte ordered \*/*

*/\*open an TCP socket\*/*

*if* ((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);

    }

*/\*send a string to the server\*/*

*// bytes = write(sockfd, buf, strlen(buf));*

*// printf("Bytes escritos %d\n", bytes);*

*return* sockfd;

}

## getip.h

*#ifndef* GET\_IP\_H

*#define* GET\_IP\_H

int getIPByName(char \*name, char \*ip);

*#endif //GET\_IP\_H*

## getip.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* "string.h"

*#include* "getip.h"

int getIPByName(char \*name, char \*ip)

{

    struct hostent \*h;

*/\**

*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.*

*\*/*

*if* ((h = gethostbyname(name)) == NULL)

    {

        herror("gethostbyname");

*return* -1;

    }

    printf("Host name  : %s\n", h->h\_name);

    printf("IP Address : %s\n", inet\_ntoa(\*((struct in\_addr \*)h->h\_addr)));

    strcpy(ip, inet\_ntoa(\*((struct in\_addr \*)h->h\_addr)));

*return* 0;

}

## downloadApp.c

*#include* <stdio.h>

*#include* <string.h>

*#include* <stdlib.h>

*#include* <unistd.h>

*#include* <ctype.h>

*#include* <sys/stat.h>

*#include* <fcntl.h>

*#include* <errno.h>

*#include* "getip.h"

*#include* "clientTCP.h"

*#define* DEFAULT\_SERVER\_FTP\_PORT 21

*#define* BUFFER\_MAX\_SIZE 255

*#define* STR\_MAX\_LEN 63

*#define* TRUE 1

*#define* FALSE 0

*#define* EXTENSION\_MAX\_SIZE 32

*#define* FILE\_NAME\_MAX\_SIZE 255

*//error codes*

*#define* REPLY\_ERROR -1

*#define* INVALID\_ARG\_N 1

*#define* INVALID\_ARG 2

*#define* HOST\_IP\_NOT\_FOUND 3

*#define* SRV\_NOT\_READY 4

*#define* LOGIN\_ERROR   5

*#define* FILE\_RETR\_ERROR 6

*#define* max(a, b)               \

    ({                          \

        \_\_typeof\_\_(a) \_a = (a); \

        \_\_typeof\_\_(b) \_b = (b); \

        \_a > \_b ? \_a : \_b;      \

    })

*#define* min(a, b)               \

    ({                          \

        \_\_typeof\_\_(a) \_a = (a); \

        \_\_typeof\_\_(b) \_b = (b); \

        \_a < \_b ? \_a : \_b;      \

    })

enum ReplyType

{

    WAIT\_FOR\_REPLY = 1,

*//1yz Positive Preliminary reply:*

*//The requested action is being initiated; expect another reply before proceeding with a new command.*

    ACTION\_COMPLETE = 2,

*//2yz Positive Completion reply:*

*//The requested action has been successfully completed. A new request may be initiated.*

    AWAITING\_INFO\_CMD = 3,

*//3yz Positive Intermediate reply:*

*//The command has been accepted, but the requested action is being held in abeyance, pending receipt of further information.*

*//The user should send another command specifying this information. This reply is used in command sequence groups.*

    RETRY\_CMD\_SEQUENCE = 4,

*//4yz Transient Negative Completion reply:*

*//The command was not accepted and the requested action did not take place, but the error condition is temporary and*

*//the action may be requested again. The user should return to the beginning of the command sequence, if any.*

    BAD\_CMD = 5,

*//5yz Permanent Negative Completion reply:*

*//The command was not accepted and the requested action did not take place. The User-process is discouraged from*

*//repeating the exact request (in the same sequence).*

};

void establishFTPConnection(int socketfd, char \*user, char \*pass);

int createReceivedFile(char \*file\_name);

void createFile(int socketDownload, char \*filename);

void sendRETRCmd(int socketfd, int socketfdDownload, char \*path);

int getNumberBeforeChar(char \*string, char c);

void handlePASVReply(int socketfd, char \*ip, int \*port);

void readReply(int socketfd, char \*replyCode, char \*info, int useInfo);

void readReplyNoInfo(int socketfd, char \*replyCode);

void readReplyWithInfo(int socketfd, char \*replyCode, char \*info);

int parseArguments(char \*url, char \*user, char \*pass, char \*host, char \*path);

int getLastCharPos(char \*str, char c);

void getFileNameFromPath(char \*path, char \*filename);

void sendFTPCmd(int sockfd, char \*cmd\_code, char \*cmd\_argument);

enum ReplyType sendCommandProcessReply(int sockfd, char \*cmd\_code, char \*cmd\_argument, int canWaitForExtraReplies, int canRetryCommand);

void replyFailureTest(enum ReplyType reply\_type, int socketfd);

int main(int argc, char \*\*argv)

{

*if* (argc != 2)

    {

        printf("Usage:\tdownload url\n\tex: download ftp://anonymous:1@speedtest.tele2.net/1KB.zip\n");

        exit(INVALID\_ARG\_N);

    }

*//parse argument*

    char user[STR\_MAX\_LEN];

    memset(user, 0, STR\_MAX\_LEN);

    char pass[STR\_MAX\_LEN];

    memset(pass, 0, STR\_MAX\_LEN);

    char host[STR\_MAX\_LEN];

    memset(host, 0, STR\_MAX\_LEN);

    char path[STR\_MAX\_LEN];

    memset(path, 0, STR\_MAX\_LEN);

*if* (parseArguments(argv[1], user, pass, host, path))

    {

        exit(INVALID\_ARG);

    }

    char host\_ip[STR\_MAX\_LEN];

    memset(host\_ip, 0, STR\_MAX\_LEN);

*if* (getIPByName(host, host\_ip))

        exit(HOST\_IP\_NOT\_FOUND);

    printf(" - Username:  %s\n", user);

    printf(" - Password:  %s\n", pass);

    printf(" - Host:      %s\n", host);

    printf(" - HOST IP:   %s\n", host\_ip);

    printf(" - Path:      %s\n", path);

*//abrir socket user - server*

    int socketfd = initSocket(DEFAULT\_SERVER\_FTP\_PORT, host\_ip);

*//receive connection greetings*

    char greetingsCode[4];

    memset(greetingsCode, 0, 4);

    readReplyNoInfo(socketfd, greetingsCode);

*if* (strcmp(greetingsCode, "220"))

    {

        close(socketfd);

*if* (greetingsCode[0] == '1')

            printf("server expecting delay, try later");

*else*

            printf("server not ready");

        exit(4);

    }

    printf("Greetings code - %s\n", greetingsCode);

    printf("Establishing connection\n");

*// establish connection*

    establishFTPConnection(socketfd, user, pass);

*//entrada em modo passivo - pasv*

    printf("Server entering passive mode\n");

*// reply\_type = sendCommandProcessReply(socketfd, "pasv", "", FALSE, TRUE);*

*// replyFailureTest(reply\_type, socketfd);*

    sendFTPCmd(socketfd, "pasv", "");

    int download\_port = 0;

    char download\_ip[STR\_MAX\_LEN];

    memset(download\_ip, 0, STR\_MAX\_LEN);

    handlePASVReply(socketfd, download\_ip, &download\_port);

    printf(" - Download Port: %d\n", download\_port);

    printf(" - Download IP:   %s\n", download\_ip);

*// abrir socket server user*

    int socketfdDownload = initSocket(download\_port, download\_ip);

*// retrieve file - retr*

    sendRETRCmd(socketfd, socketfdDownload, path);

    close(socketfdDownload);

*//close connection*

    enum ReplyType reply\_type = sendCommandProcessReply(socketfd, "quit", "", FALSE, TRUE);

    replyFailureTest(reply\_type, socketfd);

    close(socketfd);

    printf("Connection ended successfuly\n");

*return* 0;

}

void establishFTPConnection(int socketfd, char \*user, char \*pass){

    enum ReplyType reply\_type;

*do*

    {

*// username - user*

        printf("Sending username\n");

        reply\_type = sendCommandProcessReply(socketfd, "user", user, FALSE, TRUE);

*if* (reply\_type == AWAITING\_INFO\_CMD)

        {

*// password - pass*

            printf("Sending password\n");

            reply\_type = sendCommandProcessReply(socketfd, "pass", pass, FALSE, FALSE);

*if* (reply\_type != RETRY\_CMD\_SEQUENCE)

                replyFailureTest(reply\_type, socketfd);

        }

*else*

        {

            replyFailureTest(reply\_type, socketfd);

        }

    } *while* (reply\_type != ACTION\_COMPLETE);

}

int createReceivedFile(char \*file\_name){

  int file;

  int i = 1;

  char extension[EXTENSION\_MAX\_SIZE];

  char file\_name\_no\_extension[FILE\_NAME\_MAX\_SIZE];

  int last\_dot\_pos = getLastCharPos(file\_name, '.');*//position of last '.'*

*if* (last\_dot\_pos > 0)*//if it has an extension*

  {

    strcpy(extension, file\_name + last\_dot\_pos); *//get extension*

    memcpy(file\_name\_no\_extension, file\_name, last\_dot\_pos); *//get file name without extension*

    file\_name\_no\_extension[last\_dot\_pos] = '\0';

  }

*else* strcpy(file\_name\_no\_extension, file\_name);

*while* ((file = open(file\_name, O\_WRONLY | O\_CREAT | O\_EXCL | O\_APPEND, 0777)) < 0)

  {

*if* (errno == EEXIST)

    {

*if* (last\_dot\_pos > 0) *//if it has extension*

        sprintf(file\_name, "%s(%d)%s", file\_name\_no\_extension, i, extension);

*else*

        sprintf(file\_name, "%s(%d)", file\_name\_no\_extension, i);

*if* (strlen(file\_name) < FILE\_NAME\_MAX\_SIZE){

*//if the size does not exceed the maximum file*

        i++;

      }

*else* {

        perror("file name too big\n");

        exit(-1);

      }

    }

*else*

    {

      perror("error on creating file\n");

      exit(-1);

    }

  }

*return* file;

}

void createFile(int socketDownload, char \*filename){

    int fd = createReceivedFile(filename);

    char buffer[BUFFER\_MAX\_SIZE];

    int bytes;

*while* ( (bytes = read(socketDownload, buffer, BUFFER\_MAX\_SIZE) ) > 0){

        write(fd, buffer, bytes);

    }

    printf("download has ended successfuly\n");

    close(fd);

}

void sendRETRCmd(int socketfd, int socketfdDownload, char \*path){

    enum ReplyType reply\_type = sendCommandProcessReply(socketfd, "retr", path, FALSE, TRUE);

*if* (reply\_type == WAIT\_FOR\_REPLY)

    {

        char filename[STR\_MAX\_LEN];

        memset(filename, 0, STR\_MAX\_LEN);

        getFileNameFromPath(path, filename);

        printf(" - File Name: %s\n", filename);

        createFile(socketfdDownload, filename);

        char replyCode[4];

        memset(replyCode, 0, 4);

        readReplyNoInfo(socketfd, replyCode);

        printf("%s - Download Reply Code\n", replyCode);

    }

*else*

    {

        close(socketfdDownload);

        close(socketfd);

        printf("error on retrieving file\n");

        exit(FILE\_RETR\_ERROR);

    }

}

int getNumberBeforeChar(char \*string, char c){

    unsigned str\_len = strlen(string);

    int number = 0;

*for* (unsigned i = 0; i < str\_len; i++){

        char curr\_char = string[i];

*if* (curr\_char == c)

*return* number;

*else* *if* (isdigit(curr\_char)){

            number = number \* 10 + (curr\_char - '0');

        }

*else* *break*;

    }

*return* -1;

}

void handlePASVReply(int socketfd, char \*ip, int \*port){

    char buffer[BUFFER\_MAX\_SIZE];

    memset(buffer, 0, BUFFER\_MAX\_SIZE);

    char replyCode[4];

    memset(replyCode, 0, 4);

    readReplyWithInfo(socketfd, replyCode, buffer);

    printf("Code - %s PASV reply - %s\n", replyCode, buffer);

    enum ReplyType reply\_type = replyCode[0] - '0';

    replyFailureTest(reply\_type, socketfd);

    unsigned i = 0;

*while* (buffer[i++] != '(');

    unsigned numbers\_processed\_counter = 0;

    unsigned index = 0;

*while* (numbers\_processed\_counter < 4)

    {

        char curr\_char = buffer[i++];

*if* (curr\_char == ',')

        {

            numbers\_processed\_counter++;

*if* (numbers\_processed\_counter < 4)

            {

                ip[index] = '.';

            }

        }

*else*

            ip[index] = curr\_char;

        index++;

    }

    int first\_byte = getNumberBeforeChar(buffer + i, ',');

*// printf("%d\n", first\_byte);*

*while* (buffer[i++] != ',');

    int second\_byte = getNumberBeforeChar(buffer + i, ')');

*// printf("%d\n", second\_byte);*

    \*port = first\_byte \* 256 + second\_byte;

*// printf("%d\n", \*port);*

}

void replyFailureTest(enum ReplyType reply\_type, int socketfd){

*if* (reply\_type != ACTION\_COMPLETE){

        printf("Login error - Reply Code = %d\n", reply\_type);

        close(socketfd);

        exit(LOGIN\_ERROR);

    }

}

void sendFTPCmd(int sockfd, char \*cmd\_code, char \*cmd\_argument)

{

    write(sockfd, cmd\_code, strlen(cmd\_code));

    unsigned int arg\_length = strlen(cmd\_argument);

*if* (arg\_length > 0)

    {

        write(sockfd, " ", 1);

        write(sockfd, cmd\_argument, arg\_length);

    }

    write(sockfd, "\n", 1);

}

enum ReplyType sendCommandProcessReply(int sockfd, char \*cmd\_code, char \*cmd\_argument, int canWaitForExtraReplies, int canRetryCommand)

{

    sendFTPCmd(sockfd, cmd\_code, cmd\_argument);

    char reply[4];

    memset(reply, 0, 4);

    enum ReplyType reply\_type;

    int stop = FALSE;

*while* (!stop)

    {

        readReplyNoInfo(sockfd, reply);

        reply\_type = reply[0] - '0';

*switch* (reply\_type)

        {

*case* WAIT\_FOR\_REPLY:

*if* (!canWaitForExtraReplies)

                stop = TRUE;

*break*;

*case* ACTION\_COMPLETE:

*case* AWAITING\_INFO\_CMD:

*case* BAD\_CMD:

            stop = TRUE;

*break*;

*case* RETRY\_CMD\_SEQUENCE:

*if* (canRetryCommand)

                sendFTPCmd(sockfd, cmd\_code, cmd\_argument);

*else*

                stop = TRUE;

*break*;

        }

    }

*return* reply\_type;

}

void readReply(int socketfd, char \*replyCode, char \*info, int useInfo)

{

    enum State

    {

        CODE,

        MIDDLE,

        LASTLINE,

        END

    };

    enum State state = CODE;

    char curr\_char;

    int i = 0;

    int info\_index = 0;

*while* (state != END)

    {

        read(socketfd, &curr\_char, 1);

*switch* (state)

        {

*case* CODE:

*if* (curr\_char == ' ')

            {

*if* (i != 3)

                {

                    printf("Error on receiving reply code\n");

                    exit(REPLY\_ERROR);

                }

                state = LASTLINE;

            }

*else* *if* (curr\_char == '-')

            {

*if* (i != 3)

                {

                    printf("Error on receiving reply code\n");

                    exit(REPLY\_ERROR);

                }

                state = MIDDLE;

                i = 0;

            }

*else*

            {

*if* (isdigit(curr\_char))

                {

                    replyCode[i] = curr\_char;

                    i++;

                }

*else*

                {

                    i = 0;

                }

            }

*break*;

*case* MIDDLE:

*if* (i == 3 && curr\_char == ' ')

                state = LASTLINE;

*else* *if* (replyCode[i] == curr\_char)

                i++;

*else*

            {

*if* (useInfo)

                {

                    info[info\_index++] = curr\_char;

                }

                i = 0;

            }

*break*;

*case* LASTLINE:

*if* (curr\_char == '\n')

                state = END;

*else* *if* (useInfo)

            {

                info[info\_index++] = curr\_char;

            }

*break*;

*default*:

*break*;

        }

    }

}

void readReplyWithInfo(int socketfd, char \*replyCode, char \*info){

    readReply(socketfd, replyCode, info, TRUE);

}

void readReplyNoInfo(int socketfd, char \*replyCode)

{

    readReply(socketfd, replyCode, NULL, FALSE);

}

int parseArguments(char \*url, char \*user, char \*pass, char \*host, char \*path)

{

    enum State

    {

        PREFIX,

        USERNAME,

        PASSWORD,

        HOST,

        PATH

    };

    enum State state = PREFIX;

    char prefix[] = "ftp://";

    unsigned int prefix\_length = strlen(prefix);

    unsigned int url\_length = strlen(url);

    unsigned int i = 0;

    unsigned int curr\_field\_index = 0;

*while* (i < url\_length)

    {

        char curr\_char = url[i];

*switch* (state)

        {

*case* PREFIX:

*if* (i == prefix\_length - 1)

            {

                state = USERNAME;

            }

*if* (prefix[i] != curr\_char)

*return* -1;

*break*;

*case* USERNAME:

*if* (curr\_char == '@')

            {

*//EMPTY USERNAME, NO PASSWORD*

                state = HOST;

                curr\_field\_index = 0;

            }

*else* *if* (curr\_char == ':')

            {

                state = PASSWORD;

                curr\_field\_index = 0;

            }

*else*

            {

                user[curr\_field\_index] = url[i];

                curr\_field\_index++;

            }

*break*;

*case* PASSWORD:

*if* (curr\_char == '@')

            {

                state = HOST;

                curr\_field\_index = 0;

            }

*else*

            {

                pass[curr\_field\_index] = url[i];

                curr\_field\_index++;

            }

*break*;

*case* HOST:

*if* (curr\_char == '/')

            {

                state = PATH;

                curr\_field\_index = 0;

            }

*else*

            {

                host[curr\_field\_index] = url[i];

                curr\_field\_index++;

            }

*break*;

*case* PATH:

            path[curr\_field\_index] = url[i];

            curr\_field\_index++;

*break*;

*default*:

*break*;

        }

        i++;

    }

*return* 0;

}

int getLastCharPos(char \*str, char c)

{

    unsigned int str\_length = strlen(str);

*for* (int i = str\_length - 1; i >= 0; i--)

    {

*if* (str[i] == c)

*return* i;

    }

*return* -1;

}

void getFileNameFromPath(char \*path, char \*filename)

{

    int pos = getLastCharPos(path, '/');

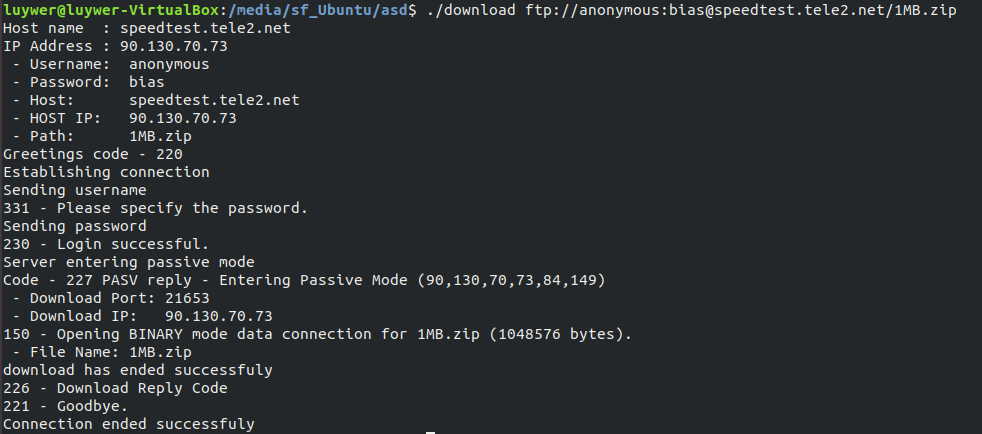
    unsigned int path\_length = strlen(path);

    unsigned int filename\_length = path\_length - (pos + 1);

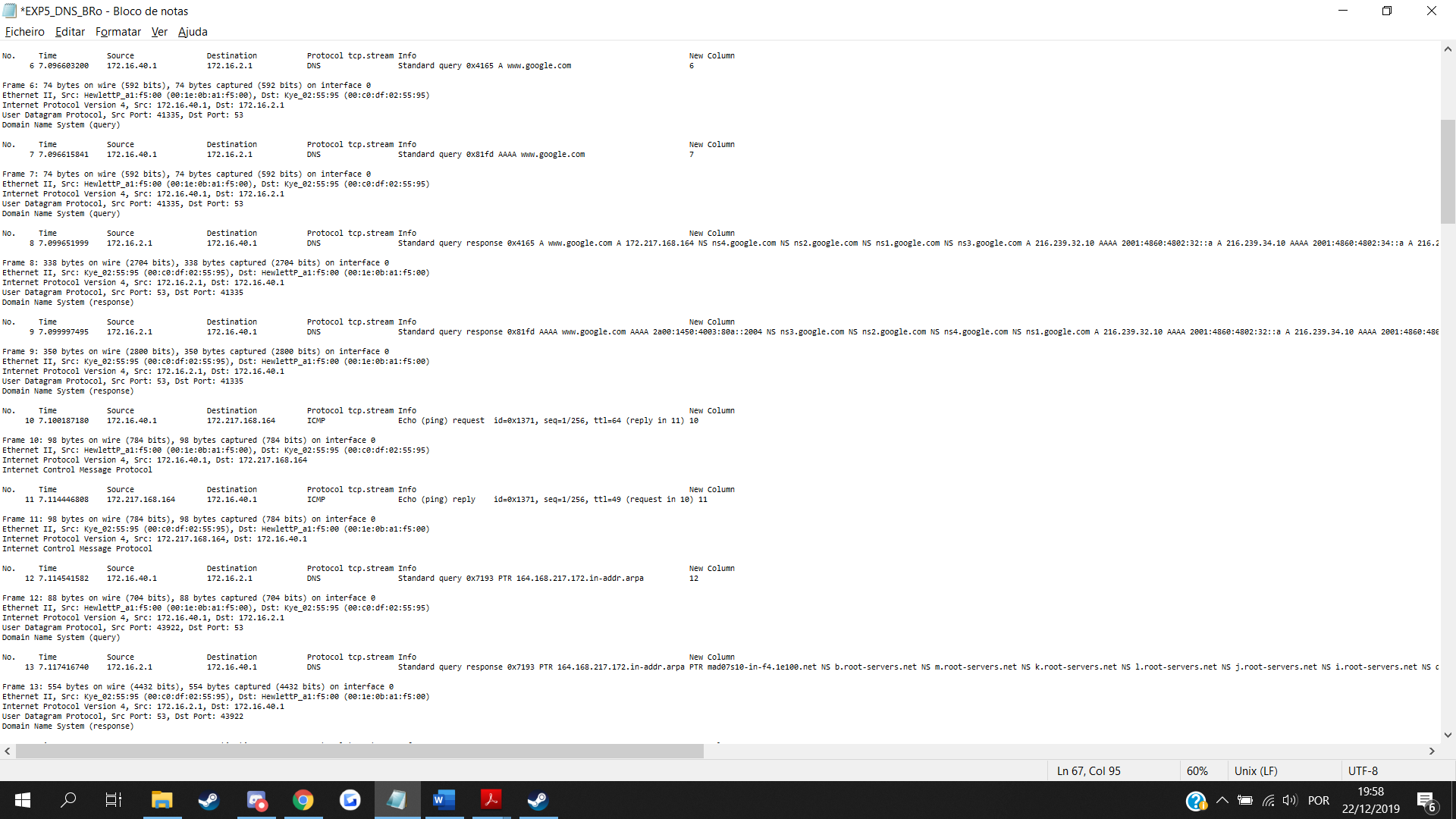
    memcpy(filename, path + max(pos + 1, 0), filename\_length);

}

# Anexo II – Execução da aplicação de download

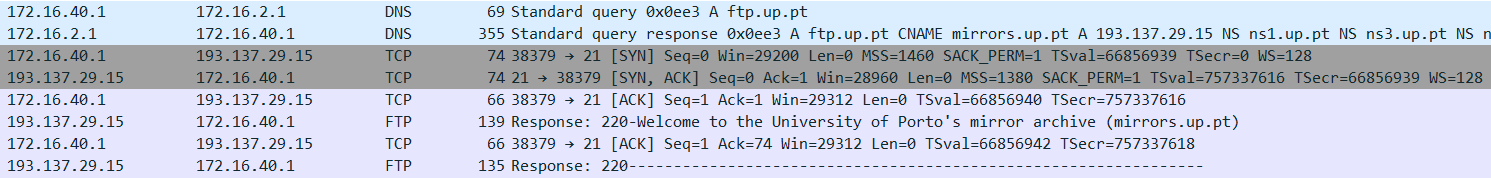


# Anexo III – DNS

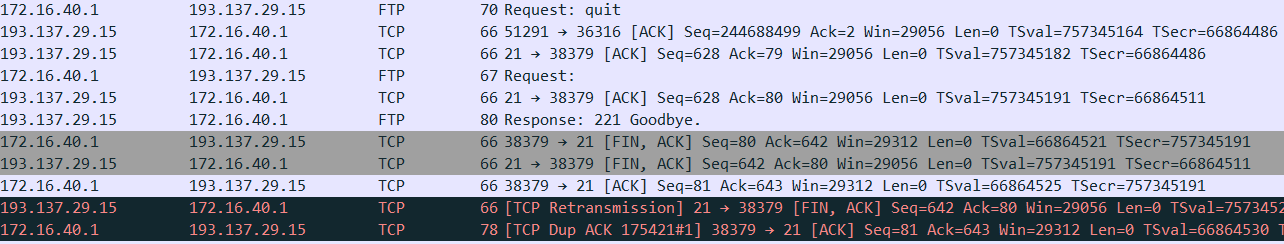


# Anexo IV – TCP Connection Phases

## Connection established

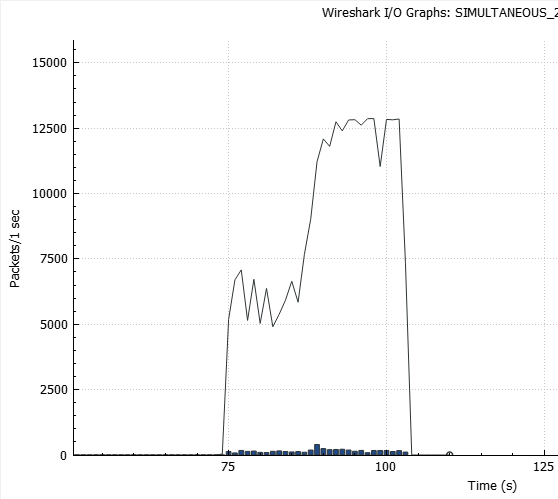


## Connection terminated



# Anexo V – Graphs

## Tux 2



## Tux 1

