

Redes de Computadores

*2º Trabalho Laboratorial*

*Mestrado Integrado em Engenharia Informática e Computação*

*(22 de dezembro de 2017)*

Bárbara Silva  **up201505628**@fe.up.pt

Catarina Ferreira **up201506671**@fe.up.pt

Julieta Frade **up201506530**@fe.up.pt

Sumário

Este relatório foi elaborado no âmbito da unidade curricular de Redes de Computadores, e trata-se de uma complementação ao segundo trabalho laboratorial, cuja essência é redes de computadores. O trabalho consiste na configuração e estudo da mesma, em que foram utilizados comandos de configuração do Router Cisco e do Cisco Switch, e no desenvolvimento de uma aplicação de download de um ficheiro de acordo com o protocolo FTP (File Transfer Protocol).

Isto posto, o trabalho foi concluído com sucesso, visto que todos os objetivos estabelecidos foram cumpridos e foi finalizada uma aplicação capaz de transferir um ficheiro, assim como a rede foi configurada corretamente.

Índice

[Introdução 3](#_Toc501584402)

[Parte 1: Aplicação de Download 3](#_Toc501584403)

[Arquitetura 3](#_Toc501584404)

[Parte 2: Configuração de Rede e Análise 4](#_Toc501584405)

[Experiência 1 – Configurar um IP de rede 4](#_Toc501584406)

[Experiência 2 – Implementar duas LAN’s virtuais no switch 6](#_Toc501584407)

[Experiência 3 – Configurar um router em Linux 6](#_Toc501584408)

[Experiência 4 – Configurar um router comercial e implementar o NAT 8](#_Toc501584409)

[Experiência 5 – DNS 9](#_Toc501584410)

[Experiência 6 – Conexões TCP 9](#_Toc501584411)

[Conclusões 10](#_Toc501584412)

[Referências 11](#_Toc501584413)

[Anexos 12](#_Toc501584414)

# *Introdução*

O trabalho tem duas grandes finalidades: a configuração de uma rede e o desenvolvimento de uma aplicação de download.

Relativamente à configuração de uma rede, o seu objetivo é permitir a execução de uma aplicação, a partir de duas VLAN’s dentro de um switch. De seguida, foi desenvolvida uma aplicação download de acordo com o protocolo FTP e com a ajuda de ligações TCP (Transmission Control Protocol) a partir de sockets.

Quanto ao relatório, o seu objetivo é expor e explicar toda a componente teórica presente neste segundo trabalho, tendo a seguinte estrutura:

* **Parte 1: Aplicação de Download**

Arquitetura da aplicação de download e respetivos resultados.

* **Parte 2: Configuração de Rede e Análise**

Análise de cada experiência.

* **Conclusões**

Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

# *Parte 1: Aplicação de Download*

A primeira parte deste trabalho foi desenvolver uma aplicação download na linguagem de programação C que aceita um link como argumento **ftp://<user>:<password>@<host>/<url-path>**. Esta aplicação consegue fazer o download de qualquer tipo de ficheiros de um servidor FTP. Para isso foi estudado o RFC959 que fala sobre o FTP e o RFC1738 que fala sobre o tratamento de informação provenientes de URL’s.

## **Arquitetura**

Primeiramente é feito o processamento do URL. Para tal, é reservado espaço para as variáveis *user*, *password*, *host* e *path* e depois é chamada a função ***parseArguments*** para obter estas mesmas variáveis a partir do URL. O nome do ficheiro é obtido a partir do *path* e o *ip address* a partir da função ***getip*** (código fornecido). A porta usada é sempre a 21.

Uma das funções principais é a ***sendCommandInterpretResponse*** que chama a função ***readResponse*** (obtém o código de resposta enviado pelo servidor) e analisa a resposta. O primeiro digito diz se a resposta é positiva (1, 2, 3) ou negativa (4 e 5). Caso o primeiro digito seja 1, é chamado a ***readResponse*** novamente para ser lida outra resposta, caso seja 2 e 3 a função retorna, caso seja 4 tenta ler-se novamente a resposta e caso seja 5 o programa termina.

Após ser aberto o *socket* pelo qual será feita a primeira conexão entre o cliente e o servidor, são enviados os comandos **USER user** e **PASS pass** para ser feito o login. Depois, é feita a entrada em modo passivo pela chamada do comando **PASV**, que vai retornar a porta necessária para a abertura de um outro *socket* que servirá para troca de dados. É enviado o comado **RETR** **filename** para pedir o ficheiro e seguidamente é feito o download do ficheiro com a ajuda da função ***createFile***. No final, são fechadas as duas conexões tanto a de transferência de comandos como a de dados.

**Resultados**

O nosso programa foi testado em diversas condições: modo anónimo, modo não anónimo, vários tipos de ficheiro, vários tamanhos do ficheiro entre outros. Este termina em caso de erro ou caso o ficheiro não exista. Todas as respostas são impressas na consola para maior controlo por parte do utilizador, o que pode ser consultado na figura 1.

# *Parte 2: Configuração de Rede e Análise*

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

O objetivo desta experiência é ligar o tux 1 ao tux4 utilizando switch.

1. O que são os pacotes ARP e para o que são usados?

O ARP (*Address Resolution Protocol*) é um protocolo de comunicação que serve para descobrir o endereço da camada de ligação associado ao endereço IPv4. Serve para mapear o endereço de rede a um endereço físico como o endereço MAC.

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

Quando fazemos *ping* do tux 1 para o tux 4, o tux 4 envia um pacote a perguntar quem é o tux com aquele IP, ou seja, a perguntar que endereço MAC tem o tux que lhe está a tentar mandar algo. Esta pergunta vem na forma de um pacote ARP com o endereço IP e endereço MAC do tux 4 (172.16.60.254 e 00:21:5a:c5:61:bb respetivamente) e com o endereço IP do tux target, ou seja, que se quer saber o MAC (172.16.60.1). Como não se sabe o MAC do tux target este está registado como 00:00:00:00:00:00. Deverá ser consultado os *logs* presentes na figura 2.

De seguida, o tux 1 responde a dizer que é ele que tem aquele IP enviando o seu endereço MAC.

No pacote de resposta presente na figura 3, o endereço IP e MAC da origem são o do tux 1 (172.16.60.1 e 00:0f:fe:8c:af:71 respetivamente) e o endereço IP e MAC do destino são o do tux 4 (172.16.60.254 e 00:21:5a:c5:61:bb).

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

O comando *ping* gera primeiro pacotes ARP para obter os endereços MAC e de seguida gera pacotes ICMP (*Internet Control Message Protocol*).

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

Quando damos *ping* ao tux 4 a partir do tux 1 os endereços (origem e destino) IP e MAC dos pacotes vão ser os destes tux’s.

**Pacote de pedido (figura 4):**

Endereço MAC da origem do pacote: 00:0f:fe:8c:af:71 (tux 1).

Endereço MAC do destino do pacote: 00:21:5a:c5:61:bb (tux 4).

Endereço IP da origem do pacote: 172.16.60.1 (tux 1).

Endereço IP do destino do pacote: 172.16.60.254 (tux 4).

**Pacote de resposta (figura 5):**

Endereço MAC da origem do pacote: 00:21:5a:c5:61:bb (tux 4).

Endereço MAC do destino do pacote: 00:0f:fe:8c:af:71 (tux 1).

Endereço IP da origem do pacote: 172.16.60.254 (tux 4).

Endereço IP do destino do pacote: 172.16.60.1 (tux 1).

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

Inspecionando o Ethernet header de um pacote conseguimos determinar o tipo da trama. Caso o tipo tiver o valor 0x0800, significa que o tipo da trama é IP, depois conseguimos analisar o IP header. Se o IP header tiver o valor 1 isso significa que o tipo de protocolo é ICMP. No entanto, se o tipo tiver o valor 0x0806, significa que o tipo da trama é ARP. Deverão ser consultadas as figuras 6 e 7.

1. Como determinar o comprimento de uma trama recetora?

O comprimento de uma trama recetora é determinado inspecionando-a usando o **wireshark**. Deverá ser consultada a figura 8.

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

A interface *loopback* é uma interface virtual da rede que permite ao computador receber respostas de si mesmo. É usada para testar se a carta de rede está configurada corretamente. Deverá ser consultada a figura 9.

## **Experiência 2 – Implementar duas LAN’s virtuais no switch**

Nesta experiência criaram se duas LAN’s virtuais (VLANY0, VLANY1), o tux1 e o tux4 foram associados à VLANY0 enquanto que o tux2 foi associado à VLANY1.

1. Como configurar vlany0?

Na régua 1, a porta T4 tem que estar ligada à porta do *switch* na régua 2. A porta T3, da régua 1, vai estar ligada à porta S0 do tux que se deseja estar ligado ao *switch*. Assim, para criar a *vlan* invocam-se os seguintes comandos no **GTKTerm** do tux escolhido:

* configure terminal
* vlan y0
* end

Depois deverá adicionar-se as portas dos tux 1 e 4:

* configure terminal
* interface fastethernet 0/[nº da porta]
* switchport mode access
* switchport access vlan y0
* end

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

Existem dois domínios de transmissão, visto que o tux 1 recebe resposta do tux 4 quando faz *ping* *broadcast*, mas não do tux 2. 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 tux4 e o que contém o tux 2.

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

Nesta experiência foi configurado o tux4 como um router estabelecendo assim ligação entre as duas VLANs criadas anteriormente.

1. Que rotas há nos tuxes? Qual o seu significado?

As rotas para as vlans associadas:

1. Tux 1 tem uma rota para a vlan 0 (172.16.y0.0) pela gateway 172.16.y0.1.
2. Tux 4 tem uma rota para a vlan 0 (172.16.y0.0) pela gateway 172.16.y0.254 e uma rota para a vlan1 (172.16.y1.0) pela gateway 172.16.y1.253.
3. Tux 2 tem uma rota para a vlan 1 (172.16.y1.0) pela gateway 172.16.y1.1.

As rotas que foram criadas durante a experiência:

1. Tux 1 tem uma rota para a vlan 1 (172.16.y1.0) pela gateway 172.16.y0.254.
2. Tux 2 tem uma rota pata a vlan 0 (172.16.y0.0) pela gateway 172.16.y1.253.

O destino das rotas é até onde o tux que está na origem da rota consegue chegar.

1. Que informação é que uma entrada da tabela de *forwarding* contém?

**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 informações sobre a rota.

**Metric**: o custo de cada 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 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).

1. Que mensagens ARP e endereços MAC associados são observados e porquê?

Quando um tux dá *ping* a outro e o tux que recebeu o *ping* 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. Deverá ser consultada a figura 10.

Essa mensagem vai ter o MAC do tux de origem associado e 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 reponde uma mensagem ARP a dizer o seu MAC *address*. Deverá ser consultada a figura 11.

Esta mensagem vai ter associado tanto o MAC *address* do tux de destino como o de origem.

1. Que ICMP packets são observados e porquê?

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

1. Quais são os endereços IP e MAC associados a um ICMP packet 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 (.253) os endereços de origem vão ser 172.16.60.1 (IP) e 00:0f:fe:8c:af:71 (MAC) e o de destino 172.16.61.253 (IP) e 00:21:5a:c5:61:bb (MAC).

## **Experiência 4 – Configurar um router comercial e implementar o NAT**

Nesta experiência foi configurado primeiramente um router comercial sem NAT ligando-o à rede do laboratório. De seguida foi configurado o router com NAT permitindo assim o acesso dos computadores da rede à internet.

1. Como se configura um router estático num router comercial?

De forma a 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*. Quanto à criação da VLAN, invocam-se os seguintes comandos no **GTKTerm** do TUX escolhido:

* configure terminal
* ip route [ip rota de destino] [máscara] [ip gateway]
* exit

1. Quais são as rotas seguidas pelos pacotes durante a experiência? Explique.

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

1. Como se configura o 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 comandos presentes na figura 12 presente nos anexos.

1. O que faz o 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 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 funções 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 de computadores fora da sua própria rede.

## **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.

1. Como configurar o serviço DNS num *host*?

De forma a configurar o serviço DNS, é necessário mudar o ficheiro **resolv.conf** que se localiza em **vi/etc** no *host* tux. Esse ficheiro tem de conter a seguinte informação:

* search netlab.fe.up.pt
* nameserver 172.16.1.1

Onde **netlab.fe.up.pt** é o nome do servidor DNS e 172.16.1.1 o seu endereço de IP. Após esta experiencia, é possível aceder à internet nos tux’s.

1. Que pacotes são trocados pelo DNS e que informações são transportadas?

Em primeiro lugar, temos um pacote enviado do *Host* para o *Server* (linha 6) que contém o *hostname* desejado, pedindo o seu endereço de IP. Deverá ser consultada a figura 13.

O servidor responde (linha 7) com um pacote que contem o endereço IP do *hostname*.

## **Experiência 6 – Conexões TCP**

Nesta experiência foi observado o comportamento do protocolo TCP utilizando para isso a aplicação desenvolvida na primeira parte do trabalho.

1. Quantas conexões TCP foram abertas pela aplicação FTP?

A aplicação FTP abriu 2 conexões TCP, uma para mandar os comandos FTP ao servidor e receber as respostas e outra para receber os dados enviados pelo servidor e enviar as respostas do cliente.

1. Em que conexão é transportado o controlo de informação?

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

1. Quais as fases da conexão TCP?

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

1. Como é que o mecanismo ARQ TCP funciona? **Quais os campos TCP relevantes? Qual a informação relevante observada nos logs?**

O TCP (*Transmission Control Protocol*) utiliza o mecanismo ARQ (*Automatic Repeat Request*). Este consiste no controlo de erros na transmissão de dados. Para tal, utiliza ***acknowledgments***, que são mensagens enviadas pelo recetor que indicam que a trama foi recebida corretamente, e ***timeouts***, que é o tempo máximo de espera por um *acknowledgment*. No caso de um *acknowledgment* não ser recebido antes do *timeout*, a trama é retransmitida até este ser recebido. Assim, é garantida uma transmissão confiável.

O TCP (*Transmission Control Protocol*) utiliza o mecanismo ARQ (*Automatic Repeat Request*) com o método da janela deslizante. Este consiste no controlo de erros na transmissão de dados. 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 do pacote a ser enviado

1. Como é que o mecanismo de controlo de congestão TCP funciona? **Como é que o fluxo de dados da conexão evoluiu ao longo do tempo? Está de acordo com o mecanismo de controlo de congestão 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.

Registamos que no inicio do primeiro download no tux1, a taxa de transferência aumentou, chegando esta taxa a um pico perto dos 7 segundos. Após o inicio do segundo download verificamos uma descida a pique seguida de uma subida a pique que estabilizou relativamente (apesar de ainda ter alguns picos) num nível mais baixo do que quando apenas o primeiro download estava a ser feito.

Sim, está de acordo pois quando a rede estava mais congestionada tinha um bitrate menor. Deverá ser consultada a figura 14.

1. De que forma é afetada a conexão de dados TCP pelo aparecimento de uma segunda conexão TCP? Como?

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.

# *Conclusões*

O segundo trabalho da unidade curricular de Redes de Computadores teve como objetivo a configuração de uma rede 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.

Em suma, o trabalho foi concluído 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.

# *Referências*

* https://www.cisco.com/c/en/us/support/docs/ip/network-address-translation-nat/26704-nat-faq-00.html

# *Anexos*

## **Imagens**

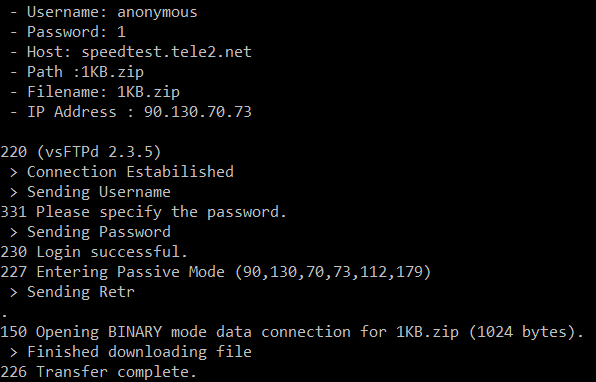


Figura 1

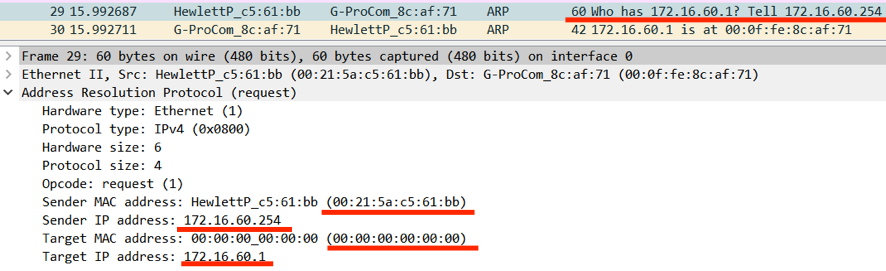


Figura 2

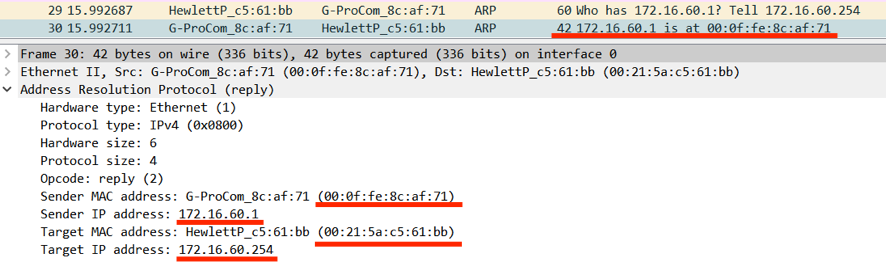


Figura 3

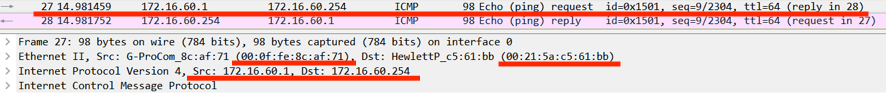


Figura 4

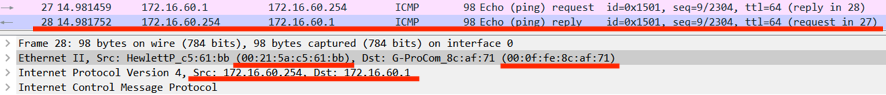


Figura 5

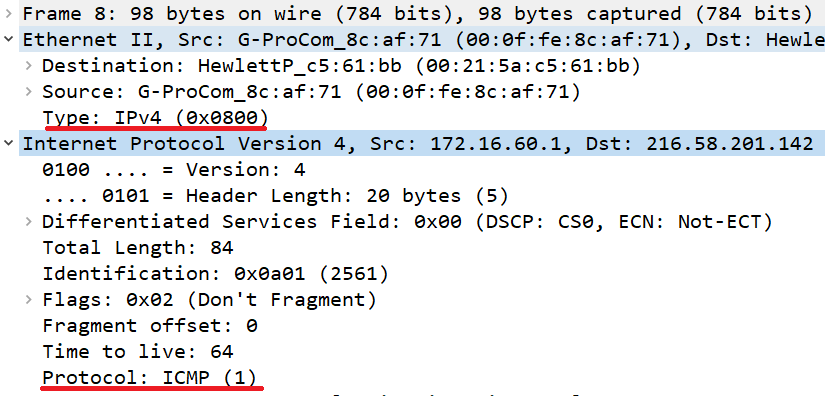


Figura 6

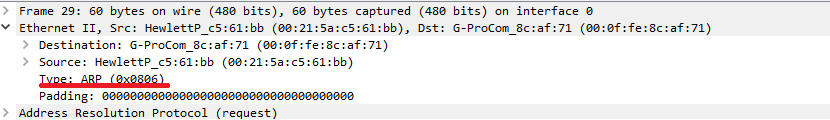


Figura 7

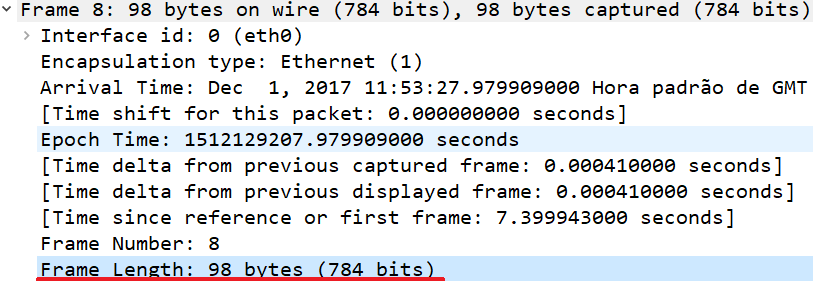


Figura 8

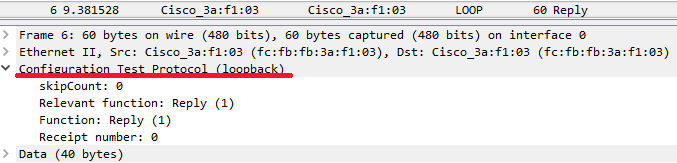


Figura 9

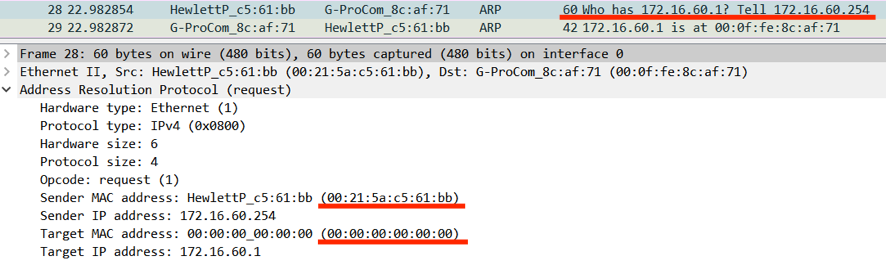


Figura 10

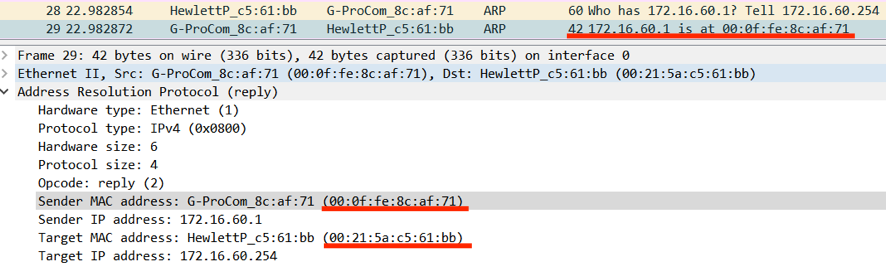


Figura 11

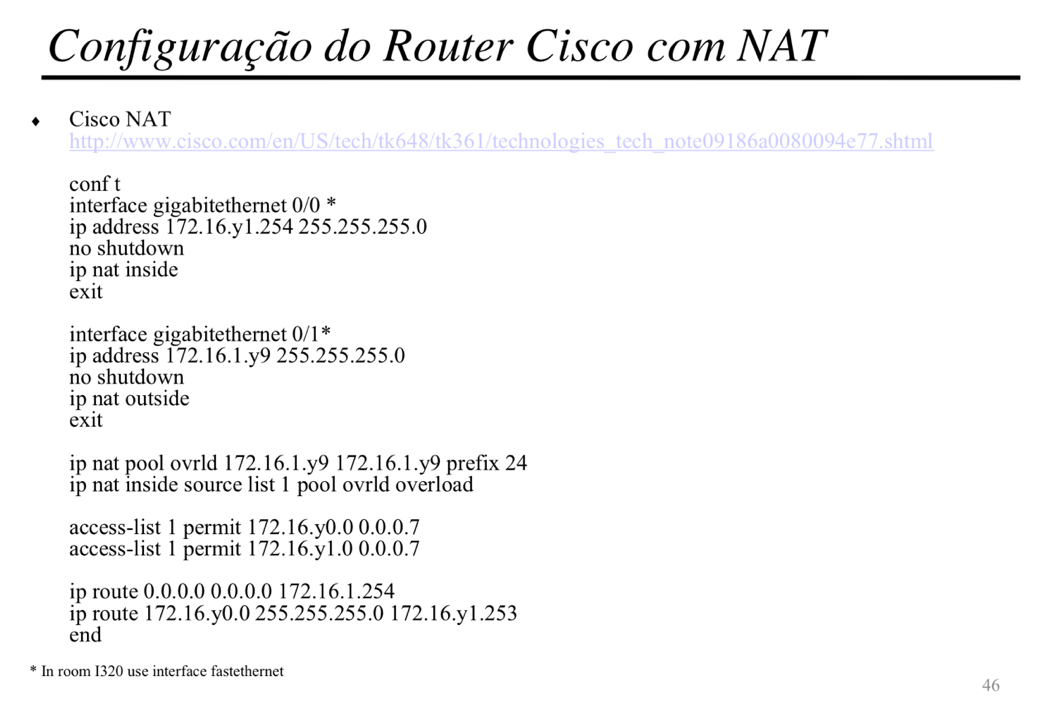


Figura 12

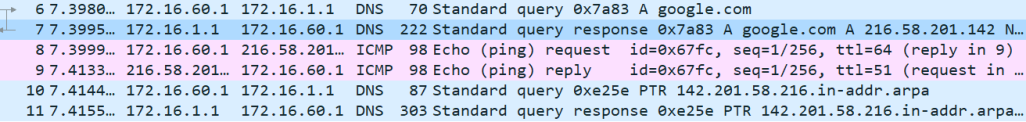


Figura 13

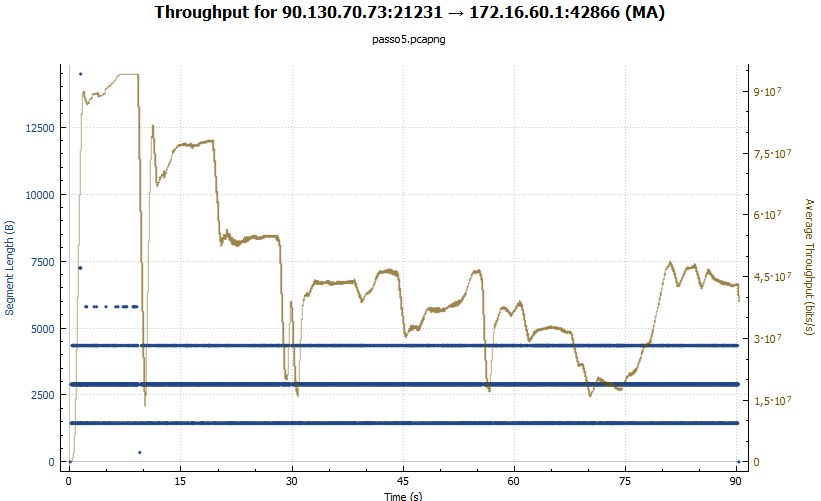


Figura 14

## **clientDownload.c**

#include <string.h>

#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 <strings.h>

#include <ctype.h>

#define **SERVER\_PORT** 21

#define **SERVER\_ADDR** "192.168.28.96"

#define **MAX\_STRING\_LENGTH** 50

#define **SOCKET\_BUF\_SIZE** 1000

void **readResponse**(int socketfd, char \*responseCode);

struct hostent \***getip**(char host[]);

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

void **parseArgument**(char \*argument, char \*user, char \*pass, char \*host, char \*path);

int **sendCommandInterpretResponse**(int socketfd, char cmd[], char commandContent[], char\* filename, int socketfdClient);

int **getServerPortFromResponse**(int socketfd);

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

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

{

    int socketfd;

    int socketfdClient =-1;

    struct sockaddr\_in server\_addr;

    struct sockaddr\_in server\_addr\_client;

    struct hostent \*h;

    char user[MAX\_STRING\_LENGTH];

    char responseCode[3];

**memset**(user, 0, MAX\_STRING\_LENGTH);

    char pass[MAX\_STRING\_LENGTH];

**memset**(pass, 0, MAX\_STRING\_LENGTH);

    char host[MAX\_STRING\_LENGTH];

**memset**(host, 0, MAX\_STRING\_LENGTH);

    char path[MAX\_STRING\_LENGTH];

**memset**(path, 0, MAX\_STRING\_LENGTH);

**parseArgument**(argv[1], user, pass, host, path);

    char filename[MAX\_STRING\_LENGTH];

**parseFilename**(path, filename);

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

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

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

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

**printf**(" - Filename: %s\n", filename);

    h = **getip**(host);

**printf**(" - IP Address : %s\n\n", **inet\_ntoa**(\*((struct in\_addr \*)h->h\_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**(**inet\_ntoa**(\*((struct in\_addr \*)h->h\_addr))); */\*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 ((socketfd = **socket**(AF\_INET, SOCK\_STREAM, 0)) < 0)

    {

**perror**("socket()");

**exit**(0);

    }

*/\*connect to the server\*/*

    if (**connect**(socketfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0)

    {

**perror**("connect()");

**exit**(0);

    }

**readResponse**(socketfd, responseCode);

    if (responseCode[0] == '2')

    {

**printf**(" > Connection Estabilished\n");

    }

**printf**(" > Sending Username\n");

    int res = **sendCommandInterpretResponse**(socketfd, "user ", user, filename, socketfdClient);

    if (res == 1)

    {

**printf**(" > Sending Password\n");

        res = **sendCommandInterpretResponse**(socketfd, "pass ", pass, filename, socketfdClient);

    }

**write**(socketfd, "pasv\n", 5);

    int serverPort = **getServerPortFromResponse**(socketfd);

*/\*server address handling\*/*

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

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

    server\_addr\_client.sin\_addr.s\_addr = **inet\_addr**(**inet\_ntoa**(\*((struct in\_addr \*)h->h\_addr))); */\*32 bit Internet address network byte ordered\*/*

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

*/\*open an TCP socket\*/*

    if ((socketfdClient = **socket**(AF\_INET, SOCK\_STREAM, 0)) < 0)

    {

**perror**("socket()");

**exit**(0);

    }

*/\*connect to the server\*/*

    if (**connect**(socketfdClient, (struct sockaddr \*)&server\_addr\_client, sizeof(server\_addr\_client)) < 0)

    {

**perror**("connect()");

**exit**(0);

    }

**printf**("\n > Sending Retr\n");

    int resRetr =**sendCommandInterpretResponse**(socketfd, "retr ", path, filename, socketfdClient);

    if(resRetr==0){

**close**(socketfdClient);

**close**(socketfd);

**exit**(0);

    }

    else **printf**(" > ERROR in RETR response\n");

**close**(socketfdClient);

**close**(socketfd);

**exit**(1);

}

*// ./download ftp://anonymous:1@speedtest.tele2.net/1KB.zip*

void **parseArgument**(char \*argument, char \*user, char \*pass, char \*host, char \*path)

{

    char start[] = "ftp://";

    int index = 0;

    int i = 0;

    int state = 0;

    int length = **strlen**(argument);

    while (i < length)

    {

        switch (state)

        {

        case 0: *//reads the ftp://*

            if (argument[i] == start[i] && i < 5)

            {

                break;

            }

            if (i == 5 && argument[i] == start[i])

                state = 1;

            else

**printf**(" > Error parsing ftp://");

            break;

        case 1: *//reads the username*

            if (argument[i] == ':')

            {

                state = 2;

                index = 0;

            }

            else

            {

                user[index] = argument[i];

                index++;

            }

            break;

        case 2:

            if (argument[i] == '@')

            {

                state = 3;

                index = 0;

            }

            else

            {

                pass[index] = argument[i];

                index++;

            }

            break;

        case 3:

            if (argument[i] == '/')

            {

                state = 4;

                index = 0;

            }

            else

            {

                host[index] = argument[i];

                index++;

            }

            break;

        case 4:

            path[index] = argument[i];

            index++;

            break;

        }

        i++;

    }

}

void **parseFilename**(char \*path, char \*filename){

    int indexPath = 0;

    int indexFilename = 0;

**memset**(filename, 0, MAX\_STRING\_LENGTH);

    for(;indexPath< **strlen**(path); indexPath++){

        if(path[indexPath]=='/'){

            indexFilename = 0;

**memset**(filename, 0, MAX\_STRING\_LENGTH);

        }

        else{

            filename[indexFilename] = path[indexPath];

            indexFilename++;

        }

    }

}

*//gets ip address according to the host's name*

struct hostent \***getip**(char host[])

{

    struct hostent \*h;

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

    {

**herror**("gethostbyname");

**exit**(1);

    }

    return h;

}

*//reads response code from the server*

void **readResponse**(int socketfd, char \*responseCode)

{

    int state = 0;

    int index = 0;

    char c;

    while (state != 3)

    {

**read**(socketfd, &c, 1);

**printf**("%c", c);

        switch (state)

        {

*//waits for 3 digit number followed by ' ' or '-'*

        case 0:

            if (c == ' ')

            {

                if (index != 3)

                {

**printf**(" > Error receiving response code\n");

                    return;

                }

                index = 0;

                state = 1;

            }

            else

            {

                if (c == '-')

                {

                    state = 2;

                    index=0;

                }

                else

                {

                    if (**isdigit**(c))

                    {

                        responseCode[index] = c;

                        index++;

                    }

                }

            }

            break;

*//reads until the end of the line*

        case 1:

            if (c == '\n')

            {

                state = 3;

            }

            break;

*//waits for response code in multiple line responses*

        case 2:

            if (c == responseCode[index])

            {

                index++;

            }

            else

            {

                if (index == 3 && c == ' ')

                {

                    state = 1;

                }

                else

                {

                 if(index==3 && c=='-'){

                    index=0;

                }

                }

            }

            break;

        }

    }

}

*//reads the server port when pasv is sent*

int **getServerPortFromResponse**(int socketfd)

{

    int state = 0;

    int index = 0;

    char firstByte[4];

**memset**(firstByte, 0, 4);

    char secondByte[4];

**memset**(secondByte, 0, 4);

    char c;

    while (state != 7)

    {

**read**(socketfd, &c, 1);

**printf**("%c", c);

        switch (state)

        {

*//waits for 3 digit number followed by ' '*

        case 0:

            if (c == ' ')

            {

                if (index != 3)

                {

**printf**(" > Error receiving response code\n");

                    return -1;

                }

                index = 0;

                state = 1;

            }

            else

            {

                index++;

            }

            break;

        case 5:

            if (c == ',')

            {

                index = 0;

                state++;

            }

            else

            {

                firstByte[index] = c;

                index++;

            }

            break;

        case 6:

            if (c == ')')

            {

                state++;

            }

            else

            {

                secondByte[index] = c;

                index++;

            }

            break;

*//reads until the first comma*

        default:

            if (c == ',')

            {

                state++;

            }

            break;

        }

    }

    int firstByteInt = **atoi**(firstByte);

    int secondByteInt = **atoi**(secondByte);

    return (firstByteInt \* 256 + secondByteInt);

}

*//sends a command, reads the response from the server and interprets it*

int **sendCommandInterpretResponse**(int socketfd, char cmd[], char commandContent[], char\* filename, int socketfdClient)

{

    char responseCode[3];

    int action = 0;

*//sends the command*

**write**(socketfd, cmd, **strlen**(cmd));

**write**(socketfd, commandContent, **strlen**(commandContent));

**write**(socketfd, "\n", 1);

    while (1)

    {

*//reads the response*

**readResponse**(socketfd, responseCode);

        action = responseCode[0] - '0';

        switch (action)

        {

*//waits for another response*

        case 1:

            if(**strcmp**(cmd, "retr ")==0){

**createFile**(socketfdClient, filename);

                break;

            }

**readResponse**(socketfd, responseCode);

            break;

*//command accepted, we can send another command*

        case 2:

            return 0;

*//needs additional information*

        case 3:

            return 1;

*//try again*

        case 4:

**write**(socketfd, cmd, **strlen**(cmd));

**write**(socketfd, commandContent, **strlen**(commandContent));

**write**(socketfd, "\r\n", 2);

            break;

        case 5:

**printf**(" > Command wasn\'t accepted. Goodbye!\n");

**close**(socketfd);

**exit**(-1);

        }

    }

}

void **createFile**(int fd, char\* filename)

{

    FILE \*file = **fopen**((char \*)filename, "wb+");

    char bufSocket[SOCKET\_BUF\_SIZE];

   int bytes;

   while ((bytes = **read**(fd, bufSocket, SOCKET\_BUF\_SIZE))>0) {

    bytes = **fwrite**(bufSocket, bytes, 1, file);

}

**fclose**(file);

**printf**(" > Finished downloading file\n");

}