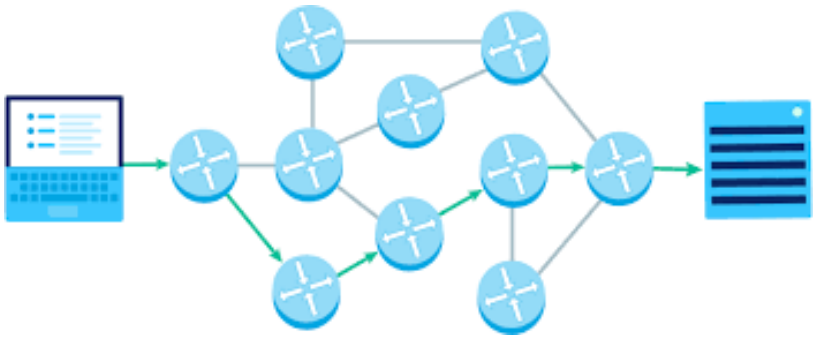
Faculdade de Engenharia da Universidade do Porto



Relatório do 2º trabalho laboratorial de RCOM

Autores:

João Marinho up201905952@fe.up.pt

Tiago Silva up201906045@fe.up.pt

ÍNDICE

SUMÁRIO

Para o segundo trabalho laboratorial desenvolvido no âmbito da unidade curricular de Redes de Computadores (RC), foram executadas duas tarefas distintas:

- Numa primeira parte, ocorreu o desenvolvimento de uma aplicação de download de ficheiros
- Numa segunda parte elaboramos diversas experiências que nos levaram a colocar em prática os conhecimentos lecionados nas aulas teóricas

Deste modo, numa visão geral, consideramos que o desenvolvimento deste trabalho foi um bom apoio para consolidar os nossos conhecimentos, pois foi uma excelente forma de termos um breve contacto com a matéria em estudo da unidade curricular neste final de semestre.

INTRODUÇÃO

Considerando a divisão já mencionada deste trabalho laboratorial, neste relatório, explicamos inicialmente o processo de desenvolvimento da aplicação de download, onde tivemos de perceber como funcionava o protocolo utilizado para perceber de que forma é que o íamos poder utilizar de forma a atingir os objetivos definidos.

De seguida, apresentamos uns breves resumos de experiências elaboradas tanto em casa como em laboratório, onde seguimos procedimentos que nos foram fornecidos pelos docentes. Posto isto, fornecemos também uma pequena síntese das análises dos resultados obtidos, que nos levou a conseguir entender melhor como funciona a rede e a comunicação das nossas máquinas neste meio.

PARTE 1 – APLICAÇÃO DE DOWNLOAD

Nesta parte do trabalho temos como objetivo desenvolver uma simples aplicação FTP que faz o download de um ficheiro em específico usando este mesmo protocolo, de acordo com o **RFC959**. Esta aplicação deve receber como argumento um URL cuja sintaxe é descrita no **RFC1738**, ftp://[<user>:<password>@]<host>/<url-path>.

ARQUITETURA DA APLICAÇÃO

Após a receção do parâmetro requisitado, é feito o seu *parsing*, isto é, são criadas 4 variáveis que simbolizam respetivamente os campos, *user*, *password*, *host*, *url-path*. No caso de não serem inseridos valores para o *user* e para a *password*, estes ficam com um valor genérico *anonymous*.

Posteriormente, é chamada a função ***getip*** (já fornecida) que, em caso de sucesso, nos retorna uma estrutura do tipo *hostent*. Desta forma, possuímos todos os valores necessários para iniciar o pedido ao servidor FTP com a chamada à função ***download***.

Primeiramente, é estabelecida a conexão com o servidor, função ***connectToHost***, além da leitura da mensagem de boas-vindas.

Segundamente, são enviados os comandos **user *username*** e **pass *password***, com os valores anteriormente obtidos através do input do utilizador. Finalizando o *login,* passamos para a chamada do comando **pasv**, que nos fornece a porta e endereço IP (do atual servidor), por onde serão enviados os dados contidos no ficheiro requerido.

Finalmente, é iniciado o download do ficheiro com o comando **retr *filename***, para o primeiro *socket*, e a leitura dos seus conteúdos será feita através da conexão estabelecida com o servidor na nova porta. Esta leitura termina com uma resposta cujo código será 226 em caso de sucesso.

De forma a executar a leitura das respostas por parte do servidor são utilizadas as funções ***receiveResponse*** e ***receiveMsg***, cuja diferença reside no facto da primeira apenas retornar o código resultante do pedido, já a segunda guarda também todo o conteúdo enviado na resposta para uma variável passada por parâmetro. Existem respostas de diversos tipos, sendo que respostas cujo código inicia com 1, 2 ou 3, corresponde uma resposta positiva, já os valores 4 e 5 são considerados erros.

Por outro lado, o envio dos comandos para o servidor é feito com as funções ***writeMsg*** e ***writeMsgToGetRes***, que diferem, tal como na leitura, no resultado retornado após a sua escrita.

DESCRIÇÃO DOS RESULTADOS

Diversos foram os casos de teste para esta aplicação, cujas variáveis a ter em conta foram:

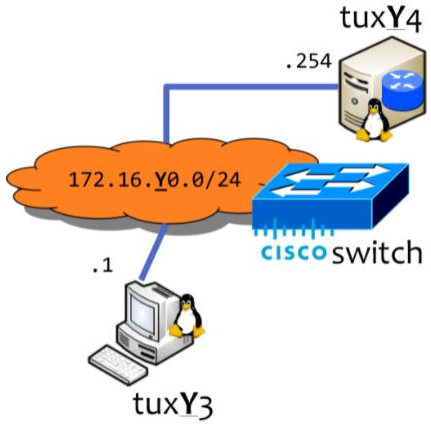
- A identidade do utilizador (anónimo ou não anónimo)
- Tipo de ficheiro e tamanho
- Respostas possíveis por parte do servidor

Em caso de erro, o programa termina, por outro lado, quando o ficheiro é transferido com sucesso, o utilizador é informado com uma mensagem para o *output stream*.

PARTE 2 – CONFIGURAÇÃO DA REDE E ANÁLISE

EXPERIÊNCIA 1 - CONFIGURAÇÃO DE UM IP DE REDE

Arquitetura da rede / Objetivos da experiência



O objetivo desta experiência consiste na configuração de IP *addresses* e conectá-los a um *switch*, de forma a conseguirmos estabelecer a conexão de uma máquina (tux3) com outra (tux4).

Deste modo, foi necessário definir em cada uma das máquinas as rotas que serviram de meio de ligação entre as duas. Assim, configuradas as rotas, podemos concluir que a conexão foi estabelecida com sucesso, quando ao executar o comando **ping 172.16.30.254** (no caso do tux3) e **ping 172.16.30.1** (no caso do tux4) de forma a conseguirmos de uma máquina pingar a outra, recebíamos respostas, ou seja, as máquinas conseguiam comunicar uma com a outra.

Comandos de configuração / Análise dos logs capturados

1) O que são os pacotes ARP e para que são usados?

Address Resolution Protocol (ARP) trata-se de um protocolo que permite mapear um endereço IP dinâmico para um endereço físico permanente numa rede local (LAN), mais conhecido como *media access control* (MAC), é possível ver o seu uso nos *logs* quando é executado o comando *ping*, por exemplo, em que uma máquina pergunta à rede quem tem o endereço IP procurado e recebe a resposta com o devido endereço MAC desse mesmo IP.

2) Quais são os endereços MAX e IP dos pacotes ARP e porquê?

Quando corremos o comando ping, o tux3 apenas sabe qual o endereço IP que tem de pingar. Porém, como não sabe qual é o endereço MAC associado a esse endereço IP, antes de enviar os pacotes ICMP (associados ao ping) começa por enviar um pacote ARP com o objetivo de descobrir qual o endereço MAC correspondente ao endereço IP alvo. Deste modo, o endereços IP e MAC serão (Figura 1):

- IP (source) - endereço IP do tux3
- MAC (source) - endereço MAC do tux 3

- IP (target) - endereço IP que se pretende pingar (tux4)
- MAC (target) - 00:00:00:00:00:00, pois este é o endereço que se pretende descobrir com o envio do pacote MAC, portanto o pacote é enviado para toda a gente de forma a chegar a quem possui o endereço IP especificado anteriormente.

Assim, quando o tux4 receber este pacote ARP, como o seu endereço IP vai ser igual ao endereço IP target do pacote, vai enviar um pacote ARP com a resposta (o seu endereço MAC) para o tux1. Neste pactote (Figura 2):

- IP (source) - endereço IP do tux4
- MAC (source) - endereço MAC do tux4
- IP (target) - endereço IP do tux3
- MAC (target) - endereço MAC do tux3

3) Quais os pacotes gerados pelo comando *ping*?

O comando ping gera dois tipos de pacotes:

Inicialmente envia pacotes ARP de forma a descobrir o endereço MAC associado ao endereço IP alvo, recebendo a resposta passa a enviar pacotes ICMP (*Internet Control Messsage Protocol*) – Figura 3.

4) Quais são os endereços MAC e IP dos pacotes ping?

Como se trata de um ***ping*** de duas máquinas na mesma rede, tanto os endereços IP e MAC serão os das máquinas em questão, no caso de *ping request* – Figura 4:

- IP Source (tux3) - 172.16.30.1
- IP Target (tux4) - 172.16.30.254
- MAC Source (tux3) - 00:21:5a:5a:7d:b7
- MAC Source (tux4) - 00:21:5a:5a:74:3e

No caso de *ping reply* os IP e MAC trocam de *Source* para *Target* e vice-versa (Figura 5).

5) Como determinar se a trama recetora é ARP, IP ou ICMP?

É possível determinar o tipo da trama recetora através de uma análise ao Ethernet header. Numa trama ARP, o Ethernet type vem como (0x806) – Figura 6. Já numa trama IP, este Ethernet type possui o valor (0x800), conseguindo ainda perceber que é uma trama ICMP através de uma análise ao protocolo da mesma (Protocol: ICMP (1)) – Figura 7.

6) Como determinar o tamanho de uma trama recetora?

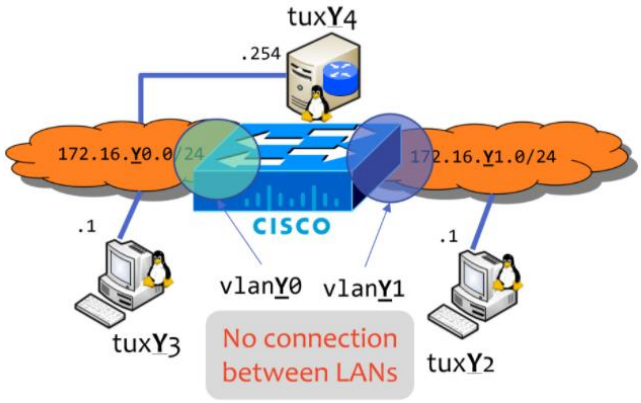
Conseguimos determinar o tamanho de uma trama recetora por análise aos pacotes recebidos através do programa **wireshark**. - Figura 8.

7) O que é a interface de *loopback* e porque é que é tão importante?

A interface de *loopback* é uma interface virtual que permite obter respostas a partir da própria origem, desta forma, é útil para casos de *debug* já que é garantida uma resposta e verificar se a rede está configurada corretamente.

EXPERIÊNCIA 2 - LANS VIRTUAIS

Arquitetura da rede / Objetivos da experiência



O objetivo desta experiência passa por configurar duas *LANs* virtuais (VLANY0, VLANY1), associando o tux1 e o tux4 à primeira e o tux2 à segunda.

Desta forma, através dos comandos corridos no switch:

configure terminal
vlan 30
end

e

configure terminal
vlan 31
end

conseguimos criar as duas LANs necessárias para a experiência, faltando apenas configurá-las.

Para proceder à configuração das mesmas, tivemos que identificar quais as portas que pertenciam a cada uma das vlans, tendo então de verificar qual a disposição dos cabos de rede. De seguida, e possuindo já todas as informações necessárias para a configuração das vlans, tivemos de correr o seguinte conjunto de comandos, novamente no switch:

configure terminal
interface fastethernet 0/X - sendo X a porta a qual cada uma das máquinas em questão está ligada ao switch
swicth mode access
switch access vlan ID - sendo ID o identificador da LAN em questão [30, 31]
end

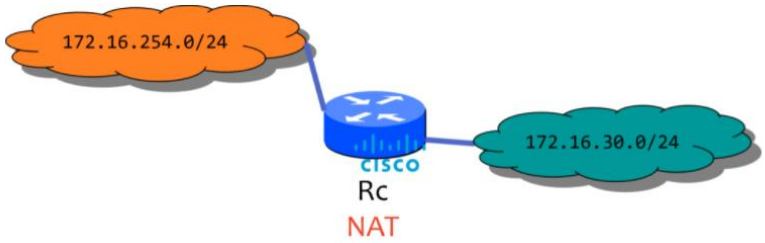
Deste modo, com as configurações terminadas tornou-se possível a comunicação entre máquinas que se encontram na mesma LAN.

Assim, através da análise dos registos podemos verificar a existência de dois domínios de transmissão, um que contém ambos os tux1 e tux4, outro que apenas contém o tux2, isto pois,

quando é feito o *broadcast* a partir do tux1 ou tux4, apenas é recebida a resposta de um dos tux que não o tux2. Por outro lado, quando é feita a transmissão a partir de tux2 não são recebidas respostas, como esperado.

EXPERIÊNCIA 3 - CONFIGURAÇÃO DO ROUTER

Arquitetura da rede / Objetivos da experiência



Comandos de configuração / Análise dos logs capturados

Parte 1 – Cisco Router Configuration

1) Como se configura um roteamento estático num router comercial?

Após uma análise, do ficheiro de configuração do Cisco Router conseguimos concluir que o nome do router é **gnu-rtr1**, possuindo duas postas *fastethernet* com os números 0 e 1.

Os endereços IP configurados são:

- 172.16.30.1 255.255.255.0
- 172.16.254.45 255.255.255.0

ambos possuindo uma máscara de 24 bits (255.255.255.0).

E deste modo é possível configurar um roteamento estático num router comercial, definindo manualmente as rotas no mesmo. No exemplo do ficheiro todos os pacotes com destino a 172.16.40.0/24 são reencaminhados para 172.16.30.2, de outra forma serão reencaminhados para 172.16.254.1. Informação esta, que se encontra presente no mesmo da seguinte maneira:

- Ip route 0.0.0.0 0.0.0.0 172.16.254.1
- Ip route 172.16.40.0 255.255.255.0 172.16.30.2

2) Como configurar a NAT num router comercial?

Recorrendo aos conteúdos do ficheiro fornecido, conseguimos perceber que é possível configurar a NAT num router comercial através das seguintes linhas:

interface FastEthernet0/1
…
nat ouside

que especifica a interface conectada com o exterior e por sua vez executa a tradução de endereços privados para públicos.

ip nat pool ovrld 172.16.254.45 172.16.254.45 prefix-length 24

permite reconhecer qual a *pool* range de IP que estão disponíveis para NATing, neste caso será apenas um endereço (172.16.254.45).

3) O que faz a NAT?

NAT ou *Network Address Translation,* funciona como um intermediário entre uma rede interna e a *Internet*, i.e., do lado especificado como *inside*, o router irá receber pacotes cujos endereços de origem são privados (rede local), traduzindo os mesmos para que possam ser utilizados no exterior (*internet*). Do mesmo modo, é feita uma tradução reversa do lado *outside*, i.e., quando o router recebe uma resposta a uma mensagem que sofreu uma tradução (NATing) irá efetuar a operação reversa, traduzindo o endereço de destino para o privado correspondente.

4) Como configurar o DNS num *host*?
5) Que pacotes são trocados pelo DNS e que informações são transportadas?

DNS ou *Domain Name Resolution*, é um mecanismo que transforma endereços que possuem um formato mais *human-friendly*, para endereços numéricos.

Inicialmente, ao executar *ping youtubas*, não foram verificados quaisquer pacotes DNS, uma vez que, foi inserida no ficheiro */etc/hosts*, static DNS resolver, a seguinte configuração:

- 142.250.200.142 youtubas

De seguida, ao executar *ping enisa.europa.eu*, foi possível intercetar pacotes DNS *query* ao servidor default de DNS.

Finalmente, no terceiro teste, alteramos o servidor de DNS para 9.9.9.9 o que levou à interceção de pacotes DNS cujo destino é o servidor atualizado.

6) Que pacotes ICMP são observados e porquê?

Através da captura de pacotes, é possível verificar a existência de pacotes do tipo ICMP. Estes são vistos como respostas enviadas (devido à execução do comando *traceroute*), pelos múltiplos saltos ao longo do percurso, em que os pacotes acabam por "morrer" devido ao chamado TTL, *Time to Leave*. A seguinte descrição pode ser vista nos pacotes obtidos:

- Time-to-live exceeded (Time to live exceeded in transit)

7) Quais são os endereços IP e MAC associados aos pacotes IP e porquê?

Os pacotes ICMP possuem os endereços IP e MAC tanto do destino como da origem, de forma a obter uma correta distribuição dos pacotes dentro da rede de origem e de destino.

8) Que rotas estão presentes na tua máquina? Qual o seu significado?

As rotas existentes na máquina podem ser obtidas através do comando, *route -n*. Este fornece uma tabela de rotas que consistem no direcionamento de um grupo de pacotes cujo destino caí sobre uma dada gama na *gateway* atribuída, i.e., indica qual a *gateway* por onde os pacotes devem ser reencaminhados quando possuem um endereço IP de destino aí especificado.

Uma possível entrada da tabela é:

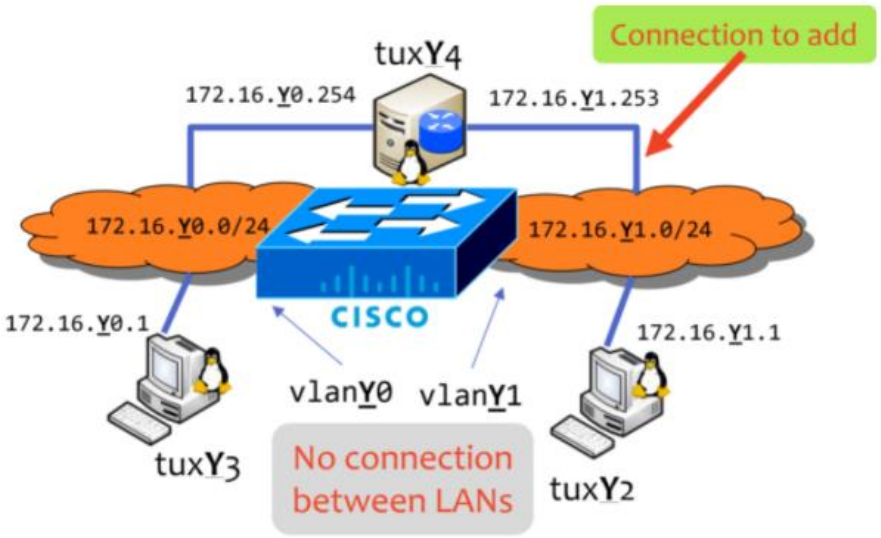
- Dest – 104.17.113.188 / Gateway – 192.168.1.1 / Genmask 255.255.255.255

que nos permite reencaminhar todos os pacotes com destino a 104.17.113.188, para a *gateway* 192.168.1.1. Já através da análise da *genmask* podemos concluir que esta rota apenas serve para o IP 104.17.133.188, uma vez que possui todos os bits ativos, ou seja, todos os 32 bits do endereço de destino têm de corresponder o especificado na coluna *destination*.

EXPERIÊNCIA 4 - CONFIGURAÇÃO DO ROUTER (LAB)

Parte 1 – Linux Router

Arquitetura da rede / Objetivos da experiência



Esta experiência vem unir aquilo que foi adquirido tanto na experiência 2 como na experiência 3. Através da reconfiguração daquilo que foi obtido em **2**, pretendemos obter uma via de comunicação entre tuxes de LANs diferentes (tux3 e tux2).

Comandos de configuração / Análise dos logs capturados

1) Que rotas há nos tuxes? Qual o seu significado?
Inicialmente, dispomos de 3 rotas:

- Rota de tux1 para a vlan0 através da *gateway* 172.16.y0.1
- Rota de tux4 para a vlan0 através da *gateway* 172.16.y0.254
- Rota de tux2 para a vlan1 através da *gateway* 172.16.y1.1

Com estas somos capazes de estabelecer uma ligação entre tuxs pertencentes à mesma LAN, ou seja, tux1 e tux4 conseguem comunicar entre si, mas não com o tux2 e vice-versa.

Através da experiência foram criadas mais 3 rotas:

- Rota de tux4 para a vlan1 através da *gateway* 172.16.y1.253
- Rota de tux2 para a vlan0 através da *gateway* 172.16.y1.253
- Rota de tux0 para a vlan1 através da *gateway* 172.16.y0.254

Estas novas rotas, permitem a comunicação entre a vlan0 e vlan1, cujo intermediário é o tux4 visto que pertence a ambas, e serve de *gateway* para as mesmas. Desta forma, o tux3 e tux2 conseguem comunicar entre si (e naturalmente com o tux4).

2) Que informação está contida numa entrada da tabela de *forwarding*?

Uma tabela de *forwarding* de um router possui as seguintes informações:

- Destino: Endereço IP do destino do pacote
- Mascará: Usada para determinar o ID da rede a partir do endereço IP do destino.
- Métrica: Um uso comum deste parâmetro é indicar o número mínimo de *hops* para atingir a rede destino.
- *Next Hop*: Endereço IP para onde o router vai reencaminhar o pacot.e

Estas informações servem para que o router consiga reencaminhar todos os pacotes que até ele chegam. Se o pacote possuir um IP destino cujo router consegue aceder diretamente, então é enviado para lá. Caso contrário, reencaminha-o para o IP obtido por um acesso à tabela de *forwarding*.

3) Que mensagens ARP e respetivos endereços MAC são observados e porquê?

No momento do ping, observa-se a troca de algumas mensagens ARP. Estas mensagem ocorrem, pois um tux está a tentar pingar outro tux, mas apenas tem conhecimento do seu endereço IP. Deste modo, ele irá enviar uma mensagem para a rede com as suas informações:

- IP(origem): o seu endereço
- MAC(origem): o seu endereço MAC
- IP(destino): endereço alvo.

Porém como este desconhece o endereço MAC do tux alvo (que é o que pretende descobrir), irá enviar uma mensagem de broadcast (para toda a rede - endereço MAC 00:00:00:00:00:00) de forma a que a mensagem consiga alcançar a máquina alvo (máquina com IP igual ao IP(destino) da mensagem ARP).

Deste modo, quando o tux alvo receber a mensagem ARP, irá enviar uma resposta à pergunta feita. Este irá enviar uma mensagem com as seguintes informações:

- IP(origem): o seu endereço IP
- MAC(origem): o seu endereço MAC
- IP(destino): IP da máquina que enviou a mensagem ARP
- MAC(destino): MAC da máquina que enviou a mensagem ARP

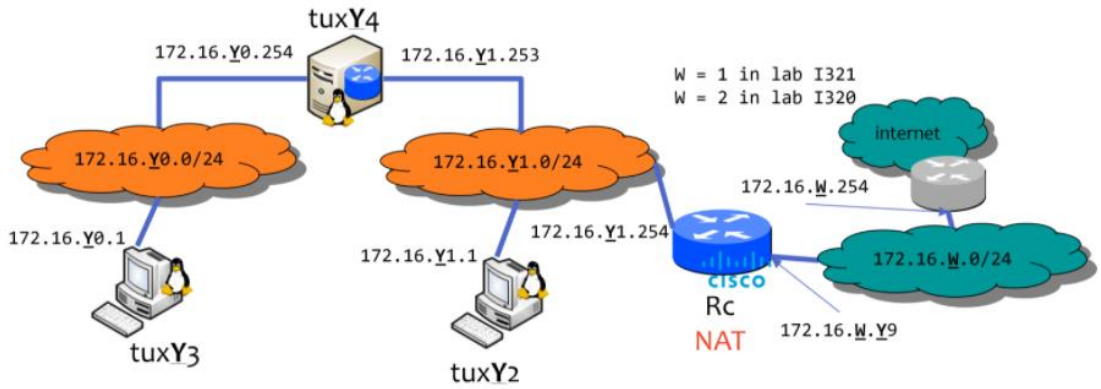
E deste modo, quando o tux que inicialmente enviou a mensagem ARP receber a resposta enviada pelo tux alvo, o objetivo deste tipo de mensagens fica concluído: este obtém o endereço MAC que necessitava,

4) Quais são os endereços IP e MAC associados aos pacotes ICMP e porquê?

Através da análise das capturas feitas na ***tarefa 9***, é possível afirmar que os endereços IP associados a pacotes ICMP são os endereços de origem e destino de pedido, neste caso dos tux's, de acordo com a vlan à qual pertencem. Já os endereços MAC irão depender se as máquinas se encontram na mesma rede ou não. Por exemplo, um *ping* do tux3 para o tux4 irá conter um IP *source* de 172.16.y0.1 e um IP *dest* de 172.16.y0.254 (IPs de cada um dos tux's), um MAC *source* de (00:21:5a:5a:7d:b7) e um MAC *dest* de (00:21:5a:5a:74:3e). Já no caso de fazer *ping* do tux3 para o tux2, iremos obter os endereços IP de ambos, isto é, IP *source* 172.16.y0.1, IP *dest* 172.16.y1.1, embora o MAC *source* se mantenha o mesmo da tentativa anterior, desta vez o MAC *dest* não será o MAC do tux2, mas sim o de tux4 (valor referido anteriormente), visto que é este que fará o redireccionamento para a o devido destino.

Parte 2 – Cisco Router

Arquitetura da rede / Objetivos da experiência



Comandos de configuração / Análise dos logs capturados

1) Quais são os caminhos percorridos pelos pacotes nas experiências descritas e porquê?

Vários são os pacotes enviados nas experiências descritas:

- *Ping* do tux3 para o ip 172.16.2.254 irá resultar num envio dos pacotes com origem no tux referido, com destino ao tux4, que serve de router para a comunicação com a segunda VLAN criada e que se encontra diretamente ligada ao Cisco Router, assim, neste será feita a tradução devido ao NAT, para que o IP da rede privada possa ser visto como um IP legal fora desta. Finalmente, o pacote chegará ao seu destino através do reencaminhamento por *default gateway*.

- *Ping* do tux3 para 104.17.113.188 (internet), segue a mesma lógica que a experiência anterior, mas na fase final, o pacote irá atravessar o router local, que irá tratar do respetivo encaminhamento para a internet.
- *Ping* do *Cisco Router* para os tux2 e tux4, os pacotes são enviados diretamente uma vez que se encontram na mesma VLAN, já com destino ao tux3 o caminho será o inverso do referido anteriormente (tux4 como gateway entre VLANs).

Ping do tux4 para fora das VLANs criadas não será possível uma vez que a lista de acessos do *Cisco Router* apenas permite (segundo a configuração utilizada), IPs cujo terminador seja entre 0-7 (máscara 0.0.0.7), visto que o terminador do IP do tux4 é 254 na VLAN0 e 253 na VLAN1, não será aceite o reencaminhamento.

CONCLUSÕES

Em suma, com o desenvolvimento deste trabalho laboratorial, conseguimos ter uma melhor perceção do funcionamento dos protocolos de comunicação, bem como, das várias preocupações que é necessário ter a nível de hardware.

Deste modo, conseguimos, mais uma vez, consolidar a matéria ao estar em contacto direto com a matéria em questão. Sentimos que tivemos um bom aproveitamento do trabalho, e que, apesar de tudo, e de todas as adversidades que nos possam ter surgido, todas elas foram concluídas a tempo e com sucesso.

ANEXO I

main.c:

```c
#include "../headers/utils.h"
#include "../headers/download.h"

int main(int argc, char * argv[]) {
    char * user = NULL, * password = NULL, * hostName = NULL, * urlPath;

    if ((argc != 2) || (strncmp(argv[1], "ftp://", 6) != 0)) {
        printError("Invalid arguments.");
        exit(-1);
    }

    int length = strlen(argv[1]) - 6;
    if(length < 3) {
        printError("Invalid arguments.");
        exit(-1);
    }

    char * newBuf = (char *)malloc(length);
    strncpy(newBuf, argv[1] + 6, length);

    char * currentString = NULL;
    if (getSubString(&user, newBuf, ":") < 0) {
        printError("Invalid arguments.");
        exit(-1);
    }

    // Anonim
    if (strlen(user) != length) {
        if (getSubString(&password, currentString, "@") < 0) {
            printError("Invalid arguments.");
            exit(-1);
        }

        if(!validCredentials(user, password)) {
            printError("Invalid credentials.");
            exit(-1);
        }
    } else {
        free(user);
        user = (char *)malloc(10);
        password = (char *)malloc(10);
        strncpy(user, "anonymous", 10);
        strncpy(password, "anonymous", 10);
        currentString = (char *)malloc(length);
        strncpy(currentString, newBuf, length);
```

```
    }

    if (getSubString(&hostName, currentString, "/") < 0) {
        printError("Invalid arguments");
        exit(-1);
    }

    if (getSubString(&urlPath, NULL, " ") < 0) {
        urlPath = (char *)malloc(1);
        urlPath = "/";
    }

    struct hostent * host;
    if ((host = getIp(hostName)) != NULL) {
        printf("Host name  : %s\n", host->h_name);
        printf("IP Address : %s\n", inet_ntoa(*((struct in_addr *) host-
>h_addr)));

        download(inet_ntoa(*((struct in_addr *) host->h_addr)), user,
password, urlPath);
    }

    free(user);
    free(password);
    free(hostName);
    free(urlPath);
}
```

download.c:

```
#include "../headers/download.h"

int connectToHost(char* server_address, 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(server_address);    /*32 bit
Internet address network byte ordered*/
    server_addr.sin_port = htons(port);                /*server TCP
port must be network byte ordered */

    /*open a TCP socket*/
    if ((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;
}

void getFileName(char* path, char** fileName) {
    char delim = '/';
    char* token = strtok(path, &delim);
    if (token == NULL) {
        *fileName = (char*)malloc(13);
        strncpy(*fileName, "fileReceived", 13);
        return;
    }

    while (token != NULL) {
        *fileName = (char*)malloc(strlen(token));
        strncpy(*fileName, token, strlen(token));
        token = strtok(NULL, &delim);
    }
}

int readFile(int fd, char* fileName) {
    //Create file
    int file_fd;
    if ((file_fd = open(fileName, O_WRONLY | O_CREAT, 0444)) < 0) {
        perror("Open file");
        return -1;
    }

    //Read information
    char buf[1];
    int res;
    printf("Receiving file...\n");

    while (1) {
        res = read(fd, &buf[0], 1);
        if (res == 0) break;
        if (res < 0) {
            close(file_fd);
            return -1;
        }
        res = write(file_fd, &buf[0], 1);
```

```
        if (res < 0) {
            close(file_fd);
            perror("Couldn't write to file");
        }
    }
    printf("File received\n");

    //Close file
    close(file_fd);
    return 0;
}

int download(char* server_address, char* user, char* password, char*
path) {
    int sockfd;

    if ((sockfd = connectToHost(server_address, SERVER_PORT)) == -1) {
        return -1;
    }

    /*receive welcome msg*/
    int ret = receiveResponse(sockfd);
    if (ret != 220) {
        close(sockfd);
        printf("Bad server");
        return -1;
    }

    /*send username*/
    int length = 6 + strlen(user);
    char* message = (char*)malloc(length);
    snprintf(message, length, "user %s", user);
    message[length - 1] = '\n';
    printf("%s", message);

    if (!writeMsg(sockfd, message, length, 331)) {
        free(message);
        close(sockfd);
        perror("Login - user");
        return -1;
    }

    /*send password*/
    length = 6 + strlen(password);
    message = (char*)realloc(message, length);
    snprintf(message, length, "pass %s", password);
    message[length - 1] = '\n';
    printf("%s", message);
```

```
    if (!writeMsg(sockfd, message, length, 230)) {
        free(message);
        close(sockfd);
        perror("Login - password");
        return -1;
    }

    /*send pasv*/
    message = (char*)realloc(message, 5);
    snprintf(message, 5, "pasv");
    message[4] = '\n';
    printf("%s", message);
    char* response = (char*)malloc(255);
    if (!writeMsgToGetRes(sockfd, message, 5, 227, response)) {
        free(message);
        close(sockfd);
        perror("Set passive mode");
        return -1;
    }

    int ip1, ip2, ip3, ip4, port1, port2;
    sscanf(response, "Entering Passive Mode (%d,%d,%d,%d,%d,%d).", &ip1,
&ip2, &ip3, &ip4, &port1, &port2);
    free(response);

    int port = port1 * 256 + port2;
    int sockfd2;

    if ((sockfd2 = connectToHost(server_address, port)) == -1) {
        return -1;
    }

    /*send retrieve*/
    length = 6 + strlen(path);
    message = (char*)realloc(message, length);
    snprintf(message, length, "retr %s", path);
    message[length - 1] = '\n';
    printf("%s", message);

    if (!writeMsg(sockfd, message, length, 150)) {
        free(message);
        close(sockfd);
        perror("Retrive");
        return -1;
    }

    /*receive file info*/
    char* fileName;
    getFileName(path, &fileName);
```

```
    readFile(sockfd2, fileName);
    free(fileName);

    /*receive response*/
    ret = receiveResponse(sockfd);

    free(message);
    close(sockfd);

    if (ret != 226) {
        perror("Transfer incomplete\n");
        return -1;
    }
    printf("transfer complete\n");
    return 0;
}
```

msgHandler:

```
#include "../headers/msgHandler.h"

int receiveResponse(int fd) {
    char buf[256];
    int res;
    int i = 0;
    char code[4];
    int end = 1;

    while (end) {
        for (int j = 0; j < 4; j++) {
            res = read(fd, &code[j], 1);
        }
        end = code[3] == '-';
        i = 0;
        do {
            res = read(fd, &buf[i++], 1);
            if (res < 0) return 0;
        } while (buf[i - 1] != '\n');
    }

    return atoi(code);
}

int writeMsg(int sockfd, char* message, size_t length, int code) {
    /* send message */
    size_t bytes = write(sockfd, message, length);
    if (bytes < 1) {
        return -1;
```

```
    }

    /*receive response*/
    return receiveResponse(sockfd) == code;
}

int receiveMsg(int fd, char* msg) {
    char buf[256];
    int res;
    int i = 0;
    char code[4];
    int end = 1;

    while (end) {
        for (int j = 0; j < 4; j++) {
            res = read(fd, &code[j], 1);
        }
        end = code[3] == '-';
        i = 0;
        memset(buf, 0, 255);
        do {
            res = read(fd, &buf[i++], 1);
            if (res < 0) return 0;
        } while (buf[i - 1] != '\n');
    }

    strncpy(msg, buf, 255);
    return atoi(code);
}

int writeMsgToGetRes(int sockfd, char* message, size_t length, int code,
char* response) {
    /* send message */
    size_t bytes = write(sockfd, message, length);
    if (bytes < 1) {
        return -1;
    }

    /*receive response*/
    return receiveMsg(sockfd, response) == code;
}
```

utils.c:

```
#include "../headers/utils.h"


int getSubString(char ** to, char * from, char * delim) {
```

```
    char * token = strtok(from, delim);
    if (token == NULL) return -1;
    *to = (char *)malloc(strlen(token));
    strncpy(*to, token, strlen(token));
    return 0;
}

void printError(char * str) {
    printf("%s: Try: ./download ftp://[<user>:<password>@]<host>/<url-
path>\n", str);
}

int validCredentials(char * user, char * password) {
    if((strcmp(user, "rcom")==0) && (strcmp(password, "rcom")==0)) return
1;
    return 0;
}

struct hostent* getIp(char * hostName) {
    struct hostent *host;

    if ((host = gethostbyname(hostName)) == NULL) {
        herror("gethostbyname()");
        return NULL;
    }

    return host;
}
```

download.h:

```
#ifndef _DOWNLOAD_H_
#define _DOWNLOAD_H_

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

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "../headers/msgHandler.h"

#define SERVER_PORT 21

/**
 * @brief Connect to host and retrieve socket fd
```

```
 *
 * @param server_address - server ip address
 * @param port - connection port
 * @return int - socket file descriptor
 */
int connectToHost(char* server_address, int port);

/**
 * @brief Get the File Name object
 *
 * @param path
 * @param fileName
 */
void getFileName(char* path, char** fileName);

/**
 * @brief
 *
 * @param fd
 * @param fileName
 * @return int
 */
int readFile(int fd, char* fileName);

/**
 * @brief
 *
 * @param server_address
 * @param user
 * @param password
 * @param path
 * @return int
 */
int download(char* server_address, char* user, char* password, char*
path);

#endif //_DOWNLOAD_H_
```

msgHandler.h:

```
#ifndef _MSGHANDLER_H_
#define _MSGHANDLER_H_

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

int receiveResponse(int fd);
```

```
int writeMsg(int sockfd, char* message, size_t length, int code);
int receiveMsg(int fd, char* msg);
int writeMsgToGetRes(int sockfd, char* message, size_t length, int code,
char* response);

#endif //_MSGHANDLER_H_
```

utils.h:

```
#ifndef _UTILS_H_
#define _UTILS_H_

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

int getSubString(char ** to, char * from, char * delim);

void printError(char * str);

int validCredentials(char * user, char * password);

struct hostent* getIp(char * hostName);

#endif //_UTILS_H_
```

logs:

```
Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: HewlettP_5a:7d:b7 (00:21:5a:5a:7d:b7)
    Sender IP address: 172.16.30.1
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 172.16.30.254
```

Figura 1

```
▾ Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: HewlettP_5a:74:3e (00:21:5a:5a:74:3e)
    Sender IP address: 172.16.30.254
    Target MAC address: HewlettP_5a:7d:b7 (00:21:5a:5a:7d:b7)
    Target IP address: 172.16.30.1
```

Figura 2

14	16.438966386	HewlettP_5a:7d:b7	Broadcast	ARP	42	Who has 172.16.30.254? Tell 172.16.30.1
15	16.439098315	HewlettP_5a:74:3e	HewlettP_5a:7d:b7	ARP	60	172.16.30.254 is at 00:21:5a:5a:74:3e
16	16.439107045	172.16.30.1	172.16.30.254	ICMP	98	Echo (ping) request id=0x31d5, seq=1/256, ttl=64 (reply in 17)
17	16.439242258	172.16.30.254	172.16.30.1	ICMP	98	Echo (ping) reply id=0x31d5, seq=1/256, ttl=64 (request in 16)

Figura 3

```
› Ethernet II, Src: HewlettP_5a:7d:b7 (00:21:5a:5a:7d:b7), Dst: HewlettP_5a:74:3e (00:21:5a:5a:74:3e)
› Internet Protocol Version 4, Src: 172.16.30.1, Dst: 172.16.30.254
```

Figura 4

```
› Ethernet II, Src: HewlettP_5a:74:3e (00:21:5a:5a:74:3e), Dst: HewlettP_5a:7d:b7 (00:21:5a:5a:7d:b7)
› Internet Protocol Version 4, Src: 172.16.30.254, Dst: 172.16.30.1
```

Figura 5

```
Frame 15: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface eth0, id 0
Ethernet II, Src: HewlettP_5a:74:3e (00:21:5a:5a:74:3e), Dst: HewlettP_5a:7d:b7 (00:21:5a:5a:7d:b7)
  › Destination: HewlettP_5a:7d:b7 (00:21:5a:5a:7d:b7)
  › Source: HewlettP_5a:74:3e (00:21:5a:5a:74:3e)
    Type: ARP (0x0806)
    Padding: 000000000000000000000000000000000000
```

Figura 6

```
▾ Ethernet II, Src: HewlettP_5a:7d:b7 (00:21:5a:5a:7d:b7), Dst: HewlettP_5a:74:3e (00:21:5a:5a:74:3e)
    › Destination: HewlettP_5a:74:3e (00:21:5a:5a:74:3e)
    › Source: HewlettP_5a:7d:b7 (00:21:5a:5a:7d:b7)
      Type: IPv4 (0x0800)
▾ Internet Protocol Version 4, Src: 172.16.30.1, Dst: 172.16.30.254
      0100 .... = Version: 4
      .... 0101 = Header Length: 20 bytes (5)
    › Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
      Total Length: 84
      Identification: 0xf724 (63268)
    › Flags: 0x40, Don't fragment
      ...0 0000 0000 0000 = Fragment Offset: 0
      Time to Live: 64
      Protocol: ICMP (1)
      Header Checksum: 0xae64 [validation disabled]
      [Header checksum status: Unverified]
      Source Address: 172.16.30.1
      Destination Address: 172.16.30.254
```

Figura 7

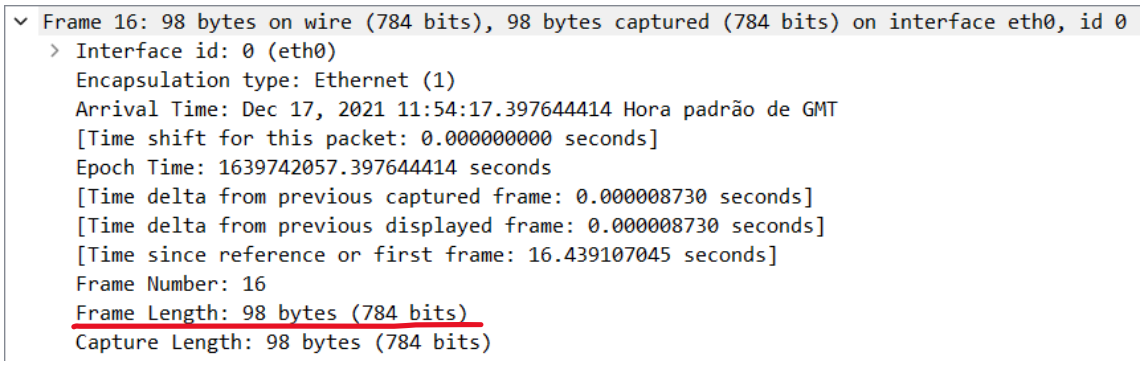

Frame 16: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface eth0, id 0
Interface id: 0 (eth0)
Encapsulation type: Ethernet (1)
Arrival Time: Dec 17, 2021 11:54:17.397644414 Hora padrão de GMT
[Time shift for this packet: 0.000000000 seconds]
Epoch Time: 1639742057.397644414 seconds
[Time delta from previous captured frame: 0.000008730 seconds]
[Time delta from previous displayed frame: 0.000008730 seconds]
[Time since reference or first frame: 16.439107045 seconds]
Frame Number: 16
Frame Length: 98 bytes (784 bits)
Capture Length: 98 bytes (784 bits)

Figura 8