Trabalho 2

Configuração de uma rede e desenvolvimento de uma aplicação de download

Relatório Final

Mestrado Integrado Em Engenharia Informática e Computação

Redes de Computadores

Professor:

Ana Cristina Costa Aguiar

Grupo 1, Turma 4:

José Miguel Botelho Mendes - 201304828

Duarte Manuel Ribeiro Pinto - 201304777

Edgar Duarte Ramos - 201305973

Faculdade de Engenharia da Universidade do Porto

Rua Roberto Frias, sn, 4200-465 Porto, Portugal

22 de Dezembro de 2015

Resumo

Este relatório incide sobre o segundo projeto da Unidade Curricular de RCOM (Redes de Computadores), do curso MIEIC.

De forma resumida, podemos afirmar que o projeto consiste na configuração de uma rede bem como no desenvolvimento de uma aplicação *download*. Este projeto divide-se, então, em duas fases de desenvolvimento, que vão ser agora descritas no relatório:

1. Desenvolvimento da aplicação de *download*, onde vai ser descrita a sua arquitetura e apresentados os resultados da sua execução.
2. A configuração de uma rede onde vão ser apresentadas as seis experiências elaboradas durante as aulas da UC.

As experiências acima referidas basearam-se na configuração de um **IP de rede**, da configuração de duas **LAN's** (*Local Area Network*), as vlan's 0 e 1, virtuais no **switch**, onde os computadores se encontravam em série, e através do **NAT** (*Network Address Translation*), e ainda de um **router** em Linux, de um **router comercial** e do **DNS** (*Domain Name System*).

Todos estes conceitos irão ser elaborados mais à frente neste relatório**.**

1 Introdução

O segundo projeto de Redes de Computadores desenvolveu-se ao longo de diversas aulas laboratoriais, tendo a primeira aula servido de exemplo dos protocolos de IETF (*Internet Engineering Task Force*), tais como o SMTP e FTP. Estes protocolos promovem a solução de problemas relacionados com ligações à Internet usando documentos RFC que descrevem os protocolos. O protocolo que implementamos neste projeto foi o FTP usando o servidor da faculdade como, por exemplo, o tom.fe.up.pt. Além disso, implementamos uma rede de computadores no laboratório e testamo-la usando aplicação do cliente FTP, que foi desenvolvido seguindo as instruções do guião e pesquisando as especificações do protocolo FTP.

No que toca à configuração de rede, o seu principal objetivo é executar uma aplicação, através de duas VLAN's. Na VLAN 2 está implementado o NAT e não na VLAN 1, sendo esta possível de ter ligação à Internet, através da VLAN 2.

Por outro lado, a aplicação de download permitiu-nos ter uma maior compreensão de como funcionam os protocolos de acesso à Internet, um servidor, o cliente e o comportamento do servidor FTP. Tendo tudo isto em conta, o grupo desenvolveu a aplicação criando o cliente FTP e a ligação TCP através de dois *sockets*.

O relatório tem a seguinte estrutura:

- **Introdução**, onde é explicado o objetivo do projeto
- **Aplicação de download**, onde é explicado o cliente FTP e a sua implementação.
- **Configuração de rede**, onde são explicados todos os passos feitos e todos os objetivos, experiência a experiência.
- **Conclusão**, onde é abordado o trabalho como um todo e analisado o que podia ser feito e a opinião geral do grupo.
- **Bibliografia**, que consiste em todos os documentos e sites utilizados pelo grupo.
- **Anexos**, que contêm o código, imagens, comandos corridos e os *logs* gravados.

2 Parte 1 - Aplicação de download

A primeira parte aborda o desenvolvimento do cliente FTP, que foi implementado na linguagem de programação C em ambiente UNIX.

Este cliente foi feito através do estudo do guião disponibilizado, certos documentos e sites e, também, o documento RFC959 que aborda o protocolo FTP e o documento RFC1738 que explica o tratamento de *URL*'s.

2.1 Arquitetura

A aplicação de download foi dividida em duas *layers*: o tratamento do URL e o cliente FTP. A layer do tratamento do URL trata de realizar um *parse* do URL e colocá-lo na struct FTPInfo, em 5 campos diferentes, o **char host**, **char password**, **char user**, **char path** e **char ftp_url**. O *parse* é feito recorrendo ao uso de uma expressão regular. Este URL é do formato

ftp://<user>:<password>@<host>/<path>. O campo **char ftp_url** contém a *string* "ftp://". Após o tratamento do URL, o **char ip** é atribuído através da função dada pelo professor, **getipAdress**. A porta usada está definida numa macro chamada **SERVER_PORT** (porta 21), sendo que é a porta usada no protocolo FTP.

```
char host[MAX_ARRAY_SIZE];
char password[MAX_ARRAY_SIZE];
char user[MAX_ARRAY_SIZE];
char path[MAX_ARRAY_SIZE];
char ftp_url[MAX_ARRAY_SIZE];
```

Figura 1: Parte da struct FTPInfo referente ao parse do URL.

As funções usadas nesta camada são a **int validURL(char * url, int size)** e **int parseURL(char * url, int size, FTPInfo * ftp)**:

- A **validURL** trata de receber o URL e comparar com a expressão regular utilizada e retorna 0 em caso de sucesso e -1 no caso de insucesso.
- A **parseURL** trata de dividir o URL em diferentes partes, o **host**, a **password**, o **user** e o **path**, acabando por colocar todos estes campos na *struct* **FTPInfo**.

```
int validURL(char * url, int size);
int parseURL(char* url, int size, FTPInfo * ftp);
```

Figura 2: Funções da camada de tratamento do URL.

A segunda *layer* trata da criação do cliente FTP. Após o tratamento do URL, é necessário o utilizador conectar-se ao cliente através de um socket TCP e, depois de se conectar, é necessário fazer download do ficheiro especificado em **char path**, ao qual foi realizado o *parse* anteriormente. Este download é feito através de um novo socket. As funções usadas nesta camada são as seguintes:

- **getipAddress**, que foi fornecida pelo professor, tratando de descodificar o IP através do URL.
- **connectSocket**, que se conecta a um *socket*.
- **closeSockets**, que fecha a conexão aos dois *sockets* criados, o socket de comunicação e o socket de onde se faz download do ficheiro. Esta função envia também o comando **quit** para fechar o cliente FTP.
- **loginHost**, que envia o comando **user <user>** e lê a sua resposta, para depois enviar o comando **pass <password>** e volta a ler a resposta dada. De seguida, retorna 0 caso o *login* esteja correto e retornando -1 se a resposta for **LOGIN_FAIL** (530).
- **setPasv**, que envia o comando **pasv** para entrar em modo passivo e lê a sua resposta para saber qual a porta a usar na criação do próximo socket. Esta porta é calculada usando o penúltimo valor * 256 + o último valor. De seguida, conecta-se a um novo socket, o socket de onde se irá fazer download.
- **retrFile**, que envia o comando **retr <path>** e lê a sua resposta. Retorna 0 caso a resposta seja **FILE_SUCCESS** (150) e então preenche o campo size da struct FTPInfo com o tamanho do ficheiro. Retorna -1 caso a resposta seja **FILE_FAIL** (550).
- **downloadFile**, que cria um novo ficheiro com o *basename* do **path** e lê através do segundo socket criado a informação e escrevendo no ficheiro criado. Acaba quando os *bytes* lidos forem igual a 0.

- **readAnswer**, que lê a resposta do servidor através do primeiro socket.
- **sendCommand**, que escreve no primeiro socket o comando recebido.

```
int getipAddress(FTPInfo * ftp);

int connectSocket(FTPInfo * ftp, char * ip, int port, int flag);

int closeSockets(FTPInfo * ftp);

int loginHost(FTPInfo * ftp);

int setPasv(FTPInfo * ftp);

int sendCommand(FTPInfo* ftp, char* command, int size);
int readAnswer(FTPInfo* ftp, char* answer, int size);

int retrFile(FTPInfo* ftp);
int downloadFile(FTPInfo *ftp);
```

Figura 3: Funções do cliente FTP.

Os descritores dos dois sockets usados, o tamanho do ficheiro, o ip e a porta a que o segundo socket se deve ligar estão também presentes na struct FTPInfo.

```
    char ip[MAX_IP_SIZE];

    int socket_comms_fd;
    int socket_data_fd;

    int fileSize;

    int pasv;

} FTPInfo;
```

Figura 4: Parte da *struct* FTPInfo referente ao cliente FTP.

A sequência das funções chamadas na **main.c** é a seguinte: **validURL**, **parseURL**, **getipAddress**, **connectSocket**, **loginHost**, **setPasv**, **retrFile**, **downloadFile**, **closeSockets**.

2.2 Resultados de download

Nos resultados do uso da aplicação, o grupo testou apenas o modo normal, com um *username* e *password*. O ficheiro escolhido é uma música com aproximadamente 3,6Mb. É possível ver também o avanço do download do ficheiro na consola, que varia consoante o tamanho definido na macro DATA_PACKET_SIZE (8192). A aplicação imprime todas as respostas conforme visto na figura 5 presente na secção anexos, apresentada mais à frente no relatório.

3 Parte 2 - Configuração da rede e análise

3.1 Experiência 1 - Configurar um IP de rede

O propósito desta experiência consiste na configuração de 2 IP's diferentes, no tux11 e no tux14 para que estes possam comunicar entre si. Após esta configuração das portas eth0 de ambos os tuxs e a adição das rotas necessárias, verificamos que existia conectividade entre os tuxs usando pings de um ip para o outro. Visto estarmos na bancada 1, usamos o comando **ifconfig eth0 172.16.10.1/24** no tux11 e **ifconfig eth0 172.16.10.254/24** no tux14.

Adicionamos também uma default *gateway* no tux11, através de **route add default gw 172.16.10.254**. Pingando ambos os ip's, obtivemos resposta em ambos os tuxs. Esta comunicação foi feita através de pacotes ARP (*Address Resolution Protocol*) que contém o endereço IP e espera uma resposta com o endereço MAC. O pacote ARP contém um endereço que varia de acordo com o protocolo usado. Após a resposta do pedido ARP, são gerados pacotes do protocolo ICMP, através do comando **ping**. Nos pacotes de ARP, o endereço IP é o endereço do computador e o MAC é o endereço físico do computador. No entanto, nos pacotes de *ping*, o IP é o endereço do computador pretendido e o MAC é o endereço do router mais próximo. Para saber o tamanho do pacote recebido, é enviado um pacote de controlo onde está escrita essa informação. De forma semelhante, para saber o protocolo usado, basta verificar o segundo e terceiro bytes do pacote de ARP. Para concluir, a interface de loopback consiste num canal de comunicação que recebe aquilo que envia, sem modificações. É usada primariamente em testes de comunicação de diagnóstico e por algumas aplicações durante o uso normal.

3.2 Experiência 2 - Implementar duas LAN's virtuais no switch

Depois de configurarmos a experiência 1, tivemos que adicionar os dois *tuxs* usados anteriormente (o tux11 e tux14) a uma *vlan* (*vlan* 10). Criamos esta *vlan* acedendo ao *switch* e correndo **vlan 10**. De seguida, adicionamos o tux11 e tux14 a essa mesma vlan, através dos comandos **interface fastethernet**, **switchport mode access**, **switchport access vlan 10**. De seguida, configuramos o IP do tux12 com **ifconfig eth0 172.16.11.1/24**. Criamos mais uma vlan (vlan 11), correndo o comando de criação acima referido. Adicionamos também o tux12 à vlan 11 com os comandos anteriores (substituindo apenas 10 por 11 no último comando). Posteriormente, tentando executar o *ping* do tux12 através do tux11 ou tux14, o *ping* não retorna resposta, pois eles não estão conectados. Existem dois domínios do *broadcast*, o primeiro consiste no tux11 e no tux14, o outro apenas no tux12. Isto pode ser concluído através dos logs guardados, pois fazendo *ping broadcast* no tux11, é detetado no tux11 e no tux14. Repetindo o processo no tux12 e no tux14, concluí-se que o domínio é igual no tux11 e no tux14, sendo diferente no tux12, que é ele próprio.

3.3 Experiência 3 - Configurar um router em linux

O objetivo desta experiência é configurar o tux14 como *router* entre as duas vlan's criadas anteriormente.

Consequentemente, criamos a interface eth1 no tux14 através de **ifconfig eth1 172.16.11.253/24**. Adicionamos também o tux14 à vlan 11 com os comandos da experiência 2. Neste ponto é necessário realizar uma outra ligação física do tux14 ao switch (acrescentando um cabo noutra porta do tux14, passando este a ter duas ligações físicas diferentes, ao invés de anteriormente que só tinha uma). De seguida, demos *enable* ao *forward* do *IP*. e desativamos o *ignore* dos *broadcasts* do protocolo ICMP. Adicionamos uma rota ao tux11 com **route add -net 172.16.11.0/24 gw 172.16.10.254** que, no primeiro argumento, identifica os endereços para o qual se quer adicionar a rota e o segundo identifica o IP para onde devem ser reencaminhados os pacotes. Usou-se os mesmos comandos no tux12 com **route add -net 172.16.10.0/24 gw 172.16.11.253**. Desta forma, criou-se a conectividade entre o tux11, tux12

e tux14. Existe uma rota no tux11, outra no tux12 e duas no tux14. Estas rotas permitem a comunicação entre os tuxs. As tabelas de redirecionamento contém 3 informações, o protocolo usado, o endereço local e o endereço remoto. Além disso, o router envia *redirects* de ICMP se: a interface de onde os pacotes vieram para o router seja a mesma de onde o pacote é routeado, se a rede do endereço IP é a mesma do endereço IP routeado, se o *kernel* está configurado para enviar *redirects* e se o pacote não diz ao router o caminho específico que tem de seguir. As mensagens de ARP e os endereços MAC observados são aqueles que foram guardados através do ***ping***, que retorna o endereço MAC através do ARP. Os pacotes de ICMP observados são causados pelo comando ***ping*** e são do tipo *echo reply* e *echo request*.

3.4 Experiência 4 - Configurar um router comercial e implementar o NAT

Nesta experiência o pretendido era configurar um router com NAT implementado. O NAT permite que os computadores da rede comuniquem com redes externas porque, sendo uma rede privada, os ip's não são reconhecidos por redes externas. Estes ip's passam pelo router, são introduzidos numa *hash table*, e o router envia um ip *routeado*, de forma a que redes externas consigam reconhecer o ip. Quando o router recebe a resposta, ocorre o processo invertido e o router envia a resposta para o ip original.

Na experiência, adicionamos o router à *VLAN* 11 da mesma forma que adicionamos os tuxs na experiência anterior, mudando apenas a porta de **interface fastethernet 0/y**. De seguida, configuramos a primeira porta do router com **interface gigabitethernet 0/0**, **ip address 172.16.11.254 255.255.255.0**, em que o primeiro argumento é o IP e o segundo a máscara utilizada e, para terminar, **no shutdown**, que mantém a configuração caso o router seja desligado. A segunda porta é configurada com os mesmos comandos da primeira, apenas mudando a porta, que é a **0/1**. Já tínhamos as rotas do tux2 criadas e apenas criamos uma *default gateway* no tux12, com **route add default gw 172.16.11.254**. As últimas rotas criadas foram as do router com **ip route 0.0.0.0 0.0.0.0 172.16.1.254** e **ip route 172.16.10.0 255.255.255.0 172.16.11.253**, criando assim uma rota interna e uma rota externa que permite que os IP's de destino 172.16.10.0-255 sejam redirecionados para o ip 172.16.11.253. Assim, configuramos rotas estáticas no router. Depois permitimos o aceitamento de redirecionamento no tux12, com **echo 1 > /proc/sys/net/ipv4/conf/all/accept_redirects**. Para verificar, voltamos a fazer **ping** e **traceroute** através dos tuxs e verificamos o redirecionamento, por exemplo um **ping** do tux12 para o tux11, vai do tux12 para o router e o router redireciona para o tux14, que envia para o tux11. Para finalizar, adicionamos a funcionalidade NAT ao router, através de **interface gigabitethernet 0/0** e **ip nat inside** e **interface gigabitethernet 0/1** e **ip nat outside**, tornando a primeira porta como interna e a segunda externa. Para adicionar a gama de endereços ao router usaram-se os comandos **ip nat pool ovrld 172.16.1.19 172.16.1.19 prefix 24** e **ip nat inside source list 1 pool ovrld overload**. Adicionamos também as *access-lists*, que mostram os acessos e permissões de pacotes, com **access-list 1 permit 172.16.10.0 0.0.0.255** e **access-list 1 permit 172.16.11.0 0.0.0.255**, onde o primeiro argumento é o IP e o segundo o máximo permitido. Como teste final, ao fazer ***ping*** a 172.16.1.254 através do tux11 resulta numa resposta positiva.

3.5 Experiência 5 - DNS

Nesta secção é suposto ter ligação à Internet, através do DNS. Para o configurar, tivemos de editar o ficheiro /etc/resolv.conf em todos os tuxs, ficheiro este que é acedido em todas as chamadas que precisam de acesso à Internet. A edição consiste em escrever "nameserver 172.16.1.1". No entanto, na nossa turma, remetendo a uma dica dada pela professora não mudamos este ficheiro, deixando-o como estava. Seguidamente, como teste, foi feito ***ping*** a www.google.com, que resultou no DNS a perguntar a informação de um domain name e a obter como resposta o tempo de vida e o tamanho do pacote de dados.

3.6 Experiência 6 - Ligações TCP

Na última experiência do projeto, o propósito era correr o cliente FTP na rede recém-criada. Correndo a aplicação no tux11 e no tux12, ela funcionou como esperado, fazendo download de um ficheiro através do servidor FTP, como descrito na Parte 1 do relatório. Assim, concluímos que a rede se encontrava bem configurada e que conseguia aceder corretamente ao protocolo FTP. Ao correr a aplicação, são abertas 2 conexões de TCP, uma que interage com o servidor e envia comandos e lê respostas e a segunda conexão que trata de fazer download do ficheiro. As conexões estão dividas em 3 fases: estabelecer a conexão, transferir dados e o término da conexão. O TCP recorre ao ARQ (Automatic Repeat reQuest) para controlar erros enquanto está a transferir dados, garantido que o segmento de TCP é passado de forma correta. Este segmento está dividido em dois: o *header* e a *data*. O *header* tem 10 campos e um opcional, e os mais relevantes são:

- 16 bits para identificar a porta emissora
- 16 bits para identificar a porta recetora
- 32 bits para o número de sequência acumulado
- 32 bits para o número de *acknowledgment*
- 4 bits para especificar o tamanho do *header* do TCP
- 9 bits para 9 flags
- 16 bits para especificar o tamanho da janela que o segmento está diposto a receber
- 16 bits para o *checksum* (bloco de data digital que deteta erros de transmissão e armazenamento)
- 16 bits para o *offset* do número de sequência

O pacote de *data* vem depois do envio do *header* que contém a informação a ser transferida. O seu tamanho pode ser calculado com o tamanho do *header*.

Além do mecanismo de ARP, o TCP tem também mecanismos de controlo de congestão, que têm como objetivo atingir *performance* máxima e evitar a congestão, que pode ser definida como o sobrecarregamento de um nó com demasiada data, deteriorando a qualidade do serviço. Os mecanismos controlam o rácio da transferência de informação, através do *acknowledgment* ou a falta do mesmo, indicando ao emissor a qualidade e condições do envio. Além disso, existem também algoritmos usados simultaneamente.

Para finalizar, o facto de existirem várias conexões de TCP não interferem com a transferência de dados, podendo até, acelerá-la, devido às limitações do tamanho da janela no pacote, como referido em cima.

Conclusão

Após a conclusão do segundo projeto de Redes e Computadores, o grupo conseguiu, de uma forma eficiente e prática, aprender os conceitos essenciais de todo o processo que este envolvia (criação de redes, DNS, etc.).

A configuração de uma rede, embora algo demorada e exigente, foi implementada de maneira simples, sendo que o grupo se sente capaz de aplicar os mesmos processos mais tarde na sua vida profissional, tendo porém noção que ainda falta aprofundar alguns conceitos para que o conhecimento seja mais consistente.

Por outro lado, a aplicação de *download,* embora tivesse sido mais simples de implementar, exigia um entendimento prévio de tudo o que esta envolvia, o que exigiu um estudo mais específico de alguns protocolos (sobre FTP por exemplo).

Todos os objetivos do grupo foram atingidos, sendo que pensamos ter realizado com sucesso este projeto. Em termos de conhecimentos, sentimos, como já referido, que este projeto foi importante, também por nunca ter sido abordado antes em nenhuma UC do curso até então realizado.

Em suma, o grupo está contente com o trabalho apresentado, sendo que sente que foi realmente uma mais-valia para cada elemento do grupo a realização do mesmo.

Referências

[1] https://en.wikipedia.org/wiki/Transmission_Control_Protocol

[2] https://en.wikipedia.org/wiki/Automatic_repeat_request

[3] https://en.wikipedia.org/wiki/Checksum

[4] https://en.wikipedia.org/wiki/Transport_layer

[5] https://pt.wikipedia.org/wiki/Payload

[6] http://stackoverflow.com/questions/259553/tcp-is-it-possible-to-achieve-higher-transfer-rate-with-multiple-connections

[7] https://wpengine.com/support/how-to-configure-your-dns/

[8] http://www.cisco.com/c/en/us/td/docs/switches/datacenter/sw/5_x/nx-os/unicast/configuration/guide/l3_cli_nxos/l3_route.html

[9] https://en.wikipedia.org/wiki/File_Transfer_Protocol

[10] http://www.cisco.com/c/en/us/support/docs/ip/routing-information-protocol-rip/13714-43.html

ANEXO

A.1 Imagens relativas a secções do relatório

```
./downloadtomislaaaav@ubuntu:~/Desktop/RCOM_Proj2/src$ ./download ftp://up201304
5g2@tom.fe.up.pt/public_html/index.html

Host: tom.fe.up.pt
User: up201304828
Password: UF5jX75g2
Path: public_html/index.html

gethostbyname: Unknown host
Exiting Program
tomislaaaav@ubuntu:~/Desktop/RCOM_Proj2/src$ ./download ftp://up201304828:UF5jX7
5g2@tom.fe.up.pt/public_html/index.html

Host: tom.fe.up.pt
User: up201304828
Password: UF5jX75g2
Path: public_html/index.html


Host: tom.fe.up.pt
User: up201304828
Password: UF5jX75g2
Path: public_html/index.html

331 Please specify the password.

230 Login successful.

Successfully logged in ip: 192.168.50.138
227 Entering Passive Mode (192,168,50,138,157,80).

Retrieving file: public_html/index.html
150 Opening BINARY mode data connection for public_html/index.html (3066 bytes).

Downloading File: 100.00%
226 Transfer complete.
221 Goodbye.

Program runned without errors.
```

Figura 5: Exemplo do cliente FTP

```
tux11:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.16.10.254   0.0.0.0         UG    0      0        0 eth0
172.16.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth0
tux11:~# arp -a
? (172.16.10.254) at 00:22:64:a6:a4:f8 [ether] on eth0
? (172.16.1.1) at <incomplete> on eth0
tux11:~#
```

Figura 6: Tabelas de ARP e rotas no tux11 na Experiência 1

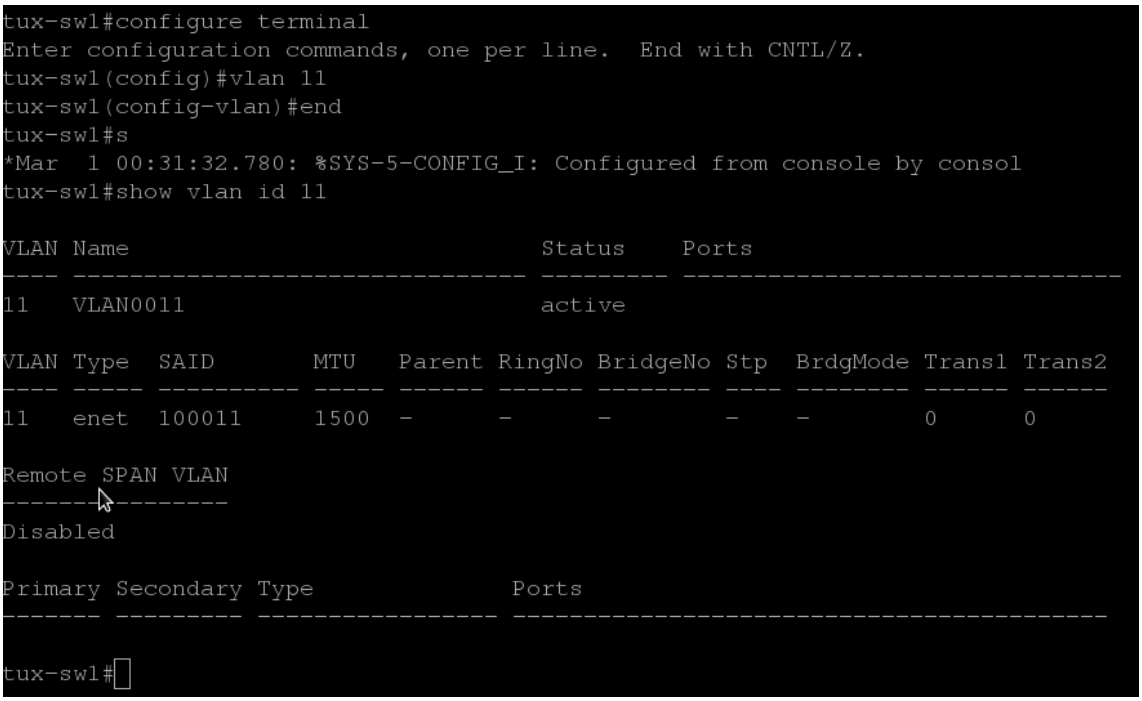

Figura 7: Criação da VLAN 11 na Experiência 2

A.2 Código do cliente FTP

Ficheiro utils.h

```
#ifndef UTILS__H

#define UTILS__H


#define MAX_ARRAY_SIZE 100

#define MAX_STRING_SIZE 255


#define SERVER_PORT 21

#define SERVER_ADDR "tom.fe.up.pt"


#define MAX_IP_SIZE    15

#define FILE_FAIL 550

#define LOGIN_FAIL 530

#define FILE_SUCCESS 150

#define DATA_PACKET_SIZE 8192
```

```
#define DONT_READ 0
#define READ 1

typedef struct {

        char host[MAX_ARRAY_SIZE];
        char password[MAX_ARRAY_SIZE];
        char user[MAX_ARRAY_SIZE];
        char path[MAX_ARRAY_SIZE];
        char ftp_url[MAX_ARRAY_SIZE];

        char ip[MAX_IP_SIZE];

        int socket_comms_fd;
        int socket_data_fd;

        int fileSize;

        int pasv;

} FTPInfo;

#endif
```

Ficheiro url.c

```
#include "url.h"
#include <string.h>
```

```
#include <stdio.h>

#include <regex.h>


int validURL(char * url, int size) {


	regex_t regExpression;

	int retReg = regcomp(&regExpression, "ftp://+[a-zA-Z0-9]+:[a-zA-Z0-9]+@+[a-zA-Z0-9._~:/?#@!$&'()*+,;=]+/[a-zA-Z0-9._~:/?#@!$&'()*+,;=]+", REG_EXTENDED);


	if (retReg) {

		fprintf(stderr, "Can't compile RegExpression\n");

		return -1;

	}


	if (!(retReg = regexec(&regExpression, url, 0, NULL, 0))) {

		return 0;

	}

	else if (retReg == REG_NOMATCH) {

		fprintf(stderr, "URL is not valid\n");

		return -1;

	}

	else {

		fprintf(stderr, "Error validating URL\n");

		return -1;

	}

}


int parseURL(char* url, int size, FTPInfo * ftp) {
```

```
char * temp_url = url + 6;

char * temp_path = strchr(temp_url, ':');
int i = 0;

while(temp_url != temp_path){
        ftp->user[i] = *temp_url;
        i++;
        temp_url++;
}
temp_url++;
ftp->user[i] = '\0';

temp_path = strchr(temp_url, '@');
i = 0;

while(temp_url != temp_path) {
        ftp->password[i] = *temp_url;
        i++;
        temp_url++;
}
temp_url++;
ftp->password[i] = '\0';

temp_path = strchr(temp_url, '/');
i = 0;
```

```
        while(temp_url != temp_path) {
                ftp->host[i] = *temp_url;
                i++;
                temp_url++;
        }
        temp_url++;
        ftp->host[i] = '\0';

        strcpy(ftp->path, temp_url);

        return 0;
}
```

Ficheiro clientFTP.c

```
#include "clientFTP.h"
#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 <sys/socket.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <libgen.h>
```

```
int getipAddress(FTPInfo * ftp)
{
        struct hostent *h;

   if ((h=gethostbyname(ftp->host)) == NULL) {
      herror("gethostbyname");
        return -1;
   }

   strcpy(ftp->ip, inet_ntoa(*((struct in_addr *)h->h_addr)));

   return 0;
}

int closeSockets(FTPInfo * ftp) {
        char answer[MAX_STRING_SIZE];

        int ret, bytesReadQuit;

        if ((ret = sendCommand(ftp, "quit\n", strlen("quit\n"))) != 0) {
                fprintf(stderr, "Error sendind quit command\n");
                return -1;
        }

        if ((bytesReadQuit = readAnswer(ftp, answer, MAX_STRING_SIZE)) == 0) {
                fprintf(stderr, "Error reading answer from quit\n");
```

```
		return -1;
	}
	fprintf(stderr, "%s\n", answer);

	close(ftp->socket_comms_fd);
	close(ftp->socket_data_fd);
	return 0;
}

int connectSocket(FTPInfo * ftp, char * ip, int port, int flag) {

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

	/*open an 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;
        }
        if (port == 21)
        ftp->socket_comms_fd = sockfd;
   else ftp->socket_data_fd = sockfd;

   int ret;
   char answer[MAX_STRING_SIZE];

   if (flag == READ) {
            if ((ret = readAnswer(ftp, answer, MAX_STRING_SIZE)) == 0) {
                        fprintf(stderr, "Error reading answer from connection\n");
                        return -1;
                }
        }

        return 0;
}

int loginHost(FTPInfo * ftp) {

        char command[4 + 1 + 1 + strlen(ftp->user)]; //o 6 vem dos espaços e do 'user' e do /n
```

```
char answer[MAX_STRING_SIZE];

int ret, bytesReadPass, bytesReadUser;

sprintf(command,"user %s\n", ftp->user);
if ((ret = sendCommand(ftp, command, strlen(command))) != 0) {
        fprintf(stderr, "Error sendind username command\n");
        return -1;
}

if ((bytesReadUser = readAnswer(ftp, answer, MAX_STRING_SIZE)) == 0) {
        fprintf(stderr, "Error reading answer from username\n");
        return -1;
}
fprintf(stderr, "%s\n", answer);


sprintf(command,"pass %s\n", ftp->password);
if ((ret = sendCommand(ftp, command, strlen(command))) != 0) {
        fprintf(stderr, "Error sendind password command\n");
        return -1;
}

if ((bytesReadPass = readAnswer(ftp, answer, MAX_STRING_SIZE)) == 0) {
        fprintf(stderr, "Error reading answer from password\n");
        return -1;
}
```

```
	fprintf(stderr, "%s\n", answer);



	int reply;
	sscanf(answer, "%d", &reply);

	if (reply == LOGIN_FAIL) {
		fprintf(stderr, "Wrong combination.\n");
		return -1;
	}

	fprintf(stderr, "Successfully logged in ip: %s\n", ftp->ip);

	return 0;
}



int setPasv(FTPInfo * ftp) {

	char answer[MAX_STRING_SIZE];

	int ret, bytesReadPasv;

	int ip1, ip2, ip3, ip4, port1, port2;

	if ((ret = sendCommand(ftp, "pasv\n", strlen("pasv\n"))) != 0) {
		fprintf(stderr, "Error sendind pasv command\n");
```

```
            return -1;
        }

        if ((bytesReadPasv = readAnswer(ftp, answer, MAX_STRING_SIZE)) == 0) {
            fprintf(stderr, "Error reading answer from pasv\n");
            return -1;
        }

        fprintf(stderr, "%s\n", answer);

        if ((sscanf(answer, "227 Entering Passive Mode (%d,%d,%d,%d,%d,%d)", &ip1, &ip2, &ip3, &ip4, &port1, &port2)) < 0){
            fprintf(stderr, "Error parsing the ports from passive mode.\n");
            return -1;
        }

        ftp->pasv = port1 * 256 + port2;

        char ip[MAX_IP_SIZE];

        // os primeiros 4 bytes serão o ip do segundo socket, que servirá para fazer download !
        sprintf(ip, "%d.%d.%d.%d", ip1, ip2, ip3, ip4);

        if (connectSocket(ftp, ip, ftp->pasv, DONT_READ) != 0) {
            fprintf(stderr, "Error connecting the data socket.\n");
            return -1;
        }
```

```
	return 0;
}

int retrFile(FTPInfo *ftp) {
	char answer[MAX_STRING_SIZE];

	char command[4 + 1 + 1 + strlen(ftp->path)];

	sprintf(command, "retr %s\n", ftp->path);

	fprintf(stderr, "Retrieving file: %s\n", ftp->path);

	int ret, bytesReadRetr;

	if ((ret = sendCommand(ftp, command, strlen(command))) != 0) {
		fprintf(stderr, "Error sendind retr command\n");
		return -1;
	}

	if ((bytesReadRetr = readAnswer(ftp, answer, MAX_STRING_SIZE)) == 0) {
		fprintf(stderr, "Error reading answer from retr\n");
		return -1;
	}
	fprintf(stderr, "%s\n", answer);

	int reply;
```

```
	sscanf(answer, "%d", &reply);

	if (reply == FILE_FAIL) {
		fprintf(stderr, "File failed opening.\n");
		return -1;
	}
	else if (reply == FILE_SUCCESS) {

		char * init = strrchr(answer, '(');

		char fill[MAX_ARRAY_SIZE];

		strcpy(fill, init);

		int size;

		sscanf(fill, "(%d bytes).", &size);
		ftp->fileSize = size;
		return 0;
	}
	else {
		fprintf(stderr, "Unexpected reply.\n");
		return -1;
	}
}

int downloadFile(FTPInfo *ftp) {
```

```
	char data_packet[DATA_PACKET_SIZE];
	int bytesRead;
	int bytesWritten = 0;

	char * fileName = basename(ftp->path);

	FILE * fileW = fopen(fileName, "w");

	while ((bytesRead = read(ftp->socket_data_fd, data_packet, DATA_PACKET_SIZE)) != 0)
{
		if (bytesRead < 0) {
			fprintf(stderr, "Error reading file.\n");
			return -1;
		}
		bytesWritten+=bytesRead;
		fprintf(stderr, "Downloading File: %.2f%%\n", (double)(bytesWritten*100)/ftp-
>fileSize);
		fwrite(data_packet, sizeof(char), bytesRead, fileW);
			if (ferror(fileW) != 0) {
				fprintf(stderr, "Error writing into file.\n");
				return -1;
			}
	}

	return 0;
}
```

```
int readAnswer(FTPInfo* ftp, char* answer, int size){
	int bytesRead;
	memset(answer, 0, size);
	bytesRead = read(ftp->socket_comms_fd, answer, size);
	return bytesRead;
}

int sendCommand(FTPInfo* ftp, char* command, int size){

	int bytesSent = write(ftp->socket_comms_fd, command, size);

	if(bytesSent <= 0){
		fprintf(stderr, "No information sent to server.\n");
		return -1;
	}

	if(bytesSent != size){
		fprintf(stderr, "Information only partly sent to server\n");
		return -1;
	}

	return 0;
}
```

Ficheiro main.c

```
#include "url.h"
#include "utils.h"
#include "clientFTP.h"
```

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

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

	FTPInfo* ftp = malloc(sizeof(FTPInfo));

	if (validURL(argv[1], sizeof(argv[1]))) {
		fprintf(stderr, "Exiting Program\n");
		return -1;
	}
	else parseURL(argv[1], sizeof(argv[1]), ftp);

	fprintf(stderr, "\nHost: %s\nUser: %s\nPassword: %s\nPath: %s\n\n", ftp->host, ftp->user, ftp->password, ftp->path);

	if (getipAddress(ftp) != 0) {
		fprintf(stderr, "Exiting Program\n");
		return -1;
	}

	if (connectSocket(ftp, ftp->ip, SERVER_PORT, READ) != 0) {
		fprintf(stderr, "Exiting Program\n");
		return -1;
	}

	fprintf(stderr, "\nHost: %s\nUser: %s\nPassword: %s\nPath: %s\n\n", ftp->host, ftp->user, ftp->password, ftp->path);
```

```
if (loginHost(ftp) != 0) {
        fprintf(stderr, "Exiting Program\n");
        return -1;
}

if (setPasv(ftp) != 0) {
        fprintf(stderr, "Exiting Program\n");
        return -1;
}

if (retrFile(ftp) != 0) {
        fprintf(stderr, "Exiting Program\n");
        return -1;
}

if (downloadFile(ftp) != 0) {
        fprintf(stderr, "Exiting Program\n");
        return -1;
}

if (closeSockets(ftp) != 0) {
        fprintf(stderr, "Exiting Program\n");
        return -1;
}

fprintf(stderr, "Program runned without errors.\n");
```

```
        free(ftp);


        return 0;
}
```

A.3 Logs guardados

Os logs encontram-se junto com este documento, ambos zipados.