U.PORTO

Mestrado Integrado em Engenharia Informática

Segundo trabalho laboratorial
Rede de computadores

RCOM

Ano Letivo de 2020/2021
201800175, Juliane de Lima Marubayashi
201800157, Guilherme Calassi
Porto, Dezembro de 2020

Índice

Índice de figuras

1 Introdução

No âmbito da unidade curricular de Redes de Computadores, foi proposto um projeto de estudo do protocolo FTP da camada de aplicação do modelo TCP/IP e da configuração de redes.

Este primeiro, foi realizado com uma aplicação de download de ficheiros em um servidor FTP com verificação das credenciais de acesso.

Já o segundo foi uma sequência de experimentos laboratoriais envolvendo switchs, routers, computadores e NATs.

Este relatório está dividido nesses dois tópicos, cada um contendo como foram realizados e suas particularidades.

2 Parte 1 - API de Download

2.1 Arquitetura da Aplicação

A arquitetura está segmentada em duas camadas majoritárias: a de aplicação e a de abstração. A primeira, controla o fluxo da execução do projeto e efetivamente faz a comunicação para com o servidor. Já a segunda, é composta por funções auxiliares e abstratas ao protocolo implementado.

2.1.1 Camada da Aplicação

Ilustrada pelo ficheiro `download.c`, esta camada é responsável por inicializar os clientes da aplicação e ligar-se ao servidor. Em termos gerais, esta é a camada principal da API. Para cumprir com seu objetivo, ela inicia os sockets, estabelece as conexões e troca informações com o servidor.

```
1  // init socket
2  int sock_requester = init_socket(ip_addr, 0);
3
4  // commands to idetification and get client port.
5  identification(sock_requester, data, port);
6  get_real_port(port, &real_port);
7
8  // init client socket
9  int sock_reader = init_socket(ip_addr, real_port);
10
11 // file to store information
12 if( (fp = fopen(data->file_name, "wb")) == NULL ) {
13         PRINT_ERR("%d\n", errno);
14         exit(-1);
15 }
16 //store information
17 while(( ret = read(sock_reader, reading, sizeof(reading)) )){
18         // read < 0, means that the connection has been closed and could not read.
19         if(ret < 0){
20                 PRINT_ERR("Error while transfering file.\n");
21                 exit(-1);
22         }
23         fwrite(reading, strlen(reading), 1, fp);
24         memset(reading, 0, strlen(reading));
25 }
26 close(sock_requester);
27 close(sock_reader);
```

2.1.2 Camada de Abstração

Essa camada é responsável por conter todo tratamento e manipulação de dados necessários para que a camada da aplicação funcione corretamente. A leitura e interpretação das respostas do servidor é orientada por uma máquina de estados, a qual lê os códigos retornados pelo cliente responsável pela conexão de controle e devolve para a camada de aplicação o seu valor. Os valores de retorno desta camada são intepretados da seguinte forma:

- Valores da forma 1xx - A máquina de estados não retorna para a camada de aplicação, visto que é esperado outra mensagem após esta. Uma excessão a este formato são os códigos 150 e 125, já que demarcam o início de tranferência de dados e não aguardam por outra mensagem de imediato. [1]
- Valores da forma 2xx - Interpretados como aceitação e sucesso. [2]
- Outros valores são retornados para a camada de aplicação para que sejam avaliados de acordo com o contexto. [3]

Ainda, foi levado em consideração a sintaxe ”xxx-”(recepção de um código seguido de um traço). Nesta situação, a máquina de estados não deve retornar para a camada de aplicação, visto que o código irá gerar multiplas linhas. Apenas perante a sintaxe ”xxx ”(recepção de um código seguido de um espaço), que irá retornar para a camada superior, com atenção à lista de exceções supra mencionadas.

2.1.3 Estruturas de Dados

Para armazenar e organizar as informações relativas à autenticação e à conexão com o servidor de um modo particular, criamos uma estrutura que permite obter de primeira mão o user, sua password, a host, o path até o ficheiro que pretende-se fazer download, o url completo e o nome do ficheiro. A struct abaixo está disponível no ficheiro `input_handler.h` e a constante definida em `macros.h`.

```
1 #define MAX_STRING_LEN 511
2 typedef struct host_request_data{
3     char user[MAX_STRING_LEN];
4     char password[MAX_STRING_LEN];
5     char host[MAX_STRING_LEN];
6     char path[MAX_STRING_LEN*2];
7     char url [MAX_STRING_LEN*5];
8     char file_name[MAX_STRING_LEN];
9 } HostRequestData;
```

2.2 Caso de Sucesso

A execução do programa é realizada através do ficheiro **download** localizado na root da API, após o projeto ter sido compilado por meio do comando **make** [4].

Para um correto funcionamento, o programa espera como argumento o endereço do ficheiro em que se deseja fazer download em um servidor ftp, seguindo a seguinte estrutura:

```
download ftp://user:pass@servidor/path
```

Com posse dessa solicitação, o programa irá tentar aceder ao servidor desejado com as credenciais indicadas. Se for um endereço válido e com as credenciais corretas, uma mensagem de boas-vindas aparecerá no ecrã e o programa então solicita que o servidor responda de modo passivo - PASV.

Abre-se outra conexão em um novo socket para a transferência dos dados que são salvos no disco rígido do cliente com o mesmo nome que estava no servidor.

3 Parte 2 - Configuração da Rede & Análises

3.1 Experiência 1

O objetivo desta esperiência foi interligar os computadores tuxy3 e tuxy4 a mesma subrede. Para tal objetivo, os comandos `ifconfig` e `route` foram utilizados.

3.1.1 Configuração

O comando `ifconfig` nos permitiu configurar o endereço ip de cada um dos computadores, já com o comando `route` estabelecemos rotas entre estes.

Uma vez configurado o ambiente de trabalho, foi testada a conectividade entre os dois computadores utilizando o comando `ping` e pôde-se concluir que, de facto, as configurações foram efetuadas com sucesso.

Em seguida, a tabela ARP dos dois computadores foi apagada e foi efetuado novamente um ping do tuxy3 para o tuxy4. O resultado foi analisado pelo programa wireshark.

3.1.2 Pacotes ARP

Uma vez que a tabela ARP tenha sido apagada dos dois computadores, o protocolo ARP foi executado. O protocolo ARP (Address Resolution Protocol) serve para mapear um endereço ipv4 na segunda camada de comunicação em um endereço físico (MAC).

Assim como pode ser conferido pelo log do wireshark [Figura A.1], primeiramente o tuxy3 envia um protocolo ARP em broadcast com os campos de IP e endereço MAC preenchidos com os valores do computador emissor (o próprio tuxy3). O protocolo pergunta qual o endereço MAC do computador que possui o ip procurado (de destino). Por não saber o endereço MAC de destino, o campo `Target MAC address` é preenchido com zeros.

Por ser um protocolo enviado em broadcast, todos os computadores da subrede irão receber o protocolo enviado pelo tuxy3. O computador que se indentificar pelo endereço IP de destino enviado pelo request, irá responder ao tuxy3 com outro protocolo ARP identificando seu endereço MAC [Figura A.2].

Uma vez registado o endereço MAC do computador de destino, o comando `ping` passa a gerar mensagens ICMP. Tais mensagens são utilizadas para fornecer relatórios de erro à fonte de origem e realizar testes de conexão [Figuras A.3 e A.4].

3.1.3 Tipos de pacotes

Para identificar se a trama recetora ethernet é ARP, IP ou ICMP deve ser inspecionado o cabeçalho do pacote. Por exemplo, caso o tipo de trama seja ARP o campo `type` do cabeçalho da trama ethernet será 0x0806, caso seja IP o campo será preenchido por 0x0800. No entanto, o datagram do endereço IP engloba o protocolo ICMP, ou seja, se o campo do tipo de serviço for preenchido com 1 no header do pacote IP, então o protocolo será ICMP.

Caso em algum momento seja preciso analisar o tamaho da trama recebida, pode-se verificar esta informação facilmente no wireshark. Na figura **??**, por exemplo, verifica-se que o tamanho da trama 16 é de 784 bits.

Ainda analisando o comando `ping` pelo wireshark , foi confirmado o ocasional envio de tramas *loopback* pelo emissor. Esta interface é importante, uma vez que permite o computador receber repostas de si mesmo e, portanto, verificar se a rede está corretamente configurada.

3.2 Experiência 2

A experiência 2 consistiu em criar duas lans virtuais (vlany0 e vlany1) não comunicáveis entre si, nas quais os computadores tuxy4 e tuxy3 fazem parte da lany0 e o computador tuxy2 da lany1.

3.2.1 Configuração

Para configurar a lany0, por exemplo, os seguintes comandos foram introduzidos na programa GKTERM o qual estava ligado ao switch:

```
configure terminal
vlan y0
end
```

Em seguida, foi preciso adicionar as portas do switch as quais deveriam ser adicionadas a lany0. Genericamente, o comando para tal seria:

```
configure terminal
interface fastetethernet 0/[num. da porta]
switchport mode access
switchport access vlan y0
end
```

Considerando que no laboratório o número da bancada era 6, da sala I321 e:

- tuxy3 : ligado a porta 6
- tuxy4 : ligado a porta 4

Os comandos efetivamente executados podem ser conferidos em anexo na sessão A.2.1.

3.2.2 Domínios da rede

Como foram configuradas duas lans não comunicáveis entre si, existem dois domínios de broadcast. Quando o tuxy3 realizar um ping em broadcast, apenas o tuxy4 (172.16.y0.254) enviará um protocolo ICMP reply. Como tuxy2 não faz parte da subrede 172.16.y0, não receberá nenhuma resposta e, portanto, não enviará respostas ao tuxy3. Isto é facilmente concluído pelo wireshark, uma vez que o ping do tuxy3 para o tuxy2 não gera nenhum log.

3.3 Experiência 3

A experiência 3 serviu como espécie de continuação à experiência 2. Anteriormente, foram criadas duas lans não comunicáveis entre si, porém nesta experiência, as lans vlany0 e vlany1 irão se comunicar utilizando o computador tuxy4 como router. Desta forma, seguindo passos semelhantes a experiência 2, o endereço 172.16.y1.253/24 do tuxy4 foi adicionado a vlan1 pela carta eth1 [Figura A.5]

3.3.1 Configuração

Para que o tuxy2 possa comunicar-se com o tuxy3, de acordo com a imagem anteriormente citada, foram adicionadas rotas da seguinte forma:

- Caso o tuxy3 queira enviar dados à rede 172.16.y1.0/24 o conteúdo deve ser redirecionado para o endereço ip 172.16.y0.254/24
- Analogamente, caso o tuxy2 queria enviar dados à rede 172.16.y0.0/24 o conteúdo deve ser redirecionado para o endereço ip 172.16.y1.253

Os comandos executados respectivamente aos tópicos acima, a fim de alcançar tais resultados, foram:

```
route add -net 172.16.60.0/24 gw 172.16.61.253  # tuxy2
route add -net 172.16.61.0/24 gw 172.16.60.254  # tuxy3
```

Adicionalmente, deve ser configurada a opção ip forward no tuxy4 de forma que ele possa fucionar como router:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
```

3.3.2 Interpretação da tabela de rotas

Para checar se os comandos de adição de rotas foram bem sucedidos, foi executado o comando `route -n`. Para o tuxy3, e.g. o resultado simplificado foi:

Destination	Gateway	GenMask	Interface
172.16.60.0	0.0.0.0	255.255.255.0	eth0
172.16.61.0	172.16.60.254	255.255.255.0	eth0

Interpretando o output, podemos traduzir tais linhas da seguinte forma: para que o tuxy3 consiga enviar dados ao IP "Destination"deve-se enviar os dados diretamente para o IP "Gateway". Um resultado semelhante foi conferido no tuxy4.

Os campos relevantes da fowarding table visualizados pelo comando `route -n` foram:

- Destination : IP de destino da informação
- Gateway: IP do computador para o qual a informação deve ser transferida para atingir a rede de destino.
- GenMask: Máscara da rede. Como a máscara da rede no laboratório possui 24 bits, podemos inferir que a GenMask dos computadores nessa rede naturalmente será 255.255.255.0.
- Flags: Pode assumir uma pluralidade de valores. Caso `U`, significa que a rota está ativa. Se apresentar a flag `G`, significa que a rota deve ser utilizada para alcançar o destino. `D` significa que a rota foi dinamicamente instalada, `M` significa que foi modificada e `R` que foi reintegrada.
- Metric: Custo associado a enviar uma mensagem do atual computador ao destino.
- Interface: Refere-se a interface de rede do computador.

3.3.3 Conexão tuxy3 com tuxy2 [ARP]

Ainda como parte de uma experiência, foi pedido para que a tabela ARP dos três computadores fosse apagada e que fosse efetuado ping do tuxy3 para o tuxy2. Como os tuxy's 3 e 2 estão em lans separadas, para que a informação seja transferida entre os dois computadores, o tuxy4 (assim como configurado) será utilizado como router.

Analisando os logs salvos pelo wireshark, verificamos que ambos os tuxy's 3 e 4 não possuem infomações do endereço MAC um do outro. Sendo assim, foram enviadas tramas ARP no início do processo de ping para que ambos os computadores pudessem reconhecer o endereço MAC de cada um. Os pares de endereços MAC e IP são referentes aos computadores de fonte e de destino.

Analogamente, o mesmo processo ocorre quando o tuxy4 reenvia a informação para o tuxy2: ambos não possuem endereço físico de um do outro e para adquirir tal informação é enviada uma trama ARP em broadcast com esse objetivo.

Neste contexto, foi possível enviar uma trama ARP do tuxy3 para o tuxy2 utilizando o tuxy4 como intermediário.

```
HewlettP_c5:61:bb  HewlettP_61:2f:4e  ARP       60 Who has 172.16.60.1? Tell 172.16.60.254
HewlettP_61:2f:4e  HewlettP_c5:61:bb  ARP       42 172.16.60.1 is at 00:21:5a:61:2f:4e
```

Figura 3.1: Tuxy3 envia trama ARP para reconhecer MAC do tuxy2 pelo tuxy4

```
HewlettP_c5:61:bb  HewlettP_61:2f:4e  ARP       60 Who has 172.16.60.1? Tell 172.16.60.254
HewlettP_61:2f:4e  HewlettP_c5:61:bb  ARP       42 172.16.60.1 is at 00:21:5a:61:2f:4e
```

Figura 3.2: Tuxy4 envia trama ARP para reconhecer MAC do tuxy3 pelo tuxy4

3.3.4 Conexão tuxy3 com tuxy2 [ICMP]

Na sequência, foram enviados pacotes ICMP.

```
172.16.60.1       172.16.61.1       ICMP      98 Echo (ping) request
172.16.61.1       172.16.60.1       ICMP      98 Echo (ping) reply
```

Figura 3.3: Pacote ICMP do tuxy3 para o tuxy2.

A respeito dos pacotes ICMP, é importante notar os pares IP e endereço MAC. Por exemplo, na imagem acima, o pacote remetente cujo IP é 172.16.60.1 (tuxy3) e com destino o pacote cujo IP é 172.16.61.1 (tuxy2), possui o endereço MAC 00:21:5a:61:2f:4e (tuxy3) como fonte. No entanto, o seu destino é o endereço MAC 00:21:5a:c5:61:bb (tuxy4). O endereço IP não é alterado, mas conforme a trama passa por diferentes computadores o endereço físico de destino é alterado.

Analogamente, quando o tuxy2 realiza o reply do ping recebido, ele envia uma trama ICMP estando o IP de destino com o mesmo valor do IP do tuxy3 e o endereço fonte o IP do tuxy2. Por outro lado, o endereço MAC de destino dessa trama é o endereço MAC do tuxy4 e com endereço MAC fonte referente ao tuxy2.

3.4 Experiência 4

Nesta experiência, foi configurado um router comercial, o qual conectava a rede criada até então à internet. Para além disso, foi preciso configurar o NAT devidamente.

3.4.1 Configuração do router

Para configurar o router, foi preciso primeiramente aderí-lo à lany1 e em seguida, configurar os endereços IP's de entrada e saída do router com os seguintes comandos:

```
# CONFIGURAR PORTA DE ENTRADA
configure terminal
interface gigabitethernet 0/[porta ligada ao switch]
# configurar endereco IP
ip address 172.16.y1.254 255.255.255.0
# no shutdown evita que a configuracao do router seja desfeita caso este seja
    desligado
no shutdown
exit

# CONFIGURAR PORTA DE SAIDA
configure terminal
interface gigabitethernet 0/[porta ligada a internet]
ip address 172.16.1.y9 255.255.255.0
no shutdown
exit
```

Para cada computador, foi preciso adicionar rotas default para que cada tux pudesse ter acesso a internet.

```
route add default gw 172.16.y1.254  # default route for tuxy2
route add default gw 172.16.y0.254  # default route for tuxy3
route add default gw 172.16.y1.254  # default route for tuxy4
```

Ambos tuxy4 e tuxy2 precisam ter como rota default, o endereço IP do router ligado a lany1.

Ainda, para que a informação do router chegue a lany0, é preciso adicionar uma rota para esta lan com o tuxy4 como gateway configurado manualmente pelo GKTERM, com o comando:

```
route ip 172.16.y0.0 255.255.255.0 172.16.y1.253
```

3.4.2 Rotas dos pacotes

Para que um computador aceda a internet, temos dois possíveis cenários:

- Caso haja rota configurada entre dois computadores, a informação há de seguir esta rota.
- Caso contrário, os pacotes irão ser enviados para rota default e o computador para o qual o pacote foi enviado irá redirecionar estes pacotes para o destino.

No caso da experiência, uma vez que tenha sido apagado a rota do tuxy2 para o tuxy3 via tuxy4, os pacotes eram redirecionados para o router e logo em seguida eram repassados para o tuxy4 na mesma. As imagens dos logs desta experiência podem ser conferidas em anexo [Figuras A.6, A.7 e A.8], onde há explicação da sequência pela legenda das imagens.

3.4.3 Configurando a NAT

A NAT (network address translation) é uma técnica que consiste em traduzir endereços privados num endereço público de forma que os computadores da rede privada possam aceder a internet. Para tal, os pacotes são enviados para fora com o endereço global do router e porta gerada pelo NAT.

Para configurar o NAT foi utilizado o código fornecido pelo guião e adaptado ao computador utilizado em laboratório. A série de comandos pode ser conferida em anexo na sessão A.4.2, assim como sua explicação.

3.5 Experiência 5

Para esta experiência foi preciso configurar o DNS (Domain Name System) responsável por traduzir endereços url, em endereços IP. O protocolo recorre a uma rede hierárquica e distribuída de servidores espalhados por diversas localidades, a fim de traduzir os endereços URL. Tais servidores possuem uma base de dados com as corresponências URL-IP.

3.5.1 Configuração

Para a configuração de um DNS, genericamente é preciso editar o ficheiro `resolv.config` e adicionar as seguintes linhas :

```
search <nome do servidor>
nameserver <ip-address>
```

Em laboratório, foram adicionadas as linhas acima de forma a aceder o servidor netlab.fe.up.pt com endereço IP 172.16.1.1:

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

3.5.2 Conexão DNS

Para que uma conexão DNS seja realizada, queries do Host para o Server são efetuadas, onde o protocolo DNS carrega o hostname desejado. Dentro da rede hierárquica, são realizadas várias trocas de tramas DNS para que por fim a mensagem alcance um servidor que contenha a tradução para tal hostname. Estas trocas podem ser conferidas na imagem em anexo na figura A.9.

3.6 Experiência 6

Nesta experiência, foi testada a API desenvolvida na primeira parte do trabalho. Com sua execução, foi observado o comportamento do protocolo TCP.

3.6.1 FTP

Com a execução da aplicação foram abertas duas conexões TCP, uma vez que para executar o protocolo FTP é precisa a abertura de dois sockets.

O controle de informação é feito na conexão TCP, a qual é responsável pelos comandos.

3.6.2 Protocolo TCP

O protocolo TCP (Transmission Control Protocol) é um protocolo da camada de transporte que promove fiabilidade da informação, evita congestionamento do receptor e verifica se os bytes foram enviados na sequencia correta no contexto da network.

Tal protocolo possui três fases: estabelecimento da conexão, fase de transmissão de dados e terminação da comunicação.

O protocolo TCP funciona com o mecânismo ARQ (Automatic Repeat Request) numa variação do método Go-Back N, uma vez que quando há um erro ou segmento fora de sequência, não é preciso reenviar todos os pacotes que ainda não receberam ACK - sendo reenviado apenas o pacote com erro e mantem-se à espera do ACK dos outros pacotes.

Para explicar um pouco sobre como a detecção de erros trabalha, é preciso adicionar algum contexto sobre como o número de sequência funciona. Cada byte numa stream (sequência de dados a ser transmitida) é numerado sequencialmente. O TCP quebra a byte stream em segmentos de tamanho máximo definido e, assim, para cada segmento é armazenado no header do TCP o número de sequência do primeiro byte do segmento para ser enviado ao receptor. Nesse cenário, quando um pacote fora de sequência é recebido pelo receptor, um ACK duplicado é enviado. Com este cenário em mente, um pacote é reenviado nas seguintes ocasiões:

- Quando 3 ACKs repetidos são enviados (pacote fora de sequência ou um pacote perdido, fast recovery);
- Quando recebe um ACK requisitando o reenvio da informação;
- Quando ocorre um timeout;

Mais especificamente, o caso de ACKs repetidos pode ser conferido na imagem em anexo [Figura A.10].

Alguns campos importantes para que este sistema de detecção de erros funcione são: campo do número de sequência, campo ACK e checkSum.

Outros campos vitais ao protocolo são os que contém os dados da porta fonte e porta de destino. No entanto, este não são relevantes para o sistema de tratamento de erros.

3.6.3 Mecânismo de controle do congestionamento TCP

O TCP é capaz de conter o congestionamento de pacotes do emissor para o receptor diminuindo a taxa de transferência de bytes.

Para explicar tal mecânismo, considere a figura A.11 em anexo. No **ponto 1**, a taxa de pacotes enviados segue um crescimento exponencial seguindo o conceito de *slow start*.

Em certo momento no **ponto 2** há uma perda de segmento por *timeout*. Assim, a taxa de trasferência de bits será diminuída pelo que:

$$Threshold = \frac{1}{2} CongestionWindow$$

$$CongestionWindow = 1$$

Detalhes sobre os conceitos de *threshold* e *congestion window* não são temas deste relatório.

Uma vez mudados os valor da *congestion window* para o valor 1, o sistema *sliding window* do emissor ficará bloqueado por certa quantia de tempo, de modo que o router possa esvaziar suas filas. Em seguida a *congestion window* irá crescer de forma exponêncial [Figura A.12 ponto 1] enquanto seu valor não exceder o valor do *threshold*. Uma vez excedido, a *congestion window* irá ser incrementada linearmente [Figura A.12 ponto 2], *congestion avoidance phase*.

Na perda de um pacote, assim como explicado anteriormente, serão enviados ACKs duplicados para o emissor e no envio de três ACKs repetidos, o valor do *congestion window* será diminuído para sua metade.

3.6.4 Alocação da banda larga

Numa experiência laboratorial seguinte foi feito o download de dois arquivos em computadores diferentes utilizando a mesma rede: inicialmente o tuxy3 começa o download do ficheiro pelo FTP e,

depois de alguns segundos, é iniciado o download do mesmo ficheiro no tuxy2. Neste momento foi possível verificar a mudança de tráfego na rede. Quando o tuxy2 terminou de realizar o download do seu ficheiro o tráfego da rede sofreu alterações novamente.

Analisando a figura A.14, verificamos que até o **ponto 1** a taxa de transferência do tuxy3 mantém-se por volta de $10^8 bit/sec$. Logo em seguida, é iniciado o download do mesmo ficheiro no tuxy 2. De forma a seguir o princípio de *Max-min fairness*, o bitrate no tuxy3 é diminuído para sua metade de maneira a ficar com uma taxa de transferência por volta de 5×10^7, uma vez que a demanda de fluxo do tuxy2 [Figura A.13] é a mesma devido ao facto de estar fazendo download do mesmo ficheiro e estarem na mesma rede.
Uma vez interrompido o download no tuxy2, a taxa de transferência no tuxy3 volta a ser 10^8, assim como observado na imagem A.14 no **ponto 3**.

4 Conclusões

Ao fim deste projeto, foi possível verificar e inferir informações à respeito da configuração de redes, principalmente acerca de sua criação com utilização de switches, routers e lans entre computadores, além da lógica por trás do DNS. Como também, construir uma base de conhecimento sólida sobre o protocolo de comunicação TCP/IP e o método FTP de fazer requisições a um servidor.

No final, com uma boa compreensão dos assuntos em questão, obtivemos resultados coerentes no funcionamento da rede local, e na transferência de dados através da API de download desenvolvida.

Referências

[1] Família Códigos 1xx. URL: https://www.serv-u.com/resource/tutorial/110-120-125-150-ftp-response-codes.

[2] Família Códigos 2xx. URL: https://www.serv-u.com/resource/tutorial/232-234-250-253-257-ftp-response-codes.

[3] Códigos Respostas FTP. URL: https://en.wikipedia.org/wiki/List_of_FTP_server_return_codes.

[4] Make Manual. URL: https://www.gnu.org/software/make/manual/make.html.

A Anexo 1 - Experiências

A.1 Experiência 1

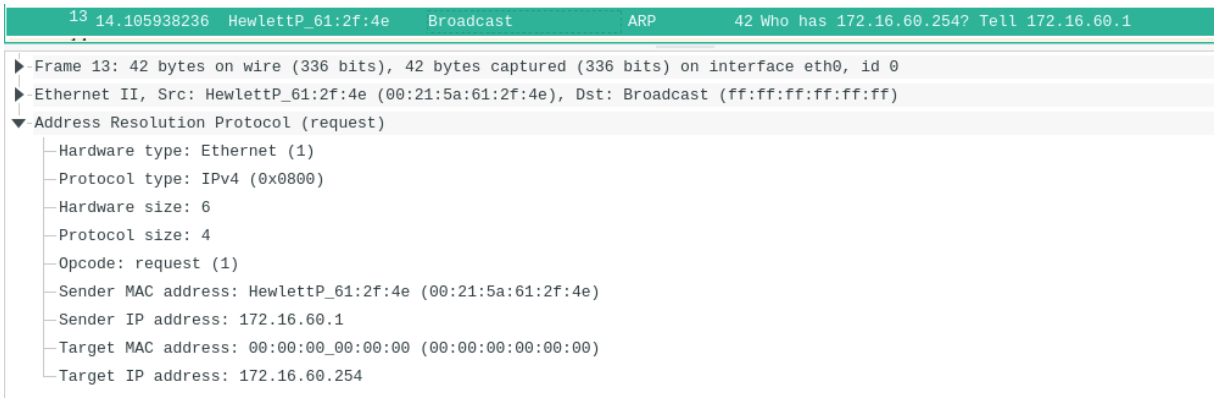

Figura A.1: Requisição de endereço MAC

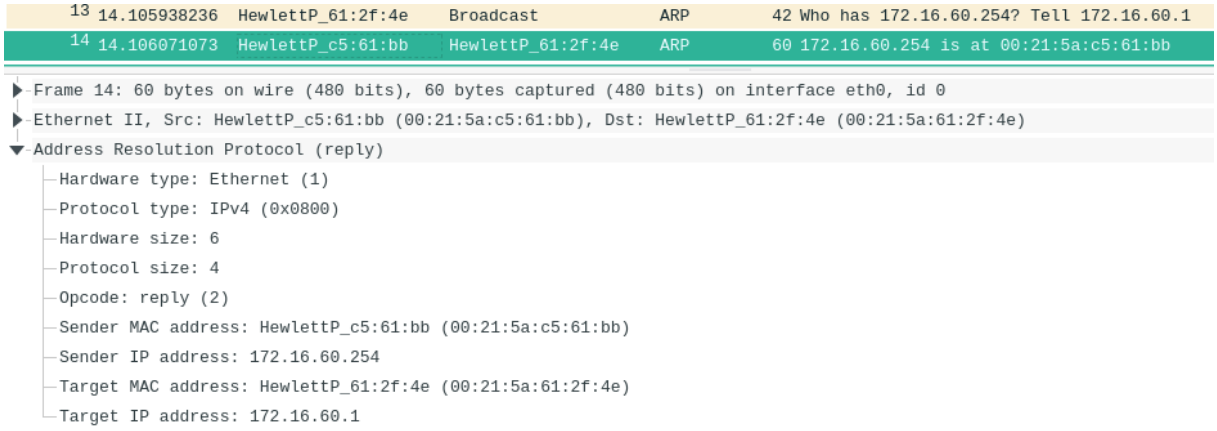

Figura A.2: Resposta de endereço MAC

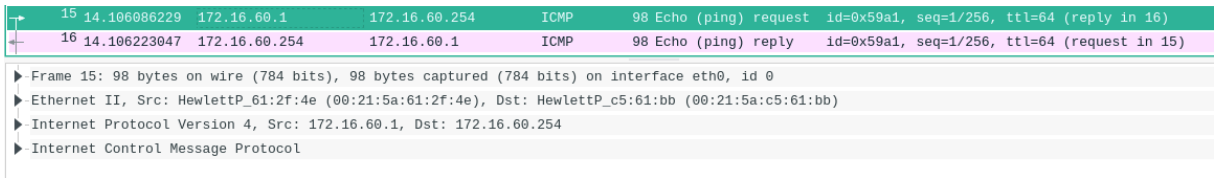

Figura A.3: Envio do protocolo ICMP

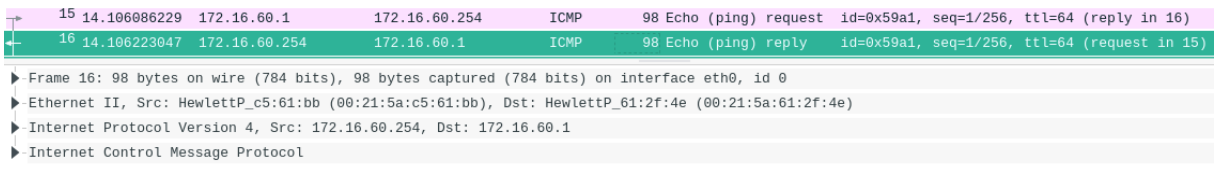

Figura A.4: Resposta do protocolo ICMP

A.2 Experiência 2

A.2.1 Comando efetivo

```
# cria vlan 60
configure terminal
vlan 60
end

# adiciona tuxy3, acessado na porta 6, a vlan 60
configure terminal
interface fastethernet 0/6
switchport mode access
switchport access vlan 60
end

# adicionar tuxy4, acessado pela porta 4, a vlan 60
configure terminal
interface fastethernet 0/4
switchport mode access
switchport access vlan 60
end
```

A.3 Experiência 3

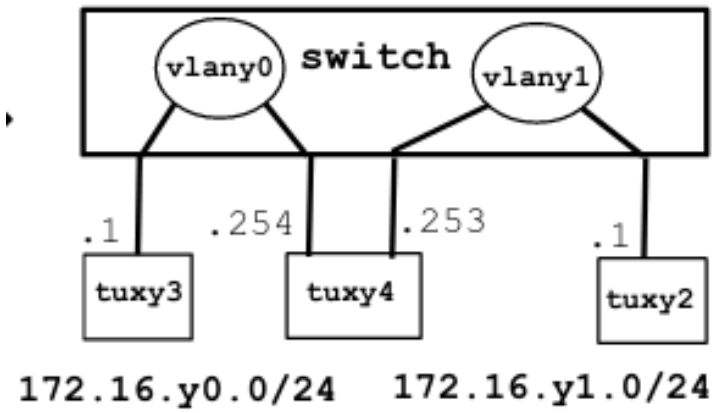

Figura A.5: Visualização da experiência

A.4 Experiência 4

A.4.1 Rota dos pacotes

```
   7 3.661248376    172.16.61.1    172.16.60.1    ICMP    98 Echo (ping) request  id=0x1c40, seq=1
› Frame 7: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface eth0, id 0
› Ethernet II, Src: HewlettP_5a:7d:9c (00:21:5a:5a:7d:9c), Dst: Cisco_d6:b1:c0 (68:ef:bd:d6:b1:c0)
› Internet Protocol Version 4, Src: 172.16.61.1, Dst: 172.16.60.1
› Internet Control Message Protocol
```

Figura A.6: Envio do pacote do tuxy2 para o Router

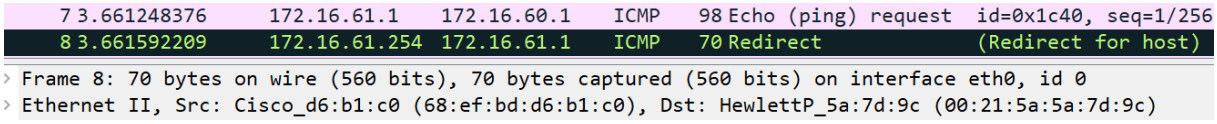

Figura A.7: Redirecionamento da resposta

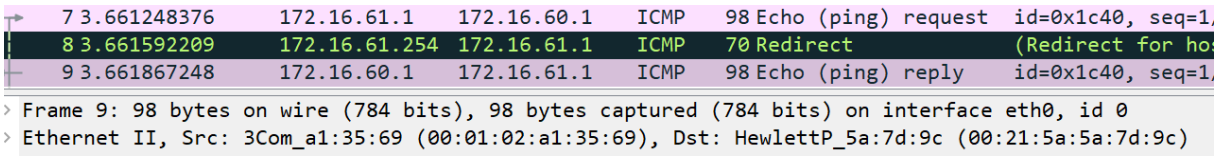

Figura A.8: Resposta do tuxy3 (IP source) via tuxy4 (MAC source) para o tuxy2 (IP e MAC dest)

A.4.2 Configurando a NAT

```
conf t
interface gigabitethernet 0/0
# ip address <ip> <mascara>
ip address 172.16.61.254 255.255.255.0
no shutdown
# ponto de entrada da nat
ip nat inside
exit

interface gigabitethernet 0/1
# ip address <ip> <mascara>
ip address 172.16.1.69 255.255.255.0
no shutdown
# ponto de saida da nat
ip nat outside
exit

# Comandos que garantem gama de enderecos
ip nat pool ovrld 172.16.1.69 172.16.1.69 prefix 24
ip nat inside source list 1 pool ovrld overload

# Na vlan0 enderecos da rede ate 7 podem aceder internet
access-list 1 permit 172.16.60.0 0.0.0.7
# Na vlan1 enderecos da rede ate 7 podem aceder internet
access-list 1 permit 172.16.61.0 0.0.7
# Como tuxy4 termina em eth0 termina em 254 e em eth1 terminam em 253, este nao
    possui internet


ip route 0.0.0.0 0.0.0.0 172.16.1.254
# define rota para a vlany0
ip route 172.16.60.0 255.255.255.0 172.16.61.253

end
```

A.5 Experiência 5

A.5.1 Conexão DNS

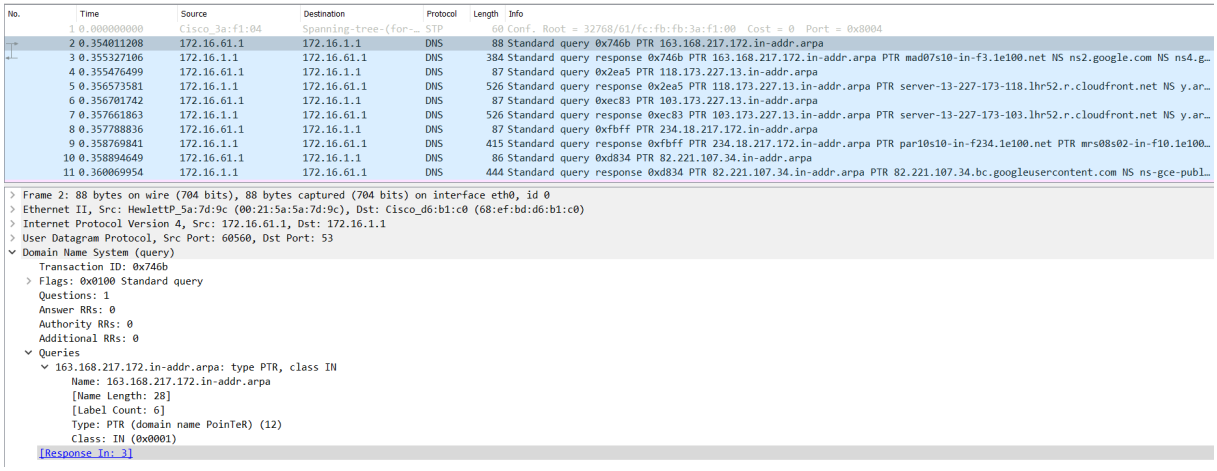

Figura A.9: Troca de pacotes DNS

A.6 Experiência 6

A.6.1 Protocolo TCP

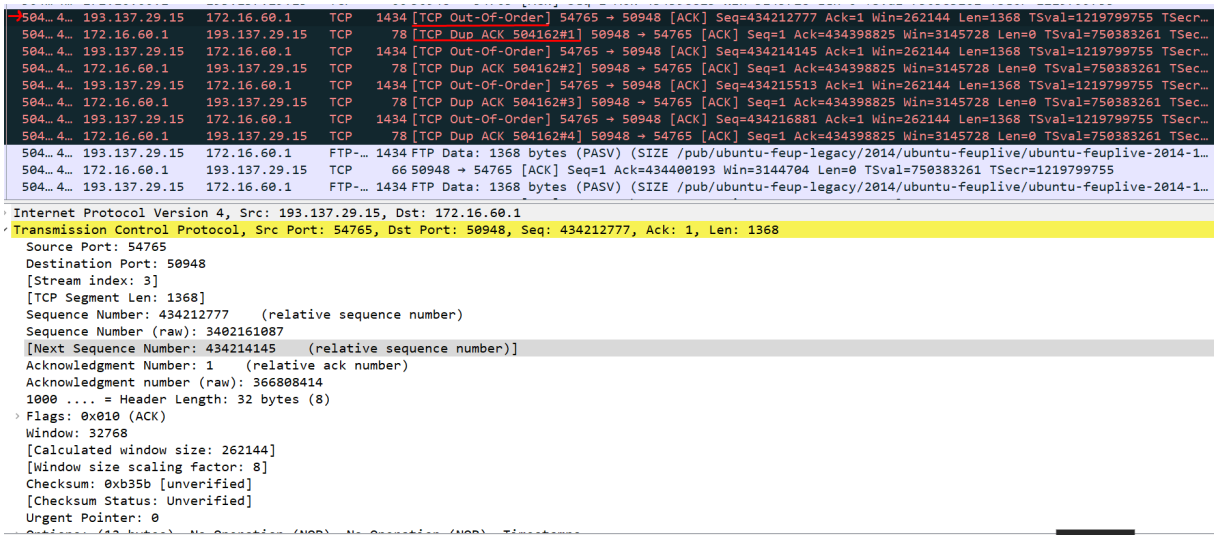

Figura A.10: Segmento fora de sequência implicou em ACKs duplicados

A.6.2 Mecânismo de controle do congestionamento TCP

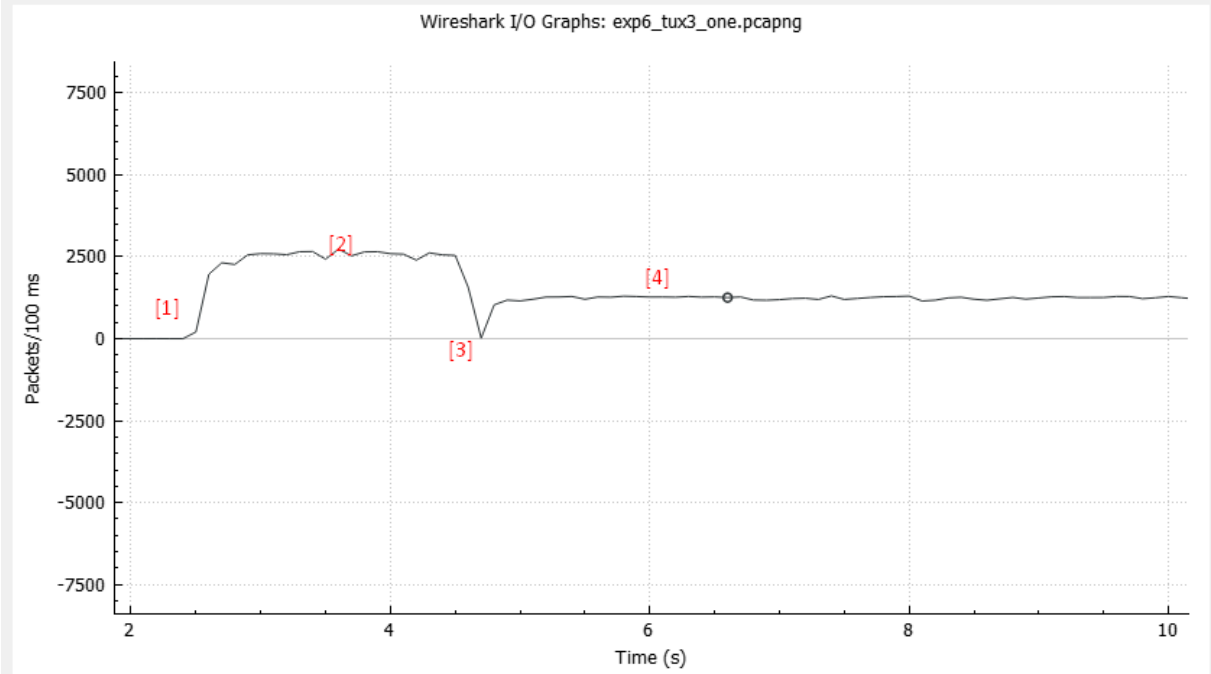

Figura A.11: Início da transmissão de um download protocolo TCP

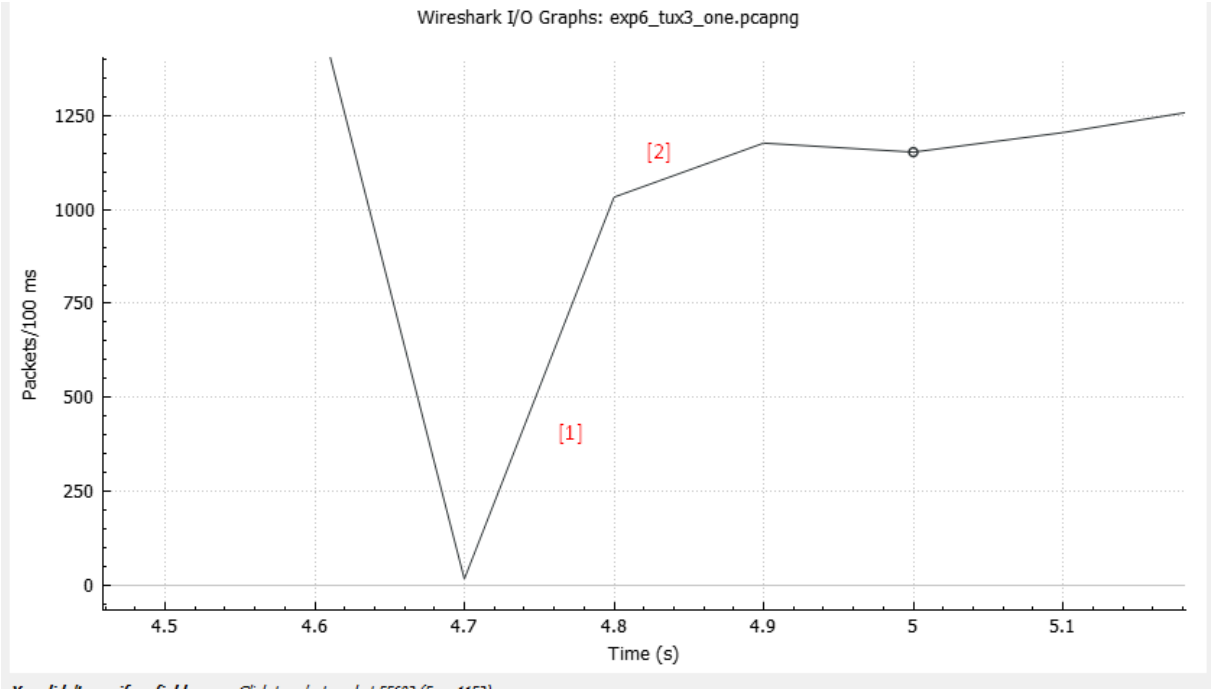

Figura A.12: Aumento da *congestion window*

A.6.3 Alocação da banda larga

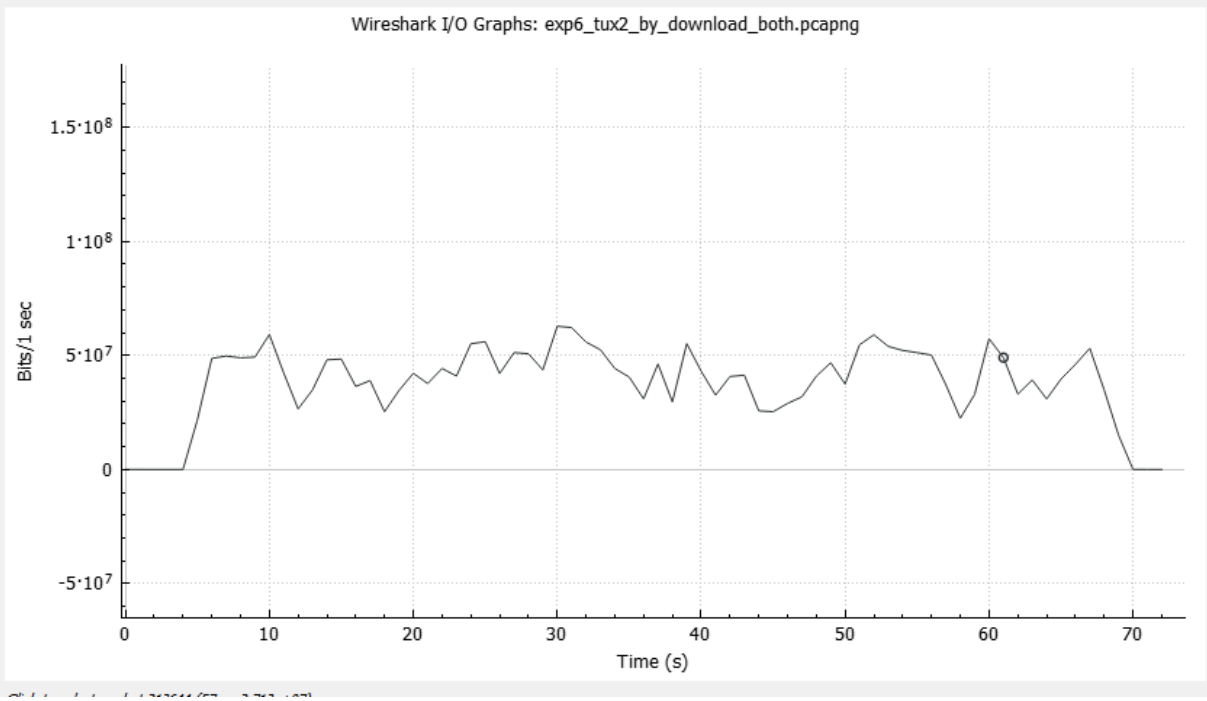

Figura A.13: Tuxy2 com bitrate por volta de $5 \times 10^7 bit/sec$

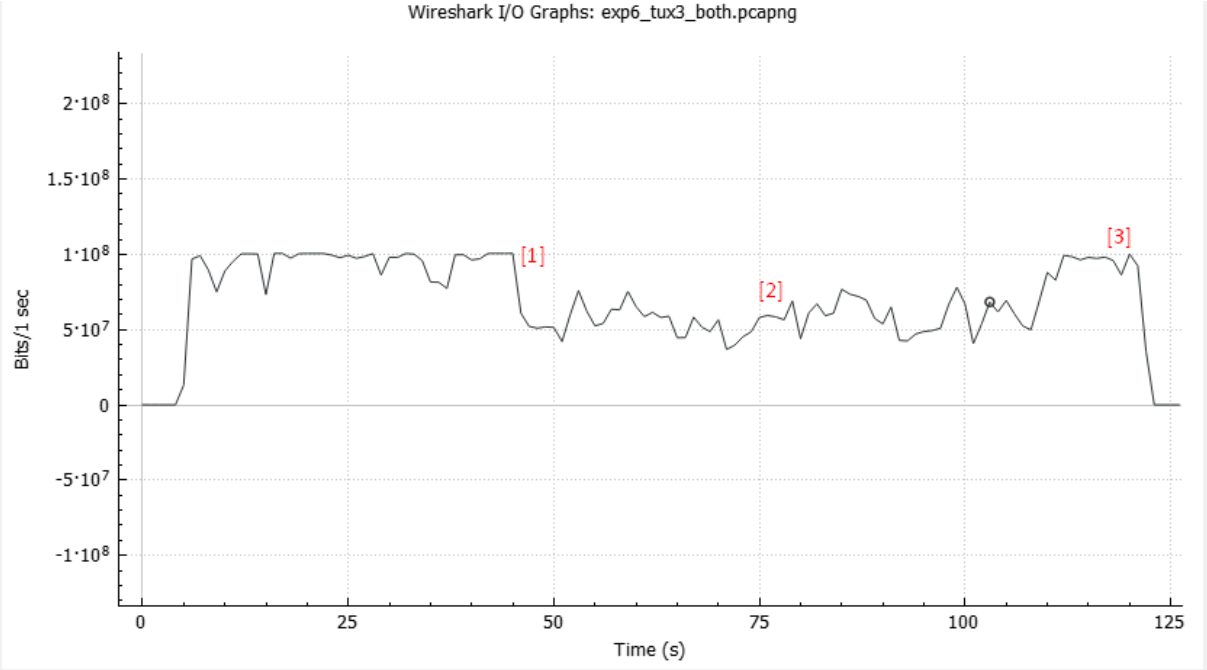

Figura A.14: Bit rate para o tuxy3 ao longo da transmissão

B Anexo 2 - Código fonte

B.1 macros.h

```
1  #include <stdio.h>
2
3  #define MAX_STRING_LEN          511     // Max size for a string malloc
4
5  #define SERVER_PORT 21              /* Default port*/
6  #define DEBUG  1                    /* Set zero to don't show prints. */
7  #define SHOW_OUTPUT         1       /* Shows API default information. */
8  #define SHOW_TEXT_RESPONSE  1       /* Shows the response text*/
9
10 // RESPONSES
11 // https://en.wikipedia.org/wiki/List_of_FTP_server_return_codes
12 #define PSV_PREL        '1'     // Positive preliminary reply.
13 #define PSV_COMPL       '2'     // Positive completion.
14 #define PSV_INTER       '3'     // Positive intermeduate completion.
15 #define NEG_TRANS       '4'     // Negative transitive completion.
16 #define NEG_PERMN       '5'     // Negative permanent completion.
17
18 #define PRINT_ERR(format, ...) \
19     do{ \
20     if (DEBUG) \
21     printf("\033[31;1mERR\033[0m: %s:%d\t\t::\033[31;1m" format "\033[0m", __FILE__,
           __LINE__, ##__VA_ARGS__ ); \
22     }while(0)
23
24 #define PRINT_SUC(format, ...) \
25     do{ \
26     if (DEBUG) \
27     printf("\033[32;1mSUC\033[0m: %s:%d\t\t::\033[32;1m" format "\033[0m", __FILE__,
           __LINE__, ##__VA_ARGS__ ); \
28     }while(0)
29
30 #define PRINT_NOTE(format, ...) \
31     do{ \
32     if (DEBUG) \
33     printf("\e[1;34mNOTE\e[0m: %s:%d\t\t::\e[1;34m" format "\e[0m", __FILE__,
           __LINE__, ##__VA_ARGS__ ); \
34     }while(0)
35
36 #define PRINTF_BLUE(format, ...) \
37     do{ \
38     if (SHOW_OUTPUT) \
39     printf("\e[1;34m"format "\e[0m", ##__VA_ARGS__ ); \
40     }while(0)
41
42
43 #define PRINTF_WHITE(format, ...) \
44     do{ \
45     if (SHOW_OUTPUT) \
46     printf("\033[1;37m"format "\033[0m", ##__VA_ARGS__ ); \
47     }while(0)
48
49
50 #define PRINTF_RESPONSE(format, ...) \
51     do{ \
52     if (SHOW_TEXT_RESPONSE) \
53     printf("\033[1;37m"format "\033[0m", ##__VA_ARGS__ ); \
54     }while(0)
```

B.2 client.h

```c
#ifndef CLIENT_H_
#define CLIENT_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 <string.h>
#include "macros.h"
#include "io.h"
#include "utils.h"


/**
 * @brief   Initialized a socket.
 *
 * @param ip_addr   IP address.
 * @param port      If the port is negative or zero, it's used the default port.
    Otherwise, the port given is used.
 * @return int      Returns the socket descriptor.
 */
int init_socket(char* ip_addr, int  port);

/**
 * @brief Reads and treat the answer of a command.
 *
 * @param sock_fd           Socket descriptor.
 * @param response_code     Response of the code.
 */
void read_rsp(int sock_fd, char* response_code);

/**
 * @brief Read the response for the pasv command.
 *
 * @param sock_fd           Scoket descriptor.
 * @param response_code     Code response.
 * @param port              Number of the port.
 */
void read_psv(int sock_fd, char* response_code, char* port);

/**
 * @brief Write the command in the socket
 *
 * @param sock_fd           Socket descriptor.
 * @param cmd               Command.
 * @param data              Data for the command.
 */
void write_cmd(int sock_fd, char* cmd, char* data);

/**
 * @brief From the port given by the pasv command get the real port.
 *
 * @param port              Port given by the pasv command, i.e "102,40"
 * @param real_port         Returns the real port as string, i.e 102*256+40
 */
void get_real_port(char port[], int* real_port);

```

```
63 #endif
```

B.3 client.c

```
1  #include "../includes/client.h"
2
3  int init_socket(char *ip_addr, int port)
4  {
5      int sockfd;
6      struct sockaddr_in server_addr;
7      int actual_port = port <= 0 ? SERVER_PORT : port;
8
9      /*server address handling*/
10     bzero((char *)&server_addr, sizeof(server_addr));
11     server_addr.sin_family = AF_INET;
12     server_addr.sin_addr.s_addr = inet_addr(ip_addr); /*32 bit Internet address
           network byte ordered*/
13     server_addr.sin_port = htons(actual_port);        /*server TCP port must be
           network byte ordered */
14
15     /*open an TCP socket*/
16     if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
17     {
18         PRINT_ERR("Not possible to open socket\n");
19         exit(0);
20     }
21
22     /*connect to the server*/
23     if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
24     {
25         PRINT_ERR("Not possible to connect to socket\n");
26         exit(0);
27     }
28
29     return sockfd;
30 }
31
32 void read_rsp(int sock_fd, char* response_code){
33     char byte;
34     int curr_state = 0;
35     int index_response = 0;
36     int is_multiple_line = 0;       // If it's a multiple line response.
37
38     while(curr_state != 2){
39         read(sock_fd, &byte, 1);
40         PRINTF_RESPONSE("%c", byte);
41
42         switch(curr_state){
43             case 0:
44                 // It's a multiple line response.
45                 if (byte == '-'){
46                     is_multiple_line = 1;
47                     curr_state = 1;
48                 }
49                 // One line response
50                 else if (byte == ' ') {
51                     // Expect another reply before proceeding with a new command
52                     if (response_code[0] == PSV_PREL && !exceptions_one_line(
                           response_code)) is_multiple_line = 1;
53                     else is_multiple_line = 0;
54                     curr_state = 1;
55                 }
56                 else response_code[index_response++] = byte;
57             break;
```

```
58
59              case 1:
60                  // If it's multiple line, goes back to the beggining of state
                        machine.
61                  if (byte == '\n' && is_multiple_line){
62                      index_response = 0;
63                      curr_state = 0;
64                  }
65                  else if (byte == '\n' && !is_multiple_line) curr_state = 2;
66              break;
67          }
68      }
69
70  }
71
72  void read_psv(int sock_fd, char* response_code, char* port){
73      char byte;
74      int curr_state = 0;
75      int index_response = 0;
76      int comman_index = 0;
77
78      while(curr_state != 5){
79          read(sock_fd, &byte, 1);
80          PRINTF_RESPONSE("%c", byte);
81
82          switch(curr_state){
83
84              // Always one line response.
85              case 0:
86                  if (byte == ' ') curr_state++;
87                  else response_code[index_response++] = byte;
88              break;
89
90              // Discard until '('
91              case 1:
92                  if (byte == '(') curr_state++;
93              break;
94
95              // Gets the ip address and discard.
96              case 2:
97                  comman_index = byte == ',' ? comman_index+1: comman_index;
98                  if (comman_index == 4){
99                      curr_state++;
100                     index_response = 0;
101                 }
102             break;
103
104             // Gets the port and store.
105             case 3:
106                 if (byte == ')') {
107                     curr_state = 4;
108                     port[index_response] = '\0';
109                 }
110                 else port[index_response++] = byte;
111             break;
112
113             // Waits for the end of line
114             case 4:
115                 if (byte == '\n') curr_state++;
116             break;
117         }
118     }
119 }
```

```
120
121 void write_cmd(int sock_fd, char* cmd, char* data){
122     write(sock_fd, cmd, strlen(cmd));
123     write(sock_fd, data, strlen(data));
124     write(sock_fd, "\n", strlen("\n"));
125 }
126
127 void get_real_port(char port[], int* real_port){
128
129     char *first_pos = malloc(4);
130     char *second_pos = malloc(4);
131     int first_pos_int, second_pos_int;
132
133     int index_comman = strcspn(port, ",");
134
135     // Separate bytes by comman.
136     memcpy(first_pos, &port[0], index_comman);
137     memcpy(second_pos, &port[index_comman+1], strlen(port)-index_comman);
138
139     // From string to integer.
140     sscanf(first_pos, "%d", &first_pos_int);
141     sscanf(second_pos, "%d", &second_pos_int);
142
143     *real_port = first_pos_int*256 + second_pos_int;
144
145     // IO
146     PRINT_SUC("Real port calculated\n");
147     io_int("REAL PORT", *real_port);
148
149     free(first_pos);
150     free(second_pos);
151 }
```

B.4 download.h

```
1  #ifndef DOWNLOAD_H
2  #define DOWNLOAD_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <errno.h>
7  #include <netdb.h>
8  #include <sys/types.h>
9  #include <netinet/in.h>
10 #include <arpa/inet.h>
11 #include "input_handler.h"
12 #include "client.h"
13 #include "io.h"
14 #include "utils.h"
15
16 struct hostent * getIP(HostRequestData *data);
17
18 /**
19  * @brief Identification steps to access the website by ftp.
20  *
21  * @param sock_fd   Socket file descriptor.
22  * @param data      Data structure with information provided by the user.
23  * @param port      Port in format i.e 410,30, where the client will access and
        retrieve file.
24  */
25 void identification(int sock_fd, HostRequestData* data, char port[]);
26
27 /**
28  * @brief Get the file name.
29  *
30  * @param data Data structure with information provided by the user.
31  */
32 void get_file_name(HostRequestData* data) ;
33
34 #endif
```

B.5 download.c

```
1  #include "../includes/download.h"
2
3  int main(int argc, char *argv[])
4  {
5          char port[10];
6          int real_port;
7
8          char response_code[4];
9          response_code[3] = '\0';
10
11         FILE* fp;
12         char *reading = malloc(10000*sizeof(char));
13         int ret;
14
15         HostRequestData *data = (HostRequestData *)malloc(sizeof(HostRequestData));
16
17         // Handle data initial data and store at struct HostRequestData;
18         input_handler(argc, argv, data);
19         struct hostent *ent = getIP(data);
20         io("IP_ADDRESS", inet_ntoa(*((struct in_addr *)ent->h_addr)));
21
22         // REQUEST ------------------------------------------------
23
24         // Init socket.
```

```
25          char *ip_addr = inet_ntoa(*((struct in_addr *)ent->h_addr));
26          int sock_requester = init_socket(ip_addr, 0);
27
28          identification(sock_requester, data, port);
29          get_real_port(port, &real_port);
30
31          // GET FILE ------------------------------------------------
32
33          // Init socket client.
34          int sock_reader = init_socket(ip_addr, real_port);
35
36          // Request file.
37          write_cmd(sock_requester, "retr ", data->path);
38          // Read answer.
39          read_rsp(sock_requester, response_code);
40          if (response_code[0] != PSV_COMPL && !exceptions_one_line(response_code)){
41                  PRINT_ERR("Not possible to transfer the file\n");
42                  exit(-1);
43          }
44
45          // Open file.
46          if( (fp = fopen(data->file_name, "wb")) == NULL ) {
47                  PRINT_ERR("%d\n", errno);
48                  exit(-1);
49          }
50
51          // Add output to file created.
52
53          while(( ret = read(sock_reader, reading, sizeof(reading)) )){
54                  // read < 0, means that the connection has been closed and could not
                        read.
55
56                  if(ret < 0){
57                          PRINT_ERR("Error while transfering file.\n");
58                          exit(-1);
59                  }
60                  fwrite(reading, strlen(reading), 1, fp);
61                  memset(reading, 0, strlen(reading));
62          }
63          free(reading);
64
65          if (exceptions_one_line(response_code)){
66                  read_rsp(sock_requester, response_code);
67                  if (response_code[0] != PSV_COMPL){
68                          PRINT_ERR("Not possible to transfer the file\n");
69                          exit(-1);
70                  }
71          }
72
73          close(sock_requester);
74          close(sock_reader);
75
76          return 0;
77  }
78
79  void identification(int sock_fd, HostRequestData *data, char port[])
80  {
81          // Get the first response code.
82          char response_code[4];
83          response_code[3] = '\0';
84
85          read_rsp(sock_fd, response_code);
86
```

```
87          // Accessing the server must be positive completion.
88          if (response_code[0] != PSV_COMPL){
89                  PRINT_ERR("Not possible to access website :: %s\n", response_code);
90                  exit(-1);
91          }
92
93          // Write the user.
94          write_cmd(sock_fd, "user ", data->user);
95          read_rsp(sock_fd, response_code);
96
97          if (response_code[0] != PSV_INTER && response_code[0] != PSV_COMPL){
98                  PRINT_ERR("Error writing user :: %s\n", response_code);
99                  exit(-1);
100         }else PRINT_SUC("User [OK]!\n");
101
102         // Write the password.
103         write_cmd(sock_fd, "pass ", data->password);
104         read_rsp(sock_fd, response_code);
105
106         if (response_code[0] != PSV_INTER && response_code[0] != PSV_COMPL){
107                 PRINT_ERR("Error writing pass :: %s\n", response_code);
108                 exit(-1);
109         }else PRINT_SUC("Pass [OK]!\n");
110
111         // Enter pasv mode.
112         write_cmd(sock_fd, "pasv", "");
113         read_psv(sock_fd, response_code, port);
114
115         if (response_code[0] != PSV_COMPL){
116                 PRINT_ERR("Error entering pasv mode :: %s\n", response_code);
117                 exit(-1);
118         }else PRINT_SUC("Get port [OK]!\n");
119
120         get_file_name(data);
121
122         // IO INTERFACE
123         label("IP CALCULATION");
124         io("PORT FIELDS", port);
125
126 }
127
128 struct hostent *getIP(HostRequestData *data)
129 {
130         struct hostent *ent;
131         if ((ent = gethostbyname(data->host)) == NULL)
132         {
133                 PRINT_ERR("Error getting host ip\n");
134                 exit(1);
135         }
136         return ent;
137 }
138
139 void get_file_name(HostRequestData* data){
140         char delim[] = "/";
141         char aux[MAX_STRING_LEN];
142         strcpy(aux, data->path);
143         char * ptr = strtok(aux, delim);
144         while(ptr != NULL){
145                 strcpy(data->file_name, ptr);
146                 ptr = strtok(NULL, delim);
147         }
148
149 }
```

B.6 input_handler.h

```
#ifndef INPUT_HANDLER_H_
#define INPUT_HANDLER_H_

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "macros.h"
#include "io.h"

typedef struct host_request_data{
    char user[MAX_STRING_LEN];
    char password[MAX_STRING_LEN];
    char host[MAX_STRING_LEN];
    char path[MAX_STRING_LEN*2];
    char url [MAX_STRING_LEN*5];
    char file_name[MAX_STRING_LEN];
} HostRequestData;

/**
 * @brief Function that handles the input given by the user
 *
 * @param argc      Number of arguments.
 * @param argv      Arguments passed.
 * @param data      Data structure to store information provided by the user.
 */
void input_handler(int argc, char **argv, HostRequestData* data);

/**
 * @brief Treats the url provided and extracts all necessary information.
 *
 * @param remain_url        URL without ftp://
 * @param remain_url_size
 * @param data              Data structure to store information provided by the user
    .
 */
void parse_input(char* remain_url, int remain_url_size, HostRequestData* data);

/**
 * @brief IO function to provide a better user interface.
 *
 * @param data          Data structure to store information provided by the user.
 */
void print_data(HostRequestData* data);


#endif
```

B.7 input_handler.c

```
#include "../includes/input_handler.h"



void input_handler(int argc, char **argv, HostRequestData* data){

    if (argc != 2){
        PRINT_ERR("Wrong number of arguments, usage:: ./download // ftp:///[<user>:<
            password>@]<host>/<url-path>");
        exit(-1);
    }
    memcpy(data->url, argv[1], strlen(argv[1]));
```

```
13
14      // Checks the presence of ftp://
15          if(strncmp("ftp://", data->url, 6) != 0){
16                  PRINT_ERR("Not a ftp:// website\n");
17          exit(-1);
18          }
19
20      // Get string after ftp://
21      int size_remain_url = strlen(data->url) - 5;
22      char remain_url[size_remain_url];
23      memcpy(remain_url, &data->url[6], size_remain_url);
24
25      // Store website info inside data struct.
26      parse_input(remain_url, size_remain_url, data);
27
28      print_data(data);
29
30
31  }
32
33  // ftp://[<user>:<password>@]<host>/<url-path>
34  void parse_input(char* remain_url, int remain_url_size, HostRequestData* data){
35      int curr_state = 0;
36      int curr_pos_parameter = 0;
37
38      for (int i = 0 ; i < remain_url_size; i++){
39          char curr_char = remain_url[i];
40
41          switch(curr_state){
42              case 0:
43                  if (curr_char == ':') {
44                      curr_state++;
45                      curr_pos_parameter = 0;
46                  }
47                  else data->user[curr_pos_parameter++] = curr_char;
48              break;
49
50              case 1:
51                  if (curr_char == '@') {
52                      curr_state++;
53                      curr_pos_parameter = 0;
54                  }
55                  else data->password[curr_pos_parameter++] = curr_char;
56              break;
57
58              case 2:
59                  if (curr_char == '/') {
60                      curr_pos_parameter = 0;
61                      curr_state ++;
62                  }
63                  else data->host[curr_pos_parameter++] = curr_char;
64              break;
65
66              case 3:
67                  data->path[curr_pos_parameter++] = curr_char;
68              break;
69          }
70      }
71
72      if (curr_state != 3){
73          PRINT_ERR("Wrong argument, usage: ftp://[<user>:<password>@]<host>/<url-path
                >\n");
74          exit(-1);
```

```
75      }
76
77  }
78
79
80  // IO
81  void print_data(HostRequestData* data){
82      // Output print.
83      label("USER INPUT");
84      io("USERNAME", data->user);
85      io("PASSWORD", data->password);
86      io("HOST   ", data->host);
87      io("PATH   ", data->path);
88      io("URL    ", data->url);
89      io("FILENAME", data->file_name);
90  }
```

B.8 io.h

```
#ifndef _IO_H_
#define _IO_H_

#include "macros.h"
#include <string.h>
#include <stdbool.h>


#define LABEL        50


void label(char* text);

void io(char* text, char* value);

void io_int(char* text, int value);

#endif
```

B.9 io.c

```
#include "../includes/io.h"


void label(char *text){
    int padding = (LABEL - strlen(text))/2;

    for (int i = 0; i < padding; i++) PRINTF_WHITE("-");
    PRINTF_WHITE("%s", text);
    for (int i = 0; i < padding; i++) PRINTF_WHITE("-");
    PRINTF_WHITE("\n");
}

void io(char* text, char* value){
        PRINTF_WHITE("%s \t", text);
        PRINTF_BLUE(" %s\n", value);
}

void io_int(char* text, int value){
        PRINTF_WHITE("%s \t", text);
        PRINTF_BLUE(" %d\n", value);
}
```

B.10 utils.h

```
#ifndef _UTILS_H_
#define _UTILS_H_

#include "macros.h"
#include <stdbool.h>

/**
 * @brief Get true if it is a exception code response that seems to be multiline but
      is just one line reply.
 *
 * @param code Pointer to code read.
 */
bool exceptions_one_line(char *code);



#endif
```

B.11 utils.c

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


bool exceptions_one_line(char *code){
        //150, 125
        return
                (*(code) == '1' && *(code+1) == '5' && *(code+2) == '0')
                || (*(code) == '1' && *(code+1) == '2' && *(code+2) == '5')
        ;
}
```