2º Trabalho Laboratorial

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

Turma 2:
Carlos Freitas - up201504749
David Falcão - up201506571
Luís Martins - up201503344

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

22 de Dezembro de 2017

Sumário

Realizado no âmbito do segundo trabalho laboratorial de Redes de Computadores, este relatório tem por base a cimentação dos conhecimentos adquiridos acerca da configuração de uma rede e do desenvolvimento de um cliente FTP (*File Transfer Protocol*).

O projeto foi concluído com sucesso. A aplicação realiza transferências sem erros e foi possível configurar corretamente a rede.

Conteúdo

1 Introdução **5**

2 Aplicação de *download* **6**
2.1 Arquitetura 6
2.2 Resultados de download 7

3 Configuração da rede **7**
3.1 Configurar uma rede IP 7
3.1.1 O que são pacotes ARP e para que são usados? 7
3.1.2 Quais são os endereços MAC e IP dos pacotes ARP e porquê? 7
3.1.3 Que pacotes gera o comando *ping*? 8
3.1.4 Quais são os endereços MAC e IP dos pacotes de *ping*? . 8
3.1.5 Como determinar se uma trama Ethernet recebida é ARP, IP, ICMP? 8
3.1.6 Como determinar o comprimento de uma trama recebida? 8
3.1.7 O que é a interface de *loopback* e porque é importante? . 8
3.2 Implementar duas LANs virtuais no *switch* 8
3.2.1 Como configurar as vlan20 e vlan21? 9
3.2.2 Quantos domínios de transmissão existem? Como se pode concluir através dos *logs*? 9
3.3 Configurar um *router* em Linux 9
3.3.1 Que rotas existem nos tuxs? Qual o significado delas? . . 9
3.3.2 Que informação contém uma entrada na tabela de encaminhamento? 9
3.3.3 Que mensagens ARP e endereços MAC associados são observados e porquê? 10
3.3.4 Que pacotes ICMP são observados e porquê? 10
3.3.5 Quais são os endereços IP e MAC associados aos pacotes ICMP e porquê? 10
3.4 Configurar um *router* comercial e implementar NAT 10
3.4.1 Como configurar uma rota estática num *router* comercial? 10
3.4.2 Quais são os caminhos seguidos pelos pacotes nas experiências realizadas? E porquê? 11
3.4.3 Como configurar o NAT num *router* comercial? 11
3.4.4 O que faz o NAT? 11
3.5 DNS 11
3.5.1 Como configurar o serviço de DNS num host? 11
3.5.2 Que pacotes são trocados pelo DNS e que informações são transportadas? 11
3.6 Ligações TCP 12
3.6.1 Quantas ligações TCP são abertas pela aplicação FTP? . 12
3.6.2 Em que ligação é transportada a informação de controlo FTP? 12
3.6.3 Quais são as fases de uma ligação TCP? 12
3.6.4 Como funciona o mecanismo ARQ TCP? Quais são as informações relevantes do TCP? 12

3.6.5 Como funciona o mecanismo de controlo de congestão TCP? Quais os campos importantes? Como evoluiu ao longo do tempo o ritmo de transferência dos dados? É de acordo com o mecanismo de controlo de congestionamento do TCP? . . . 13
3.6.6 Como reage o débito de uma ligação de dados TCP, quando existe outra ligação TCP? . . . 13

4 Conclusão **14**

A Código da aplicação de *download* **15**
A.1 Makefile . . . 15
A.2 FTP.h . . . 15
A.3 FTP.c . . . 16
A.4 parser.h . . . 23
A.5 parser.c . . . 24

B *Scripts* da *Shell* **27**
B.1 *Script* para tux21 . . . 27
B.2 *Script* para tux22 . . . 27
B.3 *Script* para tux24 . . . 27

C Configurações **28**
C.1 Experiência 1 . . . 28
C.2 Experiência 2 . . . 28
C.3 Experiência 3 . . . 28
C.4 Experiência 4 . . . 29
C.5 Experiência 5 . . . 29

D Representação da arquitetura das *Experiências* **30**

E *Logs* das *Experiências* **32**
E.1 Experiência 1 . . . 32
E.2 Experiência 2 . . . 32
E.3 Experiência 3 . . . 33
E.4 Experiência 4 . . . 35
E.5 Experiência 5 . . . 36
E.6 Experiência 6 . . . 36

1 Introdução

Este projeto foi desenvolvido no âmbito da unidade curricular de Redes de Computadores, com o objetivo de aprender a desenvolver uma aplicação de transferência usando o protocolo FTP(File Transfer Protocol) e criar e analisar uma rede.

O trabalho visou o estudo de uma rede de computadores, da sua configuração e posterior ligação a aplicação download criada pelo grupo. Para tal, foi útil seguir as recomendações e instruções fornecidas no guião.

O projeto divide-se em duas grandes componentes: o desenvolvimento de uma aplicação de download e a configuração de uma rede. Assim o relatório está dividido da seguinte forma:

- **Introdução** - Descrição sucinta do projeto e seus objetivos
- **Parte 1** - Aplicação de *download*
  - **Arquitetura** - Arquitetura e explicação sucinta da estrutura da aplicação.
  - **Resultados** - Análise dos testes realizados a aplicação
- **Parte 2** - Configuração da *rede* dividida por cada experiência sugerida no guião
  - **Configurar um IP de rede** - explicação de como foram configurados endereços IP de duas máquinas, de forma a poderem comunicar entre si.
  - **Implementar duas LANs virtuais num switch** - implementação das vlans 20 e 21, tendo a primeira 2 tuxs ligados e a segunda apenas 1.
  - **Configurar um router em Linux** - configuração do tux 4 para que fosse possível comunicar entre os três tuxs associados (comunicação entre vlans distintas).
  - **Configurar router comercial com NAT** - configuração de um router comercial com NAT para estabelecer comunicação entre as redes privadas e a Internet
  - **DNS** - configuração do serviço de DNS para que fosse possível consultar outros DNS names.
  - **Ligações TCP** - uso e análise da aplicação download na rede configurada.
- **Conclusão** - Últimas análises e opiniões do grupo em relação ao projeto
- **Anexos** - Adição do código de aplicação, dos logs de cada experiência, dos comandos de configuração.

2 Aplicação de *download*

2.1 Arquitetura

Para testar o funcionamento da rede criada, foi desenvolvida uma aplicação de download através do protocolo de transferência de ficheiros (FTP). Para tal, foram tidas em conta as normas RFC959 (para a leitura e análise das respostas provenientes do servidor) e RFC1738 (para correta utilização e tratamento dos endereços URL). Assim a aplicação está dividida em duas partes fundamentais.

A primeira parte é responsável pela leitura e interpretação dos dados fornecidos pelo utilizador na chamada da aplicação através da função *parseArgs*. Para tal, esta parte verifica através da utilização de uma expressão regular se o input tem uma estrutura válida (através da função *verifyInputRE*) e caso tal se verifique, faz parsing dos respetivos argumentos (username, password, hostname e path do ficheiro a transferir). Esta informação é posteriormente guardada numa variável global cuja estrutura é *connection_info*.

A segunda parte da aplicação é responsável pela comunicação em si. Inicialmente, com recurso à função *get_ip_addr*, descobre-se o endereço IP do hostname obtido na primeira parte da aplicação. Posteriormente é aberta uma ligação ao servidor (socket) através da porta default(21) utilizando a função *openConnection* para este mesmo endereço IP. É a partir deste ponto que se inicia a comunicação com o servidor. Para a comunicação, são enviados alguns comandos ao servidor a que este responde com mensagens de código. Esta comunicação dá-se com recurso à função *communication*. Para garantir uma interpretação correta destas respostas é analisado o primeiro dígito do código inicial de cada resposta, e age-se de acordo com este. Assim, existem as seguintes possibilidades:

- dígito 1: tenta-se ler novamente uma resposta proveniente do servidor.
- dígito 2: Houve sucesso no envio e receção da mensagem, não sendo necessário mais nada (mensagem ACK).
- dígito 3: É necessário enviar mais informação para o servidor para que este possa proceder ao pedido (exemplo após enviar o username recebe-se do servidor um comando inciado com 3 para que seja enviada a password).
- dígito 4: A mensagem enviada para o servidor é reenviada já que algum erro ocorreu no envio ou interpretação da mesma.
- dígito 5: Ocorreu um erro. Os sockets abertos são entao fechados e termina-se o programa com código 1.

Para o servidor são então enviados os comandos relativos ao login (USER e PASS através da função *logInServer*). Após um correto login, é enviado o comando PASV e calculada a nova porta para receção dos dados, fazendo parse ao conteúdo da mensagem recebida (através da função *parsePasvPort*). Abre-se então um novo socket para o mesmo servidor, mas com esta nova porta. Por fim, pede-se o ficheiro ao servidor, enviando o comando RETR com o caminho do ficheiro que se pretende, e caso este exista recebe-se um comando 1. Nesse momento, lê-se os dados do ficheiro pedido no segundo socket aberto, criando localmente um ficheiro para os guardar. Após a transferência completa do ficheiro, ou seja, quando for recebido um comando com o dígito 2 no primeiro socket e quando o segundo socket não tiver nada para ler, pode-se então proceder ao fecho de ambos os sockets e terminar o programa.

De referir também que é feito um pedido ao servidor em relação ao tamanho do ficheiro a receber de modo a que no fim da transferência seja verificada a coesão do ficheiro recebido.

2.2 Resultados de download

Esta aplicação foi testada com diversos ficheiros, tanto em modo anónimo como em modo não anónimo. Foi testada com sucesso a transferência de vários ficheiros, fazendo também variar para além do tamanho (testado até 1GB), o tipo de ficheiro (zip, iso, png, etc). De referir ainda que foram testados também vários casos de erro, tendo estes sido estes tratados corretamente pela aplicação.

3 Configuração da rede

3.1 Configurar uma rede IP

O objetivo desta experiência era configurar os endereços de IP de dois computadores (tux21 e tux24) para que estes pudessem comunicar. Após configurar os endereços IP em ambos os tuxs foi adicionada uma rota do tux21 para tux24 e foi testada a ligação através do comando ping (arquitetura da experiência representada na fig.2). Ver na figura C.1 como configurar os IPs das máquinas.

3.1.1 O que são pacotes ARP e para que são usados?

O Address Resolution Protocol (ARP) é um protocolo utilizado na resolução de endereços da camada de rede (endereços IP) em endereços da camada de ligação de dados (endereços Ethernet). Para enviar uma trama para um computador na rede, o emissor tenta descobrir o endereço MAC correspondente ao endereço IP, difundindo em *brodcast* um pacote ARP que contém o endereço IP e espera uma resposta com o endereço MAC que lhe corresponde. Exemplo:

1. Vamos considerar que o tux21 com o endereço IP: 172.16.20.1 quer comunicar com o tux24 que tem o endereço IP: 172.16.20.254 (os PCs estão na mesma rede).

2. O tux21 verifica a sua tabela ARP (pode-se ver esta informação através do comando arp –a) para saber se já existe alguma informação relativamente ao endereço físico do tux24. Caso exista, esse endereço é usado.

3. Caso o tux21 não tenha qualquer informação na tabela ARP do tux24, o protocolo ARP envia uma mensagem de *broadcast* a "questionar" (ARP Request) a quem pertence o endereço IP (neste caso 172.16.20.254).

4. O tux24 responderá à mensagem ARP enviada pelo tux21, enviando o seu endereço físico. O tux21 guardará essa informação na sua tabela ARP.

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

Como citado no exemplo acima, os endereços MAC e IP correspondem a um recetor.

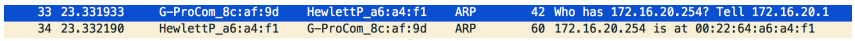


Figura 1: Protocolo ARP

3.1.3 Que pacotes gera o comando *ping*?

O ping, após obter o endereço MAC através dos pacotes ARP, gera pacotes do protocolo ICMP.

3.1.4 Quais são os endereços MAC e IP dos pacotes de *ping*?

Os pacotes de ping são enviados entre o tux1 de IP 172.16.20.1, em que o seu endereço físico é 00:0f:fe:8c:af:9d, e o tux4 de IP 172.16.20.254 com um endereço MAC de 00:22:64:a6:a4:f1. Assim sendo, verifica-se que os pacotes ICMP usam os endereços MAC obtidos pelo protocolo ARP. Estes endereços podem ser obtidos através dos logs de Wireshark como demonstrado na figura 8.

3.1.5 Como determinar se uma trama Ethernet recebida é ARP, IP, ICMP?

Para distinguir as tramas ARP das tramas IP e ICMP, é necessário analisar os 2 bytes do cabeçalho da trama Ethernet. Assim, se o tipo da trama tiver valor 0x0806 sabe-se logo que se trata de uma trama ARP. Por outro lado, como as tramas ICMP são sub-protocolo do protocolo IP, para distinguir uma trama IP de ICMP é necessário avaliar o valor do IP header. Caso esteja a 1 trata-se de uma trama ICMP, senão trama-se de uma trama IP. Esta informação está presente na figura 9

3.1.6 Como determinar o comprimento de uma trama recebida?

Se for uma trama IP, o seu cabeçalho contém essa informação. Caso se pretenda saber apenas o tamanho do pacote basta subtrair o tamanho do cabeçalho ao tamanho que se encontra na trama.

3.1.7 O que é a interface de *loopback* e porque é importante?

Um *loopback* é um canal de comunicação com apenas um ponto final. Qualquer mensagem transmitida por meio de tal canal é imediatamente recebida pelo mesmo canal.

A interface *loopback* é importante pois manda um pacote LOOP de 10 em 10 segundos para verificar a ligação e se a carta de rede se encontra configurada corretamente.

3.2 Implementar duas LANs virtuais no *switch*

O objetivo desta experiência era criar duas LANs virtuais no switch, uma com os tux21 e tux24 (VLAN 20) e outra com o tux22 (VLAN 21). Deste modo manteve-se a comunicação entre 1-4, mas nenhum destes consegue comunicar com o tux22 visto que estão em subredes diferentes (arquitetura da experiência mostrada na fig.3).

3.2.1 Como configurar as vlan20 e vlan21?

Para configurar as VLANS no switch, ligou-se este a um dos tux e utilizando o GtkTerm fez-se a configuração. Esta configuração associa cada uma das portas do switch à respetiva vlan do tux associado. Assim foram realizados os passos representados em C.2.

3.2.2 Quantos domínios de transmissão existem? Como se pode concluir através dos *logs*?

Após a configuração das VLANs, foi chamado o comando *(ping –b (...)* a partir do tux21 e do tux22. Através dos logs, analisamos que o tux21 obteve resposta ao ping broadcast por parte do tux24, enquanto o tux22 não obteve qualquer resposta. Assim concluímos que existem 2 domínios de broadcast que correspondem a cada uma das VLANs.

3.3 Configurar um *router* em Linux

O objetivo desta experiência era transformar o tux24 num router de modo a criar uma ligação entre 2 VLANs, vlan 20 e vlan 21 (arquitetura da experiência mostrada na fig.4). Para tal, a porta eth1 do tux24 foi configurada com o endereço IP 172.16.21.253 tal como podemos ver em C.3.

De seguida, foi também necessário adicionar a interface eth1 do tux24 à vlan 21, através do terminal do switch, em C.3.

Após a correta configuração da porta eth1 do tux24, foi necessário adicionar rotas no tux21 e no tux22(através do comando *route add*) de forma a que estes pudessem comunicar entre si através do tux24.

3.3.1 Que rotas existem nos tuxs? Qual o significado delas?

Para cada um dos tux21 e tux22 foi adicionada uma rota através dos comandos no terminal route add -net 172.16.21.0/24 gw 172.16.20.254 e route add -net 172.16.20.0/24 gw 172.16.21.253 respetivamente, em que o primeiro endereço identifica a gama de endereços para a qual se quer adicionar a rota, e o segundo endereço identifica o IP para o qual se deve reencaminhar o pacote. Assim estas rotas são úteis para que cada computador saiba para onde deve enviar os pacotes de forma a que eles cheguem ao destino. Neste caso, cada um dos tux21 e tux22, ao serem pedidos pacotes do seu oposto sabem que terão que enviar esses pacotes através do tux24, já que este é o único que consegue comunicar com ambos os tuxs, pois é o único que pertence a ambas as vlans. Para confirmar os dados das perguntas seguintes é possível consultar as figuras 13, 14, 15, 16

3.3.2 Que informação contém uma entrada na tabela de encaminhamento?

A tabela de encaminhamento contém para cada IP de destino, entre outras informações, qual o endereço IP para onde a trama deve reencaminhar os pacotes e a respetiva máscara. Pode existir também uma entrada default na tabela para os casos não especificados por outras entradas na tabela.

3.3.3 Que mensagens ARP e endereços MAC associados são observados e porquê?

As mensagens de ARP mostram o pedido de mapeamento do endereço de rede para um endereço físico (MAC).É possível notar nos testes realizados que as mensagens ARP pedem o endereço físico da gateway e não do destino final, já que será para esta que terão que enviar a informação uma vez que posteriormente a gateway tratará de reencaminhar para o destino final.

3.3.4 Que pacotes ICMP são observados e porquê?

Os pacotes ICMP request e ICMP reply são observados entre todos os pares de tuxs(tux21, tux 22, tux24), já que todos eles conseguem atingir os outros (existe ligação entre os 3 tuxs, sendo que o tux24 serve de ponto intermédio entre o tux 21 e tux 22). Caso tal não fosse verdade, os pacotes ICMP não chegariam ao destino.

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

Através dos logs, observando a ligação tux21 para tux22, percebe-se que os pacotes ICMP request contêm como endereço de destino o endereço MAC do tux24 e os pacotes ICMP reply, que chegam ao tux22, contêm também como endereço de origem o endereço MAC do tux24, o que era previsto, dado que é este que faz o redirecionamento da comunicação entre as duas VLANs.

3.4 Configurar um *router* comercial e implementar NAT

O objetivo desta experiência era a configuração de um *router* comercial com NAT devidamente implementado, permitindo que as redes privadas criadas pudessem comunicar para o exterior (Internet). Para tal foi configurado o router de forma a que fizesse também parte da VLAN21 (arquitetura da experiência representada na fig.5).

3.4.1 Como configurar uma rota estática num *router* comercial?

Através do seguinte comando:

- *ip route prefix mask (ip-address | interface-type interface-number [ip-address])*

Tal como foi efetuado na experiência na criação do router Rc, adicionamos uma rota estática que ligava o router a rede do laboratório

- ip route 0.0.0.0 0.0.0.0 172.16.1.254
- ip route 172.16.20.0 255.255.255.0 172.16.21.253

A segunda rota é importante para que os pacotes chegados ao router caso sejam para endereços da VLAN20 sejam reencaminhados para a carta eth1 do tux24, uma vez que é este que faz a ligação entre as 2 vlans. Se não se tratar de um endereço do tipo acima referido, então o pacote é encaminhado para a ligação do router que possibilita a ligação externa.

3.4.2 Quais são os caminhos seguidos pelos pacotes nas experiências realizadas? E porquê?

Sem ICMP redirects e a rota do tux22 para o tux24, ao fazer ping do tux22 no tux21, verificamos que os pacotes vão até o Rc, pois este tem uma rota direta para o tux24, onde são reencaminhados então para o tux24 e só depois é que vão para o tux21. Ao ser atribuída novamente a rota do tux22 para o tux24 os pacotes são diretamente encaminhados para o tux24 e depois para o tux21. Por fim ao remover mais uma vez a rota direta do tux22 para o tux24, e ao ativar o ICMP redirects quando enviamos os pacotes o router comercial responde ao tux22 com ICMP Redirect que permite a este tux fazer a ligação direta entre si e o tux24, de forma a otimizar o envio de pacotes.

3.4.3 Como configurar o NAT num *router* comercial?

Para configurar o Router da Cisco com NAT, os comandos utilizados foram os representados em C.4.

3.4.4 O que faz o NAT?

NAT (*Network address translation*) é uma técnica que consiste em reescrever, utilizando-se de uma tabela hash, os endereços IP de origem de um pacote, para que seja possível com apenas um endereço público servir vários endereços privados. Assim é possível poupar o espaço de endereçamento público, recorrendo a IPs privados.

Deste modo, é feito um mapeamento baseado no IP interno e na porta local do computador. Com esses dois dados o NAT gera um número de 16 bits usando a tabela hash. Este número é então escrito na entrada da tabela relativa a este endereço para que, quando a resposta externa chegar, seja possível reencaminhá-la para a máquina correta.

3.5 DNS

O objetivo desta experiência consistiu em adicionar um DNS. O Domain Name System (DNS) é um sistema de gestão de nomes que permite ao utilizador, através apenas de um nome, chegar a um endereço, já que existe na tabela de entradas uma entrada que relaciona cada hostname ao restivo IP (arquitetura da experiência na fig.6).

3.5.1 Como configurar o serviço de DNS num host?

Para configurar o DNS foi necessário editar o ficheiro /etc/resolv.conf de modo que neste ficasse como mostrado em C.5.

Por fim, para testar foi utilizado o comando *ping* para um domínio, por exemplo, www.google.pt e foi acedida a internet a partir de um browser.

3.5.2 Que pacotes são trocados pelo DNS e que informações são transportadas?

O DNS pede a informação contida num dado domain name, e este responde entre outras informações com o endereço IP respetivo, tal como se pode observar na figuras 22 e 23

3.6 Ligações TCP

A arquitetura desta experiência pode ser observada na fig.7.

3.6.1 Quantas ligações TCP são abertas pela aplicação FTP?

Para efetuar o download, a aplicação FTP desenvolvida abre 2 ligações TCP, a primeira para envio de comandos pelo cliente e respetivas respostas do servidor e a segunda para envio dos dados pelo servidor e respetivas respostas do cliente.

Assim, a primeira ligação dá-se entre o tux e o servidor, abrindo a porta default de comunicação FTP(porta 21) para envio de comandos, e a segunda ligação acontece entre as mesmas entidades mas desta vez a porta do servidor a abrir será a que foi especificada por este aquando da resposta ao comando pasv.

3.6.2 Em que ligação é transportada a informação de controlo FTP?

A informação de controlo FTP é enviada pela primeira ligação FTP especificada anteriormente, sendo que esta é a ligação responsável pelo envio de pedidos ao servidor e, como tal, é também responsável pelo envio das informações necessárias para que o servidor possa responder a esse mesmo pedido.

3.6.3 Quais são as fases de uma ligação TCP?

Para estabelecer uma ligação entre um participante ativo (cliente) e um participante passivo (servidor) o protocolo TCP passa por 3 fases: estabelecimento da ligação, transferência de dados e fecho da ligação. Assim, para a primeira fase recorre-se ao método *3 way handshake*.

Inicialmente o cliente envia um comando SYN para início da sua comunicação com um número de sequência A.

Após isso o servidor, notando que recebe um comando SYN, responde com uma resposta SYN+ACK sendo que o número de sequência de SYN é outro número aleatório B e o número de sequência de ACK é A+1(resposta).

Posteriormente o cliente envia o comando ACK para o servidor confirmando a sua receção do segundo SYN, e desta forma o número de sequencia desta resposta é B+1.

Após isso a comunicação entre o cliente e o servidor fica estabelecida, como se pode verificar na figura 24. De seguida dá-se a fase de transferência de dados (figura 25). No fim da da trasnferência dos pacotes de dados dá-se o fecho da ligação através do envio do comando FIN-ACK para o servindo , informando que a ligaçãp terminou. Por fim, memória alocada para o processo é libertada, como se pode verificar na figura 26.

3.6.4 Como funciona o mecanismo ARQ TCP? Quais são as informações relevantes do TCP?

O mecanismo ARQ TCP é um método de controlo de erro para transmissão de dados, que utiliza respostas do tipo acknowlegde (mensagem enviada indicando que os dados enviados estão corretos) e timeouts (tempo especificado para permitir que receba ACK do recetor relativamente ao pacote em causa). Se o emissor não receber ACK antes do tempo de timeout o pacote é retransmitido até receber uma resposta ACK antes de timeout ou atingir o máximo de retransmissões possíveis. O método utilizado em FTP é selective repeat, no qual o emissor envia vários pacotes simultaneamente sem esperar sequencialmente pelas respetivas respostas ACK.

Para análise do TCP é importante verificar que no cabeçalho de um pacote TCP estão presentes as seguintes informações:

1. portas de origem e de destino do envio
2. *sequence number* que identifica o primeiro byte da data
3. *AckNumber* que indica o próximo byte que o recetor deverá ler (implicito ACK)
4. *Window size* usado para o *flow control*
5. código detetor de erros (*checksum*)

3.6.5 Como funciona o mecanismo de controlo de congestão TCP? Quais os campos importantes? Como evoluiu ao longo do tempo o ritmo de transferência dos dados? É de acordo com o mecanismo de controlo de congestionamento do TCP?

O mecanismo de controlo de congestão TCP baseia-se no número de respostas de reconhecimentos ACK recebidos pelo remetente por unidade de tempo calculada com os dados do tempo de ida e de volta. Ou seja, pelo Round Trip Time(RTT) o protocolo consegue determinar a possibilidade de congestão com a ocorrência de timeouts.

Para evitar congestão, o TCP usa uma estratégia *multi-faced congestion-control* baseada em *congestion avoidance* com o uso de uma *congestion window*, *slow start*(para se examinar a network e determinar a capacidade válida), *fast retransmit e fast recovery* (para detetar e reparar perdas).

A congestion window é usada para limitar a quantidade de dados enviados antes de receber um ACK. Logo, de forma a que esta janela aumente se a congestão da rede diminui e diminua caso haja o aumento de congestão, o protocolo reage da seguinte maneira:

- Por cada RTT a congestion window é incrementada por um valor (additive increase)
- Sempre que ocorra um timeout, ou mais precisamente, uma perda de pacote, a congestion window é decrementada para metade, de forma a garantir o controlo do tráfego (multiplicative decrease).

Assim ao longo do tempo o ritmo de transferência como esperado, vai aumentando em varios "passos"sempre até a um máximo a partir do qual as respostas dos pacotes ja não chegam corretamente dentro do periodo de timeout e entao o número de pacotes por segundo diminui. Voltando a aumentar até haver novo pico e assim sucessivamente, tal como se pode observar na figura 27.

3.6.6 Como reage o débito de uma ligação de dados TCP, quando existe outra ligação TCP?

Ao serem utilizadas 2 ligações TCP de dados simultaneamente através do mesmo canal o débito de cada uma das ligações fica mais lento, já que o canal fica congestionado, e como dito no ponto anterior, quando o protocolo de congestão do TCP prevê a congestão da rede, este diminui a taxa de transferência. Inicialmente a taxa de transferência vai aumentado, devido ao *slow start*. Observado o gráfico da figura 28, por volta do segundo 7 o tux22 começa

a transferir o mesmo ficheiro que o tux21 já estava a transferir. Por isso a taxa de transferência desce drasticamente, devido ao *congestion control*, acabando na sua generalidade por estabilizar.

4 Conclusão

Os objetivos para este projeto foram inteiramente atingidos, quer a implementação da aplicação de download, quer a configuração de rede de computadores.

A realização deste projeto permitiu a todos os elementos do grupo ter uma melhor perceção sobre o funcionamento de redes e interiorizar os conceitos básicos do protocolo FTP.

A Código da aplicação de *download*

A.1 Makefile

```
make:
	rm -f download.o
	gcc -Wall parser.c FTP.c -o download
```

A.2 FTP.h

```
//
//  FTP.h
//

#ifndef FTP_h
#define FTP_h

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <signal.h>
#include <netdb.h>
#include <strings.h>
#include <ctype.h>

#define SERVER_PORT 21
#define MAX_IP_LENGTH 16

/**
 * Send a message to the file descriptor sokfd
 * with the following structure: message+param+"\r\n".
 * The 2 final characters represent the end of message
*/
int sendMessage(int sockfd, char* message, char* param);
/**
 * Get first number from the response gotten
*/
int getCodeResponse(int sockfd, char* response);
/**
 * Send a message to the server
 * and wait a response from it
*/
int communication(int sockfd,char* message,char* param);
/*
 * Send log in information to the server
*/
int logInServer(int sockfd);
```

```
/**
 * get the ip address from the hostname given
*/
char* get_ip_addr();
/**
 * open a new connection to the specified port
*/
int openConnection(int port,int isCommandOpen);
/**
 * get the filename from the given path
*/
char* getFilename();
/**
 * create a local file and
 * get file data from the data socket
*/
int getFile();
/**
 * main function
*/
int main(int argc, char** argv);

#endif /* FTP_h */
```

A.3 FTP.c

```
#include "parser.h"
#include "FTP.h"

static connection_info* connection;


int sendMessage(int sockfd, char* message, char* param){
  int bytes;
  char* total_message = (char*) malloc(MAX_STRING_LENGTH);
  memset(total_message, 0, MAX_STRING_LENGTH);
  strcat(total_message,message);
  if(param != NULL)
    strcat(total_message, param);
  strcat(total_message, "\r\n");
  /*send a string to the server*/
  bytes = write(sockfd, total_message, strlen(total_message));
  return bytes;
}


int readResponse(int sockfd,char* code){
  int bytes=0;
  memset(code, 0, 3);
```

```
  char maybecode[3];
  char buf;
  int i=0, state=0, finish = 0;
  do {
        bytes += read(sockfd, &buf, 1);
        printf("%c", buf);
        switch(state) {
           case 0:
                code[i] = buf;
                i++;
                if(i > 3) {
                   if(buf != ' '){
                           state = 1;
                           i=0;
                   } else
                        state = 2;
                }
                break;
           case 1:
                if(isdigit(buf)) {
                  maybecode[i] = buf;
                  i++;
                  if(i==3) {
                      state = 3;
                      i=0;
                  }
                }
                break;
           case 2:
                if(buf == '\n')
                   finish = 1;
                break;
           case 3:
                if((maybecode[0] == code[0]) &&
           (maybecode[1] == code[1]) &&
           (maybecode[2] == code[2])){
                  if(buf == '-')
                        state = 1;
                   else
                        state = 2;

                } else {
                   state = 1;
                }
                break;
        };

   }while( finish != 1);
  return bytes;
}

int readData(int sockfd, char* response) {
```

```
  int bytes = 0;
  memset(response, 0, MAX_STRING_LENGTH);
    bytes = read(sockfd, response, MAX_STRING_LENGTH);
  return bytes;
}


int getCodeResponse(int sockfd,char* response){
  int responseCode;
  responseCode = (int) response[0]-'0';
  if(responseCode == 5) {
          close(sockfd);
    exit(1);
  }

  return responseCode;
}


int readOtherResponse(int sockfd, char* response, char* message) {
    int bytes;
    char* all_resp = (char*) malloc(MAX_STRING_LENGTH);
    memset(all_resp, 0, MAX_STRING_LENGTH);
    bytes = read(sockfd, all_resp, MAX_STRING_LENGTH);
    printf("%s", all_resp);
    if(bytes > 0) {
        memset(response, 0, 3);
        response[0] = all_resp[0];
        response[1] = all_resp[1];
        response[2] = all_resp[2];
        if(strcmp(message, "pasv") == 0)
            parsePasvPort(all_resp);
        else
            parseSize(all_resp);
    }
    return bytes;
}


int communication(int sockfd,char* message,char* param){
  char* response = (char*) malloc(3);

  int finalcode;
  int bytes;

  do{

      sendMessage(sockfd, message, param);

      do{
          if((strcmp(message,"pasv") == 0) ||
             (strcmp(message, "SIZE ") == 0)) {
```

```
            bytes = readOtherResponse(sockfd, response, message);
        } else {

            bytes = readResponse(sockfd, response);
        }
        if(bytes > 0) {

            finalcode = getCodeResponse(sockfd, response);

            if(finalcode == 1 && strcmp("retr ", message) == 0) {

                //get data and create file
                getFile();
                close(connection->data_socket);
            }
        }

    }while(finalcode == 1);

  }while(finalcode == 4);

  return finalcode;
}



int logInServer(int sockfd){
    printf(" > Username will be sent\n");
  int response = communication(sockfd, "user ", connection->user);
  if(response != 3){
      return 1;
  }
  printf(" > Username correct. Password will be sent\n");
  response = communication(sockfd, "pass ", connection->password);
  if(response != 2){
      fprintf(stderr, "%s", "User or password incorrect\n");
      return 1;
  }
  printf(" > User logged in\n");
  return 0;
}


char* get_ip_addr(){
    struct hostent *h;
    char* ip = (char*) malloc(MAX_IP_LENGTH);
    memset(ip, 0, MAX_IP_LENGTH);
    printf("[HOST: %s]\n", connection->hostname);
    if ((h=gethostbyname(connection->hostname)) == NULL) {
        herror("gethostbyname");
        exit(1);
    }
```

```
    ip = inet_ntoa(*((struct in_addr *)h->h_addr));
    return ip;
}


int openConnection(int port,int isCommandConnection){

  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(connection->ip);
  server_addr.sin_port = htons(port);

  /*open an TCP socket*/
  if ((sockfd = socket(AF_INET,SOCK_STREAM,0)) < 0) {
      perror("Error open socket connection ");
      exit(0);
  }


  /*connect to the server*/
  if(connect(sockfd,(struct sockaddr *)&server_addr,sizeof(server_addr)) < 0){
      perror("connect()");
      exit(0);
  }

  char* openResponse = (char*) malloc(3);
  int code;
  //TODO verify this loop
  if(isCommandConnection){
    do{
      readResponse(sockfd, openResponse);
      code = getCodeResponse(sockfd, openResponse);
    }while(code != 2);
  }
  return sockfd;
}

char* getFilename(){
    char* filename = (char*) malloc(MAX_STRING_LENGTH);
    memset(filename, 0, MAX_STRING_LENGTH);
    unsigned int i=0, j=0, state=0;
    int length = strlen(connection->file_path);
    while(i<length){
        switch (state) {
            case 0:
                if(connection->file_path[i] != '/'){
                    filename[j] = connection->file_path[i];
                    j++;
```

```
                } else {
                    state = 1;
                }
                i++;
                break;
            case 1:
                memset(filename, 0, MAX_STRING_LENGTH);
                state = 0;
                j=0;
                break;
        }
    }
    printf("[Filename %s]\n", filename);
    return filename;
}


int getFile(){
    char* filename = getFilename();

    char* message = (char*) malloc(MAX_STRING_LENGTH);
    unsigned int bytesReaded;
    unsigned int totalBytes=0;

    FILE* filefd = fopen(filename, "w");
    if (filefd == NULL)
    {
        fprintf(stderr, "%s", " > Error opening file to write!\n");
        exit(1);
    }
    printf(" > Reading file\n");
    while((bytesReaded = readData(connection->data_socket, message)) > 0){
        totalBytes += bytesReaded;
        fseek(filefd, 0, SEEK_END);
        fwrite(message, sizeof(unsigned char), bytesReaded, filefd);
    }
    fclose(filefd);
    if(totalBytes <= 0)
        fprintf(stderr, "%s", " > Error reading the file\n");
    return totalBytes;
}

int verifyFileSize() {
    char* filename = getFilename();
    FILE* filefd = fopen(filename, "r");
    fseek(filefd, 0L, SEEK_END);
    int size = ftell(filefd);
    fseek(filefd, 0L, SEEK_SET);
    fclose(filefd);
    if(size == connection->size)
        return 1;
    else
```

```
        return 0;
}


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

    int commandSocket;
    if(argv[1] == NULL) {
        printf(" > Error.Use the structure: ftp://<username>:<password>@<host>/<file>\n");
        exit(1);
    }
    if((connection = parseArgs(argv[1])) == NULL){
       printf(" > Input values are not valid! Please try again\n");
       exit(1);
    }
    connection->ip = get_ip_addr();
    printf("[IP address: %s]\n", connection->ip);

    //open connection to the server to send commands
    commandSocket = openConnection(SERVER_PORT, 1);

    //error logging In
    if(logInServer(commandSocket) != 0){
      fprintf(stderr, "%s", " > Error logging in. Please try again!\n");
      close(commandSocket);
      exit(1);
    }
    //send size command
    printf(" > Size command will be sent\n");
    communication(commandSocket, "SIZE ", connection->file_path);

    //send pasv and get port to receive the file
    printf(" > Pasv command will be sent\n");
    communication(commandSocket, "pasv", NULL);

    printf(" > Data socket will be opened\n");
    //open the new socket to receive the file
    connection->data_socket = openConnection(connection->data_port, 0);

    printf(" > Retr message will be sent\n");
    //send retrieve command to receive the file
    int finalcommandResponse;
    finalcommandResponse = communication(commandSocket, "retr ", connection->file_path);
    close(commandSocket);
    if(finalcommandResponse != 2) {
        fprintf(stderr, "%s", " > Error getting file or sending retr\n");
        exit(1);
    } else {
        if(verifyFileSize()) {
            exit(0);
        } else {
            fprintf(stderr, "%s\n", " > The received file is probably damaged\n");
```

```
            exit(1);
        }
    }

}
```

A.4 parser.h

```
//
//  parser.h
//

#ifndef parser_h
#define parser_h

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <regex.h>

#define MAX_STRING_LENGTH 256

typedef struct{
    char* user;
    char* password;
    char* hostname;
    char* file_path;
    char* ip;
    int data_port;
    int data_socket;
    long size;

} connection_info;

/**
 * Parse user input and
 * create a new connection_info
 * structure with the information gotten
*/
connection_info* parseArgs(char* input);
/**
* Parse message gotten as response for PASV
* command and get the port to the data socket
*/
int parsePasvPort(char* msgToParse);
/**
 * Parse size message response
 * to get the file size
*/
```

```
int parseSize(char* response);

#endif /* parser_h */
```

A.5 parser.c

```
//
//  parser.c
//

#include "parser.h"

static connection_info connection;



int verifyInputRE(const char *input)
{
    int    status;
    regex_t    reg;
    char* RE = "ftp://.*:.*@.*/.*";

    if (regcomp(&reg, RE, REG_EXTENDED|REG_NOSUB) != 0) {
        return(0);      /* Report error. */
    }

    status = regexec(&reg, input, (size_t) 0, NULL, 0);
    regfree(&reg);
    if (status != 0) {
        return(0);      /* Report error. */
    }
    return(1);
}

connection_info* parseArgs(char* input) {

    if(!verifyInputRE(input))
        return NULL;

    connection.user = (char*) malloc(MAX_STRING_LENGTH);
    memset(connection.user, 0, MAX_STRING_LENGTH);
    connection.password = (char*) malloc(MAX_STRING_LENGTH);
    memset(connection.password, 0, MAX_STRING_LENGTH);
    connection.hostname = (char*) malloc(MAX_STRING_LENGTH);
    memset(connection.hostname, 0, MAX_STRING_LENGTH);
    connection.file_path = (char*) malloc(MAX_STRING_LENGTH);
    memset(connection.file_path, 0, MAX_STRING_LENGTH);

    unsigned int i=6;
```

```
    unsigned int word_index = 0;
    unsigned int state=0;
    unsigned int input_length = strlen(input);
    char elem;

    while(i < input_length){

        elem = input[i];
        switch(state){
            case 0:
                if(elem == ':') {
                    word_index = 0;
                    state = 1;

                } else {
                    connection.user[word_index] = elem;
                    word_index++;
                }
                break;
            case 1:
                if(elem == '@'){
                    word_index = 0;
                    state = 2;

                } else {
                    connection.password[word_index] = elem;
                    word_index++;
                }
                break;
            case 2:
                if(elem == '/'){
                    word_index = 0;
                    state = 3;
                } else {
                    connection.hostname[word_index] = elem;
                    word_index++;
                }
                break;
            case 3:
                connection.file_path[word_index] = elem;
                word_index++;
                break;
        }
        i++;
    }
    printf("[Username: %s]\n", connection.user);
    printf("[Password: %s]\n", connection.password);
    return &connection;
}


int parsePasvPort(char* msgToParse){
```

```
    // get only numbers
    char pasvCodes[24];
    unsigned int length = strlen(msgToParse)-29;
    int i=0;
    for(; i < length; i++){
        pasvCodes[i] = msgToParse[i+26];
    }
    // parse to get the last two numbers
    int num1, num2, escp;
    sscanf(pasvCodes, "(%d,%d,%d,%d,%d,%d)", &escp, &escp, &escp, &escp, &num1, &num2);
    connection.data_port = (num1*256+num2);
    return 0;
}

int parseSize(char* response) {
    int escp;
    sscanf(response, "%d %ld", &escp, &connection.size);
    printf(" > File size: %ld\n", connection.size);
    return 0;
}
```

B *Scripts* da *Shell*

B.1 *Script* para tux21

```
#!/bin/bash

ifconfig eth0 up
ifconfig eth0 172.16.20.1/24
route add -net 172.16.21.0/24 gw 172.16.20.254
route add default gw 172.16.20.254
route add -net 172.16.20.0/24 gw 0.0.0.0
```

B.2 *Script* para tux22

```
#!/bin/bash

ifconfig eth0 up
ifconfig eth0 172.16.21.1/24
route add default gw 172.16.21.254
route add -net 172.16.21.0/24 gw 0.0.0.0

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

echo 1 > /proc/sys/net/ipv4/conf/eth0/accept_redirects
echo 1 > /proc/sys/net/ipv4/conf/all/accept_redirects
```

B.3 *Script* para tux24

```
#!/bin/bash

ifconfig eth0 up
ifconfig eth1 up
ifconfig eth0 172.16.20.254/24
ifconfig eth1 172.16.21.253/24

route add default gw 172.16.21.254
route add -net 172.16.0.0/16 gw 0.0.0.0
route add -net 172.16.20.0/24 gw 0.0.0.0

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

C Configurações

C.1 Experiência 1

- tux21

```
ifconfig eth0 up
ifconfig eth0 172.16.20.1/24
route add default gw 172.16.20.254
```

- tux24

```
ifconfig eth0 up
ifconfig eth0 172.16.20.254/2
```

C.2 Experiência 2

```
enable
configure terminal
vlan 20
exit
vlan 21
exit
interface fastethernet 0/1
switchport mode access
switchport access vlan 20
interface fastethernet 0/5
switchport mode access
switchport access vlan 20
interface fastethernet 0/9
switchport mode access
switchport access vlan 21
end
```

C.3 Experiência 3

```
ifconfig eth1 up
ifconfig eth1 172.16.21.253/24
```

```
enable
configure terminal
interface fastethernet 0/13
switchport mode access
switchport access vlan 21
end
```

C.4 Experiência 4

```
conf t
interface gigabitethernet 0/0
ip address 172.16.21.254 255.255.255.0
no shutdown
ip nat inside
exit

interface gigabitethernet 0/1
ip address 172.16.1.29 255.255.255.0
no shutdown
ip nat inside
exit

ip nat pool ovrld 172.16.1.29 172.16.1.29 prefix 24
ip nat inside source list 1 pool vorld overload
acess-list 1 permit 172.16.20.0 0.0.0.7
acess-list 1 permit 172.16.21.0 0.0.0.7

ip route 0.0.0.0 0.0.0.0 172.16.1.254
ip route 172.16.20.0 255.255.255.0 172.16.21.253
end
```

C.5 Experiência 5

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

D Representação da arquitetura das *Experiências*

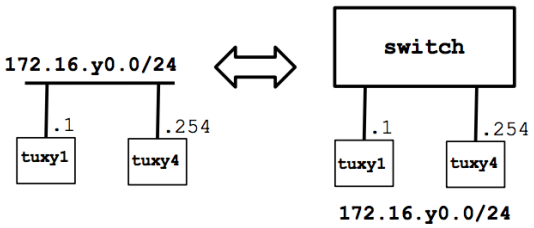


Figura 2: Arquitetura exp. 1

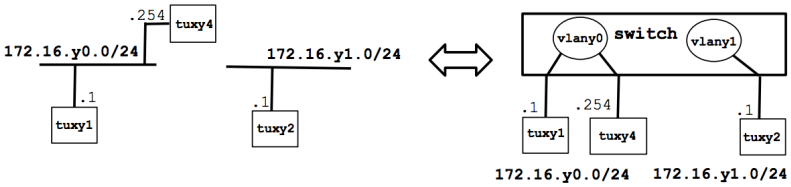


Figura 3: Arquitetura exp. 2

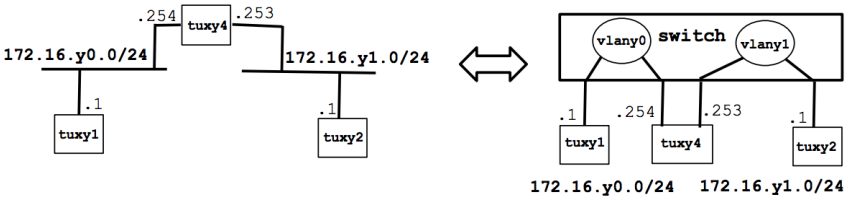


Figura 4: Arquitetura exp. 3

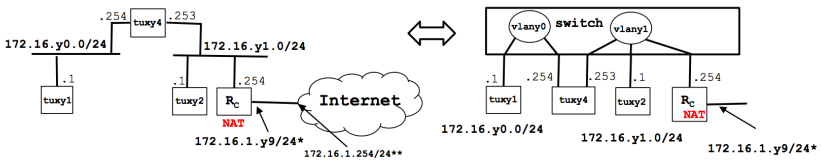


Figura 5: Arquitetura exp. 4

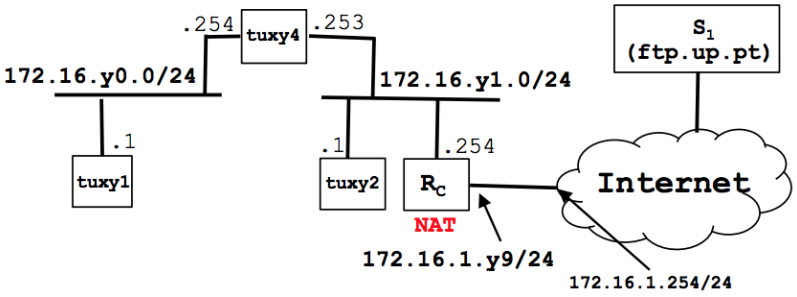


Figura 6: Arquitetura exp. 5

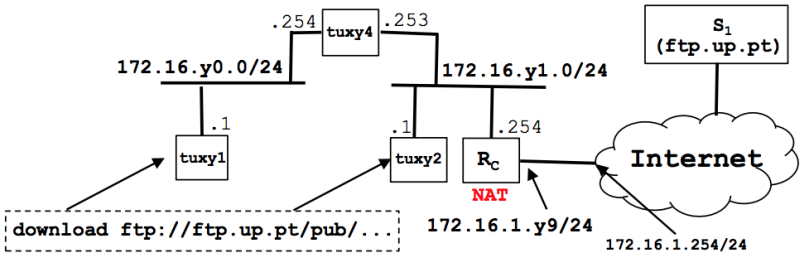


Figura 7: Arquitetura exp. 6

E *Logs* das *Experiências*

E.1 Experiência 1

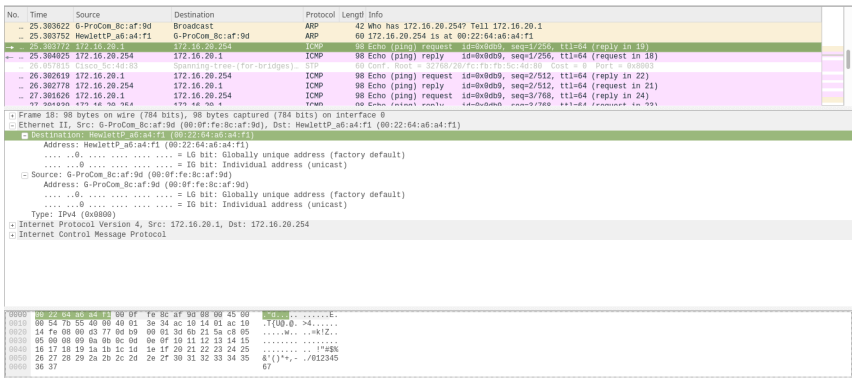


Figura 8: Obter endereços MAC e IP dos pacotes de ping,tipo de tramas de ethernet

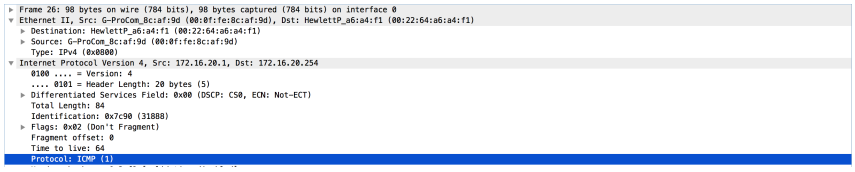


Figura 9: Identificação tipos tramas Ethernet

E.2 Experiência 2

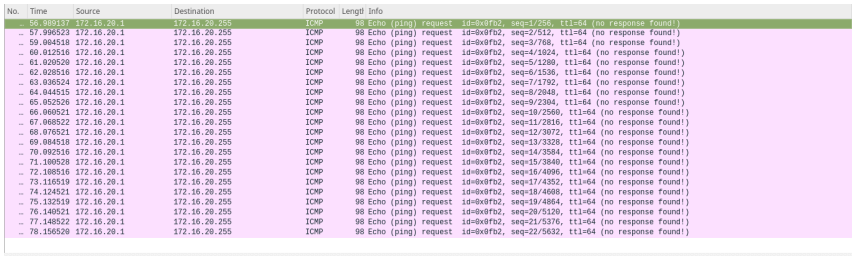
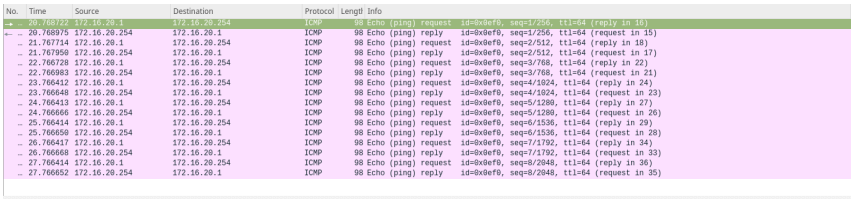
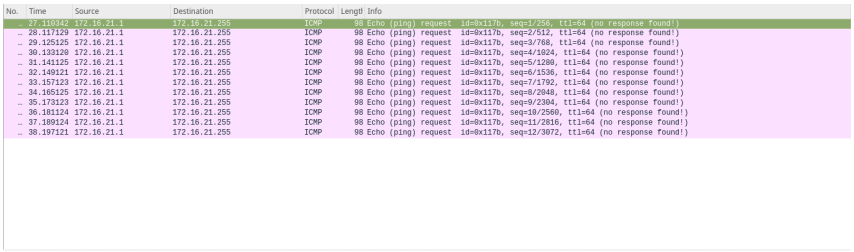


Figura 10: Ping do tux21 para o broadcast 172.16.20.255 no tux21



No.	Time	Source	Destination	Protocol	Length	Info
→	20.768722	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x0ef0, seq=1/256, ttl=64 (reply in 16)
←	20.769075	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x0ef0, seq=1/256, ttl=64 (request in 15)
	21.767714	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x0ef0, seq=2/512, ttl=64 (reply in 18)
	21.767950	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x0ef0, seq=2/512, ttl=64 (request in 17)
	22.766728	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x0ef0, seq=3/768, ttl=64 (reply in 22)
	22.766983	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x0ef0, seq=3/768, ttl=64 (request in 21)
	23.766412	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x0ef0, seq=4/1024, ttl=64 (reply in 24)
	23.766648	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x0ef0, seq=4/1024, ttl=64 (request in 23)
	24.766413	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x0ef0, seq=5/1280, ttl=64 (reply in 27)
	24.766666	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x0ef0, seq=5/1280, ttl=64 (request in 26)
	25.766414	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x0ef0, seq=6/1536, ttl=64 (reply in 29)
	25.766650	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x0ef0, seq=6/1536, ttl=64 (request in 28)
	26.766417	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x0ef0, seq=7/1792, ttl=64 (reply in 34)
	26.766668	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x0ef0, seq=7/1792, ttl=64 (request in 33)
	27.766414	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x0ef0, seq=8/2048, ttl=64 (reply in 36)
	27.766652	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x0ef0, seq=8/2048, ttl=64 (request in 35)

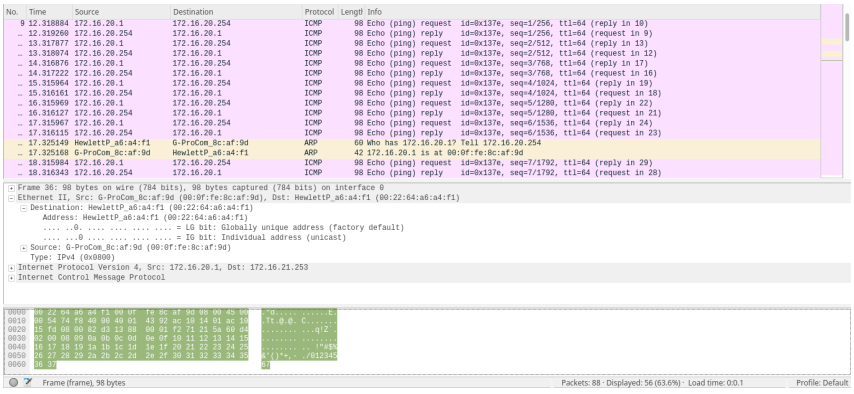
Figura 11: Ping do tux21 para o tux24



No.	Time	Source	Destination	Protocol	Length	Info
	27.110342	172.16.21.1	172.16.21.255	ICMP	98	Echo (ping) request id=0x117b, seq=1/256, ttl=64 (no response found!)
	28.117129	172.16.21.1	172.16.21.255	ICMP	98	Echo (ping) request id=0x117b, seq=2/512, ttl=64 (no response found!)
	29.125125	172.16.21.1	172.16.21.255	ICMP	98	Echo (ping) request id=0x117b, seq=3/768, ttl=64 (no response found!)
	30.133120	172.16.21.1	172.16.21.255	ICMP	98	Echo (ping) request id=0x117b, seq=4/1024, ttl=64 (no response found!)
	31.141125	172.16.21.1	172.16.21.255	ICMP	98	Echo (ping) request id=0x117b, seq=5/1280, ttl=64 (no response found!)
	32.149121	172.16.21.1	172.16.21.255	ICMP	98	Echo (ping) request id=0x117b, seq=6/1536, ttl=64 (no response found!)
	33.157123	172.16.21.1	172.16.21.255	ICMP	98	Echo (ping) request id=0x117b, seq=7/1792, ttl=64 (no response found!)
	34.165125	172.16.21.1	172.16.21.255	ICMP	98	Echo (ping) request id=0x117b, seq=8/2048, ttl=64 (no response found!)
	35.173123	172.16.21.1	172.16.21.255	ICMP	98	Echo (ping) request id=0x117b, seq=9/2304, ttl=64 (no response found!)
	36.181124	172.16.21.1	172.16.21.255	ICMP	98	Echo (ping) request id=0x117b, seq=10/2560, ttl=64 (no response found!)
	37.189124	172.16.21.1	172.16.21.255	ICMP	98	Echo (ping) request id=0x117b, seq=11/2816, ttl=64 (no response found!)
	38.197121	172.16.21.1	172.16.21.255	ICMP	98	Echo (ping) request id=0x117b, seq=12/3072, ttl=64 (no response found!)

Figura 12: Ping broadcast do tux2 para 172.16.20.255 no tux2

E.3 Experiência 3



No.	Time	Source	Destination	Protocol	Length	Info
9	12.310884	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x137e, seq=1/256, ttl=64 (reply in 10)
	12.319260	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x137e, seq=1/256, ttl=64 (request in 9)
	13.317877	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x137e, seq=2/512, ttl=64 (reply in 13)
	13.318074	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x137e, seq=2/512, ttl=64 (request in 12)
	14.316876	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x137e, seq=3/768, ttl=64 (reply in 17)
	14.317222	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x137e, seq=3/768, ttl=64 (request in 16)
	15.315964	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x137e, seq=4/1024, ttl=64 (reply in 19)
	15.316161	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x137e, seq=4/1024, ttl=64 (request in 18)
	16.315969	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x137e, seq=5/1280, ttl=64 (reply in 22)
	16.316127	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x137e, seq=5/1280, ttl=64 (request in 21)
	17.315967	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x137e, seq=6/1536, ttl=64 (reply in 24)
	17.316115	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x137e, seq=6/1536, ttl=64 (request in 23)
	17.325149	HewlettP_a6:a4:f1	G-ProCom_8c:af:9d	ARP	60	Who has 172.16.20.1? Tell 172.16.20.254
	17.325168	G-ProCom_8c:af:9d	HewlettP_a6:a4:f1	ARP	42	172.16.20.1 is at 00:0f:fe:8c:af:9d
	18.315984	172.16.20.1	172.16.20.254	ICMP	98	Echo (ping) request id=0x137e, seq=7/1792, ttl=64 (reply in 29)
	18.316343	172.16.20.254	172.16.20.1	ICMP	98	Echo (ping) reply id=0x137e, seq=7/1792, ttl=64 (request in 28)

```
Frame 36: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: G-ProCom_8c:af:9d (00:0f:fe:8c:af:9d), Dst: HewlettP_a6:a4:f1 (00:22:64:a6:a4:f1)
    Destination: HewlettP_a6:a4:f1 (00:22:64:a6:a4:f1)
        Address: HewlettP_a6:a4:f1 (00:22:64:a6:a4:f1)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Source: G-ProCom_8c:af:9d (00:0f:fe:8c:af:9d)
    Type: IPv4 (0x0800)
Internet Protocol Version 4, Src: 172.16.20.1, Dst: 172.16.21.253
Internet Control Message Protocol
```

```
0000  00 22 64 a6 a4 f1 00 0f  fe 8c af 9d 08 00 45 00   ."d..... ......E.
0010  00 54 74 f8 40 00 40 01  43 92 ac 10 14 01 ac 10   .Tt.@.@. C.......
0020  15 fd 00 00 82 d3 13 08  00 01 f2 71 21 5a 60 dd   ........ ...q!Z`.
0030  02 00 08 09 0a 0b 0c 0d  0e 0f 10 11 12 13 14 15   ........ ........
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ........ .. !"#$%
0050  26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35   &'()*+,- ./012345
0060  36 37                                             67
```

Figura 13: Captura de ping do tux1 para o tux4 na interface eth0

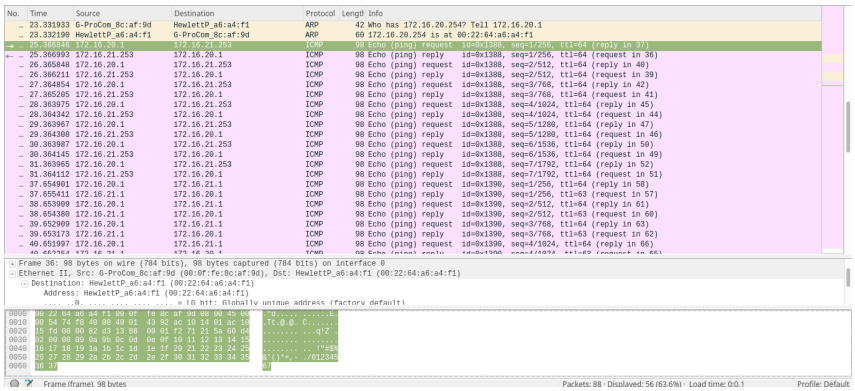


Figura 14: Captura de ping do tux1 para o tux4 na interface eth1 e também do tux1 para o tux2

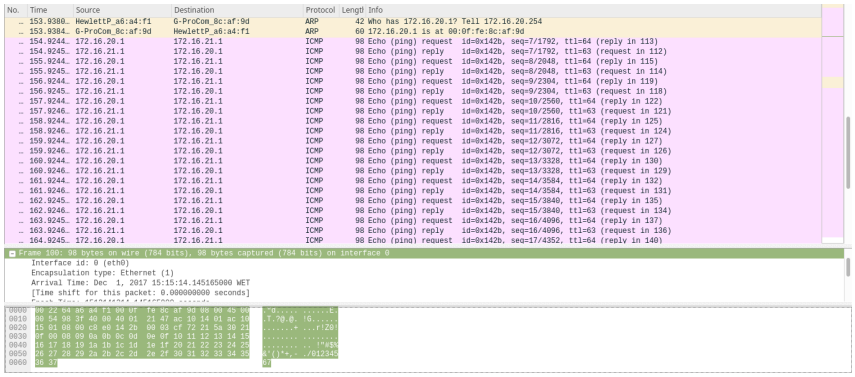


Figura 15: Captura no tux4 na interface eth0 enquanto o tux1 dá ping ao tux2

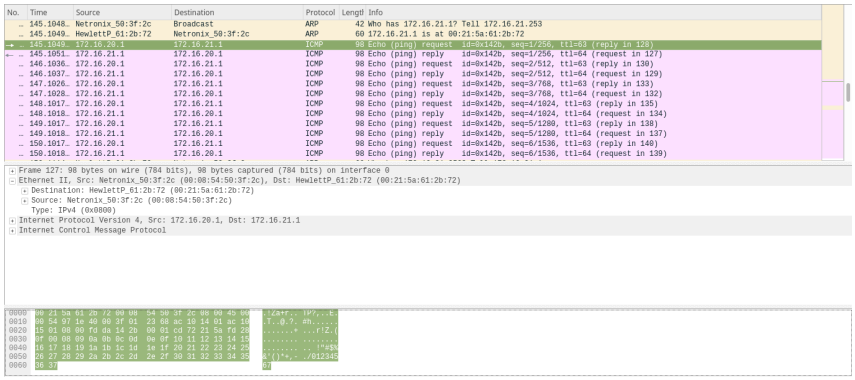


Figura 16: Captura no tux4 na interface eth1 enquanto o tux1 dá ping ao tux2

E.4 Experiência 4

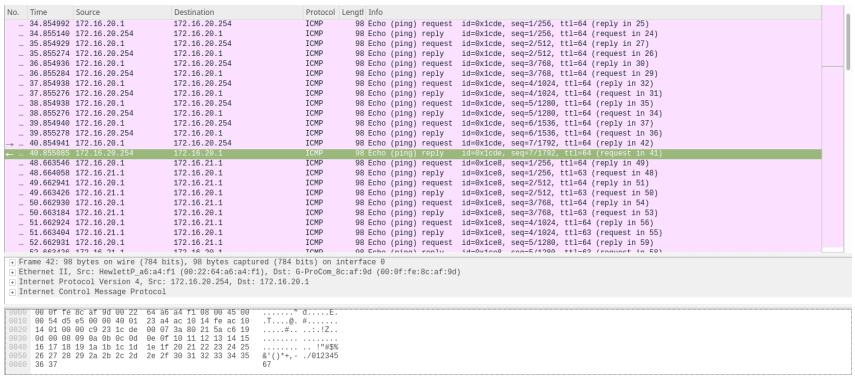


Figura 17: Captura do tux1 dar ping ao tux2 e ao tux4

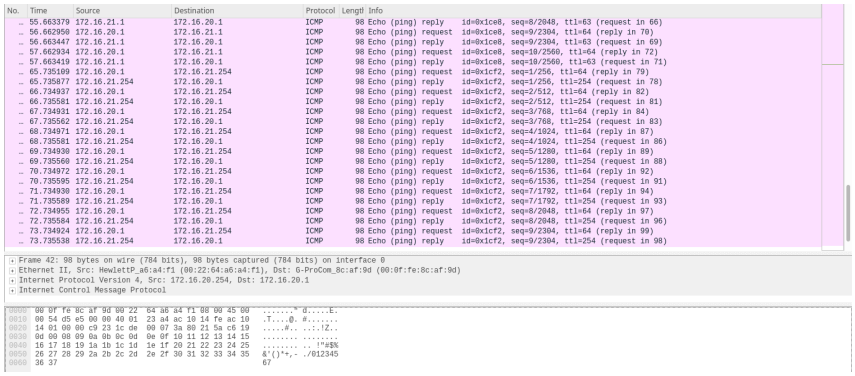


Figura 18: Captura do tux1 dar ping ao Rc criado

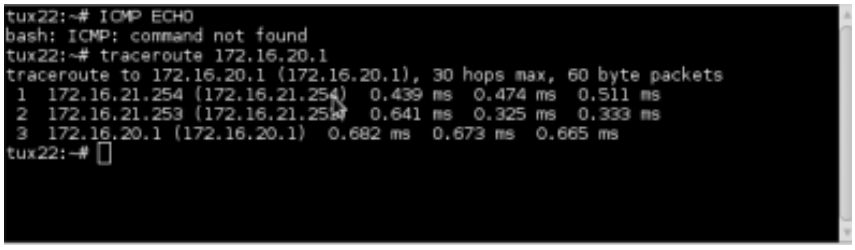


Figura 19: Traceroute com ICMP redirect desativado e sem rota direta de tux2 para tux4

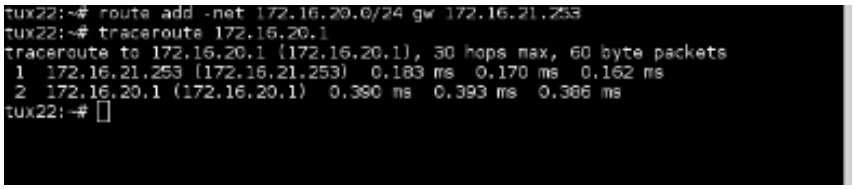


Figura 20: Traceroute com ICMP redirect desativado e com rota direta de tux2 para tux4

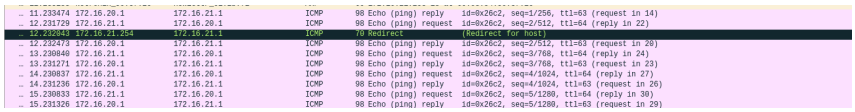


Figura 21: ICMP redirect ativado sem rota direta do tux2 para tux4

E.5 Experiência 5

Figura 22: resposta ao comando DNS

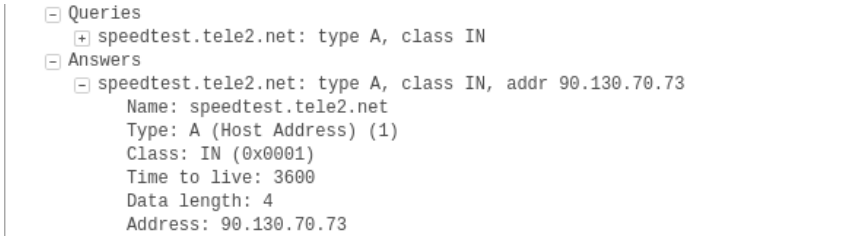


Figura 23: Informações do pacote de resposta DNS

E.6 Experiência 6

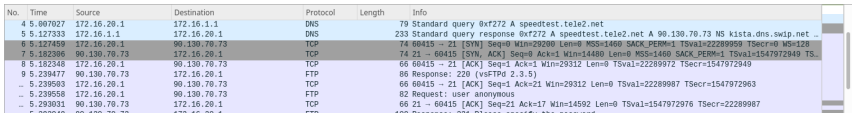
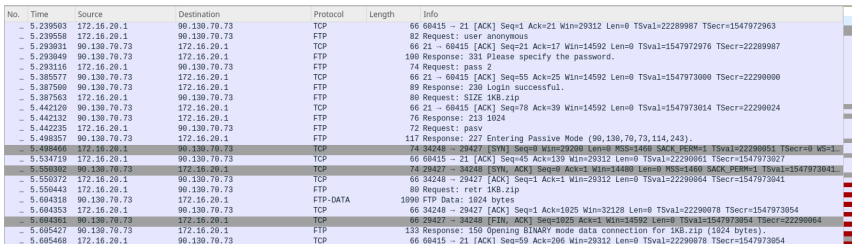
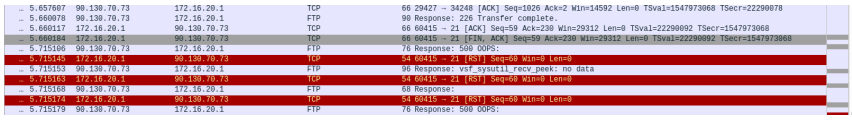


Figura 24: Fase inicial do protocolo TCP - estabelecer a conexão



No.	Time	Source	Destination	Protocol	Length	Info
…	5.239503	172.16.20.1	90.130.70.73	TCP	66	60415 → 21 [ACK] Seq=1 Ack=21 Win=29312 Len=0 TSval=22289987 TSecr=1547972963
…	5.239558	172.16.20.1	90.130.70.73	FTP	82	Request: user anonymous
…	5.293031	90.130.70.73	172.16.20.1	TCP	66	21 → 60415 [ACK] Seq=21 Ack=17 Win=14592 Len=0 TSval=1547972976 TSecr=22289987
…	5.293049	90.130.70.73	172.16.20.1	FTP	100	Response: 331 Please specify the password.
…	5.293116	172.16.20.1	90.130.70.73	FTP	74	Request: pass 2
…	5.385577	90.130.70.73	172.16.20.1	TCP	66	21 → 60415 [ACK] Seq=55 Ack=25 Win=14592 Len=0 TSval=1547973000 TSecr=22290000
…	5.387500	90.130.70.73	172.16.20.1	FTP	89	Response: 230 Login successful.
…	5.387563	172.16.20.1	90.130.70.73	FTP	80	Request: SIZE 1KB.zip
…	5.442120	90.130.70.73	172.16.20.1	TCP	66	21 → 60415 [ACK] Seq=78 Ack=39 Win=14592 Len=0 TSval=1547973014 TSecr=22290024
…	5.442132	90.130.70.73	172.16.20.1	FTP	76	Response: 213 1024
…	5.442235	172.16.20.1	90.130.70.73	FTP	72	Request: pasv
…	5.498357	90.130.70.73	172.16.20.1	FTP	117	Response: 227 Entering Passive Mode (90,130,70,73,114,243).
…	5.498466	172.16.20.1	90.130.70.73	TCP	74	34248 → 29427 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=22290051 TSecr=0 WS=1…
…	5.534719	172.16.20.1	90.130.70.73	TCP	66	60415 → 21 [ACK] Seq=45 Ack=130 Win=29312 Len=0 TSval=22290061 TSecr=1547973027
…	5.550302	90.130.70.73	172.16.20.1	TCP	74	29427 → 34248 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PERM=1 TSval=1547973041…
…	5.550372	172.16.20.1	90.130.70.73	TCP	66	34248 → 29427 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=22290064 TSecr=1547973041
…	5.550443	172.16.20.1	90.130.70.73	FTP	80	Request: retr 1KB.zip
…	5.604318	90.130.70.73	172.16.20.1	FTP-DATA	1090	FTP Data: 1024 bytes
…	5.604353	172.16.20.1	90.130.70.73	TCP	66	34248 → 29427 [ACK] Seq=1 Ack=1025 Win=32128 Len=0 TSval=22290078 TSecr=1547973054
…	5.604361	90.130.70.73	172.16.20.1	TCP	66	29427 → 34248 [FIN, ACK] Seq=1025 Ack=1 Win=14592 Len=0 TSval=1547973054 TSecr=22290064
…	5.605427	90.130.70.73	172.16.20.1	FTP	133	Response: 150 Opening BINARY mode data connection for 1KB.zip (1024 bytes).
…	5.605468	172.16.20.1	90.130.70.73	TCP	66	60415 → 21 [ACK] Seq=59 Ack=200 Win=29312 Len=0 TSval=22290078 TSecr=1547973054

Figura 25: Protocolo TCP com transferência de dados



…	5.657607	90.130.70.73	172.16.20.1	TCP	66	29427 → 34248 [ACK] Seq=1026 Ack=2 Win=14592 Len=0 TSval=1547973068 TSecr=22290078
…	5.660078	90.130.70.73	172.16.20.1	FTP	90	Response: 226 Transfer complete.
…	5.660117	172.16.20.1	90.130.70.73	TCP	66	60415 → 21 [ACK] Seq=59 Ack=230 Win=29312 Len=0 TSval=22290092 TSecr=1547973068
…	5.660184	172.16.20.1	90.130.70.73	TCP	66	60415 → 21 [FIN, ACK] Seq=59 Ack=230 Win=29312 Len=0 TSval=22290092 TSecr=1547973068
…	5.715106	90.130.70.73	172.16.20.1	FTP	76	Response: 500 OOPS:
…	5.715145	172.16.20.1	90.130.70.73	TCP	54	60415 → 21 [RST] Seq=60 Win=0 Len=0
…	5.715153	90.130.70.73	172.16.20.1	FTP	90	Response: vsf_sysutil_recv_peek: no data
…	5.715163	172.16.20.1	90.130.70.73	TCP	54	60415 → 21 [RST] Seq=60 Win=0 Len=0
…	5.715168	90.130.70.73	172.16.20.1	FTP	68	Response:
…	5.715174	172.16.20.1	90.130.70.73	TCP	54	60415 → 21 [RST] Seq=60 Win=0 Len=0
…	5.715179	90.130.70.73	172.16.20.1	FTP	78	Response: 500 OOPS:

Figura 26: Terminação da conexão do protocolo TCP

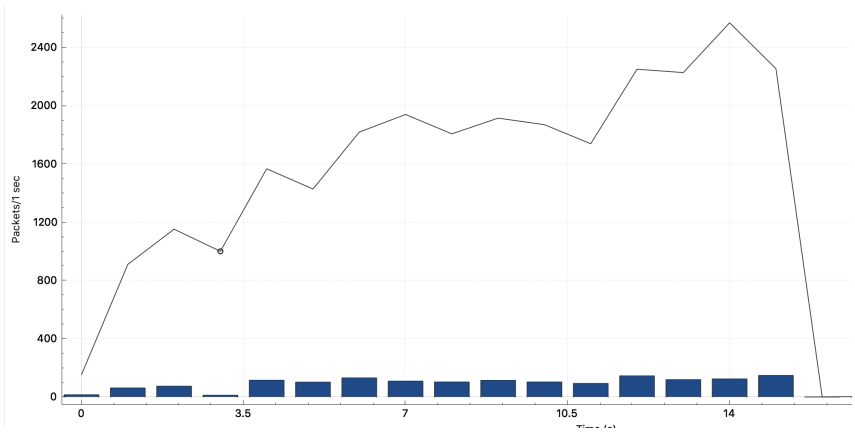


Figura 27: Gráfico dos pacotes transferidos por unidade de tempo

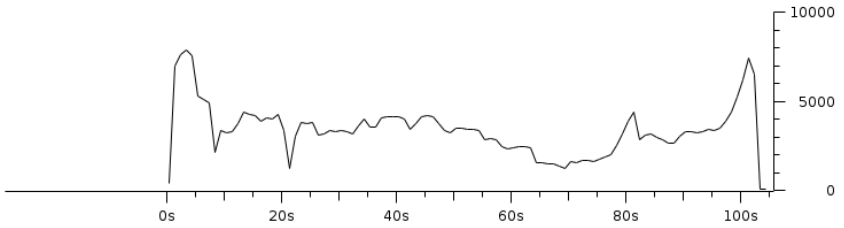


Figura 28: Gráfico dos pacotes transferidos por unidade de tempo com 2 tuxs