Redes de Computadores

2º Trabalho Laboratorial

Rede de Computadores

Licenciatura em Engenharia Informática e Computação
1º Semestre - 2022/2023 | 3LEIC10

22 de dezembro de 2022

Bruna Brasil Leão Marques – up202007191@fe.up.pt
Francisca Horta Guimarães – up202004229@fe.up.pt

Índice

1. Sumário

No âmbito da unidade curricular de Redes de Computadores do primeiro semestre do terceiro ano da Licenciatura em Engenharia Informática e Computação, foi-nos proposta a realização de um segundo projeto cujo objetivo passa por explorar o processo de criação de uma aplicação de *download* de ficheiros usando o protocolo FTP e TCP/IP, juntamente com o desenvolvimento de uma rede de computadores na qual será testada a aplicação *download* referida anteriormente.

Concluímos este trabalho com sucesso, visto que os objetivos propostos foram cumpridos.

2. Introdução

Este projeto tem dois grandes objetivos: desenvolver a aplicação de download e configurar uma rede em laboratório. Relativamente à aplicação download, esta foi desenvolvida seguindo o protocolo FTP (*File Transfer Protocol*) com ligações TCP (*Transmission Control Protocol*) a partir de *sockets*. Quanto à configuração da rede em laboratório, o seu objetivo é permitir executar a aplicação *download* a partir de duas bridges dentro de um *switch*.

Assim, este relatório estará dividido em essencialmente três partes: Parte I: Aplicação de Download; Parte II: Configuração de Rede e Análise; Conclusões. Na Parte I, iremos examinar a arquitetura e respectivos resultados da nossa aplicação *download*. De seguida, na Parte II, iremos analisar com mais detalhe cada experiência realizada para a configuração da rede. Por último, concluiremos este relatório refletindo sobre os objetivos concluídos.

3. Parte I: Aplicação de Download

A primeira parte deste trabalho passa por desenvolver uma aplicação *download* que aceita um link, ftp://:@/, como argumento. Esta aplicação faz o *download* de qualquer tipo de ficheiros de um servidor FTP.

3.1. Arquitetura

A aplicação está dividida em duas partes: primeiramente é processada a string de input do utilizador e, de seguida, a ligação com o servidor, login e download do ficheiro indicado pelo utilizador são os aspetos focados.

Utilizamos uma estrutura de dados, **urlData**, onde guardamos toda a informação disponibilizada pela string de input do utilizador, isto é, o **user**, **password**, **host**, **url_path**, **fileName** e **ip.** Este input é validado pela função parseUrlData() que inicializa a estrutura **urlData** através da função strtok() da biblioteca C.

Após a inicialização de todas as variáveis, a função download() é chamada. É nesta função que o processo de *download* do ficheiro acontece e para isso começamos por chamar startConnection(). Esta função é responsável por ligar o cliente FTP ao servidor através de um *socket*.

De seguida, o cliente faz login no servidor com o username e a palavra passe disponibilizada previamente pelo utilizador. Utilizamos, para isso, a função sendCommandToServer() que envia comandos ao servidor, e aguardamos pela resposta do servidor com a função readServerReply(). Esta função chama getline() até o quarto caractere corresponder a um espaço, uma vez que os três primeiros constituem o código.

Após efetuado o login, colocamos o servidor em modo passivo, utilizando mais uma vez, a função sendCommandToServer(). Obtemos como resposta uma string com o endereço ip e a porta onde iremos fazer o download do ficheiro pelo que é chamada a função getIpAndPort() que processa a string e retorna o ip e a porta. Com este endereço e porta chamamos mais uma vez a função startConnection().

Estabelecida a nova conexão, pedimos ao servidor o ficheiro desejado para *download* utilizando novamente as funções sendCommandToServer() e readServerReply() para tal. Após o pedido do ficheiro, lemos o mesmo através da função readFile() que o guarda no nosso disco.

Terminada a leitura do ficheiro resta-nos enviar, com auxílio da função sendCommandToServer(), o comando de término da ligação.

3.2. Resultados

Esta aplicação foi testada com vários ficheiros, tanto em modo anónimo como em modo não anónimo. Verificámos que a aplicação consegue transferir ficheiros de vários tamanhos, tendo o ficheiro com máximo tamanho cerca de 521MB,

Em caso de erro em qualquer parte do programa, é impresso na consola não só o alerta do erro mas também em que parte do programa ocorreu para que o utilizador saiba qual o passo que falhou na execução da aplicação.

4. Parte II: Configuração de Rede e Análise

4.1. Experiência 1 - Configurar um IP de rede

O objetivo desta experiência é configurar os endereços IP de dois computadores (tux3 e tux4) para que estes consigam comunicar entre si. Utilizamos o comando *ifconfig* para configurar os endereços IP das duas máquinas e depois de configurar as portas eth0 e adicionar as rotas necessárias, utilizamos o comando *ping* para verificar a conectividade entre os mesmos.

ARP (*Address Resolution Protocol*) é um protocolo de comunicação utilizado para a resolução de endereços da camada de rede - IP (*Internet Protocol*). Serve para mapear o endereço de rede a um endereço físico como o endereço MAC (*Medium Access Control*).

Assim, ao utilizarmos o comando *ping* do tux3 para o tux4, o emissor (tux3) tenta descobrir o endereço MAC correspondente ao endereço IP, difundindo em broadcast um pacote ARP que contém o endereço de IP, esperando uma resposta com o endereço MAC que lhe corresponde. Nesta experiência, o pacote de pedido contém os endereços IP e MAC do emissor (tux3) (172.16.Y0.1 e 00:21:5a:61:2c:54, respetivamente) e o endereço IP do receptor (tux4) (172.16.Y0.254), como ainda não se conhece o MAC do receptor este está registado como 00:00:00:00:00:00 (Figura 1). Já no pacote de resposta, é enviado o endereço IP e MAC do novo emissor (tux4) (172.16.Y0.254 e 00:22:64:19:09:5c, respetivamente) com o endereço IP e MAC do receptor (tux3) (172.16.Y0.1 e 00:21:5a:61:2c:54) (Figura 2).

O comando *ping*, depois de gerar pacotes ARP (quando ainda não conhece os endereços MAC), gera pacotes do protocolo ICMP (*Internet Control Message Protocol*) para transferir mensagens de controlo entre endereços IP. Nesta experiência, o pacote de pedido tem um endereço MAC de origem 00:21:5a:61:2c:54 e destino 00:22:64:19:09:5c e endereço IP de origem 172.16.50.1 e destino 172,16.50.254 (Figura 3). Já o pacote de resposta, tem endereço MAC de origem 00:22:64:19:09:5c e destino 00:21:5a:61:2c:54 e endereço IP de origem 172,16.50.254 e destino 172.16.50.1 (Figura 4).

Sobre a trama receptora Ethernet, para conseguirmos distinguir se é ARP, IP ou ICMP inspecionamos o Ethernet header nos *wiresharks* e se no parâmetro do tipo o valor for 0x0800,

trata-se de uma trama IP. Inspecionando, de seguida, o IP header, se no parâmetro do protocolo o valor 1 estiver presente, concluímos que o protocolo é ICMP. Por outro lado, se o Ethernet header tiver o valor 0x0806 no parâmetro tipo, a trama é ARP (Figuras 5 e 6).

Já para determinarmos o comprimento de uma trama receptora, adquirimos essa informação também nos wiresharks na secção da *Frame* (Figura 7).

Por último, ficamos a conhecer a importância de uma interface loopback visto ser uma interface de rede virtual utilizada para realizar testes de diagnóstico pelo computador. Esta interface permite ter um endereço de IP no router, sempre ativo, o que permite a independência de uma interface física.

4.2. Experiência 2 - Implementar duas *bridges* no switch

Na experiência 2, o objetivo é criar duas LAN's virtuais, a bridgeY0 e a bridgeY1 e associar o tux3 e o tux4 à primeira e o tux2 à segunda.

Para a configurar uma bridge foi necessário entrar na consola de configuração do switch e executar o comando */interface bridge add name=bridgeY0*. Em seguida, remove-se da bridge default as portas do switch que ligamos o tux3 e 4 (*/interface bridge port remove [find interface=etherZ],* onde Z é o identificador da porta) e adiciona-se essas portas às respectivas bridges (*/interface bridge port add bridge=bridgeY0 interface=etherZ*). Este processo repete-se para a bridge Y1, obtendo assim duas sub-redes individuais.

Nesta experiência, executamos o comando ping no tux3 para o tux2 e falha (Figura 8), tal como esperado uma vez que não há nenhuma rota entre as VLAN's. Depois, no tux3, foi feito ping em broadcast, ping -b 172.16.Y0.255, onde se poderia esperar uma resposta do tux4 uma vez que se encontram na mesma sub-rede. No entanto, como *echo-ignore-broadcast* está ativado por default, não obtém resposta.

Após isso, repete-se este processo mas a partir do tux2. Neste caso, não foi obtida nenhuma resposta pois nenhum dispositivo está configurado na bridge21 para além do próprio tux2 (Figura 9). Deste modo, conclui-se que existem dois domínios de *broadcasts* com os endereços 172.16.Y0.255 e 172.16.Y1.255.

4.3. Experiência 3 - Configurar um router em Linux

Na experiência 3, o tux4 é configurado como um router entre as duas sub-redes criadas na experiência anterior, possibilitando assim a comunicação entre o tux3 e tux2. Para isso, liga-se à interface Ethernet 1 da máquina 4 e adiciona-se esta interface à sub-rede em que o tux2 está.

Para criar a ligação entre o tux3 e o tux2, adiciona-se uma rota ao tux3 utilizando o comando *route add -net 172.16.Y1.0/24 gw 172.16.Y0.254*, em que o primeiro endereço identifica o domínio para a qual se quer adicionar a rota e o segundo identifica o IP a encaminhar o pacote, neste caso o IP do tux4 na interface eth0. De seguida, repete-se este processo para o tux2, onde 172.16.Y1.253 é o endereço do domínio e 172.16.Y1.253 é o IP do tux4 na interface eth1 a ser utilizado como gateway. Além disso, todos os tuxs tem uma rota para o domínio em que pertencem em que a gateway é 0.0.0.0 uma vez que os pacotes de dados não precisam de ser redirecionados.

É possível observar a tabela de forwarding ao correr *route -n*, obtendo as seguintes informações:

- Destination: o destino da rota;
- Gateway: o IP por onde a rota passará;

- Genmask: máscara de endereço usada para corresponder um endereço IP ao valor mostrado no campo Destino; usado para determinar o ID da rede a partir do endereço IP do destino;
- Flags: características da rota;
- Metric: atribui um valor a cada rota para garantir que as ideais sejam escolhidas para enviar pacotes, assim, se existirem várias rotas, a rota com a métrica mais baixa geralmente é escolhida;
- Ref: número de referências para a rota;
- Use: contador do número de vezes que esta rota foi consultada;
- Interface: nome da interface de rede utilizada por esta rota, no caso deste projeto eth0 ou eth1.

Uma mensagem ARP é enviada quando se faz um pedido para saber o endereço MAC de um certo endereço IP, se não o conhecer ainda.

Depois de definir as rotas, torna-se possível fazer *ping* a partir do tux3 a todas as interfaces dos outros tuxs. Quando faz *ping* para o tux4, pela Figura 10, é possível observar que a interface eth0 do tux3 enviou um pedido ARP para saber o endereço MAC da interface eth0 do tux4, enquanto que o tux4 mandou um pedido para saber o endereço MAC do tux3. Isto repete-se quando faz *ping* para a interface eth1 do tux4 e para o tux2, cada um envia o pedido para tomar conhecimento do endereço MAC do tux3 (Figura 11 e Figura 12).

Pelos logs das figuras citadas acima, também é possível observar pacotes ICMP de *request* e *reply*, já que uma vez adicionadas as rotas, todos os três tuxs conseguem se comunicar uns com os outros. Caso contrário, seriam enviados pacotes ICMP de *Host Unreachable*.

A estes pacotes ICMP estão associados os endereços IP e MAC dos tuxs de origem e destino. Na Figura 13, um caso exemplo, encontra-se sublinhado o endereço MAC e IP do *source* (tux3) e do *destination* (tux4 da eth1).

4.4. Experiência 4 - Configurar um router comercial e implementar o NAT

Nesta experiência, configuramos um *commercial router* com NAT implementado.

Para configurar uma *static route* num *commercial router,* ligamos o cabo de série S0 do tux4 à entrada de mtik do router, para assim poder configurá-lo pelo GTKTerm. No GTKTerm, adicionamos a rota com o comando /ip route add dst-address=172.16.Y0.0/24 gateway=172.16.Y1.253.

Se existir uma rota específica para onde se quer enviar os pacotes, eles seguem essa rota. Caso não exista, os pacotes são direcionados para a rota default, isto é, a rota para o router.

É de salientar que num dos passos nesta experiência, os redirecionamentos são desativados no tux2 (*echo 0 > /proc/sys/net/ipv4/conf/eth0/accept_redirects e echo 0 > /proc/sys/net/ipv4/conf/all/accept_redirects*). Deste modo, o tux não guarda na tabela forwarding as entradas resultantes de um redirecionamento de outro tux.

Depois disso, define-se o router **Rc** como default route para o tux2 e para o tux4. Quando se elimina a rota no tux2 para o domínio de endereço 172.16.Y0.0 com a gateway como o eth1 do tux4, o tux2 passa a não ter nenhuma rota para tux3. Assim, o tux2 passa a encaminhar os pacotes para a sua rota default, o router **Rc**, e como este tem uma rota para o domínio, re-encaminha o pacote do tux2 de forma a chegar ao tux3. Se o tux2 tiver os *redirects* ativos, apenas será necessário visitar o router a primeira vez. Caso contrário, como explicado acima, todos os pacotes têm de passar pelo router. No caso do tux3, os pacotes que envia seguem pela gateway da interface eth0 do tux4, 172.16.Y0.254.

O NAT – *Network Address Translation* – traduz os endereços IP privados por endereços IP públicos possibilitando que os computadores de uma rede interna, como as que foram criadas, tenham acesso à rede pública. Desta forma, o router que implementa o NAT encaminha os pacotes para o endereço correto, dentro ou fora da rede local.

4.5. Experiência 5 - DNS

Na experiência 5, configuramos o DNS (*Domain Name System*) no tux3, tux4 e tux2. Um servidor DNS é usado para traduzir os *hostnames* para os seus respectivos endereços IP, neste caso, *services.netlab.fe.up.pt* que contém uma base de dados com endereços IP públicos e os seus respectivos *hostnames*.

Para configurarmos o serviço DNS, é necessário editar o ficheiro **resolv.conf** que se localiza em /etc/resolv.conf no host tux. Esse ficheiro terá de ser editado para conter a seguinte informação: nameserver 172.16.1.1 (ou 172.16.2.1 se nos encontrarmos no segundo laboratório). Após esta configuração é possível aceder à internet nos tuxs.

Assim, podemos observar na Figura 14, que é enviado um pacote do Host para o Server que contém o *hostname* (**www.google.com**) desejado, pedindo o seu endereço de IP. O servidor responde com um pacote que contém o endereço IP do *hostname*.

4.6. Experiência 6 - Ligações TCP

Depois de a rede estar totalmente configurada, a aplicação *download* desenvolvida na Parte I foi caso de estudo. Com recurso à transferência de um ficheiro de um servidor FTP, o teste foi feito e concluído com sucesso.

Para isso, a nossa aplicação abriu duas conexões TCP (*Transmission Control Protocol*), uma para enviar comandos FTP ao servidor e receber respostas e outra para receber os dados enviados pelo servidor e responder ao cliente. Uma conexão TCP tem três fases: estabelecer a conexão, trocar dados e encerrar a conexão. Assim, é na conexão TCP responsável pela troca de comandos que o controlo de informação é transportado.

O TCP utiliza o mecanismo ARQ (*Automatic Repeat Request*), um método de controlo de erros na transmissão de dados. Este método utiliza acknowledgments, mensagens que confirmam a recepção correta da trama de dados, e timeouts, o máximo de tempo permitido para esperar por um acknowledgment, com o objetivo de garantir que uma transmissão é confiável através de um serviço não confiável. Se antes do timeout não for recebido um acknowledgment, a trama é retransmitida até um acknowledgement ser recebido.

Para além disto, o TCP tem um mecanismo de controlo de congestão, que se baseia em manter uma janela de congestão que consiste numa estimativa do número de octetos que a rede consegue encaminhar não enviando mais octetos do que o mínimo da janela definida pelo recetor e pela janela de congestão. Nesta experiência, observamos que no início do primeiro *download* no tux3 a taxa de transferência aumenta, no entanto, quando iniciamos o *download* no tux2 enquanto a transferência no tux3 ainda ocorre, verificamos uma descida significativa seguida de uma subida que se estabiliza num nível mais baixo que o original (quando apenas ocorria o download no tux3). Assim, o fluxo de dados da conexão está de acordo com o mecanismo de controlo de congestão.

A existência de uma transferência de dados em simultâneo com o aparecimento de uma segunda conexão pode levar a uma queda na taxa de transmissão, já que a taxa de transferência é distribuída igualmente para cada ligação.

5. Conclusões

Após a conclusão deste trabalho, podemos afirmar que interiorizamos os conceitos necessários para a implementação do que nos era pedido no guião.

Implementámos a aplicação *download*, percebendo melhor o protocolo FTP, o que nos possibilitou a comparação com o protocolo desenvolvido no projeto anterior, e configuramos uma rede de computadores, que nos permitiu compreender, a um nível mais detalhado, como se configura uma rede, e assim, poder aplicar esta prática a nível profissional.

Em suma, todos os objetivos foram concluídos com sucesso, o que contribuiu positivamente para um aprofundamento dos conhecimentos acerca de protocolos de ligação, bastante utilizados no dia-a-dia.

6. Referências

1. *Address Resolution Protocol* [online]. [visto a 18 de dezembro de 2022]. Disponível em: https://en.wikipedia.org/wiki/Address_Resolution_Protocol
2. *Automatic repeat request* [online]. [visto a 20 de dezembro de 2022]. Disponível em: https://en.wikipedia.org/wiki/Automatic_repeat_request
3. RODRIGUES, Luís. *Protocolos em Redes de Dados - Aula 09 - Controlo da congestão e QoS* [online]. [visto a 20 de dezembro de 2022]. Disponível em: http://www.gsd.inesc-id.pt/ler/docencia/prd0304/handouts09.pdf

7. Anexos

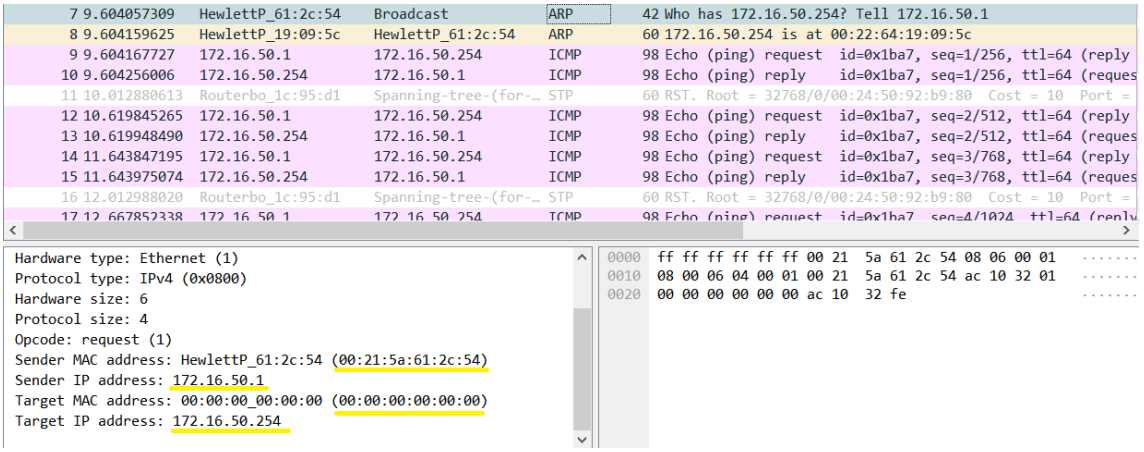


Figura 1

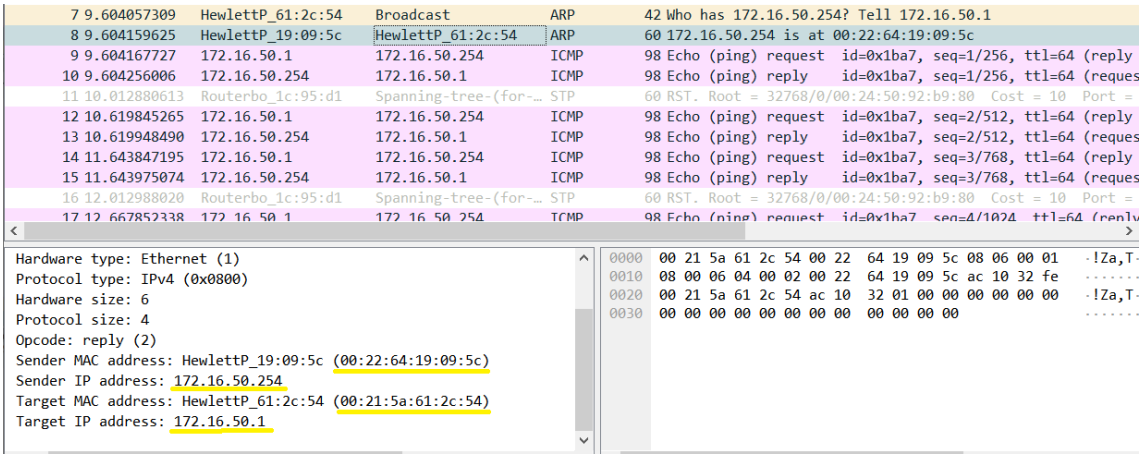


Figura 2

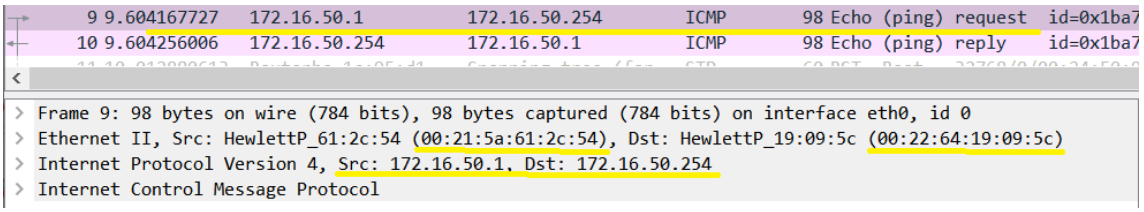


Figura 3

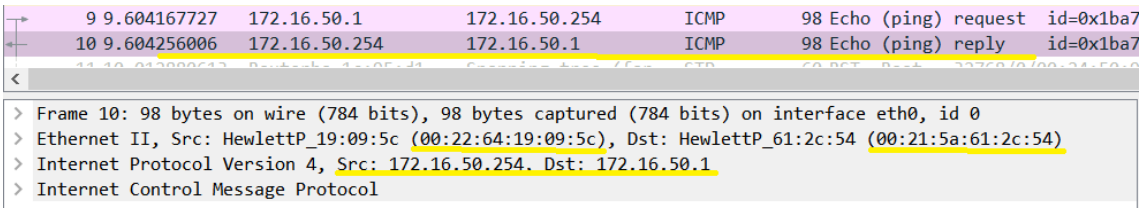


Figura 4

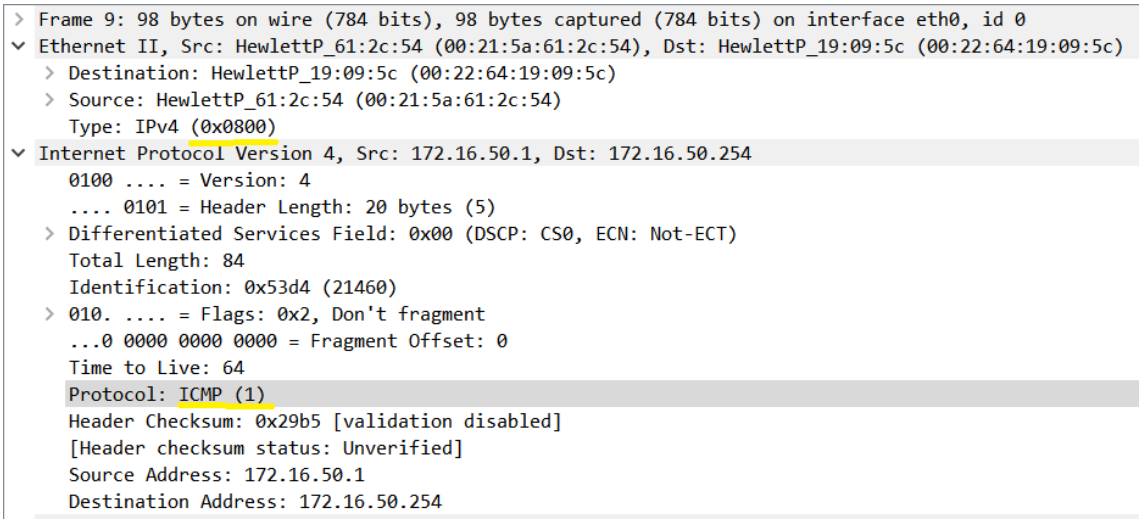


Figura 5

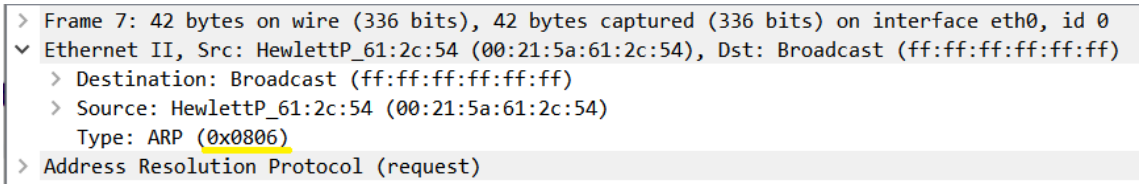


Figura 6

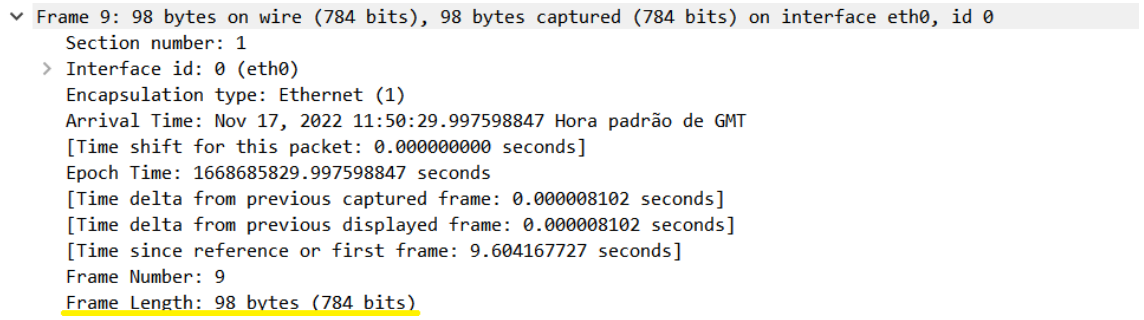
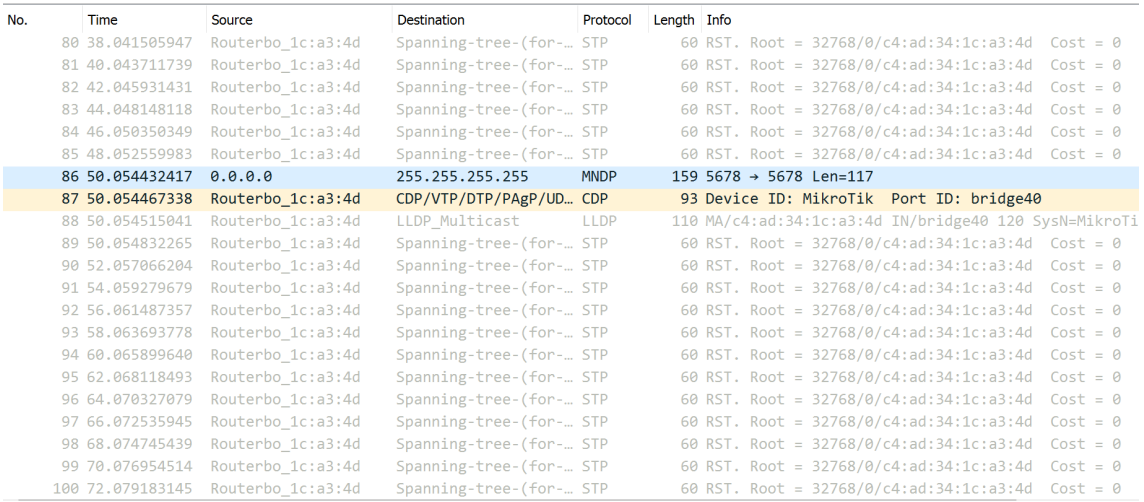
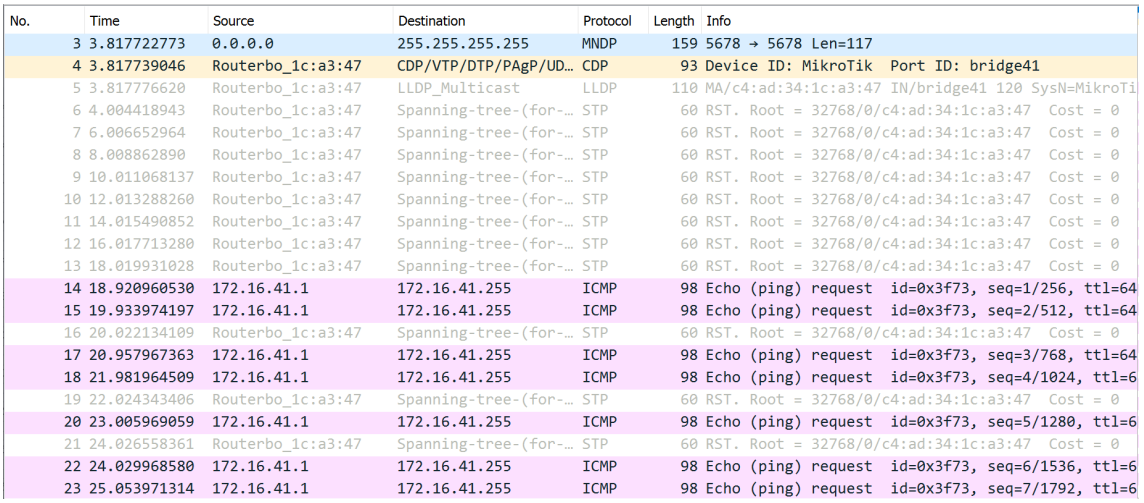


Figura 7



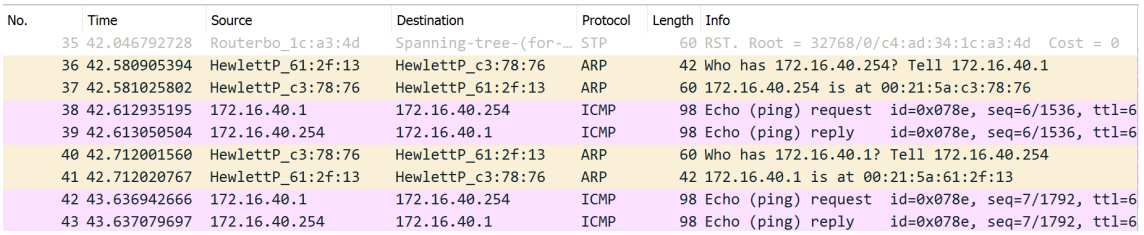
No.	Time	Source	Destination	Protocol	Length	Info
80	38.041505947	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
81	40.043711739	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
82	42.045931431	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
83	44.048148118	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
84	46.050350349	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
85	48.052559983	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
86	50.054432417	0.0.0.0	255.255.255.255	MNDP	159	5678 → 5678 Len=117
87	50.054467338	Routerbo_1c:a3:4d	CDP/VTP/DTP/PAgP/UD…	CDP	93	Device ID: MikroTik Port ID: bridge40
88	50.054515041	Routerbo_1c:a3:4d	LLDP_Multicast	LLDP	110	MA/c4:ad:34:1c:a3:4d IN/bridge40 120 SysN=MikroTi
89	50.054832265	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
90	52.057066204	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
91	54.059279679	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
92	56.061487357	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
93	58.063693778	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
94	60.065899640	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
95	62.068118493	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
96	64.070327079	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
97	66.072535945	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
98	68.074745439	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
99	70.076954514	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
100	72.079183145	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0

Figura 8



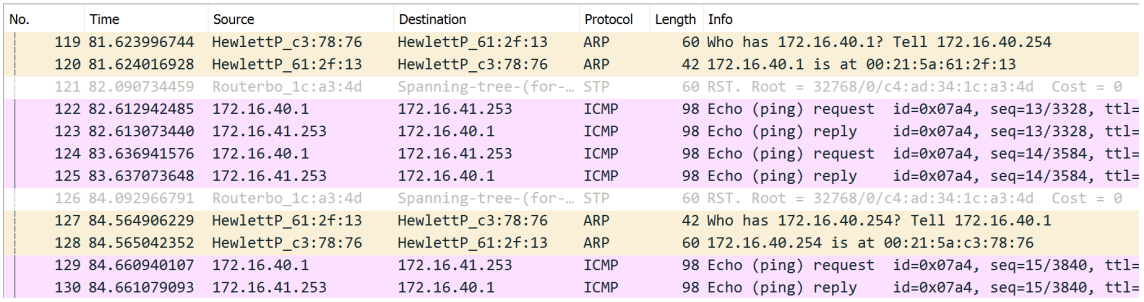
No.	Time	Source	Destination	Protocol	Length	Info
3	3.817722773	0.0.0.0	255.255.255.255	MNDP	159	5678 → 5678 Len=117
4	3.817739046	Routerbo_1c:a3:47	CDP/VTP/DTP/PAgP/UD…	CDP	93	Device ID: MikroTik Port ID: bridge41
5	3.817776620	Routerbo_1c:a3:47	LLDP_Multicast	LLDP	110	MA/c4:ad:34:1c:a3:47 IN/bridge41 120 SysN=MikroTi
6	4.004418943	Routerbo_1c:a3:47	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:47 Cost = 0
7	6.006652964	Routerbo_1c:a3:47	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:47 Cost = 0
8	8.008862890	Routerbo_1c:a3:47	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:47 Cost = 0
9	10.011068137	Routerbo_1c:a3:47	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:47 Cost = 0
10	12.013288260	Routerbo_1c:a3:47	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:47 Cost = 0
11	14.015490852	Routerbo_1c:a3:47	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:47 Cost = 0
12	16.017713280	Routerbo_1c:a3:47	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:47 Cost = 0
13	18.019931028	Routerbo_1c:a3:47	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:47 Cost = 0
14	18.920960530	172.16.41.1	172.16.41.255	ICMP	98	Echo (ping) request id=0x3f73, seq=1/256, ttl=64
15	19.933974197	172.16.41.1	172.16.41.255	ICMP	98	Echo (ping) request id=0x3f73, seq=2/512, ttl=64
16	20.022134109	Routerbo_1c:a3:47	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:47 Cost = 0
17	20.957967363	172.16.41.1	172.16.41.255	ICMP	98	Echo (ping) request id=0x3f73, seq=3/768, ttl=64
18	21.981964509	172.16.41.1	172.16.41.255	ICMP	98	Echo (ping) request id=0x3f73, seq=4/1024, ttl=6
19	22.024343406	Routerbo_1c:a3:47	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:47 Cost = 0
20	23.005969059	172.16.41.1	172.16.41.255	ICMP	98	Echo (ping) request id=0x3f73, seq=5/1280, ttl=6
21	24.026558361	Routerbo_1c:a3:47	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:47 Cost = 0
22	24.029968580	172.16.41.1	172.16.41.255	ICMP	98	Echo (ping) request id=0x3f73, seq=6/1536, ttl=6
23	25.053971314	172.16.41.1	172.16.41.255	ICMP	98	Echo (ping) request id=0x3f73, seq=7/1792, ttl=6

Figura 9



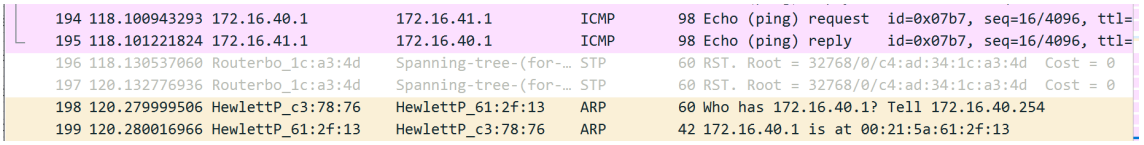
No.	Time	Source	Destination	Protocol	Length	Info
35	42.046792728	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
36	42.580905394	HewlettP_61:2f:13	HewlettP_c3:78:76	ARP	42	Who has 172.16.40.254? Tell 172.16.40.1
37	42.581025802	HewlettP_c3:78:76	HewlettP_61:2f:13	ARP	60	172.16.40.254 is at 00:21:5a:c3:78:76
38	42.612935195	172.16.40.1	172.16.40.254	ICMP	98	Echo (ping) request id=0x078e, seq=6/1536, ttl=6
39	42.613050504	172.16.40.254	172.16.40.1	ICMP	98	Echo (ping) reply id=0x078e, seq=6/1536, ttl=6
40	42.712001560	HewlettP_c3:78:76	HewlettP_61:2f:13	ARP	60	Who has 172.16.40.1? Tell 172.16.40.254
41	42.712020767	HewlettP_61:2f:13	HewlettP_c3:78:76	ARP	42	172.16.40.1 is at 00:21:5a:61:2f:13
42	43.636942666	172.16.40.1	172.16.40.254	ICMP	98	Echo (ping) request id=0x078e, seq=7/1792, ttl=6
43	43.637079697	172.16.40.254	172.16.40.1	ICMP	98	Echo (ping) reply id=0x078e, seq=7/1792, ttl=6

Figura 10



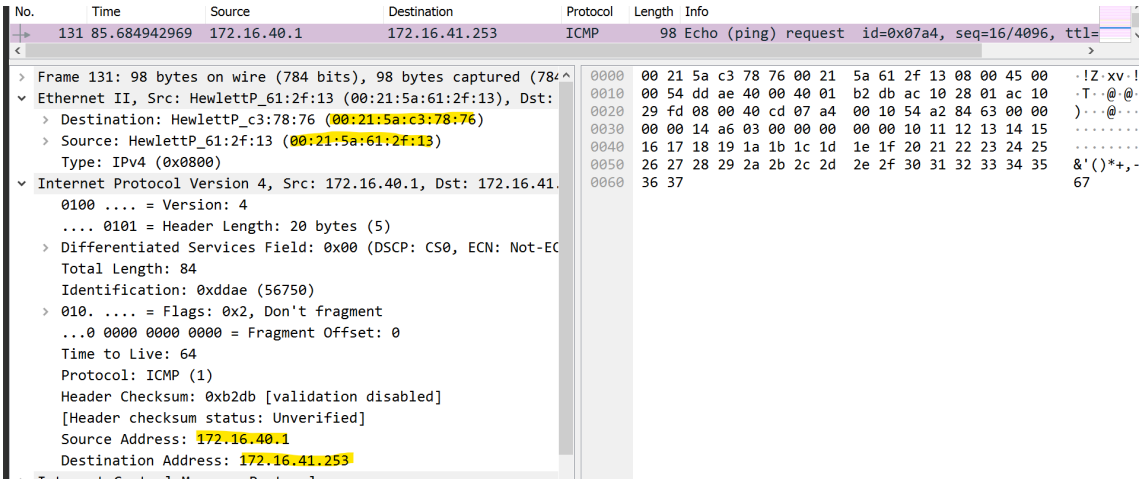
No.	Time	Source	Destination	Protocol	Length	Info
119	81.623996744	HewlettP_c3:78:76	HewlettP_61:2f:13	ARP	60	Who has 172.16.40.1? Tell 172.16.40.254
120	81.624016928	HewlettP_61:2f:13	HewlettP_c3:78:76	ARP	42	172.16.40.1 is at 00:21:5a:61:2f:13
121	82.090734459	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
122	82.612942485	172.16.40.1	172.16.41.253	ICMP	98	Echo (ping) request id=0x07a4, seq=13/3328, ttl=
123	82.613073440	172.16.41.253	172.16.40.1	ICMP	98	Echo (ping) reply id=0x07a4, seq=13/3328, ttl=
124	83.636941576	172.16.40.1	172.16.41.253	ICMP	98	Echo (ping) request id=0x07a4, seq=14/3584, ttl=
125	83.637073648	172.16.41.253	172.16.40.1	ICMP	98	Echo (ping) reply id=0x07a4, seq=14/3584, ttl=
126	84.092966791	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
127	84.564906229	HewlettP_61:2f:13	HewlettP_c3:78:76	ARP	42	Who has 172.16.40.254? Tell 172.16.40.1
128	84.565042352	HewlettP_c3:78:76	HewlettP_61:2f:13	ARP	60	172.16.40.254 is at 00:21:5a:c3:78:76
129	84.660940107	172.16.40.1	172.16.41.253	ICMP	98	Echo (ping) request id=0x07a4, seq=15/3840, ttl=
130	84.661079093	172.16.41.253	172.16.40.1	ICMP	98	Echo (ping) reply id=0x07a4, seq=15/3840, ttl=

Figura 11



194	118.100943293	172.16.40.1	172.16.41.1	ICMP	98	Echo (ping) request id=0x07b7, seq=16/4096, ttl=
195	118.101221824	172.16.41.1	172.16.40.1	ICMP	98	Echo (ping) reply id=0x07b7, seq=16/4096, ttl=
196	118.130537060	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
197	120.132776936	Routerbo_1c:a3:4d	Spanning-tree-(for-…	STP	60	RST. Root = 32768/0/c4:ad:34:1c:a3:4d Cost = 0
198	120.279999506	HewlettP_c3:78:76	HewlettP_61:2f:13	ARP	60	Who has 172.16.40.1? Tell 172.16.40.254
199	120.280016966	HewlettP_61:2f:13	HewlettP_c3:78:76	ARP	42	172.16.40.1 is at 00:21:5a:61:2f:13

Figura 12

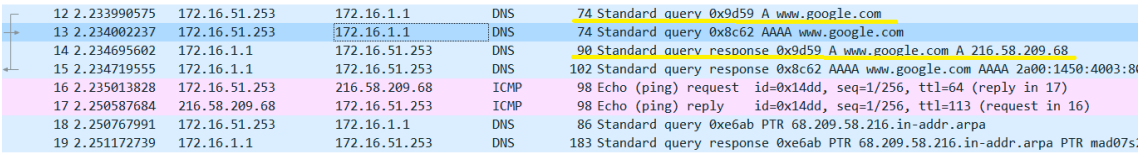


No.	Time	Source	Destination	Protocol	Length	Info
131	85.684942969	172.16.40.1	172.16.41.253	ICMP	98	Echo (ping) request id=0x07a4, seq=16/4096, ttl=

```
Frame 131: 98 bytes on wire (784 bits), 98 bytes captured (78
Ethernet II, Src: HewlettP_61:2f:13 (00:21:5a:61:2f:13), Dst:
  Destination: HewlettP_c3:78:76 (00:21:5a:c3:78:76)
  Source: HewlettP_61:2f:13 (00:21:5a:61:2f:13)
  Type: IPv4 (0x0800)
Internet Protocol Version 4, Src: 172.16.40.1, Dst: 172.16.41.
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-EC
  Total Length: 84
  Identification: 0xddae (56750)
  010. .... = Flags: 0x2, Don't fragment
  ...0 0000 0000 0000 = Fragment Offset: 0
  Time to Live: 64
  Protocol: ICMP (1)
  Header Checksum: 0xb2db [validation disabled]
  [Header checksum status: Unverified]
  Source Address: 172.16.40.1
  Destination Address: 172.16.41.253
```

```
0000  00 21 5a c3 78 76 00 21  5a 61 2f 13 08 00 45 00   ·!Z·xv·!  Za/···E·
0010  00 54 dd ae 40 00 40 01  b2 db ac 10 28 01 ac 10   ·T··@·@·  ····(···
0020  29 fd 08 00 40 cd 07 a4  00 10 54 a2 84 63 00 00   )···@···  ··T··c··
0030  00 00 14 a6 03 00 00 00  00 00 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



12	2.233990575	172.16.51.253	172.16.1.1	DNS	74 Standard query 0x9d59 A www.google.com
13	2.234002237	172.16.51.253	172.16.1.1	DNS	74 Standard query 0x8c62 AAAA www.google.com
14	2.234695602	172.16.1.1	172.16.51.253	DNS	90 Standard query response 0x9d59 A www.google.com A 216.58.209.68
15	2.234719555	172.16.1.1	172.16.51.253	DNS	102 Standard query response 0x8c62 AAAA www.google.com AAAA 2a00:1450:4003:8
16	2.235013828	172.16.51.253	216.58.209.68	ICMP	98 Echo (ping) request id=0x14dd, seq=1/256, ttl=64 (reply in 17)
17	2.250587684	216.58.209.68	172.16.51.253	ICMP	98 Echo (ping) reply id=0x14dd, seq=1/256, ttl=113 (request in 16)
18	2.250767991	172.16.51.253	172.16.1.1	DNS	86 Standard query 0xe6ab PTR 68.209.58.216.in-addr.arpa
19	2.251172739	172.16.1.1	172.16.51.253	DNS	183 Standard query response 0xe6ab PTR 68.209.58.216.in-addr.arpa PTR mad07s

Figura 14

download.c

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>
#include <fcntl.h>

#define TRUE 1

typedef struct urlData {
    char user[128];
    char password[128];
    char host[256];
    char url_path[240];
    char fileName[128];
    char ip[128];
} urlData;

// function given by professors
int getIp(char *host, struct urlData *urlData){
    struct hostent *h;

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

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

```
    printf("IP Address: %s\n", inet_ntoa(*((struct in_addr *)
h->h_addr)));

    return 0;
}

int getFileName(struct urlData *urlData){
  char fullpath[256];

  strcpy(fullpath, urlData->url_path);

  char* token = strtok(fullpath, "/");

  while(token != NULL){
    strcpy(urlData->fileName, token);
    token = strtok(NULL, "/");
  }

  return 0;
}

int parseUrlData(char *url, struct urlData *urlData ){

    char* ftp = strtok(url, "/");
    char* credentials = strtok(NULL, "/");  //
[<user>:<password>@]<host>
    char* url_path = strtok(NULL, "");      // url-path

    if (strcmp(ftp, "ftp:") != 0){
        printf("Error: Not using ftp protocol\n");
        return -1;
    }

    char* user = strtok(credentials, ":");
    char* password = strtok(NULL, "@");

    // no user:password given
    if (password == NULL){
        user = "anonymous";
        password = "anonymous";
        strcpy(urlData->host, credentials);
    } else {
```

```
        strcpy(urlData->host, strtok(NULL, ""));
    }

    strcpy(urlData->url_path, url_path);
    strcpy(urlData->user, user);
    strcpy(urlData->password, password);

    if(getIp(urlData->host, urlData) != 0){
        printf("Error: resolving host name\n");
        return 1;
    }

    if(getFileName(urlData) != 0){
        perror("Error: reading filename\n");
        exit(1);
    }

    return 0;
}

// function given by professors
int startConnection(char *ip, int port, int *sockfd){
    struct sockaddr_in server_addr;

    /*server address handling*/
    bzero((char *) &server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(ip);    /*32 bit
Internet address network byte ordered*/
    server_addr.sin_port = htons(port);        /*server TCP port
must be network byte ordered */

    /*open a TCP socket*/
    if ((*sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Error: socket()");
        exit(-1);
    }

    /*connect to the server*/
    if (connect(*sockfd, (struct sockaddr *) &server_addr,
sizeof(server_addr)) < 0) {
        perror("Error: connect()");
        exit(-1);
```

```
    }

    return 0;
}

void gerIpAndPort(char * ip, int *port, FILE * readSockect){
    char *line = NULL;
    size_t len = 0;

    while(TRUE){
        getline(&line, &len, readSockect);
        printf("> %s", line);
        if(line[3] == ' '){
            break;
        }
    }

    int ip_address[4];
    int port_address[2];

    sscanf(line, "227 Entering Passive Mode (%d, %d, %d, %d, %d,
%d).\n", &ip_address[0], &ip_address[1], &ip_address[2],
&ip_address[3], &port_address[0], &port_address[1]);
    free(line);

    sprintf(ip, "%d.%d.%d.%d", ip_address[0], ip_address[1],
ip_address[2], ip_address[3]);
    *port = port_address[0] * 256 + port_address[1];
}

int sendCommandToServer (int sockfd, char * command){
    int bSent;

    bSent = write(sockfd, command, strlen(command));

    if (bSent == 0){
        printf("sendCommand: Connection closed\n");
        return 1;
    } else if (bSent == -1){
        printf("Error: sending command\n");
        return 1;
    }
```

```
    printf("%s\n",command);

    return 0;
}

int readServerReply(FILE * readSockect){
    long code;
    char *buf;
    size_t bufsize = 0;

    while(getline(&buf, &bufsize, readSockect) != -1){

        printf("> %s", buf);

        if(buf[3] == ' '){
            code = atoi(buf);

            if(code <= 559 && code >= 500){
                printf("Error!!\n");
                exit(1);
            }
            break;
        }
    }

    return 0;
}

int readFile(char *fileName, int sockfdReceive ){
    int file;

    if ((file = open(fileName, O_WRONLY | O_CREAT, 0666)) == -1) {
        perror("Error: Couldn't open file");
        return -1;
    }

    char c[1];
    int b;

    while (TRUE){
        b = read(sockfdReceive, &c[0], 1);
        if (b == 0) break;
        if (b < 0){
```

```
            close(file);
            return -1;
        }

        b = write(file, &c[0], 1);
        if (b < 0){
            close(file);
            perror("Error: Couldn't write to file");
        }
    }

    close(file);

    return 0;
}

int download(struct urlData urlData){

    int sockfd, sockfdReceive;

    if(startConnection(urlData.ip, 21, &sockfd) != 0){
        printf("Error: Starting connection\n");
        return -1;
    }

    FILE * readSockect = fdopen(sockfd, "r");
    readServerReply(readSockect);

    // credentials
    char command[256];
    sprintf(command, "user %s\n", urlData.user);

    sendCommandToServer(sockfd,command);
    readServerReply(readSockect);

    sprintf(command, "pass %s\n", urlData.password);

    sendCommandToServer(sockfd,command);
    readServerReply(readSockect);

    // set mode
    sprintf(command, "pasv \n");
    sendCommandToServer(sockfd,command);
```

```
    //read ip and port
    char ip[32];
    int port;
    gerIpAndPort(ip, &port, readSockect);
    printf("ip: %s\n",ip);
    printf("port: %i\n", port);

    //open connection to receive data
    if(startConnection(ip, port, &sockfdReceive) != 0){
        printf("Error: Starting connection\n");
        return -1;
    }

    sprintf(command, "retr %s\r\n",urlData.url_path);
    sendCommandToServer(sockfd,command);
    readServerReply(readSockect);

    readFile(urlData.fileName, sockfdReceive);

    //close
    sprintf(command, "quit \r\n");
    sendCommandToServer(sockfd,command);

    return 0;
}

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

    if(argc != 2){
          printf("Incorrect program usage\n"
                "Usage: download
ftp://<user>:<password>@<host>/<url>\n");
        exit(1);
    }

    struct urlData urlData;

    if(parseUrlData(argv[1], &urlData) != 0){
        printf("Error: Parsing input\n");
        return 1;
    }
```

```
    if(download(urlData) == -1){
        printf("Error: Downloading file\n");
        exit(-1);
    }

    return 0;
}
```